Group-Centric Secure Information Sharing Models

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

By

Ram Narayan Krishnan
Master of Science
New Jersey Institute of Technology, 2003
Bachelor of Science
Pondicherry University, 2002

Co-Director: Dr. Ravi Sandhu, Professor
University of Texas, San Antonio
Co-Director: Dr. Daniel A. Menascé, Professor
Department of Computer Science
George Mason University

Fall Semester 2009
George Mason University
Fairfax, VA

Dedication

To the members of my near and extended family whose never ending love and support inspired this endeavor.

Acknowledgments

I would like to express my most sincere appreciation and profound gratitude to my advisor Prof. Ravi Sandhu for his guidance and support from the formative stages of this thesis to its completion. Learning and working with him has been my most rewarding experience both professionally and personally. I am further thankful to him for continuing to advice even after leaving GMU.

I would like to extend my sincere gratitude to my dissertation co-director Prof. Daniel Menasce and committee members Prof. Larry Kerschberg, Prof. Robert Simon, and Prof. Duminda Wijesekera for their valuable comments and suggestions. I would also like to thank them for facilitating smooth completion of this thesis in spite of Prof. Sandhu's move to University of Texas at San Antonio. Special thanks to Prof. Menasce for his continued advice and support by stepping in to act as my dissertation co-director.

I would also like to thank Prof. Jianwei Niu and Prof. William Winsborough of University of Texas at San Antonio for their immense support and valuable advice that greatly helped me with many chapters in this thesis.

Table of Contents

List of Tables

List of Figures

Abstract

GROUP-CENTRIC SECURE INFORMATION SHARING MODELS

Ram Narayan Krishnan, PhD

George Mason University, 2009

Dissertation Co-Director: Dr. Ravi Sandhu

Dissertation Co-Director: Dr. Daniel A. Menascé

In this dissertation, we introduce a novel approach for secure information sharing characterized as "Group-Centric". Traditional approaches to information sharing include "Dissemination-Centric" and "Query-Centric" sharing. "Dissemination-Centric" sharing focuses on attaching attributes and policies to an object as it is disseminated from producers to consumers in a system. In "Query-Centric" sharing, information seekers construct appropriate queries to obtain authorized information from the system. The primary focus of this mode of sharing has been on preventing inference of unauthorized information from authorized information obtained by querying a database. In contrast, Group-Centric sharing envisions bringing the users and objects together in a group to facilitate sharing for some purpose. The metaphors "secure meeting room" and "subscription service" characterize the Group-Centric approach where participants and information come together to share for some common purpose and authorizations depend upon relative membership period of users (participants) and objects (information).

In this dissertation, we follow the Policy, Enforcement and Implementation (PEI) framework to develop respective models for Group-Centric Secure Information Sharing (g-SIS). The PEI framework facilitates security policy and design decisions to be made at three

distinct yet related layers of secure systems design. At the policy layer, we develop the foundations for a theory of g-SIS by characterizing a set of core properties and specifying a family of models. We focus on semantics of group operations: Join and Leave for users and Add and Remove for objects, each of which can have several variations. We use Linear Temporal Logic (LTL) to characterize the core properties of a group in terms of these operations. We also characterize additional properties for specific types of these operations. We specify the authorization behavior for a family of g-SIS models and prove that these models satisfy the core g-SIS properties.

At the enforcement layer, we specify an architecture for g-SIS based on super-distribution, micro-distribution and a hybrid object distribution model. As we will see, the hybrid model addresses the limitations of super-distribution and micro-distribution model. Further, we characterize and define the problem of "stale-safety" in g-SIS. In a distributed system such as g-SIS, "stale-safety" is concerned with enforcing safe authorization behavior given that authorization decisions will inevitably be made based on stale attribute information. Attribute staleness arises due to the physical distribution of authorization information, decision and enforcement points. While it may not be practical to eliminate staleness, we can limit unsafe access decisions made based on stale authorization information such as user and object attributes. We propose and formally specify stale-safe security properties of varying strength. Again, we use LTL to formalize these properties allowing them to be verified using automated techniques such as model checking. We model the authorization information, decision and enforcement points of the g-SIS system as finite state machines and verify using model checking that the model satisfies the stale-safe security properties.

Finally, at the implementation layer, we discuss Trusted Computing Technology based protocols and models for g-SIS. A fundamental requirement for g-SIS is that protection needs to extend to clients. Trusted Computing Technology provides a hardware root of trust through the Trusted Platform Module (TPM). A Trusted Reference Monitor (TRM) on client platforms faithfully enforces group policies. We provide concrete TPM based protocols and outline an implementation model to realize the enforcement models discussed earlier. As a proof-of-concept, we implement a critical protocol, called the provisioning,

protocol that is concerned with secure provisioning of group credentials on user's platform. At the end of the protocol, the group credentials will only be accessible to the TRM in the user's platform in a trustworthy platform state.

Chapter 1: Introduction and Motivation

Secure Information Sharing (SIS) or sharing information while protecting it is one of the earliest problems to be recognized in computer security, and yet remains a challenging problem to solve. The application scenarios of SIS are endless ranging from revenue centric retail Digital Rights Management (DRM) and sensitivity centric intellectual property to national security centric secret information. Post 9/11, the need-to-share principle has supplanted the traditional need-to-know which caused failure to "connect the dots." In our information age, businesses collaborate not only with allies, but also with competitors. For the individual citizen, modern health care requires timely sharing of medical information amongst care providers while maintaining privacy. The explosive phenomena of social networking enables individuals to interact without geographic barriers, but with an expectation of security and privacy.

In all of these cases, we see the need to "share but protect." In this dissertation, we introduce the theory of Group-Centric SIS (g-SIS) and develop formal models for g-SIS. Intuitively, users and information come together in a group to facilitate sharing. We identify two metaphors: a secure meeting room and a subscription service. A meeting room brings people together to "share" information for some common purpose. The purpose can range from collaboration on a specific goal-oriented task (such as designing a new product or merger and acquisition) to participation in a shared activity (such as a semester long class) to a dynamic coalition (such as a mission-oriented group driven towards completion of a particular task). The subscription metaphor speaks to potentially larger scale sharing with a publisher disseminating information to subscribers who in turn participate in blogs and forums. We show that these simple and familiar metaphors enable a rich space of policies

that will be systematically investigated. In particular, we show that the temporal interactions of users joining and leaving the group and information being added and removed is critical to determination of who can see what in the group.

Secure meeting room: In general, a meeting room has the notion of simultaneous presence of participants engaged in the meeting. Visualize a conversation room (albeit virtual) where users may join, leave and re-join, but only hear the conversation occurring during their participation period. Users bring objects (e.g. documents) to this room wherein they are asynchronously accessible to participants from different stakeholders and third-party agents. Users' participation may be intermittent as influenced by their availability, need to know, etc. Let us consider some example scenarios:

Program committee meeting: Typically committee members are restricted to conversations occurring in their presence. Alice, a committee member, may be excused from the room when her paper or proposal is being discussed and may re-join after that discussion has concluded. The conversation that occurred during her absence is not accessible to her. In a different setting, such conversations may be recorded in a smart board and made available to her on return.

Collaborative product development: Consider collaborative product design between ABC Corp. and XYZ Corp. Say ABC establishes a group by gathering engineers from across the company. Certain sensitive documents are provided to these ABC engineers, but are not accessible to XYZ engineers. Documents created after the XYZ engineers join the group are shared by both ABC and XYZ engineers. Such documents may represent new intellectual property during the collaboration period. Moreover, both XYZ and ABC engineers retain access to such new documents even after leaving the group. In a different consulting scenario, incoming XYZ engineers may access the sensitive ABC documents during their membership period, but lose access once the collaboration ends.

Employee stock options: Stock option benefits typically change over time. New employees only get to see benefits as of their joining and not previously existing ones. Similarly,

organizations may share some information with existing employees, but withhold it from future employees. In these cases, certain objects are shared only with existing group members and not with future members. Furthermore, when employees leave the company, they may be allowed to retain certain information (such as their profile, recommendations, etc.), but denied access to sensitive proprietary information (such as design documents, code, etc.).

Subscription service: This metaphor speaks potentially to large-scale sharing. Here access to content may depend upon when the subscription began and the terms of subscription.

Magazine Subscription: Consider an online news magazine ABS Corp. that offers four levels of membership. Level 1 ($10/year) subscribers can access news articles that are published after they started paying the subscription fee. If they cancel their subscription, they completely lose access. In addition to Level 1 services, Level 2 ($12/year) subscribers can retain access to news articles that they paid for even after canceling their subscription. In addition to Level 2 services, Level 3 ($15/year) subscribers can access rich archives filled with post-news analysis, predictions, annotations and opinions from experts but lose all access on cancelling their subscription. Level 4 ($17/year) subscribers, in addition to Level 3 benefits, can view all articles that they had access to before leaving, even after they discontinue the service.

Secure multicast: In secure multicast [1], typically new members joining the group cannot access content distributed prior to their join time (backward secrecy). Similarly, members leaving the group can no longer access any new content (forward secrecy). Thus access is dependant on when the users join and leave the group.

Clearly, the "secure meeting room" metaphor suggests a smaller scale information sharing scenario whereas "subscription service" indicates a potentially larger-scale. These examples illustrate two important principles in the group-centric approach. The first principle is "share but differentiate". Sharing is enabled by joining and adding information to a group. Yet, users access is differentiated by the time at which they join and the time at which the requested information is added to the group, as well as possibly by other attributes. The

second principle is the notion of "multiple groups" with possibly overlapping users. The relationship between these groups can be of any number of varieties familiar to computer scientists. One well-known structure is that of a hierarchy, where users at a higher level dominate those at the lower levels. Another common relationship is that of mutual exclusion where the same user is prohibited from joining conflicting groups.

1.1 Motivation

Traditional access control models do capture certain important SIS aspects but have not been satisfactory in practice. For example, Discretionary Access Control or DAC [2–4] captures the notion of owner control. In other words, as owner of an object, the user has the ultimate authority to decide who can access the object. While this is an important SIS aspect, DAC is fundamentally limited in that it controls access only to the original object but not to copies. If objects could be read, one can read and create a copy of this object. Objects are only protected up to the point when read access is granted to a user. From then on, the owner loses confidentiality of the object potentially to other unintended users[1]. Further, DAC is also too fine-grained in practice since the secure information sharing responsibility falls on the owner of the information. The system provides no guidance as to how information can be effectively shared.

Next, Mandatory Access Control or MAC[2] models such as that of Bell-LaPadula (BLP) [6] capture the important notion of information flow. In BLP, information can only flow from a subject of lower clearance to that of higher clearance and not vice-versa. The intended objective is that of confidentiality of objects at higher security clearance from that of subjects executing at lower clearance. This is common in military where, for example, only Generals can see certain information and not Soldiers and potentially malicious subjects executing with the General's clearance cannot share information with those executing

[1]It should be noted that DAC is effective with respect to write controls, since writing to the original authoritative object is very different from writing to a copy. However, for reading, an up to date copy is as good as the original.

[2]In this dissertation, we use the term MAC synonymous to Lattice-Based Access Control [5].

with a Soldier clearance. However, such lattice based information flow has proven to be too rigid in practice for modern information sharing scenarios. For example, the system does not allow to set up a group of Generals and Soldiers for a specific mission to which other Generals and Soldiers may not have access[3].

The modern concept of Role-Based Access Control (RBAC) [7] can be viewed as an evolution of access control to simplify administration in organizations bringing in additional concepts such as hierarchies and constraints. It has also been shown that RBAC is policy neutral in the sense that it can be configured to enforce both DAC and MAC policies [8]. However, RBAC is too general and does not directly address information sharing. Attribute based access control models such as that of UCON [9] also suffer from being too general and providing no explicit framework or guidance for secure sharing.

We can categorize two primary issues in the models discussed so far in the context of information sharing: Copy Control and Manageability. The traditional approach to information sharing in the last two decades, which we characterize as Dissemination-Centric, focuses on attaching attributes and policies to an object as it is disseminated from producers to consumers in a system. These policies are sometimes described as being "sticky" [10–12]. As an object is disseminated further down a supply chain the policies may get modified, such modification itself being controlled by existing policies. This mode of information sharing goes back to early discussions on originator-control systems [13–16] in the 1980s and Digital and Enterprise Rights Management systems in the 1990s and 2000s. XrML [17] and ODRL [18] are recent examples of policy languages developed for this purpose. Another mode of sharing which can be characterized as Query-Centric has primarily focussed on the problem of inference or de-aggregation [19]. In Query-Centric sharing, information seekers construct appropriate queries to obtain authorized information from the system. The primary focus of this mode of sharing has been on preventing inference of unauthorized information from authorized information obtained by querying a database.

[3]Compartments are inadequate in practice because it may not be clear in the beginning if the requirement warrants establishing a new compartment by going through costly administrative processes (since the group may not be long-lived).

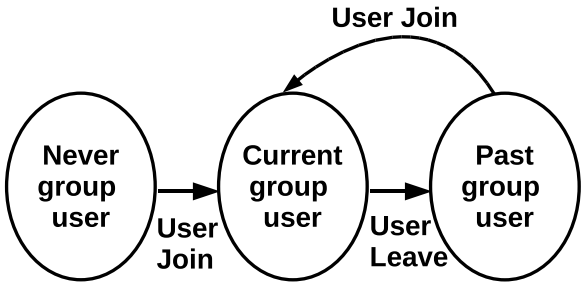

Figure 1.1: User Membership States.

While such systems mainly focussed on copy/usage control, the manageability problem has remained largely unaddressed. Group-Centric sharing differs in that it advocates bringing the users and objects together to facilitate sharing by focussing on semantics of group operations. We envision that Dissemination and Group-Centric sharing will co-exist in a mutually supportive manner. For example, objects could be added with "sticky" policies in a Group-Centric model. Furthermore, from a practical perspective, a complete Group-Centric information sharing system would exercise advantages of appropriate models such as DAC, ORCON and RBAC (for owner control, administration, etc.), Lattice Based Access Control such as the Bell-LaPadula model (for restricting information flow), UCON (for dynamic aspects such as rate limiting access), etc.

1.2 Dissertation Scope

There are various aspects to g-SIS such as operational and administrative aspects in a group and inter-group relationships. Operational aspects include questions such as what are the characteristics or security properties of a group, what are the semantics of user operations such as Join and Leave and object operations such as Add and Remove and models with and without versioning of objects (where a write operation on an object creates a new version of that object). Administrative aspects include questions such as who authorizes, Join, Leave, Add, Remove, etc. Finally, it is important to study useful inter-group relationships such as hierarchical groups, conditional membership, mutual exclusion, etc.

Our primary focus in this dissertation is on the operational aspects of g-SIS including the semantics of the basic group operations and their temporal interactions. We propose an abstract set of group operations: Join and Leave for users, Add and Remove for objects. Users may Join, Leave and re-Join the group. This is illustrated in figure 1.1. Similarly, objects may be Added, Removed and re-Added to the group. Further each of these operations could be of various types such as Lossy/Lossless, Restorative/Non-Restorative, etc. For example, in Lossless Join, a joining user never loses access to objects authorized prior to joining the group. In Restorative Join, the joining user may regain access to objects authorized during past membership period. Temporal aspects of access control have been previously studied, where they are introduced as extensions to prior models such as that of Role Based Access Control (e.g., [20, 21]). Temporal aspects in such models typically focus on when a user can activate his/her authorizations. For instance, a normal user may be allowed to activate his/her permissions only between 8AM to 5PM while an administrative user may activate at anytime. Temporal semantics in g-SIS is fundamentally different in that, we focus on the authorizations enabled depending on the time group operations such as join, leave, add and remove occur.

We recognize the importance of authorization for these operations. It is clearly not sufficient for a security policy to only specify the semantics of group operations. A complete model must also specify the authorization for these operations. In simple cases, a distinguished group owner may be responsible for all of these operations. More realistically the authorization will be decentralized and distributed. The problem of decentralized authorization and its administration has been investigated in the access control literature for over three decades [22–27]. We believe that authorizations concerning the operational aspects that bear on group membership is a more immediate and novel problem, and this will be the focus of this dissertation. Without a basic understanding of group operation semantics, we believe that it would be premature to consider administrative models. Henceforth, we leave the development of an administrative g-SIS model for future work.

Finally, we also perform a case study of the application of g-SIS in a concrete inter-organizational collaboration scenario. For this specific scenario, we develop an administrative model and operational model. The administrative model specifies authorizations for various aspects such as establishing a group, joining a group, etc. The operational model specifies the authorizations for a user to create a subject and exercise his/her privileges in the group or the organization.

We conclude by exploring inter-group relationships such as membership domination and conditional membership and perform a case-study of application of such relationships and operational semantics to promote sharing between different lattices of the Bell-LaPadula model.

Within the above-mentioned scope, we explore Policy, Enforcement and Implementation models (PEI models) for g-SIS based on the PEI framework for secure systems design introduced in [28]. In the policy layer, we formally develop a family of g-SIS models by specifying a core set of properties that are required of any g-SIS model. We develop various enforcement models for g-SIS and specify additional enforcement level security properties that are concerned with making safe authorization decisions in the context of stale authorization information. Finally, we outline an implementation model for g-SIS using Trusted Computing Technology and implement a provisioning protocol as a proof-of-concept. We assume that a Trusted Execution Environment exists on the client for a reference monitor to faithfully enforce group policies.

1.3 Thesis Organization

We begin by providing necessary background in chapter 2 comprising brief reviews of the PEI framework, Linear Temporal Logic, Model Checking and Trusted Computing Technology. At the policy layer (chapter 3), we develop the foundations for a theory of g-SIS and characterize a specific family of models in this arena. We focus on semantics of group operations: Join and Leave for users and Add and Remove for objects, each of which can have several variations called types. We use Linear Temporal Logic (LTL) [29] to first

characterize the core properties of a group in terms of these operations. We prove the consistency and independence of these core properties. We then characterize additional properties for specific types of group operations. We specify the authorization behavior for a family of g-SIS models and prove that these models satisfy the core g-SIS properties.

At the enforcement layer (chapter 4), we specify an architecture for g-SIS based on super-distribution, micro-distribution and a hybrid object distribution model. As we will see, the hybrid model addresses the limitations of super-distribution and micro-distribution model. Further, we address the problem of "stale-safety" in the g-SIS architecture. As we will see, g-SIS, like any other distributed system, suffers from access violation due to stale authorization information. We propose stale-safe security properties of varying strength and prove using model checking that the g-SIS enforcement model satisfies appropriate stale-safe properties.

Finally, at the implementation layer (chapter 5), we discuss Trusted Computing Technology based protocols and models for g-SIS. A fundamental requirement for g-SIS is that protection needs to extend to clients. Trusted Computing Technology provides a hardware root of trust through the Trusted Platform Module (TPM). A Trusted Reference Monitor (TRM) on client platforms faithfully enforces group policies. We outline an implementation model and provide concrete TPM based protocols to realize the enforcement model discussed earlier. As a proof-of-concept, we implement one of the protocols, called the provisioning protocol, that is concerned with secure provisioning of group credentials on user's platform. At the end of the protocol, the group credentials will only be accessible to the TRM in the user's platform in a trustworthy platform state. We conclude in chapter 6.

1.4 Contributions

The contributions of this thesis can be respectively categorized at the Policy, Enforcement and Implementation layers of Group-Centric Secure Information Sharing design.

Policy Layer Contributions

- Major contributions at the Policy layer are as follows:

  ⇒ A formal model of a single group g-SIS model is developed. This includes formal characterization of the following:

    * Identification of a core set of properties that are required of any g-SIS specification.
    * Proof of consistency and independence of core properties.
    * Identification and formal specification of a set of useful group operation semantics.

  ⇒ A family of g-SIS specifications, called the π-system, supporting a variety of g-SIS operation semantics are specified.

  ⇒ A formal proof that the π-system satisfies the mandatory core g-SIS properties.

- Other contributions include:

  ⇒ An object versioning model is proposed and the g-SIS core properties are extended to support versioning.

  ⇒ A case-study of inter-organizational collaboration with a complete specification including an administrative and operational model that supports object versioning is presented.

  ⇒ An initial framework for developing more sophisticated models for Group-Centric sharing is proposed.

Enforcement Layer Contributions

- Major contributions at the Enforcement layer are as follows:

  ⇒ A set of enforcement level security properties, called stale-safety, is identified and formalized for g-SIS. These properties ensure that in the proposed distributed

g-SIS architecture if a user is able to perform an action on an object, the authorization for that action is guaranteed to have held in the recent past.

⇒ Specification of a complete g-SIS architecture using Finite State Machines (FSM).

⇒ Formal verification of weak and strong stale-safe properties using model checking against the FSMs.

- Other contributions include:

  ⇒ A g-SIS architecture supporting super and micro object distribution is proposed.

  ⇒ A more practical hybrid approach using split-key RSA is specified.

Implementation Layer Contributions At the implementation layer, no fundamental novelty is claimed. However, since the g-SIS system would not be complete without an implementation model, a collection of Trusted Platform Module (TPM) based protocols is specified that realizes the g-SIS architecture. The feasibility of the proposed approach using standard platforms with TPM is ascertained by means of a proof-of-concept. Specifically, a critical component called the provisioning protocol is implemented. In the protocol, an authoritative server securely provisions the group key on the user's machine in such a manner that the group key is accessible to a Trusted Reference Monitor only in a trustworthy platform state. This protocol involves exercising the TPM capabilities extensively and thus serves as an excellent means for in-depth understanding of the underlying system.

Chapter 2: Background

In this section, we provide a brief overview of necessary background on the PEI framework, Linear Temporal Logic, Model Checking and Trusted Computing which are extensively used in this dissertation.

2.1 PEI Framework

The Policy-Mechanism separation principle is a well-known approach for secure systems design and has been recognized as such for many decades (see for example [30]). This decoupling allows for building flexible systems that support a wide range of policies. Policy is concerned with "what" security policy is to be enforced while mechanism is concerned with "how" the security policy can be realized. While this abstract distinction was sufficient in the early times when most systems dealt with a single operating system with limited resources and applications, it has remained a major challenge to close the gap in two steps in modern complex distributed systems with varying levels of trust and service dependencies. The Policy, Enforcement and Implementation (PEI) framework [28], illustrated in figure 2.1, has been recently proposed to close this gap.

In figure 2.1, the Objective layer specifies the desired security and functional objectives and is necessarily informal. The purpose of the Policy layer is to formally specify the security objectives. A notable example of a policy model is the lattice model for mandatory access control [6]. Security policy is specified using users, subjects, objects, roles, groups, etc. and security analysis at this layer involves verification that desirable high-level security properties hold in the system. The simple security and star properties for information flow in the Bell-LaPadula model [6] are examples of such high-level properties that are concerned, for example, only with subjects and objects and their actions. Note that in the Policy layer

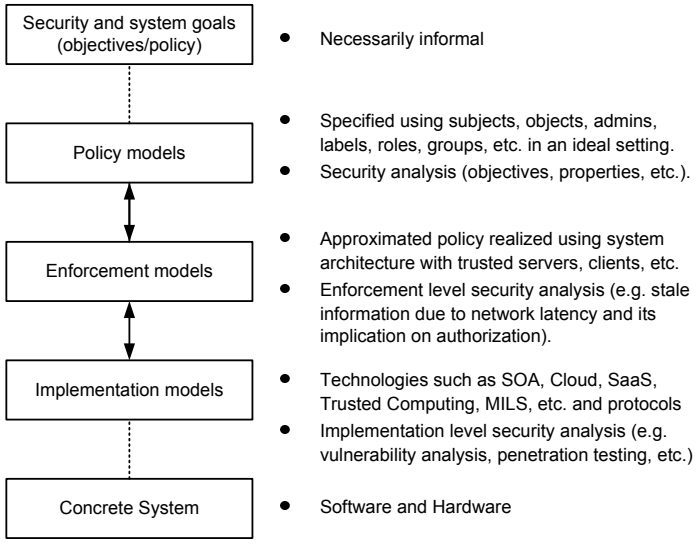

Figure 2.1: The PEI framework.

we assume an ideal setting for policy specification and ignore practical realities such as stale authorization information at a Policy Decision Point in a distributed system due to, for example, network delay.

The Enforcement layer is concerned with "how" to realize the policy specified in the Policy layer by specifying system architecture using, for example, trusted servers, third-party service providers, clients, etc. At this layer, real-world limitations can no longer be ignored and the models must explicitly address such issues. For example, it is inevitable that authorization information will be stale at the Policy Decision Point in a distributed system if only due to network delay. The Enforcement layer should explicitly address how to limit staleness and set realistic expectations. We strongly believe that it is unrealistic to expect ideal policy enforcement in most non-trivial systems and thus appropriately name the policy specified at the Policy layer as Ideal Policy and the one that is enforced in the Enforcement layer as the Approximate policy. Like the Policy layer, security analysis can be performed in the Enforcement layer as well. For instance, in this dissertation, we specify a few enforcement level security properties for stale-safety and successfully verify that our enforcement model satisfies those properties.

Table 2.1: Intuitive summary of temporal operators used in this dissertation

Future/Past	Operator	Read as	Explanation
Future	$\bigcirc$	Next	($\bigcirc$ p) means that the formula p holds in the next state.
	$\Box$	Henceforth	($\Box$ p) means that the formula p will continuously hold in all future states starting from the current state.
	$\mathcal{U}$	Until	(p $\mathcal{U}$ q) means that q will occur sometime in the future and p will hold at least until the first occurrence of q.
	$\mathcal{W}$	Unless	(p $\mathcal{W}$ q) is a weaker form of (p $\mathcal{U}$ q). It says that p holds either until the next occurrence of q or if q never occurs, it holds throughout.
Past	$\ominus$	Previous	($\ominus$ p) means that formula p held in the previous state.
	$\blacklozenge$	Once	($\blacklozenge$ p) means that formula p held at least once in the past.
	$\mathcal{S}$	Since	(p $\mathcal{S}$ q) means that q happened in the past and p held continuously from the position following the last occurrence of q to the present.

The Implementation layer is concerned with closing the final gap between the Enforcement layer and realizing the concrete system. The implementation layer specifies models accommodating the technologies that will be used for final implementation. Examples of such technologies include trusted computing using Trusted Platform Module, cloud computing, service-oriented architecture, etc. The last layer, labeled Concrete System, involves actual implementation and code generation.

2.2 Linear Temporal Logic

In this dissertation, we extensively use Linear Temporal Logic (LTL) [29] for specifying g-SIS policies and properties. A brief overview of LTL operators used in this dissertation is given in table 2.1. Temporal logic is a specification language for expressing properties related to a sequence of states in terms of temporal logic operators and logic connectives (e.g., $\wedge$ and $\vee$). Temporal logic operators are of two types: Past and Future. The past operators $\ominus$, $\blacklozenge$ and $\mathcal{S}$ (read previous, once and since respectively) have the following semantics. Formula ($\ominus$ p) means that the p was true in the previous state. Note that $\ominus$ p is false in the very first state. Formula ($\blacklozenge$ p) means that the p holds at least once in the past (i.e., in some previous state). Note that if p is true in the current state, ($\blacklozenge$ p) is trivially

true. Formula ($p \mathcal{S} q$) means that q held sometime in the past and p has held continuously following the last occurrence of q to the present. Again, if q is true in the current state, ($p \mathcal{S} q$) is trivially true (even if p is false in the current state). The future operators $\bigcirc$, $\Box$, $\mathcal{U}$ and $\mathcal{W}$ (read next, henceforth, until and unless respectively) have the following semantics. Formula ($\bigcirc p$) means that p is true in the next state. Formula ($\Box p$) means that p holds in all future states, including the current state. Formula ($p \mathcal{U} q$) means that q will occur sometime in a future state and formula p will continuously hold in every state at least until that future state. If q is true in the current state, ($p \mathcal{U} q$) is trivially true in the current state (even if p is false in the current state). Finally, the unless operator ($\mathcal{W}$) is a weaker form of the until operator ($\mathcal{U}$). Formula ($p \mathcal{U} q$) predicts the future occurrence of q and thus will be false if q never occurs. Formula ($p \mathcal{W} q$), on the other hand, says that p will hold until the next occurrence of q. But if q never occurs in the future, p will hold indefinitely.

2.3 Model Checking

Model checking [31, 32] is an automated verification technique that analyzes a finite model of a system (i.e., a finite state machine (FSM) that produces computation traces consisting of infinite sequences of states) and exhaustively explores the state space of the model to determine whether desired properties hold in the model. In the case that a property is false, a model checker produces a counterexample consisting of a trace that violates the property, which can be used to correct the model or modify the property specification.

SMV [33, 34] is a family of model checking tools based on Binary Decision Diagrams (BDDs). BDDs represent states very compactly. In SMV, models are represented by using variables that are assigned values in each step of the FSM. Properties to be checked are specified by temporal logic [35] formulas. SMV provides built-in finite data types, such as boolean, enumerated type, integer range, arrays, and bit vectors. In SMV, the initial state is defined by assigning initial values to state variables. State transitions are specified by

assigning values to be assumed by each state variable x in the next state, which is denoted by next(x). Each such value is given by an expression over variables in either the current or the next state. The assignments are effectively performed simultaneously to obtain the subsequent state. SMV allows nondeterministic assignment, i.e., the value of variable is chosen arbitrarily from the set of possible values. Expression operators !, &, |, and -> represent logical operators "not", "and", "or" and "implies", respectively. Comments follow the symbol "- -". SMV supports macros, which are replaced by their definitions, so they do not increase the system's state space.

2.4 Trusted Computing

Trusted Computing Technology is an industry standard proposed by the Trusted Computing Group (TCG) [36]. It is widely accepted that software only mechanisms cannot provide high assurance. This motivated TCG to provide a root of trust at the hardware level through a security chip called the Trusted Platform Module (TPM). The technology has evolved to a great degree now and we only provide a very brief overview here. The TPM mainly offers three novel features: Trusted Storage for keys, Trusted Capabilities and Platform Configuration Registers (PCR's) for storing integrity measurements.

Trusted Storage for keys is provided by encrypting user's keys with a chain of keys. The root key called Storage Root Key (SRK), at the top of the chain is stored within the TPM and is not accessible outside the TPM. The private part of SRK never leaves the TPM. SRK can be used to encrypt data (keys or arbitrary blob) that can later be decrypted only with the same TPM. A chain of keys can be created (keys in the leaves encrypted with the parent and so on and so forth) where the root key is SRK.

A TPM has many hardware Platform Configuration Registers (PCR). A PCR is a register that is capable of holding 160-bit SHA1 hash values. The idea is that as a machine boots up, all the software that are loaded is measured in sequence thus resulting in a final 160-bit SHA1 hash value that reflects the specific boot sequence and the state of software

loaded in the main memory of the machine. This value is registered in the PCR and may be reported to a remote entity which can verify the trustworthiness of the machine by comparing with a well-known PCR value.

Trusted Capabilities are capabilities exposed by the TPM that are guaranteed to be trustworthy. Users cannot modify the behavior of these capabilities.

Seal is a trusted capability exposed by the TPM. In the simplest case, a seal operation takes a key or a data blob and appends it to a PCR value and encrypts it using the SRK. This secret can later be unsealed only if the current PCR value in the TPM is same as the value mentioned the sealed blob. Thus a seal operation allows an entity to specify the software environment under which a blob may be accessed by any entity in a platform.

CertifyKey is another trusted capability where a public part of a key-pair and the PCR value under which the private counterpart may be accessed, is collectively signed using the TPM. If (P_{priv}, P_{pub}) is an asymmetric key pair, then a certify key operation, $\{P_{pub}||PCR\}\_Sign$, means that a) P_{priv} is protected using the SRK, b) P_{priv} is non-migratable, i.e., it cannot be used in any other platform other than the TPM that created it, and c) P_{priv} is sealed to a software state of PCR. Thus any external entity can encrypt a secret using P_{pub} with the assurance that the secret can be decrypted only under a trustworthy platform software state reflected by PCR.

Chapter 3: Policy Models For Group-Centric Secure Information Sharing

In this chapter, we develop the foundations for g-SIS by specifying policy models based on the PEI framwork [28]. We discuss and formally specify the core g-SIS properties, group operation semantics and g-SIS specifications.

3.1 Formal Specification of g-SIS

In this section, we present a collection of core properties that must be satisfied by all g-SIS specifications. After that, we discuss several candidate g-SIS operation semantics and provide specifications for a specific family of these operation semantics. We begin by defining the g-SIS language below.

3.1.1 g-SIS Language

We use Linear Temporal Logic (LTL) to characterize g-SIS properties and specifications. A brief overview of temporal operators used in this dissertation is given in table 2.1. To formalize the LTL language we use and its semantics, suppose U is a finite set of users, O is a finite set of objects, G is a finite set of groups and R is a finite set of actions, such as read and write. We do not introduce subjects in the model which will be needed in more elaborate models presented in this dissertation[1]. Let P be a set of predicates over sorts U, O, G and/or R, and let $\{A, B\}$ be a partition of P. Predicates in A are called actions and intuitively encode actions or events that occurred in the transition to the current state.

[1]Note that a user is a representation of a human in the system and the user may create various subjects (programs/processes) that execute on his/her behalf. For specification of the core properties, the authorization of subjects need not be distinguished from that of the users who control them.

Predicates in B are used to encode aspects of a given state, such as operations that are authorized or not authorized. We let $\mathcal{F}$ be the set of atomic formulas obtained by applying a predicate $p \in P$ to a list of arguments of the appropriate number and sorts. LTL formulas are constructed from $\mathcal{F}$ by applying logical connectives and temporal operators in the usual way.

For the purpose of this dissertation, a *g-SIS language* is required to satisfy the following (the language represented here should be a sub-language of any elaborate g-SIS language designed in the future). It must include a collection of join-group events, leave-group events, add-object events, and remove-object events: $A = \{\text{join}_i | 1 \leq i \leq m\} \cup \{\text{leave}_i | 1 \leq i \leq n\} \cup \{\text{add}_i | 1 \leq i \leq p\} \cup \{\text{remove}_i | 1 \leq i \leq q\}$, $B = \{\text{Authz}\}$, and $R = \{r, w\}$, where r and w refer to the right to exercise "read" and "write" operations respectively. Also, an atomic formula in a g-SIS language should be formed in the natural way: for all $u \in U$, $o \in O$, $g \in G$, $op \in R$, $\text{join}_i(u, g), \text{add}_i(o, g), \ldots, \text{Authz}(u, o, op) \in \mathcal{F}$.

Formally, a state is a function from variable-free atomic formulas $\mathcal{F}$ into the set $\{\text{True}, \text{False}\}$. We use Σ to denote the set of all states. A *trace* σ is an infinite sequence of states, that is, it is an ω-sequence in Σ^ω. In the following, we often wish to write sub-formulas that state, for example, some type of join event occurs. It is therefore convenient to introduce the following notation:

$$\text{Join}(u, g) = (\text{join}_1(u, g) \vee \text{join}_2(u, g) \vee ... \vee \text{join}_m(u, g))$$

$$\text{Leave}(u, g) = (\text{leave}_1(u, g) \vee \text{leave}_2(u, g) \vee ... \vee \text{leave}_n(u, g))$$

$$\text{Add}(o, g) = (\text{add}_1(o, g) \vee \text{add}_2(o, g) \vee ... \vee \text{add}_p(o, g))$$

$$\text{Remove}(o, g) = (\text{remove}_1(o, g) \vee ... \vee \text{remove}_q(o, g))$$

Note that Join (u, g) holds in a state just in case the transition to the state is a Join event. The properties we consider treat the authorization a user has to access an object independent of actions involving other users and objects. Thus, it is often convenient to

omit the parameters in all of the predicates. For instance, when we write Authz $\rightarrow$ (Join $\wedge$ ($\neg$(Leave $\vee$ Remove) $\mathcal{S}$ Add)) we mean $\forall u \in U.\forall o \in O.\forall op \in R.\forall g \in G.\text{Authz}(u, o, op) \rightarrow$ (Join$(u, g) \wedge$ ($\neg$(Leave$(u, g) \vee$ Remove$(o, g))$ $\mathcal{S}$ Add(o, g))). Note that Join, Leave, Add, Remove and Authz, all refer to the same set of u, o and g. In addition to using this notation in formulas, we continue to use these words to informally refer to intuitive notions of corresponding operations.

Well-Formed Traces We now introduce a few formulas that define *well-formed* g-SIS traces. A trace in g-SIS is a sequence of group events such as Join, Leave, Add and Remove. The well-formedness constraints specify how and in what order such events may occur.

A. An object cannot be Added and Removed and a user cannot Join and Leave in the same state. Note that here and below we introduce names of the form τ_j for each of the formulas for later reference. The equality introduces shorthands for the respective formulas.

$$\tau_0 = \Box(\neg(\text{Add} \wedge \text{Remove}) \wedge \neg(\text{Join} \wedge \text{Leave}))$$

B. For a given user/object, two types of the same operation cannot occur in the same state.

$$\tau_1 = \forall i, j.\ \Box((i \neq j) \rightarrow \neg(\text{join}_i \wedge \text{join}_j)) \wedge \forall k, l.\ \Box((k \neq l) \rightarrow \neg(\text{leave}_k \wedge \text{leave}_l)) \wedge$$

$$\forall m, n.\ \Box((m \neq n) \rightarrow \neg(\text{add}_m \wedge \text{add}_n)) \wedge \forall p, q.\ \Box((p \neq q) \rightarrow \neg(\text{remove}_p \wedge \text{remove}_q))$$

C. If a user u joins a group, u cannot join again unless u first leaves the group. A similar rule applies for Add and Remove.

$$\tau_2 = \Box(\text{Join} \rightarrow \bigcirc\ (\neg\text{Join}\ \mathcal{W}\ \text{Leave})) \wedge \Box(\text{Leave} \rightarrow \bigcirc\ (\neg\text{Leave}\ \mathcal{W}\ \text{Join})) \wedge$$

$$\Box(\text{Add} \rightarrow \bigcirc\ (\neg\text{Add}\ \mathcal{W}\ \text{Remove})) \wedge \Box(\text{Remove} \rightarrow \bigcirc\ (\neg\text{Remove}\ \mathcal{W}\ \text{Add}))$$

D. A Leave event cannot occur before Join. Similarly an object cannot be removed from the group before adding.

$$\tau_3 = \Box(\text{Leave} \rightarrow \blacklozenge\text{Join}) \wedge \Box(\text{Remove} \rightarrow \blacklozenge\text{Add})$$

The language we defined above allows us to develop g-SIS specifications which formally define the precise conditions under which authorization can hold. A g-SIS specification is syntactically correct if it is stated in terms of joins, leaves, adds and removes and satisfies the well-formedness constraints. We formally define the requirements for the syntactic correctness of a g-SIS specification.

Definition 3.1 (Syntactic Correctness)**.** A g-SIS specification is *syntactically correct* if it is of the form:

$$\gamma = \forall u \in U.\forall o \in O.\forall op \in R.\forall g \in G.\Box(\text{Authz}(u, o, op) \leftrightarrow \psi(u, o, g)) \wedge \bigwedge_{0 \leq i \leq 3} \tau_i$$

in which ψ is an LTL formula constructed by using temporal operators and predicates in A, and the conjunction τ_i specifies the well-formedness requirements of a g-SIS trace.

A g-SIS specification is *semantically correct* if it satisfies the core properties specified below.

3.1.2 Core Properties

We specify the Core properties, all of which must be satisfied by any g-SIS specification.[2] Subsequently, we specify a few useful additional optional properties. We specify the core properties with the assumption that Join, Leave, Add and Remove are the only events that influence authorization in g-SIS. In the future, these properties can be extended to models involving additional aspects (e.g. attributes of users and objects).

[2]Note that each of these formulas defines a *safety* property [37] in the sense that any trace that does not satisfy the property can be recognized as such by examining a finite prefix of the trace.

In the following, we specify the properties for a read-only g-SIS model. In the subsequent sections, we extend these properties to accommodate write operations in g-SIS. Thus Authz in the following formulas refers to a user's authorization to perform a read operation on the object in question.

1. **Persistence Properties:** These properties consider the conditions under which authorization may change.

   Authorization Persistence: When a user u is authorized to access an object o, it remains so at least until a group event involving u or o occurs.[3]

   $$\varphi_0 = \Box(\text{Authz} \rightarrow (\text{Authz } \mathcal{W} \ (\text{Join} \vee \text{Leave} \vee \text{Add} \vee \text{Remove})))$$

   Revocation Persistence: When a user u is not authorized to access an object o, it remains so at least until a group event involving u or o occurs.

   $$\varphi_1 = \Box(\neg\text{Authz} \rightarrow (\neg\text{Authz } \mathcal{W} \ (\text{Join} \vee \text{Leave} \vee \text{Add} \vee \text{Remove})))$$

   A generalized statement of these properties may be "Authorization does not change unless an authorization changing event occurs." With this generalization, we believe persistence property is required of all access control systems. The following properties are more specifically targeted at g-SIS. They seek to recognize the additional authorizations enabled and disabled by group membership and non-membership respectively.

2. **Authorization Provenance:** Intuitively, a user will not be authorized to read an object until a point at which both the user and object are simultaneously group members.

   $$\varphi_2 = (\neg\text{Authz } \mathcal{W} \ (\text{Authz} \wedge (\neg\text{Leave } \mathcal{S} \ \text{Join}) \wedge (\neg\text{Remove } \mathcal{S} \ \text{Add})))$$

[3] As we will see later, authorization may no longer hold even when certain variations of authorization enabling events such as Join and Add occur.

Two important observations can be made from formula φ_2. First, if Authz holds in a given state then there was an overlapping period of membership between the user and object at least once in the past. Next, authorization to read an object cannot begin for the *first time* during a user's non-membership period (that is, only joining a group can enable authorization).

3. **Bounded Authorization:** These properties require that authorizations not increase during non-membership periods of users and objects (note that authorizations may decrease). Authorizations that hold during the non-membership period of users and object should have held at the time of Leave and Remove respectively.

   Bounded User Authorization: The set of all objects that a user can access during non-membership period is bounded at Leave time. This set cannot grow until the user re-joins.

$$\varphi_3 = \Box((\text{Leave} \wedge \neg\text{Authz}) \rightarrow (\neg\text{Authz}\ \mathcal{W}\ \text{Join}))$$

   The above property states that additional authorizations cannot be granted to a user during non-membership period. Any object that is accessible after Leave should have been authorized at the time of Leave.

   Bounded Object Authorization: The set of all users who can access a removed object is bounded at Remove time, which cannot grow until re-Add.

$$\varphi_4 = \Box((\text{Remove} \wedge \neg\text{Authz}) \rightarrow (\neg\text{Authz}\ \mathcal{W}\ \text{Add}))$$

4. **Availability:** Availability specifies the conditions under which authorization *must* succeed.

$$\varphi_5 = \Box(\text{Join} \rightarrow ((\text{Add} \rightarrow (\text{Authz}\ \mathcal{W}\ (\text{Leave} \vee \text{Remove})))\ \mathcal{W}\ \text{Leave}))$$

This property states that after a user joins a group, any object that is added subsequently should be authorized. Obviously, the user should be a current member when the object in question is added.

We believe that properties φ_0 to φ_4 are truly core and foundational and will apply to sophisticated g-SIS models beyond those that will be studied and formalized in this dissertation. We expect that property φ_5 may need to be relaxed in situations where selective access within a group is required. For instance, a user may be required to belong to a particular role in addition to being a group member in order to access the object[4]. Hence adding an object may not always guarantee immediate access to existing group users. Finally, whether or not additional core properties are desirable remains an open question. Since these core properties are required of any g-SIS model, a g-SIS specification is semantically correct only if it satisfies all of these core properties. We define this below:

Definition 3.2 (Semantic Correctness)**.** A g-SIS specification γ is *semantically correct* if:

$$\gamma \vDash \bigwedge_{0 \leq i \leq 5} \varphi_i$$

Thus, in summary, a g-SIS specification must obey the well-formedness requirements (syntactic correctness, definition 3.1) and the core properties (semantic correctness, definition 3.2).

3.1.3 Consistency and Independence of Core Properties

In this section, we answer two important questions about the g-SIS core properties. First, are the core properties consistent and thus are satisfiable by some g-SIS specification? In other words, is there at least one well-formed trace in which all the core properties hold? Next, are the core properties independent? That is, is it possible to prove or refute one of the properties from the rest? If not, the core properties are independent. In other words,

[4]Such a property may be characterized by specifying that two users with the same attributes joining a group at the same time will have the same authorizations both before and after joining the group.

the core properties specify the requirements of orthogonal aspects of g-SIS. We state these two theorems and formally prove that the core properties are both consistent and mutually independent. We use model checking to prove these theorems.

Theorem 3.1 (Consistency)**.** The g-SIS core properties are satisfiable. More precisely, there is a well formed trace that satisfies them all. Formally stated,

$$\text{there exists a } \sigma \in \Sigma^{\omega} \text{ such that } \sigma \models \bigwedge_{i \in [0..3]} \tau_i \wedge \bigwedge_{j \in [0..5]} \varphi_j$$

Recall from section 3.1.1 that in the g-SIS language, Σ refers to the set of all possible traces of group events such as Join, Leave, Add and Remove. The Consistency theorem states that there exists such a trace that is well-formed in which the all the core properties hold.

We utilize the model checker NuSMV [34] to prove this theorem. Model checking is an automated formal analysis approach that takes a finite model of a system and its properties written in temporal logic formulas as input and determines whether the properties hold in the system. In the case that a property fails to hold, a model checker produces a counterexample consisting of a trace that shows how the failure can arise and can be used to correct the model or the property specification.

A NuSMV model describes how variables can be modified in each step of a system execution. The model is specified using NuSMV. We discuss the code for the model and results in appendix A.1. Here, the NuSMV model is expressed in terms of events Join, Leave, Add and Remove (declared as boolean variables) that are allowed to occur concurrently in a non-deterministic manner. The theorem is expressed as an implication having the well-formed traces (the τ's) in the antecedent and the conjunction of core properties as the consequent. NuSMV takes the model and the LTL formula and determines whether the formula holds. The output from NuSMV in appendix A.1 shows that the LTL formula holds against the model, thus proving the Consistency Theorem.

Theorem 3.2 (Independence)**.** The g-SIS core properties are mutually independent. That is the following two formulas are satisfiable:

$$\forall i \in [0..5].\ \neg((\bigwedge_{k\in[0..3]} \tau_k \wedge \bigwedge_{i\neq j, j\in[0..5]} \varphi_j) \rightarrow \varphi_i) \tag{3.1}$$

$$\forall i \in [0..5].\ \neg((\bigwedge_{k\in[0..3]} \tau_k \wedge \bigwedge_{i\neq j, j\in[0..5]} \varphi_j) \rightarrow \neg\varphi_i) \tag{3.2}$$

The above theorem states that it is neither possible to prove nor refute any one of the core properties if the remaining core properties are known to be true. That is, each core property states an orthogonal aspect of g-SIS. Again, the proof is discussed in appendix A.2. Note that equation (3.2) follows immediately from Theorem 3.1.

3.1.4 Group Operation Semantics

In this section and the following, we discuss a few additional properties that are based on specific variations of group operations. Unlike the core properties, all g-SIS specifications are not required to satisfy these properties. Instead, these properties define certain group operation semantics that are useful for many applications. So only some g-SIS specifications at the designers discretion will be required to conform to these properties.

Membership Semantics

Membership Properties characterize the semantics of authorizations enabled when a user joins or an object is added and those which are disabled when a user leaves or an object is removed from the group. Later, we consider properties when a user or an object is re-admitted to the group, characterized as Membership Renewal Properties.

Strict Join (SJ) Vs Liberal Join (LJ): In SJ, the joining user may only access some or all of the objects added after Join time. LJ additionally allows the user to access some or all of the objects that were added prior to join time. Suppose that in figure 3.1 the second Join $(u1, g)$ is an SJ. Then $u1$ can access $o4$ and $o5$ but cannot access $o2$ and $o3$. If the Join

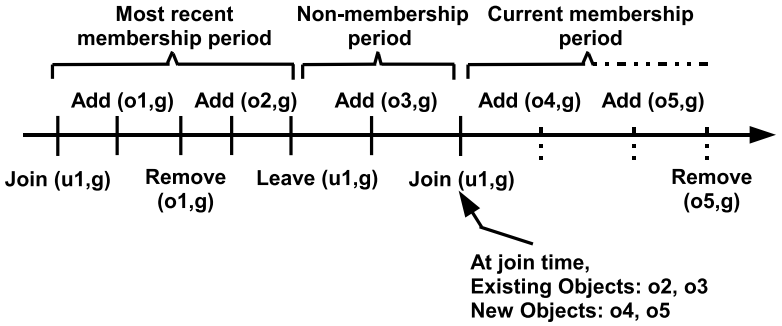

Figure 3.1: User Operations Illustration.

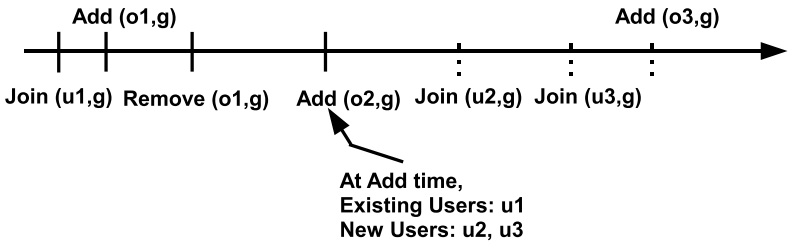

Figure 3.2: Object Operations Illustration.

was an LJ instead of SJ, $u1$ can also access $o2$ and $o3$. This can be formalized by requiring that join_i, a type of Join, would be admitted as SJ only if it satisfies α_0 stated in table 3.1. In a g-SIS specification with LJ, there exists at least one well-formed trace for which Authz does not satisfy α_0.

Strict Leave (SL) Vs Liberal Leave (LL): In SL, the leaving user loses access to all objects. In LL, the leaving user may retain access to some or all of the objects authorized prior to Leave time. In figure 3.1, on SL, $u1$ loses access to all group objects ($o1$ and $o2$) authorized during the membership period. An LL will allow $u1$ to retain access to $o2$ (and possibly $o1$, depending on the type of Remove of $o1$). A type of Leave, leave_i, would be admitted as SL only if it satisfies α_1 stated in table 3.1. In a g-SIS specification with LL, there exists at least one well-formed trace that does not satisfy α_1.

Strict Add (SA) Vs Liberal Add (LA): In SA, the added object may only be accessed by only some or all of the users who joined before Add time. In LA, the added object may also

Table 3.1: Summary of group membership semantics

Operation	Explanation	Property
Strict Join (SJ)	Only objects added after join time can be accessed	$\boldsymbol{\alpha_0} = \Box(\text{Authz} \rightarrow \blacklozenge(\text{Add} \wedge (\neg\text{Leave } \mathcal{S} \text{ join}_i)))$
Liberal Join (LJ)	Can access objects added before and after join time	There exists a well-formed trace that does not satisfy α_0
Strict Leave (SL)	Lose access to all objects on leave	$\boldsymbol{\alpha_1} = \Box(\text{Authz} \rightarrow (\neg\text{leave}_i \ \mathcal{S} \ \text{Join}))$
Liberal Leave (LL)	Retain access to objects authorized before leave time	There exists a well-formed trace that does not satisfy α_1
Strict Add (SA)	Only users who joined prior to add time can access	$\boldsymbol{\alpha_2} = \Box(\text{add}_i \rightarrow (\neg\blacklozenge\text{Join} \rightarrow (\neg\text{Authz } \mathcal{W} \text{ Add})))$
Liberal Add (LA)	Users who joined before or after add time may access	There exists a well-formed trace that does not satisfy α_2
Strict Remove (SR)	All users lose access on remove	$\boldsymbol{\alpha_3} = \Box(\text{remove}_i \rightarrow (\neg\text{Authz } \mathcal{W} \text{ Add}))$
Liberal Remove (LR)	Users who had access at remove time retain access	There exists a well-formed trace that does not satisfy α_3

be accessed by some or all of the users that join (e.g., LJ) later. If Add $(o2, g)$ in figure 3.2 is an SA, only $u1$ can access the object. Users $u2$ and $u3$, joining later, cannot access this object. But on LA current user $u1$ and future users $u2$ and $u3$ may access $o2$. A type of Add, add_i, would be admitted as SA only if it satisfies α_2 stated in table 3.1. In a g-SIS specification with LA, there exists at least one well-formed trace that does not satisfy α_2.

Strict Remove (SR) Vs Liberal Remove (LR): In SR, the removed object cannot be accessed by any user. In LR, some or all of the users who had access at Remove time may retain access (of course, users joining later are not allowed to access the removed object—this respects the Authorization Provenance core property). In figure 3.2, if Remove $(o1, g)$ is an SR, every group user (including $u1$) loses access to $o1$. If Remove $(o1, g)$ is an LR, $u1$ can continue to access $o1$. However $u2$ and $u3$ will not have access to $o1$. A type of Remove, remove_i, would be admitted as SR only if it satisfies α_3 stated in table 3.1. In a g-SIS specification with LR, there exists a well-formed trace that does not satisfy α_3.

Membership Renewal Semantics

Membership Renewal Properties characterize what, if any, authorizations from previous membership period(s) are enabled or disabled when users re-join and subsequently leave the group. In the meeting room metaphor, Alice may leave the room and re-enter later. These properties are concerned with her authorizations from her previous sessions in the room and its continuity when she leaves the room again. As the name implies, these properties are applicable only to returning users and are discussed below.

Unlike Membership Properties, these properties apply only to users and not to objects since a user may have authorizations from past membership periods at re-Join time. Since the Renewal Properties are concerned about the status of such authorizations, a decision needs to be made for the user at re-Join time. Objects, being a passive entity, do not have past authorizations and hence renewal decisions need not be made at re-Add time[5].

Lossless Vs Lossy Join: In Lossless Join, a re-joining user does not lose authorization(s) held immediately prior to re-joining. A Join operation that causes a user to lose some or all prior authorizations is called Lossy. Suppose in figure 3.1 $u1$ retains access to $o2$ at the time of Leave (due to LL for example). When $u1$ re-joins subsequently, in a Lossless Join (regardless of whether it is a SJ or LJ), access to $o2$ will not be revoked. If access to $o2$ is revoked by re-joining the group, the Join is Lossy. Formula β_0 (table 3.2) characterizes Lossless Join. In a g-SIS specification with Lossy Join, there exists at least one well-formed trace that does not satisfy the above property.

A Lossy Join is useful in scenarios when authorizations from past membership and those from the new membership are in conflict of interest. For example, if a student registers for a course, drops after the mid-term and re-registers the following semester, he/she may be required to relinquish access to exercise solutions and other materials from past enrollment. The student would be given a Lossy Join in this scenario.

Non-Restorative Vs Restorative Join: In a Non-Restorative Join, authorizations from past membership periods may not be explicitly restored at the time of re-join. On the

[5]Note that for certain applications, it may be useful to consider equivalent semantics of object operations.

other hand, a Restorative Join explicitly restores authorizations from past membership periods. Suppose in figure 3.1 when $u1$ leaves, SL is applied and SJ is applied on re-join. A Restorative SJ in this scenario will allow $u1$ to re-gain access to $o2$ (and possibly $o1$) from past membership period. Note that the notion of Restorative LJ is subtle but important. Suppose $o1$ was removed with LR and an SL is applied at the time of Leave. In this case, $u1$ will continue to access $o1$ until the time of Leave. If LJ is applied on re-join, a Restorative LJ will allow $u1$ to re-gain access to $o1$, but a Non-Restorative LJ will not.

Formalizing Non-Restorative Join is complicated because we want our characterization to be independent of the exact semantics of the Join operation in question. Intuitively, we want to require that the Non-restorative Join does not add any authorizations that it would not have added if the user had a different history. However LTL does not enable one to compare different traces. The solution we take is to consider two different users within a single trace. Because the two users can have different histories with the same trace, this strategy enables us to formalize the property. We first state two formulas ρ_1 and ρ_2 which will then be used to specify the property β_1 as shown in table 3.2.

In formula ρ_1, users $u1$ and $u2$ both join the group at the same time by means of the same type of Join. Further, it looks at a case where $u1$ is authorized to access an object in the current state and $u2$ is not. ρ_2 says that this should also be the case in the previous state (and vice-versa). The Non-Restorative Join property is characterized by formula β_1. It states that if two users Join the group at the same time with the same type of Join, then any difference in access at Join time is the result of some operation prior to the current Join operation. Let us use formula ρ_2 to understand the intuition. Because both $u1$ and $u2$ Join at the same time with same type, any access that is necessarily enabled by this Join for $u1$, would also be enabled for $u2$. Any additional access that $u1$ may have that $u2$ does not have could arise only because $u1$ had access to that object before joining the group. This captures the fact that access is not restored from past but is a consequence of the type of Leave operation applied to the user when he/she left the group.

Table 3.2: Summary of group membership renewal semantics

Operation	Explanation	Property
Lossless Join	Authorizations prior to join time is not lost	$\boldsymbol{\beta_0} = \Box((\text{Join} \wedge \neg\text{Remove} \wedge \ominus \text{ Authz}) \rightarrow \text{Authz})$
Lossy Join	Authorizations from prior to join may be lost	There exists a well-formed trace that does not satisfy β_0
Non-Restorative Join	Authorizations from past membership periods not explicitly restored	$\rho_1 = (\text{join}_i(u1, g) \wedge \text{join}_i(u2, g) \wedge \text{Authz}(u1, o, g, r) \wedge \neg\text{Authz}(u2, o, g, r))$ $\rho_2 = \ominus\ (\text{Authz}(u1, o, g, r) \wedge \neg\text{Authz}(u2, o, g, r))$ $\boldsymbol{\beta_1} = \forall i \Box(\rho_1 \rightarrow \rho_2)$
Restorative Join	Authorizations from past membership may be restored	There exists a well-formed trace that does not satisfy β_1
Gainless Leave	Authorizations that never held during most recent membership period cannot be obtained	$\boldsymbol{\beta_2} = \Box((\text{Leave} \wedge (\neg\text{Join}\ \mathcal{U}\ (\text{Authz} \wedge \neg\text{Join}))) \rightarrow \ominus\ ((\neg\text{Authz} \wedge \neg\text{Join})\ \mathcal{S}\ (\text{Authz} \wedge (\neg\text{Join}\ \mathcal{S}\ \text{Join}))))$
Gainful Leave	New authorizations may be granted at Leave time	There exists a well-formed trace that does not satisfy β_2
Non-Restorative Leave	Authorizations that the user had prior to joining the group are not explicitly restored	$\boldsymbol{\beta_3} = \Box(\text{Leave} \wedge \text{Authz} \rightarrow \ominus\ \text{Authz})$
Restorative Leave	Authorizations from prior to join time may be restored	There exists a well-formed trace that does not satisfy β_3

In Restorative Join, there exists at least one well-formed trace that does not satisfy the Non-Restorative Join property. If a user joins a group using Restorative Join, some or all of the accesses to objects authorized during past membership period may be restored. Note that this is in addition to the authorizations that current Join may enable. The formula $\Box(\text{Join} \wedge ((\neg\text{Leave} \wedge \neg\text{Remove})\ \mathcal{S}\ (\text{Leave} \wedge \ominus\ \text{Authz})) \rightarrow \text{Authz})$, for example, characterizes a type of Restorative Join where *all* past authorizations are restored.

A Restorative Join is applicable in scenarios where an incentive is provided for a user to re-join the group. On the other hand, in subscription service scenarios, a Restorative and Non-Restorative Join may be priced differently, which may decide what prior authorizations to their past subscription materials will be restored.

Gainless Vs Gainful Leave: After re-joining the group, a subsequent Leave operation could either be Gainless or Gainful. In Gainless Leave, authorizations that never held during

current membership period cannot be obtained by leaving the group. On the other hand, a Gainful Leave allows new authorizations to be granted at the time of Leave. Suppose that in figure 3.1 a Lossless SJ is applied when $u1$ re-joins the group. Because the re-join is of type Lossless SJ, only $o4$ and possibly $o5$ can be accessed. If $u1$ leaves the group in the future with LL, a Gainless LL will not grant any new authorizations other than that to $o4$ and $o5$. A Gainful LL, for example, may additionally grant access to $o2$. Note that this is still consistent with the Authorization Provenance property because $o2$ and $u1$ had an overlapping period of membership. A Gainful Leave is useful in scenarios where an incentive is provided for a user to leave the group. This is commonplace in voluntary retirement and severance packages.

β_2 in table 3.2 characterizes Gainless Leave. This formula states that if the user is authorized to access an object during non-membership period then it should have been authorized during the most recent membership period. In a g-SIS specification with Gainful Leave, there exists at least one well-formed trace that does not satisfy the Gainless Leave property.

Non-Restorative Vs Restorative Leave: In Non-Restorative Leave, authorizations that the user had prior to joining the group are not explicitly restored at Leave time. In Restorative Leave, some or all of such authorizations are restored at Leave time. Suppose in figure 3.1 $u1$ left the group with LL and re-joins with Lossy SJ. In this case, $u1$ possibly loses access to both $o1$ and $o2$ at re-join time. Later on, if $u1$ leaves with Gainful LL, a Restorative Leave will allow $u1$ to re-gain access to $o1$ and $o2$ at the time of leave, but a Non-Restorative leave will not.

In the meeting room metaphor, suppose Alice is re-invited as a consultant on demand and is required to relinquish her past authorizations due to a conflict of interest with new authorizations that will be enabled by current membership. After Alice performs her duties and leaves the group, it is natural that she will regain access to objects for which she lost authorization when joining the group. Formula β_3 in table 3.2 characterizes Non-Restorative Leave.

In Restorative Leave, there exists at least one well-formed trace that does not satisfy the Non-Restorative Leave Property. For example, the formula $\Box((\text{Leave} \wedge \neg\text{Remove} \wedge \ominus((\neg\text{Leave} \wedge \neg\text{Remove})\ \mathcal{S}\ (\text{Join} \wedge \ominus\ \text{Authz}))) \rightarrow \text{Authz})$ characterizes a specific type of Restorative Leave where access to *all* objects authorized prior to Join is restored.

3.1.5 Formal Analysis

In this section, we discuss the construction of the specification of a family of g-SIS models, called the π-system. The π-system is a g-SIS specification that formally defines the conditions under which authorization can hold. We successfully verify using a model checker called NuSMV [34] that the π-system we develop is semantically correct—that is, it satisfies the core g-SIS properties.

We also show that the π-system satisfies a subset of Membership and Membership Renewal properties. To this end, the π-system allows any variation of Membership operations (Strict/Liberal). Thus, different users and objects may be given different types of respective operations in the π-system. For example, SJ for $u1$, LJ for $u2$, SL for $u1$ and subsequently LJ for $u1$ and similarly for objects. However, for Membership Renewal operations, we confine our scope to Join operations that are of type Lossless and Non-Restorative and Leave operations of type Gainless and Non-Restorative. These specific types of renewal operations are the most basic since they do not require us to treat past membership authorizations explicitly. Further, other renewal semantics are likely application dependant. For example, what exact authorizations will be disabled at Join time in Lossy Join depends on the application in question (similarly for Restorative Join, Gainful Leave and Restorative Leave).

Remark 3.1. The π-system only allows group operations of type indicated below:

$$\forall i.\text{Type}(\text{join}_i) \in \{\text{SJ}, \text{LJ}\} \times \{\text{Lossless}\} \times \{\text{Non-Restorative}\}$$

$$\forall i.\text{Type}(\text{leave}_i) \in \{\text{SL}, \text{LL}\} \times \{\text{Gainless}\} \times \{\text{Non-Restorative}\}$$

$$\forall i.\text{Type}(\text{add}_i) \in \{\text{SA}, \text{LA}\}$$

$$\forall i.\text{Type}(\text{remove}_i) \in \{\text{SR}, \text{LR}\}$$

Henceforth, for convenience, we assume that SJ and LJ refer to operations of type Lossless and Non-Restorative join. Similarly, SL and LL refer to Gainless and Non-Restorative leave operations.

Furthermore, note that as per the definition of Strict Join (table 3.1), the user may access *some or all* objects that are added after Join time. However, in the π-system, we will use the following specific interpretation of Strict and Liberal operations which effectively replaces "some or all" with "all":

- On SJ, the joining user may access *all* the objects added after Join time. Objects added before Join time may not be accessed. On LJ, the joining user may access *all* the current objects added before and after Join time.

- On SL, the users lose access to all objects. On LL, access to *all* the group objects authorized at the time of Leave may be retained.

- On SA, *all* the users who joined prior to Add time may access. Users joining later may not access. On LA, *all* the users who joined prior to and after Add time may access.

- On SR, all the users lose access to the object. On LR, *all* the users who had access at Remove time may retain access.

π-system Specification

There are two scenarios to consider when a user requests access to an object: (i) the user Join event occurred prior to object Add event or (ii) the object Add event occurred prior to user Join event. Intuitively, if the specification correctly addresses these two scenarios, it would be complete. We now consider these two scenarios separately. Formula λ_1 addresses the scenario where the object is added after the user joined the group (figure 3.3).

$$\lambda_1 = ((\neg \text{SL} \wedge \neg \text{SR}) \; \mathcal{S} \; ((\text{SA} \vee \text{LA}) \wedge ((\neg \text{LL} \wedge \neg \text{SL}) \; \mathcal{S} \; (\text{SJ} \vee \text{LJ}))))$$

Since Join occurred prior to Add, regardless of whether the object was LA'ed[6] or SA'ed or whether the user was SJ'ed or LJ'ed, the user should be authorized to access the object in both cases as per the core Availability property. Formula λ_1 says that the user has not been SL'ed and the object has not been SR'ed since it was added with SA or LA. Further, when the Add occurred, the user was a current member (that is, no SL or LL since SJ or LJ).

In figure 3.3, an SL or SR since object add time denies access to the requested object. However, it is alright for an LL or LR to occur during that period. Recall that an LR authorizes current users at remove time to retain access and LL authorizes a leaving user to retain access to objects authorized during the membership period. Similarly, if the user was not a current member when the object was added (for e.g., joined and left the group before the object was added), authorization cannot hold as per our core Provenance property.

Scenario (ii) where an Add occurs prior to Join is more interesting. As shown in figure 3.4, there are four possible cases. Let us first consider cases (a) and (b) where the object is SA'ed to the group. Recall that an SA'ed object can be accessed only by *existing* users (that is, the users who joined the group prior to object Add). Clearly, regardless of the type of Join, the user is not authorized to access objects that were SA'ed prior to the user Join time. Thus Authz cannot hold in cases (a) and (b).

[6]We use the abbreviation of the form "LA'ed" to refer to the fact that the object in question is Liberally Added.

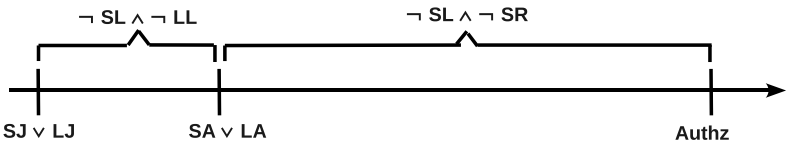

Figure 3.3: Formula λ_1.

Consider cases (c) and (d) where the object is LA'ed to the group. In case (c), the object is LA'ed and the user is SJ'ed. An SJ'ed user is authorized to access only those objects that were added after join time. Thus (c) is also a failed case. Authorization is successful in case (d) where both Add and Join are Liberal operations. An LJ'ed user can access all current LA'ed objects and any newly added object in the future (LA or SA). We can now formulate λ_2 as shown below.

$$\lambda_2 = ((\neg \text{SL} \wedge \neg \text{SR})\ \mathcal{S}\ (\text{LJ} \wedge ((\neg \text{SR} \wedge \neg \text{LR})\ \mathcal{S}\ \text{LA})))$$

Figure 3.5 illustrates λ_2. It says that the user has not been SL'ed and the object has not been SR'ed since the user LJ'ed the group. Further, at Join time, the object in question was still part of the group (that is, it has not been LR'ed or SR'ed since it was added). We can now formally specify the π-system.

Definition 3.3 (π-system)**.** The π-system is given by:

$$\pi = \Box(\text{Authz} \leftrightarrow \lambda_1 \vee \lambda_2) \wedge \bigwedge_{0 \leq j \leq 3} \tau_j$$

Note that π is a syntactically correct g-SIS specification by definition. π says that a user is authorized to access an object if and only if λ_1 or λ_2 holds and the trace is well-formed.

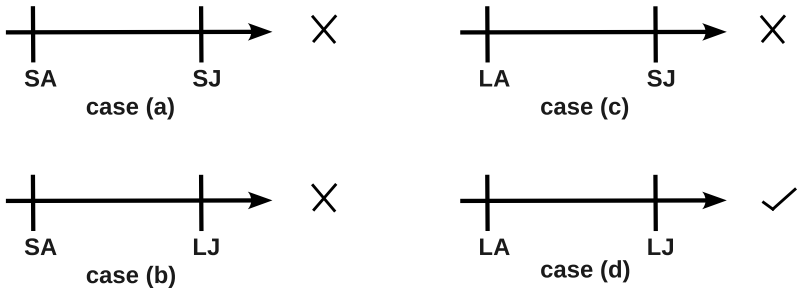

Figure 3.4: Cases when Add occurs prior to Join.

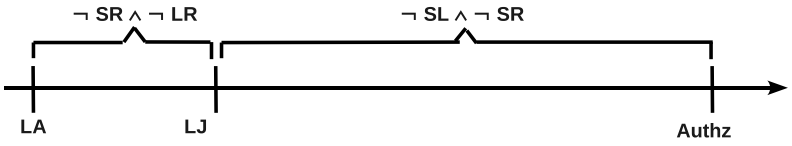

Figure 3.5: Formula λ_2.

Entailment Theorems

In this section, we show that the π-system entails the Core and Membership Renewal properties. Since Membership operations are mixed (i.e., both Strict and Liberal operations are allowed), a specific Membership property cannot be verified for this specification. Later, we show the verification of Membership Properties in a specification where the operations are fixed for all users and objects in a group. We begin with the Mixed Operations and Membership Renewal Semantics Entailment Theorem.

Theorem 3.3 (Mixed Operations Entailment Theorem)**.** The π-system entails the Core Properties (φ_0 to φ_5):

$$\pi \vDash \bigwedge_{0 \leq q \leq 5} \varphi_q$$

Theorem 3.4 (Membership Renewal Semantics Entailment Theorem)**.** The π-system entails the Membership Renewal Properties (β_0 to β_3):

$$\pi \vDash \bigwedge_{0 \leq r \leq 3} \beta_r$$

Again, we utilize the model checker NuSMV [34] to prove these theorems. The model of the π-system is specified using NuSMV. We discuss the code for the model and results in appendix B.1. Here, the NuSMV model is expressed in terms of events Join, Leave, Add and Remove (declared as boolean variables) that are allowed to occur concurrently in a non-deterministic manner. The theorem is expressed as an implication having the π-system in the antecedent and the Core and Membership Renewal properties in the consequent. NuSMV takes the model and the LTL formula and determines whether the formula holds in all possible traces generated by the model. The output from NuSMV in appendix B.1 shows that the LTL formula holds against the model. Thus we verify the theorems, i.e., the π-system satisfies the Core and Membership Renewal properties.

The significance of this Mixed Operation Entailment Theorem is two-fold. First, it shows that any specification that one derives from the π-system is guaranteed to be a *g-SIS* specification (i.e., it would satisfy the core properties.) For example, one can derive various fixed operation g-SIS specifications from the π-system. By substituting all operations in formula π with either Strict or Liberal versions of membership semantics, we obtain the Most Restrictive and Most Permissive π-system respectively. Note that if the most restrictive model (SJ, SL, SA, SR) permits a user to access an object, the most permissive model (LJ, LL, LA, LR) should also grant access to the same object. Such fixed operation specifications are guaranteed to be admitted as g-SIS. Next, the core g-SIS properties are consistent. That is, the core properties can be satisfied, namely, by the π-system, and do not conflict with each other.

As mentioned earlier, the Most Restrictive π-system specification is one where only Strict operations are allowed. Since Liberal operations are not allowed in such a specification, we

substitute Liberal operations with "False" in formulas λ_1 and λ_2 and thereby obtain the μ-system defined below.

Definition 3.4 (μ-system)**.** The μ-system is given by:

$$\mu = \Box(\text{Authz} \leftrightarrow ((\neg\text{SL} \wedge \neg\text{SR})\ \mathcal{S}\ (\text{SA} \wedge (\neg\text{SL}\ \mathcal{S}\ \text{SJ})))) \wedge \bigwedge_{0 \leq i \leq 3} \tau_i$$

Note that specification μ is a syntactically and semantically correct g-SIS specification since it is a specific form of π. Further, since only Strict operations are allowed, μ must satisfy all the Strict versions of Membership Properties (formulas α_0 to α_3).

Theorem 3.5 (Most Restrictive Entailment Theorem)**.** The μ-system entails the Core Properties (φ_0 to φ_5), Membership (α_0 to α_3) and Membership Renewal Properties (β_0 to β_3).

$$\mu \vDash (\bigwedge_{0 \leq j \leq 5} \varphi_j \wedge \bigwedge_{0 \leq k \leq 3} \alpha_k \wedge \bigwedge_{0 \leq l \leq 3} \beta_l)$$

We take a similar approach to that of Theorem 3.3 to prove this theorem. Again, the proof and the discussion of code and NuSMV results is given in appendix B.2.

3.1.6 A Family of Fixed Operation g-SIS Models

Consider g-SIS models where the group operations are fixed for all users and objects. For example, a g-SIS model may only allow Liberal operations for all users and objects (LJ, LL, LA, LR) in the group. That is, every user admitted to the group will be given LJ. Similarly, objects will be added only with LA and so on. Another example of a fixed operations model is (SJ, SL, SA, SR) where all operations are Strict. Note that the fixed operation (SJ, SL, SA, SR) model is more restrictive in terms of authorization than the (LJ, LL, LA, LR) model. If a more restrictive model (such as (SJ, SL, SA, SR)) permits a user to access an object, a less restrictive model ((LJ, LL, LA, LR)) would also grant access to the same object.

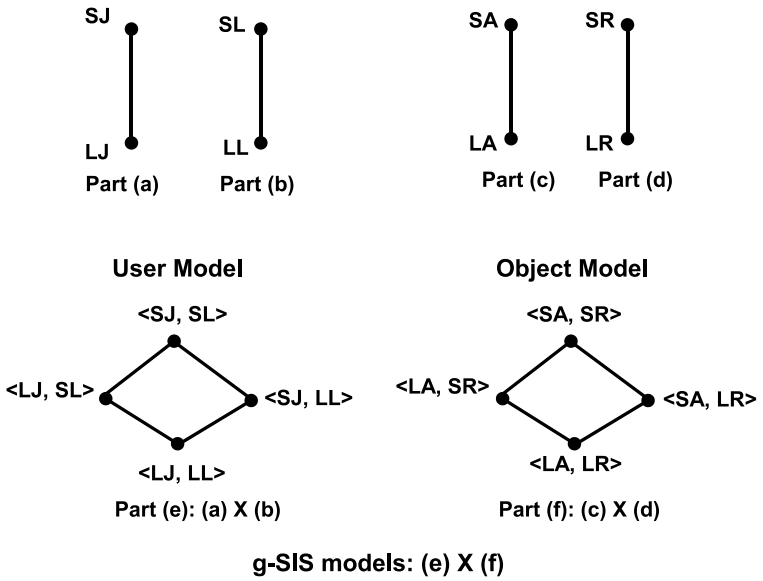

Figure 3.6: A family of g-SIS models: *The cartesian product of User and Object Model results in a lattice of 16 g-SIS models with fixed operation types.*

Thus if the type of operations are fixed for all users and objects, there are 16 possible models ranging from the most restrictive model allowing only Strict operations: (SJ, SL, SA, SR) to the most permissive model allowing only Liberal operations: (LJ, LL, LA, LR). This is illustrated in figure 3.6. Parts (a) through (d) show that the Strict operation is more restrictive than the Liberal operation. Parts (e) and (f) show the user and object model that is obtained by the Cartesian product of user and object operations respectively. Finally, a lattice of 16 g-SIS models is obtained by the Cartesian product (e) $\times$ (f).

Lattice Reduction

The following observations can be made about 16 possible fixed operation g-SIS models in figure 3.6:

1. The type of object add has no significance in a model allowing only SJ. This is because, with SJ, joining users can only access newly added objects. Thus regardless of how the object is added, a joining user cannot access objects added prior to join

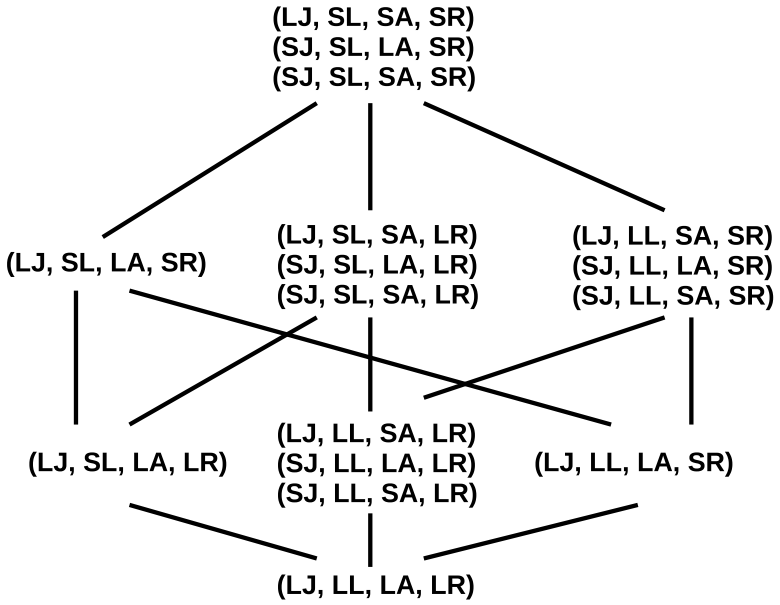

Figure 3.7: Reduced lattice with 8 fixed operation g-SIS models.

time. And if the object is added after join time, the users can access the object regardless of how it is added (as per the core availability property).

2. Similarly, the type of user join has no significance in a model allowing only SA.

3. LR has no significance in a model with SJ. LR allows users who had access at the time of remove to retain access. These are users who joined with SJ prior to the object add time. Users who joined with SJ after the removed object's add time had no access to begin with. Thus an SJ model supporting only LR is as though the model has no support for remove operation for a given user once the user is authorized to access that object.

4. Similarly, LR has no significance in a model with SA.

Thus, based on these observations, we only have 8 unique g-SIS models with fixed operation types (as illustrated in figure 3.7):

1. (SJ, SL, SA, SR)=(SJ, SL, LA, SR)=(LJ, SL, SA, SR)
2. (SJ, SL, SA, LR)=(SJ, SL, LA, LR)=(LJ, SL, SA, LR)
3. (SJ, LL, SA, SR)=(SJ, LL, LA, SR)=(LJ, LL, SA, SR)
4. (SJ, LL, SA, LR)=(SJ, LL, LA, LR)=(LJ, LL, SA, LR)
5. (LJ, SL, LA, SR)
6. (LJ, SL, LA, LR)
7. (LJ, LL, LA, SR)
8. (LJ, LL, LA, LR)

Thus the type of Add has no significance on the authorization in these 8 specifications. In many usage scenarios, object operations do not remain fixed for all group objects. Certain objects may need to be added with SA to restrict access to existing group members while others may be added with LA when such a restriction is not required. Similarly, certain objects may be removed with SR while others with LR. Thus, object operations may differ from one object to another. On the other hand, in many dynamic scenarios such as in emergency response or clinical work flow systems, even the user operations may differ from one user to another. For instance, physicians may be given an LJ so they have access to all the patient records. However, nurses are rotated in shifts and thus may be given SJ and SL. Thus the nurses get to see patient information that is pertinent during their shift period.

One can derive additional specifications from the π-system where some of the membership operations may vary while others remain fixed for various users and objects.

Usage Scenarios

We now discuss two usage scenarios: a large-scale subscriptions scenario where the operations are fixed for all users and a small-scale collaboration scenario where the operations could be mixed.

Subscription Models In general, user operations define the semantics of most subscription models. Thus most subscription models fall into one of the four categories: (SJ, SL), (SJ, LL), (LJ, SL) and (LJ, LL). Consider a premier online news magazine ABS Corp. that offers four levels of membership:

1. Level 1; \$10/year (SJ, SL): These subscribers can only access news articles that are published after they started paying the subscription fee. Level 1 subscribers cannot access ABS's archives (effected by SJ). If they cancel their subscription, they lose access to all news articles.

2. Level 2; \$12/year (SJ, LL): Similar to Level 1 but subscribers can retain access to news articles that they paid for even after canceling their subscription.

3. Level 3; \$15/year (LJ, SL): Level 3 subscribers can access rich archives filled with post-news analysis, predictions, annotations and opinions from experts. But if they cancel their subscription, they lose access to all news articles including archives.

4. Level 4; \$17/year (LJ, LL): Similar to Level 3, but even after canceling membership, subscribers can login and view all articles that they had access before leaving.

Object operations do not fundamentally change the subscription model's semantics. Nevertheless, they model useful scenarios. For example, if an object is added with SA, only existing users in the group may access. Thus SA objects model sales promotion or discounted price available only for current group members.

Mission Oriented Group Consider a g-SIS model with the operation types: (LJ, SL, SA/LA, SR) where all operations are fixed except object Add. Objects can be added to the group by type SA or LA. Let us consider a simple collaboration scenario where the group is mission oriented, so many users may join and leave the group to contribute and receive information over time.

Consider two users Alice and Bob who join the group at the same time. If Bob wants to ensure that any information he shares with Alice is not accessible to future users who

may join the group, he can add objects with SA. SA objects are only accessible to *existing* members at add time. This allows current members of the group to share information privately. On the other hand, to the mission's end, information can be made available to future users by LA'ing objects to the group. Suppose Alice leaves the group and later Cathy joins with LJ. Cathy cannot access SA objects shared between Alice and Bob before her join time. Cathy can only access existing LA objects.

3.1.7 Read-Write g-SIS (single object version)

We specified the core properties for read-only authorizations in g-SIS in section 3.1.2. In this section, we specify the core properties for write authorizations in g-SIS. We assume that objects are not versioned. That is, updating an object always overwrites the object and only one version of the object exists at any time. In the following section, we study a g-SIS model with object versions. We now specify the core properties for write authorization for a Read-Write g-SIS model with no versioning of objects.

1. **Persistence Properties:** These properties consider the conditions under which authorization to write may change.

   Authorization Persistence: When a user u is authorized to write to an object o, it remains so at least until a group event involving u or o occurs.

   $$\vartheta_0 = \Box(\text{Authz}(u, o, w) \rightarrow (\text{Authz}(u, o, w)\ \mathcal{W}\ (\text{Join} \vee \text{Leave} \vee \text{Add} \vee \text{Remove})))$$

   Revocation Persistence: When a user u is not authorized to write to an object o, it remains so at least until a group event involving u or o occurs.

   $$\vartheta_1 = \Box(\neg\text{Authz}(u, o, w) \rightarrow (\neg\text{Authz}(u, o, w)\ \mathcal{W}\ (\text{Join} \vee \text{Leave} \vee \text{Add} \vee \text{Remove})))$$

2. **Authorization Provenance:** Unlike authorization provenance for read, we require that in order to write to an object, both the user and the object are current members in the group. This is formalized in ϑ_2 below. An important observation is that the user needs to be authorized to read an object in order to write.

$$\vartheta_2 = \Box(\text{Authz}(u, o, w) \rightarrow (\text{Authz}(u, o, r) \wedge (\neg\text{Leave}(u, g)\ \mathcal{S}\ \text{Join}(u, g))$$

$$\wedge (\neg\text{Remove}(o, g)\ \mathcal{S}\ \text{Add}(o, g))))$$

3. **Bounded Authorization:** These properties specify the conditions under which write authorizations may change during non-membership period.

   Bounded User Authorization: A past user cannot write to any object unless he/she re-joins the group. This is stated in formula ϑ_3 below.

$$\vartheta_3 = \Box(\text{Leave} \rightarrow (\neg\text{Authz}(u, o, w)\ \mathcal{W}\ \text{Join}))$$

   Bounded Object Authorization: Recall that the bounded user authorization property for read stated that any read authorization that a user has during non-membership period should have held at the time of leave. Consider a leaving user who retains read authorization to a few objects at leave time. With write operation, if such objects are updated by existing group users, the past user can continue to read new information. To prevent such unwanted information flow, we constrain that past users may continue to read objects authorized at leave time as long as the objects are not updated by some existing group user. To specify this property, we introduce a new operation called Update (o,g) which occurs whenever some user writes to an object o in group g. Formula ϑ_4 below states that if an object is updated, any past user cannot read the object unless the user re-joins the group.

$$\vartheta_4 = \Box((\text{Update}(o, g) \wedge (\neg\text{Join}\ \mathcal{S}\ \text{Leave})) \rightarrow (\neg\text{Authz}(u, o, r)\ \mathcal{W}\ \text{Join}))$$

Property ϑ_5 below states that past objects cannot be updated by any user unless it is re-added.

$$\vartheta_5 = \Box(\text{Remove} \rightarrow (\neg\text{Authz}(u, o, w)\ \mathcal{W}\ \text{Add}))$$

4. **Availability:** Availability specifies the conditions under which authorization *must* succeed.

$$\vartheta_6 = \Box(\text{Join} \rightarrow ((\text{Add} \rightarrow (\text{Authz}(u, o, w)\ \mathcal{W}\ (\text{Leave} \vee \text{Remove})))\ \mathcal{W}\ \text{Leave}))$$

This property states that after a user joins a group, any object that is added subsequently should be authorized to write. Obviously, the user should be a current member when the object in question is added.

Like earlier, we require that any read-write single version g-SIS specification satisfy all of the core properties specified above (ϑ_0 to ϑ_6).

3.1.8 Read-Write g-SIS (object versioning)

Earlier, we introduced a family of g-SIS models where read and write operations were performed on the object on the whole. In such a setting, it is possible that if multiple users update the same object at the same time, the latest write will always take precedence and overwrite the object. In a distributed system such as g-SIS, the same object may be updated by multiple parties, potentially offline without the intervention of a central server. As we will see later in the following chapter, we will present an enforcement model based on super-distribution where objects may be obtained from any source and accessed offline by group users if authorized. To this end, in this section, we discuss a versioning model in which each update operation results in a new version of the same object. Note that with the introduction of versioning, access control decision needs to be made on the new versions. It is possible that different versions of the same object may have different authorizations.

Versioning Model

With object versioning, we introduce a number of additional group operations to manage versions. We retain the 4 operations from the earlier single version g-SIS model (Join, Leave, Add and Remove). The Add operation is used to add a version from an external entity to the group and Remove operation removes the added version from the group. The added version could be updated using an Update operation which results in a new version. Such updated versions may be merged back with the source entity that initially shared it with the group using the Merge operation.

Furthermore, new objects may be created internally in the group. We assume that a new object is created with an initial version of $v0$. Such a new version may be updated subsequently using the Update operation. Any version of the internally created object may be exported to an external entity using the Export operation.

Any version in the group may be suspended and subsequently resumed using the Suspend and Resume operations respectively. We assume that Update may be performed only on existing object versions in the group. Thus any version that is removed or suspended can not be updated. Note that a Remove operation can only be performed on the version that was added by an external entity to the group using a corresponding Add operation. New versions created by updating such a version can not be removed. They may only be suspended/resumed or exported in the future.

We now discuss the life-cycle of an object to elucidate our versioning model. There are two cases of an object version: (a) it was externally created in some source entity (e.g. organization) and shared with the group, or (b) it was internally created in the group.

Case (a) Externally Created Object: Figures 3.8 and 3.9 show the life-cycle of an object version that was externally created in some source entity and shared with the group. As shown in figure 3.8, such a version may be shared with the group using the Add operation. Once added, it may be updated using the Update operation resulting in creation of a new version of the same object. The life-cycle of such updated versions is shown in figure 3.9.

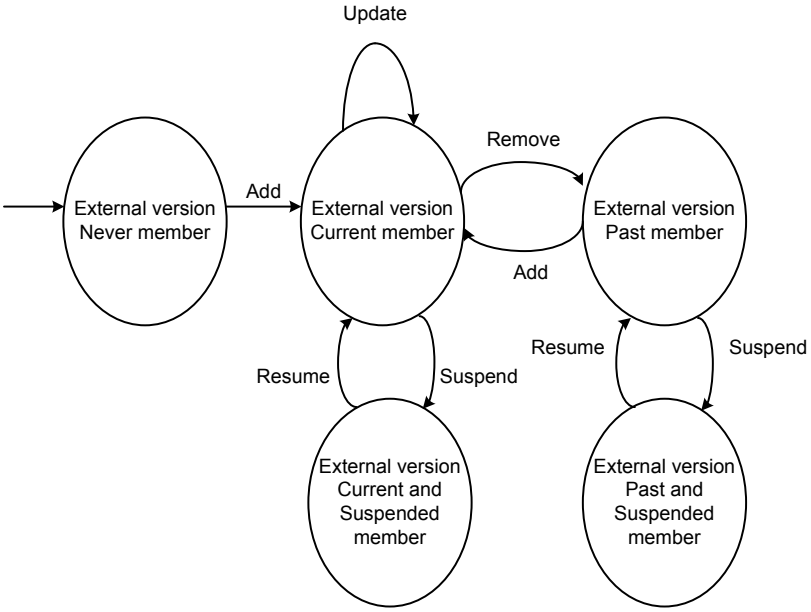

Figure 3.8: External Object Lifecycle I.

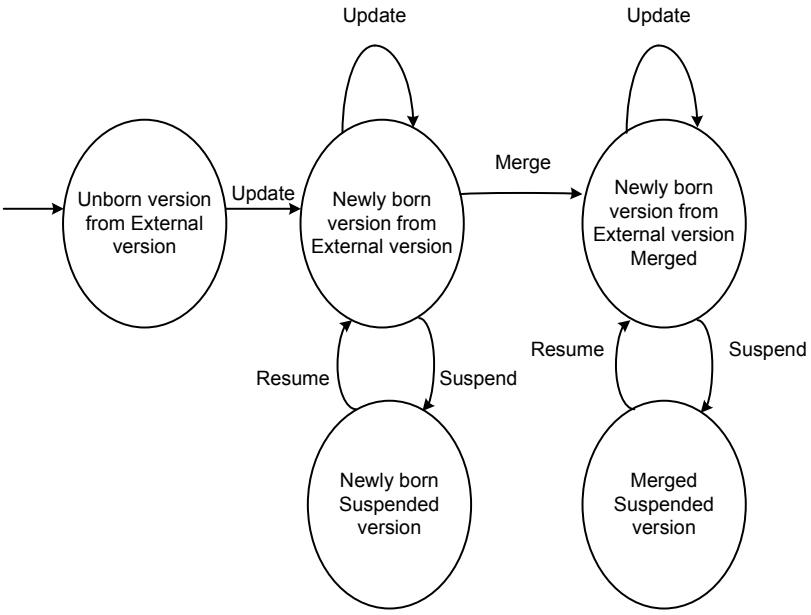

Figure 3.9: External Object Lifecycle II.

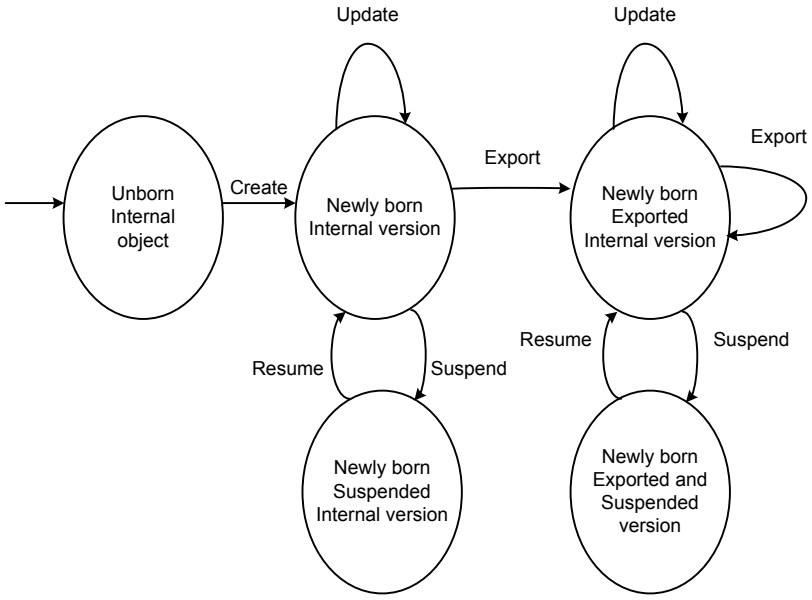

Figure 3.10: Internal Object Lifecycle I.

However, Update does not change the state of the previously existing updated version. Thus the previously existing version remains in the same state even after the Update operation. This added version may be removed using the Remove operation and may subsequently be re-added. Note that in any state, the version may be suspended and resumed any number of times. Suspended versions are not accessible to any user in the group unless it is resumed using Resume. Also, a removed version cannot be updated.

As shown in figure 3.9, a new version may be created from the externally created object using the Update operation. Such a new version created from the external version may be merged back only to the external entity that initially shared the version. Note that subsequent updates create a new version but does not change the state of the version being updated. Again, the version in any state may be suspended and subsequently resumed.

Case (b) Internally Created Object: Figures 3.10 and 3.11 show the life-cycle of an object that is created internally in the group. As shown in figure 3.10, a new object with an initial version of $v0$ may be created using the Create operation. Such an internally created version and any subsequent version created using Update may be exported to an external

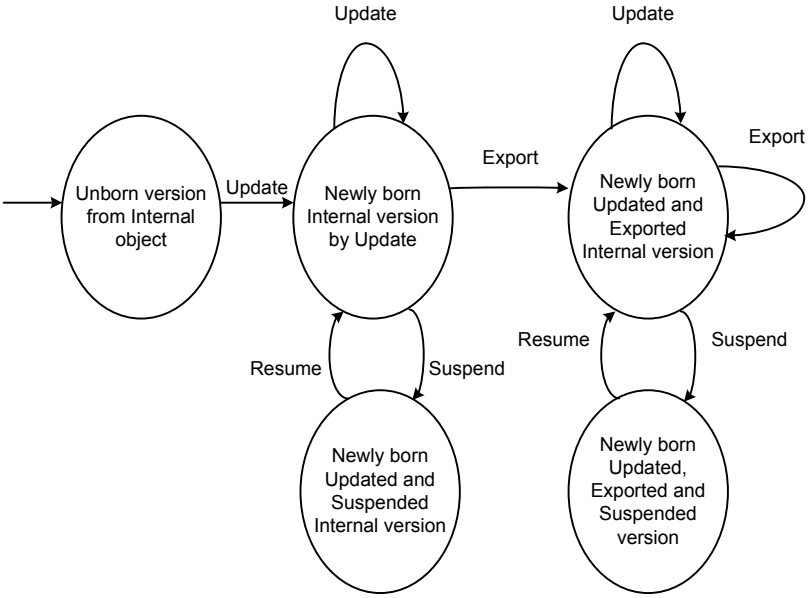

Figure 3.11: Internal Object Lifecycle II.

entity using the Export operation. Versions may be suspended and resumed as earlier. As shown in figure 3.11, new versions of internally created object may be created using the Update operation which may in turn be exported to an external entity. Note that in figures 3.10 and 3.11 the versions may be exported multiple times but to different entities.

Clearly, this versioning model allows two or more entities (such as organizations) to share their resources (users and objects) using the group. New versions created from the shared object may be merged back to the source entity. Also, new objects and updated versions may be internally created in the group. This may represent new ideas and intellectual property created during the sharing process. Such internally created objects may be exported to an external entity depending on who the stake-holders are in the sharing and collaboration process.

We emphasize that we have not specified the administrative model for authorizing operations such as Merge, Export much as we did not specify administrative models for Join, Leave, Add and Remove.

Table 3.3: Summary of group operations

Operation	Explanation
Join (u,g)	Join user u to group g.
Leave (u,g)	Leave user u from group g.
Add (o.v$_i$,g)	Add version v_i of object o to group g.
Remove (o.v$_i$,g)	Remove version v_i of object o from group g.
Create (o.v$_0$,g)	Create object o in group g. This creates a version 0 of o.
Update (o.v$_i$,o.v$_j$,g)	Update version v_i of object o in group g resulting in a new version v_j.
Suspend (o.v$_i$,g)	Suspend version v_i of object o in group g.
Resume (o.v$_i$,g)	Resume version v_i of object o in group g.
Export (o.v$_i$,fromg, tog)	Export version v_i of object o in group $fromg$ to group tog.
Merge (o.v$_i$,fromg, tog)	Merge version v_i of object o in group g to group tog.

Well-Formed Traces

Table 3.3 gives an overview of group operations in the versioning model. Like earlier, each of these operations could be of different type and each operation displayed in the table denote a disjunction of such types. Where obvious, we omit the parameters for these operations for convenience. The following formulas specify constraints on g-SIS language that separate well-formed traces from all possible traces.

A. Two operations cannot occur on the same user or object in the same state.

$$\tau_0 \equiv \Box(\neg(\text{Add} \wedge \text{Remove}) \wedge \neg(\text{Join} \wedge \text{Leave}) \wedge \neg(\text{Suspend} \wedge \text{Resume}) \wedge$$

$$\neg(\text{Create} \wedge \text{Suspend}) \wedge \neg(\text{Create} \wedge \text{Update}) \wedge \neg(\text{Create} \wedge \text{Export}) \wedge$$

$$\neg(\text{Add} \wedge \text{Suspend}) \wedge \neg(\text{Add} \wedge \text{Update}) \wedge \neg(\text{Add} \wedge \text{Merge}) \wedge$$

$$\neg(\text{Remove} \wedge \text{Suspend}) \wedge \neg(\text{Remove} \wedge \text{Update}) \wedge \neg(\text{Remove} \wedge \text{Merge}) \wedge$$

$$\neg(\text{Update} \wedge \text{Suspend}) \wedge \neg(\text{Update} \wedge \text{Export}) \wedge (\text{Update} \wedge \text{Merge}))$$

B. For any given user or object, two types of operations cannot occur at the same time.

$$\tau_1 \equiv \forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{join}_i \wedge \text{join}_j)) \wedge \forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{leave}_i \wedge \text{leave}_j)) \wedge$$

$$\forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{add}_i \wedge \text{add}_j)) \wedge \forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{remove}_i \wedge \text{remove}_j))$$

$$\forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{suspend}_i \wedge \text{suspend}_j)) \wedge \forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{resume}_i \wedge \text{resume}_j))$$

$$\forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{create}_i \wedge \text{create}_j)) \wedge \forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{update}_i \wedge \text{update}_j))$$

$$\forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{export}_i \wedge \text{export}_j)) \wedge \forall i,j\ \Box((i \neq j) \rightarrow \neg(\text{merge}_i \wedge \text{merge}_j))$$

C. An enabling and disabling operation should occur alternatively. Note that merge and export operations can never re-occur for the same object version between two groups. Similarly, the create operation can never re-occur for the same object version in the group.

$$\tau_2 \equiv \Box(\text{Join} \rightarrow \bigcirc\ (\neg\text{Join}\ \mathcal{W}\ \text{Leave})) \wedge \Box(\text{Leave} \rightarrow \bigcirc\ (\neg\text{Leave}\ \mathcal{W}\ \text{Join})) \wedge$$

$$\Box(\text{Add} \rightarrow \bigcirc\ (\neg\text{Add}\ \mathcal{W}\ \text{Remove})) \wedge \Box(\text{Remove} \rightarrow \bigcirc\ (\neg\text{Remove}\ \mathcal{W}\ \text{Add})) \wedge$$

$$\Box(\text{Suspend} \rightarrow \bigcirc\ (\neg\text{Suspend}\ \mathcal{W}\ \text{Resume})) \wedge \Box(\text{Resume} \rightarrow \bigcirc\ (\neg\text{Resume}\ \mathcal{W}\ \text{Suspend})) \wedge$$

$$\Box(\text{Export}(o.\text{v}_i, fg, tg) \rightarrow \bigcirc \neg \Diamond\ \text{Export}(o.\text{v}_i, fg, tg)) \wedge$$

$$\Box(\text{Merge}(o.\text{v}_i, fg, tg) \rightarrow \bigcirc \neg \Diamond\ \text{Merge}(o.\text{v}_i, fg, tg)) \wedge$$

$$\Box(\text{Create}(o.\text{v}_0, g) \rightarrow \bigcirc \neg \Diamond\ \text{Create}(o.\text{v}_0, g))$$

D. In a group, a disabling event cannot occur before an enabling event. Also, update, merge and export can occur on an object version only after it is added to or created in the group.

$$\tau_3 \equiv \Box(\text{Leave} \rightarrow \blacklozenge\text{Join}) \wedge \Box(\text{Remove} \rightarrow \blacklozenge\text{Add}) \wedge \Box(\text{Resume} \rightarrow \blacklozenge\text{Suspend}) \wedge$$

$$\Box(\text{Suspend}(o.\text{v}_i, g) \rightarrow \blacklozenge(\text{Add}(o.\text{v}_i, g) \vee \text{Create}(o.\text{v}_i, g) \vee \exists \text{v}_j.\ \text{Update}(o.\text{v}_j, o.\text{v}_i, g))) \wedge$$

$$\Box(\text{Update}(o.\text{v}_i, o.\text{v}_j, g) \rightarrow \blacklozenge(\text{Add}(o.\text{v}_i, g) \vee \text{Create}(o.\text{v}_i, g) \vee \exists \text{v}_k.\text{Update}(o.\text{v}_k, o.\text{v}_i, g))) \wedge$$

$$\Box(\text{Merge}(o.\text{v}_i, fg, tg) \rightarrow \blacklozenge\ \text{Add}(o.\text{v}_i, fg) \vee$$

$$\blacklozenge\ (\exists \text{v}_j.\text{Add}(o.\text{v}_j, fg) \wedge \exists \text{v}_k.\text{Update}(o.\text{v}_k, o.\text{v}_i, fg))) \wedge$$

$$\Box(\text{Export}(o.\text{v}_i, fg, tg) \rightarrow \blacklozenge\ \text{Create}(o.\text{v}_i, fg) \vee$$

$$\blacklozenge\ (\text{Create}(o.\text{v}_0, fg) \wedge \exists \text{v}_j.\text{Update}(o.\text{v}_j, o.\text{v}_i, fg)))$$

Core Properties

Following are the core g-SIS properties for read and write authorizations.

1. **Persistence Property:** Note here that the create group operation is not a required condition for Authz to change since create is concerned with creating a new (different) object. The $-$ in Authz in the antecedent and consequent both refer to either a read or write action.

$$\kappa_0 = \Box(\text{Authz}(u, o.\text{v}_i, -) \rightarrow (\text{Authz}(u, o.\text{v}_i, -)\ \mathcal{W}\ (\text{Join} \vee \text{Leave} \vee \text{Add} \vee \text{Remove} \vee$$

$$\text{Update} \vee \text{Suspend} \vee \text{Resume} \vee \text{Export} \vee \text{Merge})))$$

$$\kappa_1 = \Box(\neg\text{Authz}(u, o.\text{v}_i, -) \rightarrow (\neg\text{Authz}(u, o.\text{v}_i, -)\ \mathcal{W}\ (\text{Join} \vee \text{Leave} \vee \text{Add} \vee \text{Remove} \vee$$

$$\text{Update} \vee \text{Suspend} \vee \text{Resume} \vee \text{Export} \vee \text{Merge})))$$

2. **Authorization Provenance:** We accommodate the possibility of Create, Suspend, Resume and Update operations to specify provenance for read and write.

$$\kappa_2 = (\neg \text{Authz}(u, o.\text{v}_i, r)\ \mathcal{W}\ (\text{Authz}(u, o.\text{v}_i, r) \wedge$$

$$(\neg \text{Leave}(u, g)\ \mathcal{S}\ \text{Join}(u, g)) \wedge ((\neg \text{Remove}(o.\text{v}_i, g) \wedge \neg \text{Suspend}(o.\text{v}_i, g))\ \mathcal{S}$$

$$(\text{Resume}(o.\text{v}_i, g) \vee \text{Add}(o.\text{v}_i, g) \vee \text{Create}(o.\text{v}_i, g) \vee \exists \text{v}_j.\text{Update}(o.\text{v}_j, o.\text{v}_i, g)))))$$

For write, the user needs to be a current member to update any version in the group. Also, the user can update only those versions that he/she is authorized to read. Note that Export and Merge operation do not impact authorizations in the group.

$$\kappa_3 = \Box(\text{Authz}(u, o.\text{v}_i, w) \rightarrow \text{Authz}(u, o.\text{v}_i, r) \wedge (\neg \text{Leave}(u, g)\ \mathcal{S}\ \text{Join}(u, g))$$

$$\wedge ((\neg \text{Remove}(o.\text{v}_i, g) \wedge \neg \text{Suspend}(o.\text{v}_i, g))\ \mathcal{S}\ (\text{Resume}(o.\text{v}_i, g)$$

$$\vee \text{Add}(o.\text{v}_i, g) \vee \text{Create}(o.\text{v}_i, g) \vee \exists \text{v}_j.\text{Update}(o.\text{v}_j, o.\text{v}_i, g)))))$$

3. **Bounded Authorization:** Properties κ_4 and κ_5 are similar to the single version g-SIS.

$$\kappa_4 = \Box((\text{Leave}(u, g) \wedge \neg \text{Authz}(u, o.\text{v}_i, r)) \rightarrow (\neg \text{Authz}(u, o.\text{v}_i, r)\ \mathcal{W}\ \text{Join}(u, g)))$$

$$\kappa_5 = \Box((\text{Remove}(o.\text{v}_i, g) \wedge \neg \text{Authz}(u, o.\text{v}_i, r)) \rightarrow (\neg \text{Authz}(u, o.\text{v}_i, r)\ \mathcal{W}\ \text{Add}(o.\text{v}_i, g)))$$

Property κ_6 states that group users cannot read or write to a suspended version unless it is resumed. Also a past user cannot write to any version in the group (Property κ_7).

$$\kappa_6 = \Box(\text{Suspend}(o.\text{v}_i, g) \rightarrow \forall u.\ (\neg\text{Authz}(u, o.\text{v}_i, r) \wedge \neg\text{Authz}(u, o.\text{v}_i, w))\ \mathcal{W}\ \text{Resume}(o.\text{v}_i, g))$$

$$\kappa_7 = \Box(\text{Leave}(u, g) \rightarrow \forall \text{v}_i.\ (\neg\text{Authz}(u, o.\text{v}_i, w)\ \mathcal{W}\ \text{Join}(u, g)))$$

4. **Availability:** A current group user is authorized to read or update any version that is created (by means of Add or Create or Update operation) after the user's join time.

$$\kappa_8 = \Box(\text{Join}(u, g) \rightarrow ((\text{Add}(o.\text{v}_i, g) \vee \text{Create}(o.\text{v}_i, g) \vee \exists \text{v}_j.\text{Update}(o.\text{v}_j, o.\text{v}_i, g) \rightarrow \text{Authz}(u, o.\text{v}_i, -)\ \mathcal{W}\ (\text{Leave}(u, g) \vee \text{Remove}(o.\text{v}_i, g)))\ \mathcal{W}\ \text{Leave}(u, g)))$$

5. **Version Dependency Properties:** κ_9 states that if a current user can read or write the base version (version 0) of an object o, then the user can read or write all unsuspended versions of o in the group.

$$\kappa_9 = \Box((\text{Authz}(u, o.\text{v}_0, -) \wedge (\neg\text{Leave}(u, g)\ \mathcal{S}\ \text{Join}(u, g))) \rightarrow \forall \text{v}_i.\ (((\neg\text{Suspend}(o.\text{v}_i, g) \wedge \neg\text{Remove}(o.\text{v}_i, g))\ \mathcal{S}\ (\exists \text{v}_j.\text{Update}(o.\text{v}_j, o.\text{v}_i, g) \vee \text{Create}(o.\text{v}_i, g) \vee \text{Resume}(o.\text{v}_i, g))) \rightarrow \text{Authz}(u, o.\text{v}_i, -))))$$

κ_{10} states that if a user can read some version of an object o, then all prior unsuspended versions of o can be read by that user.

$$\kappa_{10} = \Box(\text{Authz}(u, o.\text{v}_i, r) \rightarrow \blacklozenge((\exists \text{v}_j.\text{Update}(o.\text{v}_j, \text{v}_i, g) \vee \text{Create}(o.\text{v}_i, g)) \wedge$$

$$(\forall \text{v}_k.(\neg\text{Suspend}(o.\text{v}_k, g) \wedge \neg\text{Remove}(o.\text{v}_k, g)) \mathcal{S} (\exists \text{v}_l.\text{Update}(o.\text{v}_l, \text{v}_k, g)$$

$$\vee \text{Create}(o.\text{v}_k, g)) \rightarrow \text{Authz}(u, o.\text{v}_k, r))))$$

κ_{11} states that if a user u can write some version of o, then u can write to all unsuspended versions of o.

$$\kappa_{11} = \Box(\text{Authz}(u, o.\text{v}_i, w) \rightarrow \forall \text{v}_j.\ ((\neg\text{Suspend}(o.\text{v}_j) \wedge \neg\text{Remove}(o.\text{v}_i, g))\ \mathcal{S}$$

$$(\exists \text{v}_x.\ \text{Update}(o.\text{v}_x, o.\text{v}_j, g) \vee \text{Resume}(o.\text{v}_j, g))) \rightarrow \text{Authz}(u, o.\text{v}_j, w))$$

6. **Object Create Property:** This property states that only current users in the group can create an object.

$$\kappa_{12} = \Box(\text{Create}(o.\text{v}_0, g) \rightarrow (\neg\text{Leave}(u, g)\ \mathcal{S}\ \text{Join}(u, g)))$$

7. **Version Publish Properties:** Property κ_{13} states that a version of an object can be exported to an external entity only if the object was created internally in the group. Property κ_{14} states that a version of an object can be merged to an external entity only if some previous version of that object was added to the group from that external entity. The $\mathcal{B}$ operator is a weaker version of $\mathcal{S}$ where if q in $(p\ \mathcal{B}\ q)$ never occurred in the past, then p holds all the way from the initial state.

$$\kappa_{13} = \Box(\text{Export}(o.\text{v}_i, fg, tg) \rightarrow \blacklozenge\ \text{Create}(o.\text{v}_0, fg) \wedge (\neg\text{Suspend}(o.\text{v}_i, fg)\ \mathcal{S}$$

$$(\text{Resume}(o.\text{v}_i, fg) \vee \text{Create}(o.\text{v}_i, fg) \vee \text{Update}(o.\text{v}_j, o.\text{v}_i, fg))))$$

$$\kappa_{14} = \Box(\text{Merge}(o.\text{v}_i, fg, tg) \rightarrow \blacklozenge\, \exists \text{v}_x.\text{Add}(o.\text{v}_x, fg) \wedge ((\neg\text{Suspend}(o.\text{v}_i, fg)$$

$$\wedge \neg\text{Remove}(o.\text{v}_i, fg))\ \mathcal{B}\ (\exists \text{v}_j.\text{Update}(o.\text{v}_j, o.\text{v}_i, fg) \vee \text{Resume}(o.\text{v}_i, fg))))$$

All of these core properties (κ_0 to κ_14) must be satisfied by any read-write g-SIS specification with object versioning. Depending on the specific needs of the application, one of the three sets of core properties discussed earlier may be used to develop g-SIS specifications. We specified core properties for a read-only g-SIS model (with new object Add), a read-write single version g-SIS model and a read-write g-SIS model with object versioning.

3.2 Case Study: Inter-Organizational Collaboration

In this section, we present a case-study of group-centric sharing built around an inter-organizational collaboration scenario. Collaboration, in general, can be of many different types and at different scales. In most inter-organizational and similar collaboration scenarios, a group is established between participating organizations to promote sharing and collaboration. Each organization, a stake holder, may share its resources (e.g. users and objects) with others by adding them to the common group. As collaboration proceeds, objects may be created (e.g. new intellectual property) which may be transferred back to the stake holders. Such collaboration scenarios have many characteristics: small-scale vs large-scale, short-term vs long-term, dynamic vs static membership, equal vs unequal stakeholders, unilateral vs bi/multi-lateral administration, uniform vs differentiated permissions, etc. Joint product design between two or more organizations, merger and acquisition scenarios, program committee meetings, micro-collaboration such as in a research paper where multiple authors collaborate, collaboration for hire between a product team in an organization and temporary consultants/contractors, interest groups such as in IETF where members participate in writing RFCs, etc. are few examples of collaboration scenarios with such characteristics. On the other hand, we have massively large-scale, highly distributed collaboration scenarios where users may not represent a specific organization and the notion

of group tends to be fuzzy. Examples of such scenarios include Wikipedia, Crowd-Sourcing (where humans answer questions that are hard for a computer to solve efficiently such as describing a photograph), etc. In this case study, we focus on the inter-organizational collaboration scenario.

3.2.1 Collaboration Scenarios

We discuss three distinct collaboration scenarios below. Later we specify a complete formal policy model for one of them. We then generalize these in to a unified framework for developing policy models for group-centric collaboration.

Scenario #1 (Bilateral Organizational Collaboration) Consider two organizations A and B that want to collaborate for some common purpose such as developing Intellectual Property (IP). We refer to A and B as Source Organizations (SO). A Collaboration Group (CG) is established and each organization shares its resources (users, objects, etc.) by adding them to the group. Thus the objective of the collaboration is to share assets, develop IP and take-away identical set of results. All users in CG are treated equally in terms of the privileges that they can exercise in the group. While user and object membership change is expected, it is not expected to be frequent. User authentication to CG is performed through an inter-organizational federated identity system. That is, there is no local id for CG and users use their SO id to authenticate and participate in CG. Further, we also assume federated CG administration. That is, the respective SO (A and B) is responsible for adding and removing its users and objects to CG. For example, if A decides to remove its user from CG, B is not required to participate in that administrative process. However, a collective CG admin is established for controlling objects such as IP that may be created in CG. CG admins are representatives from A and B. CG admins decide what information from CG flows back to participating SOs. In this case, during and at the end of collaboration, CG admins export intermediate results and the final IP back to both A and B.

Scenario #2 (Organizational Collaboration for Hire) Consider organizations A and B where A is a defense services organization and B is a consulting company. This is similar to scenario #1, except A is the ultimate stakeholder and sets of consultants from B are admitted to CG periodically. Org A users in CG are long-term members, whereas different sets of B consultants are brought into CG at different periods of time for a specific purpose. Thus, B users are authorized to access objects added only during their period of membership but A users may access all objects in CG. Users in CG may be substituted with another user temporarily from corresponding SO. Substituted users inherit all permissions of the outgoing user. Authentication to CG is performed through a federated id between A and B. However, A controls membership to CG. That is A may decide to add and remove users and objects to CG at its discretion. At the end of collaboration, A takes away the results. B is compensated for its service.

Scenario #3 (Qualified Open Collaboration) Consider an interest group such as in IETF where many users jointly work on an RFC. Here users share, collaborate and finally publish or release the results of collaboration, say, to the public. All users are treated equally in CG. Users are authenticated to CG through a federated id system or based on a local identity infrastructure. Membership change is expected to be highly frequent with different users coming in and going out and possibly re-joining while contributing information and collaborating in CG. We assume that CG is self-administered. That is membership to CG is controlled by CG admin(s) who also control final release of the results of the collaboration. These scenarios albeit relatively simple, illustrate the relevance of groups in a wide variety of collaboration scenarios.

3.2.2 Formal Specification of Bilateral Inter-Organizational Collaboration

We formally specify a complete policy model for the bilateral inter-organizational collaboration discussed in scenario #1 earlier. We closely follow the attribute-based UCON model

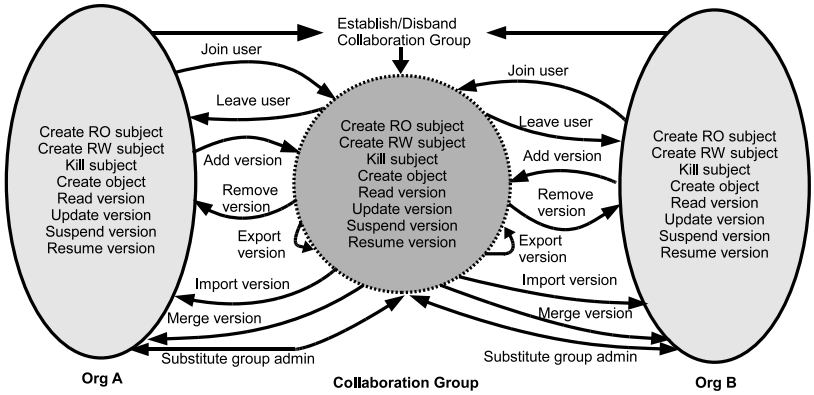

Figure 3.12: Each entry gives the operation name followed by its principal target (specified in further detail in the subsequent tables). Operations labeled over arcs belong to the administrative model and those within the group or organizations (all identical) belong to the operational model.

for usage control [9]. Authorization to perform an operation (such as reading an object) is expressed over the attributes (e.g. user id, group membership, etc.) of the entities involved in the operation (such as the user and the object). A unique and powerful capability of UCON is that if the operation succeeds, the attributes may be updated. For example, if the operation is to remove a user from a group, the user's membership attribute is updated accordingly. In the following, for convenience, we interchangeably use the terms "organization" and "org" and "collaboration group" and "group" respectively. In UCON terminology, our model is a UCON_A model, involving only authorizations but no obligations or conditions.

Overview

We assume a federated administrative model for managing user and object membership in the group. User membership in the group is controlled by respective org's admins. Each collaborating org has total control on users and objects that they want to contribute or remove from the group.

In a distributed sharing setting such as inter-organizational collaboration, we expect that objects will be versioned. We assume a general version model where each write operation on

an object creates a new version of that object (see section 3.1.8). Orgs can share a specific version of an object with other orgs by adding it to the collaboration group. The same version may be shared by an org with one or more collaboration groups. Newer versions of the object created in the org are not automatically available to users in the group unless explicitly shared. Similarly, newer versions created in the group from the version shared by an org are not automatically available to users in the org unless explicitly shared. Such new versions in the group can be merged back to that org as allowed by the group admins.

New objects (and new versions of those objects) may be created in the group during collaboration. The group's admins (representatives from each collaborating org) may export specific versions of such objects to all collaborating organizations (recall that in scenario #1 each org is an equal stakeholder). We use a pull model here where specific versions in the group are marked as importable and it is the responsibility of collaborating orgs' admins to import such versions back to their respective orgs as and when they are made available.

Figure 3.12 illustrates a group that is established between two orgs A and B (more generally, any number of orgs may participate). Operations labeled over the arcs are administrative operations and those that are contained within the group and the orgs belong to the operational model where users create subjects (processes) and exercise their privileges. Thus an administrative model specifies the authorization for operations such as Join and Leave, while the operational model specifies the authorizations for user to create a subject and exercise privileges in the group and organization. We discuss these two models below.

Attributes

Table 3.4 specifies the sets and attributes in the system. UNIV_ORG, UNIV_CG, UNIV_U, UNIV_S, UNIV_O, UNIV_V represent the universal sets (name spaces) of orgs, groups established between such orgs, users, subjects, objects and object versions respectively. Similarly sets ORG, CG, U, S and O represent the existing sets of respective entities. Note that since each object in the set O require the name space of UNIV_V, we do not maintain

Table 3.4: Attribute Definitions

Universal sets of names:
UNIV_ORG: The universe of organizations
UNIV_CG: The universe of collaboration groups
UNIV_U: The universe of users
UNIV_S: The universe of subjects
UNIV_O: The universe of objects
UNIV_V: The universe of versions

Existing sets of names:
ORG: Set of all existing collaborating organizations
CG: Set of all existing groups established between orgs in ORG
U: Set of all existing users
S: Set of all existing subjects
O: Set of all existing objects

User Attributes:
Att (U) = {uorg, ucg, orgadmin, cgadmin}
uorg : $U \rightarrow ORG$
ucg : $U \rightarrow 2^{CG}$
orgadmin : $U \rightarrow \{True, False\}$
cgadmin : $U \rightarrow 2^{CG}$

Objects Attributes:
Att (O) = {member, currV}
member : $O \rightarrow ORG \cup CG$
currV : $O \rightarrow 2^{UNIV\_V}$

Object Version Attributes:
Att (O, UNIV_V) = {vMember, vSuspended, importable}
vMember : $O \times UNIV\_V \hookrightarrow 2^{ORG \cup CG}$
vSuspended : $O \times UNIV\_V \hookrightarrow \{True, False\}$
importable : $O \times UNIV\_V \hookrightarrow \{True, False\}$
As shown, vMember, vSuspended and importable are partial functions that are undefined for object versions that do not currently exist.

Group Attributes:
Att (CG) = {assoc}
assoc : $CG \rightarrow 2^{ORG}$

Subject Attributes:
Att (S) = {sOwner, type, belongsTo}
sOwner : $S \rightarrow U$
type : $S \rightarrow$ {ro, rw}
belongsTo : $S \rightarrow ORG \cup CG$

a single set of existing versions. As will be discussed later, we address this by maintaining an attribute of the object, currV, which keeps track of the existing versions of that object.

User Attributes: "User" is the representation of a human in the system. We assume that a user can be a member of only one organization[7]. Thus attribute uorg specifies the membership organization of a user. But there could be any number of groups established between the organizations and the user may be a member of more than one group. Thus ucg maps the user to a set of groups as indicated by 2^{CG} (the power set of CG). The orgadmin attribute specifies if the user is an administrator of his/her membership organization. The user could be assigned as an administrator of one or more groups as specified by the cgadmin attribute.

Object Attributes: Each object is a member of a single entity (an org or a group) as represented by the memberattribute. The object implicitly becomes a member of an org or a group depending on where it is created. The currV attribute maintains the list of current versions of an object.

Object Version Attributes: Each object may have many versions and specific versions may be shared by organizations with one or more groups. Also, specific versions in a group may be made available to all participating organizations by the group's admins. Thus given an object and version, attribute vMember lists the orgs and/or groups to which the version belongs and vSuspended specifies if the version has been suspended. In our model, we support a Suspend operation instead of a Delete operation. In the case the version was created in a group, importable specifies if the version can be imported by all orgs associated with the group (we will discuss this process in detail in section 3.2.2).

Group Attributes: The assoc attribute of a group lists the set of collaborating orgs that are associated with the group.

[7]This may not be realistic in practice since many users are typically members of multiple organizations. For simplicity, we assume a single organization user membership for the purpose of this case-study.

Subject Attributes: "Subjects" represent processes that execute on behalf of the user. The sOwner attribute specifies the user that owns the subject (the creator of the subject), type specifies whether the subject is read-only or read-write and belongsTo specifies the org or group in which the subject was created.

Administrative Model

In tables 3.5 and 3.6, we formally specify a set of administrative operations that are relevant to the collaboration group (as illustrated in figure 3.12). In the models we discuss, we confine the group operation semantics to Liberal Join, Strict Leave for users and Liberal Add, Strict Remove for objects (that is we follow the fixed operation model (LJ, SL, LA, SR) from section 3.1.6). Note that any variation of these semantics will only affect the authorizations of Join, Leave, Add and Remove operations leaving the rest of the specification intact. Note that we do not consider organizational administrative operations such as the user joining and leaving the org due to space limitations and organizational variations. In the tables, the notation used is similar to that of UCON. the first column specifies the operation that is to be performed, the second column specifies the conditions under which the operation is authorized and the final column specifies the attributes and sets that need to be updated if the operation succeeds. In the third column, an attribute/set name denoted with a superscripted prime refers to its new value after the update and that without the superscripted prime refers to its existing value before the update.

- **Establish** collaboration group: A set of administrative users come together to form a collaboration group between their representative organizations. In the first column in table 3.5, $uSet$ is the set of users and cg is the group to be established. Note that cg cannot be an existing group name. Since UNIV_CG is the set of universe of group names and CG the existing group names, we require that cg be picked from the unused group names set UNIV_CG $-$ CG. In all of the following operations, we take a similar approach whenever a new name is required.

Table 3.5: Administrative model

Operation	Authz Requirement ($\rightarrow$)	Updates
$\forall uSet \subseteq$ U. $\forall cg \in$ UNIV_CG $-$ CG. **Establish**$(uSet, cg)$	$\forall u1, u2 \in uSet.$(orgadmin$(u1) \wedge$ $(u1 \neq u2 \rightarrow$ uorg$(u1) \neq$ uorg$(u2)))$	assoc$'(cg) = \bigcup_{\forall u \in uSet}$ uorg(u) $\forall u \in uSet.$ cgadmin$'(u) =$ cgadmin$(u) \cup \{cg\}$ CG$' =$ CG $\cup \{cg\}$
$\forall u1, u2 \in$ U.$\forall cg \in$ CG. **Join**$(u1, u2, cg)$	orgadmin$(u1) \wedge cg \in$ cgadmin$(u1) \wedge$ (uorg$(u1) =$ uorg$(u2)) \wedge$ $cg \notin$ ucg$(u2) \wedge$ uorg$(u1) \in$ assoc(cg)	ucg$'(u2) =$ ucg$(u2) \cup \{cg\}$
$\forall u1, u2 \in$ U.$\forall cg \in$ CG. **Leave**$(u1, u2, cg)$	orgadmin$(u1) \wedge cg \in$ cgadmin$(u1) \wedge$ (uorg$(u1) =$ uorg$(u2)) \wedge$ $cg \in$ ucg$(u2) \wedge$ uorg$(u1) \in$ assoc(cg)	ucg$'(u2) =$ ucg$(u2) - \{cg\}$ $\forall s \in$ S. sOwner$(s) = u2 \wedge$ belongsTo$(s) = cg$ S$' =$ S $- \{s\}$
$\forall u \in$ U.$\forall o \in$ O. $\forall v \in$ currV(o).$\forall cg \in$ CG. **Add**(u, o, v, cg)	uorg$(u) =$ member$(o) \wedge$ uorg$(u) \in$ assoc$(cg) \wedge$ $cg \in$ cgadmin$(u) \wedge$ orgadmin$(u) \wedge$ uorg$(u) \in$ vMember$(o, v) \wedge$ $cg \notin$ vMember(o, v)	vMember$'(o, v) =$ vMember$(o, v) \cup \{cg\}$
$\forall u \in$ U.$\forall o \in$ O. $\forall v \in$ currV(o).$\forall cg \in$ CG. **Remove**(u, o, v, cg)	uorg$(u) =$ member$(o) \wedge$ $\wedge$uorg$(u) \in$ assoc$(cg) \wedge$ $cg \in$ cgadmin$(u) \wedge$ orgadmin$(u) \wedge$ uorg$(u) \in$ vMember$(o, v) \wedge$ $cg \in$ vMember(o, v)	vMember$'(o, v) =$ vMember$(o, v) - \{cg\}$
$\forall u1, u2 \in$ U.$\forall cg \in$ CG. **Substitute**$(u1, u2, cg)$	orgadmin$(u2) \wedge cg \in$ cgadmin$(u1) \wedge$ uorg$(u1) =$ uorg$(u2) \wedge$ uorg$(u1) \in$ assoc(cg)	cgadmin$'(u1) =$ cgadmin$(u1) - \{cg\}$ cgadmin$'(u2) =$ cgadmin$(u2) \cup \{cg\}$

In column 2, for Establish to succeed, we require that there be exactly one administrative user in the set $uSet$ from each collaborating organization. That is any two users in $uSet$ cannot belong to the same organization. In other scenarios, this operation may simply be performed by one admin user from a single organization.

Subsequently, in column 3, the assoc attribute of cg is updated to the set of organizations of the users in $uSet$ and every user in $uSet$ is made an administrator of cg by setting the respective user's cgadmin attribute. Finally, since the group name cg is now used, we update the existing group names set CG accordingly.

- **Join** user to group: An org's admin user $u1$ is allowed to join the same org's user $u2$ to the group cg if an association exists between $u1$ and $u2$'s org and cg. Note that $u1$ should also be an admin of cg (that is, $u1$ participated in cg's Establish process and became the group admin at that time or was later substituted to be the group admin). $u2$'s ucg attribute is updated to reflect the fact that the user is now a member of cg.

- **Leave** user from group: Leave operation is similar to join, except all the subjects executing on the user's behalf that were created in the group should be killed. To this end, we update the existing subjects set S by removing all the subjects executing in the group that are owned by the leaving user. As we will see in the operational model, this ensures that such subjects cannot continue performing any actions.

- **Add** object version to group: A user u is allowed to add a specific version v of an object o to a group cg if u is an admin of the organization to which the object version belongs and also an admin of cg. After add, the vMember attribute of the object version is updated to reflect the fact that the version is now a member of cg.

- **Remove** object version from group: This is similar to Add operation. Again, u is both an admin of the group from which the version is being removed and the org that initially shared the version with the group. In contrast to Leave operation, subjects need not be killed because the user is still a current member in the group.

- **Substitute** group admin: In the event a cg admin is unavailable or has to leave the group, we provide a Substitute operation to replace the leaving admin. Here, current cg and org admin $u1$ needs to be substituted with another org admin $u2$. Note that we make $u2$ a cgadmin in the update column. This operation is essentially delegation of permissions and more sophisticated approaches may be taken, for example, as discussed in delegation in RBAC [38–40]. For our purpose, this simple substitution operation serves to illustrate the requirement.

- **Export** a version in group to all source orgs: We use a pull model for exporting any object version created newly in the group. That is, the cg admins mark an object version in cg as importable and the collaborating orgs can subsequently copy that version to their respective orgs using the following Import operation. In table 3.6, a set of cg admins come together to authorize Export. Export requires that every user in $uSet$ is a cg admin and users in $uSet$ represent every org associated with cg. The object version should belong to cg and not have been suspended. Note that Export can only be performed on versions of an object that was natively created in cg (as ensured by the requirement member(o)=cg). The importable attribute of the object version is set to True.

- **Import** a version from group to org: The semantics of import operation is to copy the importable object version in cg to the actor's organization. The cg admin u (who is also an admin of his/her organization) copies the importable object version $o1, v1$ to object version $o2, v2$ in u's organization (a new version $v2$ of the object $o2$ is created in org as part of Import). This is allowed if an association exists between u's organization and cg and importable attribute of $o1, v1$ is True. Similar to Export, Import can only be performed on versions of an object that was natively created in cg. The currV attribute of $o2$ is updated to reflect the fact that $v2$ has been used. Note that the imported and exported objects may be added to another group. To prevent such information flow, additional attributes need to be brought into the model.

Table 3.6: Administrative model (continued)

Operation	Authz Requirement ($\rightarrow$)	Updates
$\forall uSet \subseteq \text{U}.\forall cg \in \text{CG}.$ $\forall o \in \text{O}.\forall v \in \text{currV}(o).$ **Export**$(uSet, cg, o, v)$	$\forall u1 \in uSet.cg \in \text{cgadmin}(u1) \wedge$ $\text{assoc}(cg) = \bigcup_{\forall u2 \in uSet} \text{uorg}(u2) \wedge$ $\neg\text{vSuspended}(o, v) \wedge$ $cg \in \text{vMember}(o, v) \wedge$ $\text{member}(o) = cg \wedge \neg\text{importable}(o, v)$	$\text{importable}'(o, v) = \text{True}$
$\forall u \in \text{U}.\forall o1, o2 \in \text{O}.$ $\forall v1 \in \text{currV}(o1).$ $\forall cg \in \text{CG}.$ **Import**$(u, o1, v1, o2, cg)$	$\text{member}(o1) = cg \wedge$ $\text{importable}(o1, v1) \wedge$ $cg \in \text{cgadmin}(u) \wedge \text{orgadmin}(u) \wedge$ $\text{uorg}(u) \in \text{assoc}(cg) \wedge$ $\neg\text{vSuspended}(o1, v1) \wedge$ $\text{member}(o2) = \text{uorg}(u)$	$\text{currV}'(o2) = \text{currV}(o2) \cup \{v2\}$ $/*where\ v2\ \in\ \text{UNIV\_V} - \text{currV}(o2)$ *is a newly generated version id* $*/$ $\text{vMember}'(o2, v2) =$ $\text{vMember}(o2, v2) \cup \text{uorg}(u)$
$\forall uSet \subseteq \text{U}.\forall cg \in \text{CG}.$ $\forall o \in \text{O}.\forall v \in \text{currV}(o).$ **Merge**$(uSet, cg, o, v)$	$\bigcup_{\forall u1 \in uSet} \text{uorg}(u1) = \text{assoc}(cg) \wedge$ $\forall u2 \in uSet.(cg \in \text{cgadmin}(u2)) \wedge$ $\exists u3 \in uSet.\text{uorg}(u3) = \text{member}(o) \wedge$ $cg \in \text{vMember}(o, v) \wedge$ $\text{member}(o) \notin \text{vMember}(o, v)$	$\text{vMember}'(o, v) =$ $\text{vMember}(o, v) \cup \text{member}(o)$
$\forall uSet \subseteq \text{U}.\forall cg \in \text{CG}.$ **Disband**$(uSet, cg, disband)$	$\forall u1 \in uSet.(\text{orgadmin}(u1) \wedge$ $cg \in \text{cgadmin}(u1)) \wedge$ $\bigcup_{\forall u \in uSet} \text{uorg}(u) = assoc(cg)$	$\forall u2 \in \text{U}.\text{uorg}(u2) \in \text{assoc}(cg).$ $(\text{ucg}'(u2) = \text{ucg}(u2) - \{cg\}$ $\text{cgadmin}'(u2) = \text{cgadmin}(u2) - \{cg\})$ $\text{assoc}'(cg) = NULL$ $\forall o \in \text{O}.cg = \text{member}(o).$ $\text{member}'(o) = NULL$ $\forall o \in \text{O}.$ $\forall v \in \text{currV}(o).cg \in \text{vMember}(o, v).$ $\text{vMember}'(o, v) =$ $\text{vMember}(o, v) - \{cg\}$ $\text{CG}' = \text{CG} - \{cg\}$ $\text{S}' = \text{S} - \bigcup_{\forall s \in \text{S.belongsTo}(s) = cg} s$

- **Merge** a version to org: The "merge" operation is performed on new object versions in *cg* that were created from a version that was previously shared by an organization with *cg*. Versions can only be merged back to an organization that shared an older version in the past. Hence, we verify that the object is actually a member of one of the collaborating organizations. Similar to Export, we need one admin from each collaborating organization in *cg* to authorize this operation. This is required to ensure that content of the version that is to be merged back to one of the collaborating organizations is appropriate.

- **Disband** group: The semantics of this operation is to delete the group. We assume that each of the collaborating organizations have copied all the group objects back to their respective organizations before Disband operation is performed. Similar to group creation, we require that one administrator from each associated organization in the group be present to authorize this operation. Once disbanded, the corresponding attributes of every user, object and object version in the group are updated as shown. Also, all subjects executing in the group should be killed. At this point, any object that has not been copied to collaborating orgs become inaccessible.

Operational Model

For the operational model, the user-subject relationship is critical. We assume "user" represents a human who can create "subjects" in the system that execute on his/her behalf. We assume that each user may have multiple subjects, however each subject is owned by one user. The user may be a member of one or more collaboration groups established between any number of orgs. A typical user will then create a subject to read and write objects in the user's org and those in one or more collaboration groups of which he/she is a member. While it may be reasonable to trust the user in this scenario, we cannot trust subjects (programs) running on behalf of the user due to trojan horse issues. Clearly, it is critical to control information flow here. For instance, we should not allow a subject to copy arbitrary information from the org to one of the user's membership groups or copy

information from one group (between org A and B) to another group (between A and C) if the user is member of both groups.

To this end, in our user-subject model, we allow two types of subjects specified at creation time by the user: read-only and read-write. A user can create a read-only subject to freely read information from his/her org and all groups that he/she is a member of. However read-only subjects cannot write information anywhere. Such a subject is convenient for the purpose of aggregation. The user can use a read-only subject to aggregate information from various groups in one place in order to read them, scan for alerts, etc. If the user wants to write information, he/she needs to create a read-write subject. However, due to the possibility of malicious subjects (trojan horses that may copy arbitrary information from one group to other or from an org to group and vice versa), we restrict read-write subjects to one group specified at creation time. Suppose an org A user is a member of group 1 and group 2 that are established between orgs A and B and A and C respectively. A read-write subject created by the user in group 1 will not be allowed to read information from group 1 and write to group 2. However, it can read and write any object that is in group 1. If the user needs to write to objects in group 2, he/she needs to create a separate read-write subject in group 2.

We discuss a set of operations involved in the operational model as specified in tables 3.7 and 3.8. The operations contained within the group or the orgs in figure 3.12 belong to the operational model. These operations allow a user to create subjects and exercise privileges in a group or an org.

- **CreateRO** subject: A user can create a read-only subject in either an organization or a group if he/she is a member of the respective entity. Note that the subject's type attribute is set to read-only and the creating user is made the owner of the subject. The parameter *entity* specifies where the subject should be rooted (the user's organization or a specific group of which the user is a member) and belongsTo attribute is set accordingly. Recall that read-only subjects may read objects from multiple entities (groups and source org). However, they cannot write to any object.

Table 3.7: Operational model

Operation	Authz Requirement ($\rightarrow$)	Updates
$\forall u \in \text{U}.\forall s \in \text{UNIV\_S} - \text{S}.$ $\forall entity \in ORG \cup \text{CG}.$ **CreateRO**$(u, s, entity)$	$\text{uorg}(u) = entity \vee entity \in \text{ucg}(u)$	$\text{sOwner}'(s) = u$ $\text{type}'(s) = ro$ $\text{belongsTo}'(s) = entity$ $\text{S}' = \text{S} \cup \{s\}$
$\forall u \in \text{U}.\forall s \in \text{UNIV\_S} - \text{S}.$ $\forall entity \in ORG \cup \text{CG}.$ **CreateRW**$(u, s, entity)$	$\text{uorg}(u) = entity \vee entity \in \text{ucg}(u)$	$\text{sOwner}'(s) = u$ $\text{type}'(s) = rw$ $\text{belongsTo}'(s) = entity$ $\text{S}' = \text{S} \cup \{s\}$
$\forall u \in \text{U}.\forall s \in \text{S}.$ **Kill**(u, s)	$\text{sOwner}(s) = u \vee (\text{orgadmin}(u) \wedge$ $\text{belongsTo}(s) = \text{uorg}(u)) \vee$ $\text{belongsTo}(s) \in \text{cgadmin}(u)$	$\text{sOwner}'(s) = NULL$ $\text{type}'(s) = NULL$ $\text{belongsTo}'(s) = NULL$ $\text{S}' = \text{S} - \{s\}$
$\forall s \in \text{S}.\forall o \in \text{O}.$ $\forall v \in \text{currV}(o).$ **Read**(s, o, v)	$((\text{type}(s) = rw \wedge$ $\text{belongsTo}(s) \in \text{vMember}(o, v))$ $\vee$ $(\text{type}(s) = ro \wedge$ $(\text{uorg}(\text{sOwner}(s)) \in \text{vMember}(o, v)$ $\vee \text{vMember}(o, v) \cap \text{ucg}(\text{sOwner}(s)) \neq \phi)))$ $\wedge$ $\neg\text{vSuspended}(o, v)$	None

- **CreateRW** subject: A user can create a read-write subject in either an organization or a group if he/she is a member of the respective entity. The subject's type attribute is set to read-write and the creating user is made the owner of the subject. The parameter *entity* specifies where the subject should be rooted (the user's organization or a group) and belongsTo attribute is set accordingly. Note that read-write subjects can read and write objects only in the entity specified by the belongsTo attribute.

- **Kill** subject: A subject can be killed by either the owner of the subject or by the administrator of the organization or the group where the subject is rooted. The existing subjects set S is updated once the subject is killed.

- **Read** an object version: A subject can only read un-suspended object versions. If the subject is of type read-only, the subject can read object versions from the owner's organization or any of the groups of which the user who owns the subject is a member. On the other hand, if it is a read-write subject, the subject can read object versions only from the organization or group where it is rooted (as specified by the belongsTo attribute). Note that we need an ongoing authorization check even after the initial authorization for Read. In the event s is no longer an element of S (which happens if the subject is killed or if the user leaves group) or the version being read gets suspended or ceases to exist in the group or org, the read operation should stop as indicated below.

$$\text{stopped}(s, o, v, \mathit{Read}) \leftarrow s \notin S \vee \text{vSuspended}(o, v) \vee (\text{uorg}(\text{sOwner}(s)) \notin \text{vMember}(o, v) \wedge (\text{vMember}(o, v) \cap \text{ucg}(\text{sOwner}(s))) = \phi)$$

  The notion of stopped is adopted form the UCON model [9]. An enforcement model will provide additional details of how the stopped operation may be enforced.

- **Update** an object version: Subjects of type read-write may update un-suspended object versions. Updating an object version creates a new version of the object in the same organization or group where the subject is rooted. As shown, the new version $v2$ is made a member of the organization or group where the subject is rooted. This operation should stop in the event $s \notin$ S or the version being updated ceases to exist.

$$\text{stopped}(s, o, v1, \mathit{Update}) \leftarrow s \notin S \ \vee \text{belongsTo}(s) \notin \text{vMember}(o, v1)$$

- **Create** an object: A read-write subject can create a new object in the organization or group where it is rooted. As shown, the object is created with a new version v which is made a member of the subject's rooted org or group.

Table 3.8: Operational model (continued)

Operation	Authz Requirement ($\rightarrow$)	Updates
$\forall s \in \mathrm{S}.\forall o \in \mathrm{O}.$ $\forall v1 \in \mathrm{currV}(o).$ **Update**$(s, o, v1)$	$\mathrm{type}(s) = rw \wedge$ $\mathrm{belongsTo}(s) \in \mathrm{vMember}(o, v1) \wedge$ $\neg \mathrm{vSuspended}(o, v1)$	$\mathrm{currV}'(o) = \mathrm{currV}(o) \cup \{v2\}$ $/* where\ v2 \in \mathrm{UNIV\_V} - \mathrm{currV}(o)$ *is a newly generated version id* $*/$ $\mathrm{vMember}'(o, v2) = \mathrm{belongsTo}(s)$ $\mathrm{vSuspended}'(o, v2) = \mathrm{False}$ $\mathrm{importable}'(o, v2) = \mathrm{False}$
$\forall s \in \mathrm{S}.\forall o \in \mathrm{UNIV\_O} - \mathrm{O}.$ **Create**(s, o)	$\mathrm{type}(s) = rw$	$\mathrm{currV}'(o) = \mathrm{currV}(o) \cup \{v\}$ $/* where\ v \in \mathrm{UNIV\_V}$ *is a newly generated version id* $*/$ $\mathrm{vMember}'(o, v) = \mathrm{belongsTo}(s)$ $\mathrm{member}'(o) = \mathrm{belongsTo}(s)$ $\mathrm{vSuspended}'(o, v) = \mathrm{False}$ $\mathrm{importable}'(o, v) = \mathrm{False}$ $\mathrm{O}' = \mathrm{O} \cup \{o\}$
$\forall s \in \mathrm{S}.\forall o \in \mathrm{O}.$ $\forall v \in \mathrm{currV}(o).$ **Suspend**(s, o, v)	$\mathrm{type}(s) = rw \wedge$ $\mathrm{belongsTo}(s) \in \mathrm{vMember}(o, v) \wedge$ $\neg \mathrm{vSuspended}(o, v)$	$\mathrm{vSuspended}'(o, v) = \mathrm{True}$
$\forall s \in \mathrm{S}.\forall o \in \mathrm{O}.\forall v \in \mathrm{currV}(o).$ **Resume**(s, o, v)	$\mathrm{type}(s) = rw \wedge \mathrm{belongsTo}(s) \in \mathrm{vMember}(o, v) \wedge$ $\mathrm{vSuspended}(o, v)$	$\mathrm{vSuspended}'(o, v) = \mathrm{False}$

- **Suspend/Resume** a version: A read-write subject can suspend or resume an object version that is member of the organization or group where the subject is rooted. The vSuspended attribute is set to True or False accordingly.

3.3 Group-Centric Collaboration Framework

In this section, we present a framework for developing group-centric collaboration models. The framework consolidates many issues that are typical in most inter-organizational collaboration scenarios as discussed earlier.

Formal specification of one of the scenarios allowed us to realize a host of issues that needed to be resolved even in the most simple inter-organizational collaboration scenario where two or more orgs establish a group to share their resources. The resources we considered included only users and objects. There were no sub-groups or hierarchical groups within the collaboration group and all users had the same permissions in the group (that is permissions did not depend on other user attributes such as role). Based on this detailed case study, we consolidate various alternatives and issues into a framework for developing group-centric collaboration models.

At a high level, organizational collaboration scenarios can be classified into three distinct phases. In the Begin Collaboration phase, participating organizations make various fundamental decisions about the collaboration. For example, which entities may participate in the collaboration, the entity that controls the collaboration group, etc. Next, in the Collaboration Phase, users participate in the collaboration. During this phase, new ideas are born and in general, the group evolves. The collaborating organizations need to decide on how to handle issues such as who controls membership to the collaboration group, what is the policy if a member exits the source organization, etc. Finally, in the End Collaboration phase, some or all of participating organizations take away the results of the collaboration (this may also happen during the Collaboration Phase). In the following discussion, we refer to the organizations that participate in the collaboration by contributing users and objects as Source Organizations (SO). The group that is established for collaboration is referred to

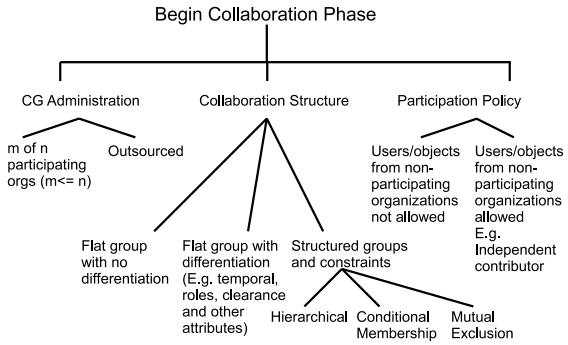

Figure 3.13: Begin Collaboration Phase.

as Collaboration Group (CG). Any other organization that facilitates the collaboration is referred to as third-party.

Begin Collaboration Phase

In this phase (figure 3.13), there are at least three important high-level decisions to be made. First a decision on who is responsible for administration of the CG needs to be made. For example, when 3 SOs A, B and C collaborate, CG may be controlled and hosted at one of the SOs or control may be shared between two of the SOs or all of them. Alternatively, the control could be outsourced to a third-party. Next, participating SOs need to decide if CG users will be differentiated in terms of privileges that they can exercise in the group where a specific group structure may be established for this purpose. In the simplest case, it could be a flat group where all users are treated equally or users may be differentiated as per their attributes such as their roles, time of joining the group, etc. In more complex settings, various structures and constraints could be established. For example, the groups could be set up in a hierarchical fashion where membership at higher group may automatically grant permissions to lower groups. Further constraints such as mutual exclusion (where membership in one group may be mutually exclusive to membership in one or more other groups) and/or conditional membership (where membership in a source group may be required for continued membership in another group) may be imposed. A

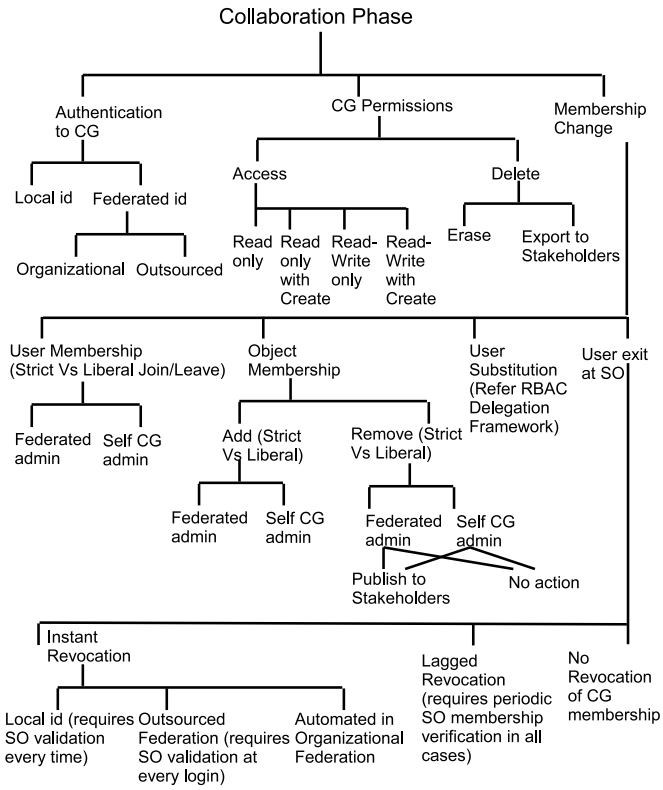

Figure 3.14: Collaboration Phase.

final issue is whether users and objects from a non-participating organization are allowed to be present in CG. In some scenarios this may be desirable where although only two organizations formally initiate and establish a collaboration, a limited number of users may be admitted from a different organization. The users may also be individual contributors who may not belong to any organization. In other cases, this may not be allowed due to the sensitivity of collaboration.

Collaboration Phase

During this phase (figure 3.14), collaboration occurs and users and objects may be added and removed and the CG evolves. First, users need to be authenticated to CG. Authentication can be carried out locally by CG or could be federated to respective SOs. While maintaining a local identity infrastructure is not desirable in inter-organization collaboration, it

is practical in scenarios where users don't belong to any organization, yet collaborate. In federated id systems, respective organizations authenticate their users and send a verifiable assertion that the user has been authenticated. CG can verify the assertion and decide to let the user in or not. Clearly, it is the responsibility of SOs to correctly authenticate their users.

The next issue is handling membership change. As mentioned earlier in section 3.2.2, users may join and leave the group with SJ or LJ and SL or LL respectively. Administration of user join and leave may be federated or controlled locally by the CG. In federated admin, the SOs admit and remove their users to/from CG at their discretion. In self CG admin, the CG admins are responsible for managing users. Admission policy could vary depending on the scenario. For example, only faculty members from accredited universities may be allowed to join CG.

Similar to user membership, object membership may be either federated or self administered by CG. In federated admin, participating SOs decide what objects they want to contribute to CG. Note that federated object remove is tricky in some situations. Suppose that organization A adds an object to CG and organization B members in CG update those objects. In federated remove, if organization A decides to remove the object from CG, organization B CG members may lose access to that object. Clearly, the object update model is relevant in this scenario. If updating an object, creates a new version, we have the flexibility to support any type of policy for object remove. Thus if organization A removes the original object, in some scenarios, newer versions of that object may be allowed to remain in CG. On the other hand, object remove may be controlled by CG admins locally. At remove time, they may choose to either export the removed object to all the stakeholders in the collaboration or take no action, in which case the object is simply removed from the group with no information flowing back to other stakeholders. Further objects could be added and removed in Strict or Liberal fashion as mentioned in section 3.2.2.

The next issue is whether user substitution is allowed which involves questions related to delegation of substituting user's permissions in CG. For example, it could be temporary or

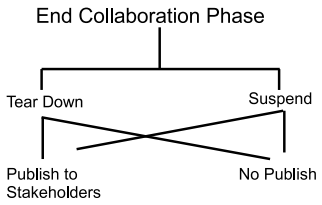

Figure 3.15: End Collaboration Phase.

permanent substitution, single-step (where only one level of substitution allowed) vs multi-step (where subsequent substitutions may be allowed), etc. There are a number of such alternatives and many issues in the delegation framework [38–40] proposed for role-based access control apply here.

A final issue is how to handle CG users leaving their respective SOs. In certain scenarios, an instant or lagged revocation to CG may be required. In other scenarios, user's exit from SO may not affect their membership in CG. Instant revocation to CG can be enforced if the user's validation with the SO is carried out with every login. This is simple in organizational federated id scenarios. However, in both local id and outsourced federation, user's membership in SO needs to be verified every time the user attempts to login to CG. Similar issues exist in lagged revocation, except that membership validation with SO is performed periodically instead of doing it at every login. Clearly, instant revocation may incur significant performance penalty while lagged revocation may incur assurance penalty while achieving better performance.

End Collaboration Phase

When the collaboration ends (figure 3.15), CG may be torn down or suspended if future collaboration is anticipated. In either case, the results of the collaboration may need to published to the stakeholders of the collaboration. In some cases, participating organizations may all be equal stakeholders, while in others, only some of the participating organizations may be stakeholders and the remaining SOs may not own the final outcome. In certain unique scenarios (such as is confidential military operations), the results of collaboration

Table 3.9: Informal Policy model for the 3 usage scenarios based on the framework

Issues	Scenario #1	Scenario #2	Scenario #3
1. Who creates and owns CG?	Collective admin	Org A admins	CG admins
2. Differentiate Users?	No	Yes. Timeliness of membership.	No
3. User/object membership	Federated	Federated	Self-admin
4. Users/objects from 3rd party?	No	No	N/A
5. Authentication to CG	Federated id	Federated id	Local id
6. Membership policy	(LJ, SL), (LA, SR)	(LJ, SL) for A, (SJ, SL) for B, Mixed operations for objects (SA/LA, SR/LR)	(LJ, SL), (LA, SR)
9. Handling user exit from SO	Handled by federated id	Handled by federated id	N/A
10. Handling object deletes in SO	SR from group	SR/LR at CG admin discretion	SR/LR at CG admin discretion
11. User substitution	Yes. Single-step, temporary, complete inheritance.	Yes. Single-step, temporary, complete inheritance.	No
12. Disband CG	Orgs A and B take away results	Only org A takes away results	CG admins publish

may need to be destroyed after the mission is accomplished in which case there may not be any tangible information flowing back to stakeholders.

Based on this framework, table 3.9 outlines an informal policy model for the three scenarios discussed in section 3.2.1 (recall that the complete model presented was for scenario #1).

Chapter 4: Enforcement Models for g-SIS

In this chapter, we discuss an enforcement model for the π-system of read-only g-SIS policy models discussed in section 3.1.5. An enforcement model identifies various components such as trusted servers, clients and administrators and specifies the interaction between them to realize the g-SIS policy model. We discuss three object distribution approaches in g-SIS: Super-Distribution (SD), Micro-Distribution (MD) and a hybrid approach that combines the advantages of SD and MD. We also identify and analyze the problem of "stale-safety" in g-SIS. "Stale-safety" is concerned with enforcing safe authorization policy given that authorization decision will inevitably be made based on stale authorization information in distributed systems such as g-SIS. We specify stale-safe security properties of varying strength and formally verify these properties against the SD based g-SIS enforcement model using model checking.

4.1 g-SIS Architecture

An important g-SIS objective that we are interested in is to enable offline access. We strongly believe that some degree of offline access is highly desirable in SIS where disconnected access attempts are likely. A motivation for offline access in the context of SD in P2P networks in mobile devices can be found in [41]. Offline periods of access can be restricted by various measures such as time, usage, etc. Thus, we expect that sometimes access decisions will be made locally without contacting a central authority. This requires that authorization information such as user attributes be stored locally and used in a trustworthy manner. Nevertheless, authorization information needs to be periodically refreshed with the central authority. In this section, we assume that the users interact with other entities using access machines running a Trusted Reference Monitor (TRM) whose software configuration

can be verified. Authorization information such as the group key(s) will only be available to the TRM in a trustworthy state. Thus the TRM can faithfully enforce group policies locally. Later, we discuss TPM-based protocols for interaction between various entities in the system.

4.1.1 System Characterization

The g-SIS system consists of users and objects, trusted access machines (using which users access group objects), a Group Administrator (GA) who is responsible for updating user and object attributes and a Control Center (CC) that maintains the attributes. An access control policy is specified using user and object attributes. A user's access machine has a Trusted Reference Monitor (TRM) that maintains a local copy of user attributes which are refreshed periodically with the CC so as to reflect changes, if any, in their values. There are many approaches to trigger a refresh of user attributes. For example, a refresh could be triggered based on time-out or a count on the number of times the user attributes (e.g. group key) may be used by the TRM to decrypt group objects. Offline access to secure clock is not feasible in TPMs today and so we take the usage count based approach for refresh (see for example [42, 43]). Object attributes are embedded in the object itself. An object removed from the group is listed in the Object Revocation List (ORL) which is provided to the TRM as part of refresh. A g-SIS system can be characterized as follows:

User attributes	$\{\text{id}, \text{Join\_TS}, \text{Leave\_TS}, \text{ORL}, \text{gKey}, \text{N}\}$
Object attributes	$\{\text{id}, \text{Add\_TS}\}$
Access Policy	$\text{Authz}(u, o, read) \rightarrow \text{Join\_TS}(u) \leq \text{Add\_TS}(o)$
	$\wedge\ \text{Leave\_TS}(u) = \text{NULL} \wedge\ o \notin \text{ORL}(u)$

User attribute Join_TS is join time of a user, Leave_TS is the leave time of user (if the user has left the group, NULL otherwise), ORL is the Object Revocation List that identifies

the list of objects that have been removed from the group along with their history of add and remove time-stamps, gKey represents the group keys (symmetric or asymmetric key pair) using which objects can be encrypted and decrypted and N is the usage count that limits usage of gKey before a refresh of user attributes is required with CC. Object attribute Add_TS is the time at which an object was added to the group. Attribute id represents a unique identity for each user and object.

Authz specified here is an attribute-based specification of the fixed operation (SJ, SL, LA, SR) model discussed in section 3.1.6. The access policy here specifies that a user is allowed to read an object as long as both the user and object are current members of the group and the object was added after the time at which the user joined the group. While any fixed operation model based on the π-system (see sections 3.1.5 and 3.1.6) can be specified and enforced by the TRM using these sets of attributes, we use the access policy specified here as a running example in this chapter.

4.1.2 System Architecture

Figure 4.1 shows the g-SIS architecture which illustrates the interaction between various components. The GA controls group membership and policies. The CC is responsible for maintaining authoritative attributes of group users and objects on behalf of the GA.

- *User Join (steps 1.1-1.4)*: Joining a group involves obtaining authorization from the GA followed by obtaining group credentials from the CC. In step 1.1, the TRM on user's access machine contacts the GA and requests authorization to join a group. The GA authorizes the user join in step 1.2 (by setting AUTH to TRUE). The user furnishes the authorization to join the group and the evidence that the access machine is in a good software state to the CC in step 1.3. The integrity evidence is a signed hash that proves that the chain of software loaded during the boot-up process including the system steady-state is trustworthy. The CC remotely verifies GA's authorization, that the user's access machine is trustworthy (using the integrity evidence) and has a known TRM that is responsible for enforcing g-SIS policies. In step 1.4, the CC

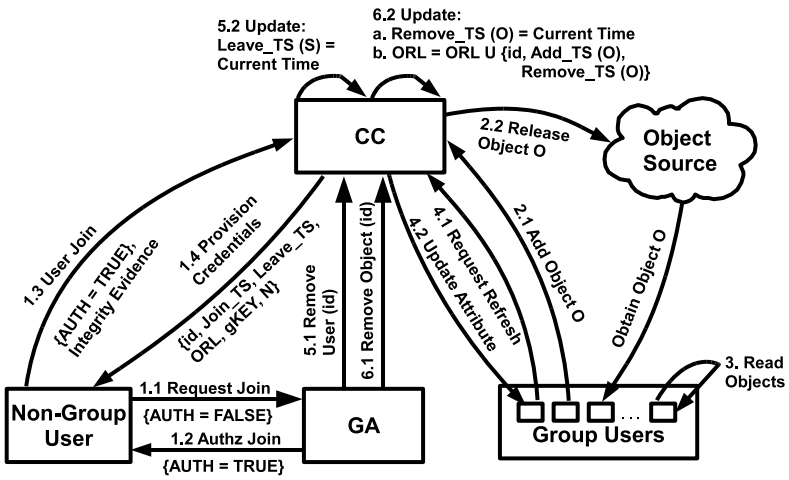

Figure 4.1: g-SIS Architecture.

provisions the attributes after setting them with appropriate values. We refer to these attributes as group credentials or simply credentials. Note that we assume that the credentials are provisioned such that only the TRM can access them.

- *Add Object (steps 2.1-2.2)*: From here on, the user is considered a group member. Objects may be added to the group by users after the CC sets the Add_TS (step 2.1). The CC verifies the object[1] and sets the Add_TS attribute. We assume object attributes are embedded in the object itself. The CC then releases the protected object to the Object Source (step 2.2). The Object Source acts as the object repository from which users may obtain objects. It is possible for the CC itself to act as the Object Source, but in practice, this may be an independent entity.

- *Read Objects (step 3)*: Users may read group objects as per the group policy using the credentials obtained from the CC. This is locally mediated and enforced by the TRM. Note that the objects may be obtained from Object Source and stored locally. Because of the presence of a TRM on user's access machines, these objects may be

[1]We treat this as an abstract step because verification of object depends on the specific application. In one case, this could simply be proof-reading while in another it could be a verification that the content is appropriate.

read offline conforming to the policy. Recall that various authorization policies are possible as discussed in section 4.1.1.

- *Attribute Refresh (steps 4.1-4.2)*: The TRM refreshes user attributes with the CC periodically. More generally, g-SIS policy, like the one discussed in section 4.1.1, may be updated or replaced in these steps. A usage count limits the number of times the credentials (e.g., gKey) may be used to read group objects (like consumable rights). Thus objects may be read until the usage count is exhausted and the TRM will be required to refresh user attributes in steps 4.1 and 4.2 before any further read access can be granted.

- *Administrative Actions (steps 5.1-5.2 and 6.1-6.2)*: The GA may have to remove a user or object from the group. In step 5.1, the GA instructs the CC to remove a user. The CC in turn marks the user for removal by locally setting the user's Leave_TS attribute in step 5.2. This attribute update is communicated to the user's TRM during the refresh steps 4.1 and 4.2. In the case of object remove, an Object Revocation List or ORL is provisioned by the CC on the user's access machine. Thus for object removal (steps 6.1-6.2), the object's id, Add_TS and Remove_TS are added to the ORL.

4.1.3 Super Vs Micro-Distribution in g-SIS

We now discuss SD and MD approaches for object distribution and access in the g-SIS architecture. In figure 4.1, steps 2.1 and 2.2 are object distribution steps and step 3 is the offline object read step. We compare and point out the pros and cons of each approach and present a superior hybrid approach in the following section. In this dissertation, *Super-Distribution* (SD) refers to widespread distribution of protected group objects while only authorized users may read them. Here all group users share a single group key using which objects are encrypted and decrypted. *Micro-Distribution* (MD) refers to custom encryption of objects for each authorized user. Figures 4.2 and 4.3 illustrate the difference between SD

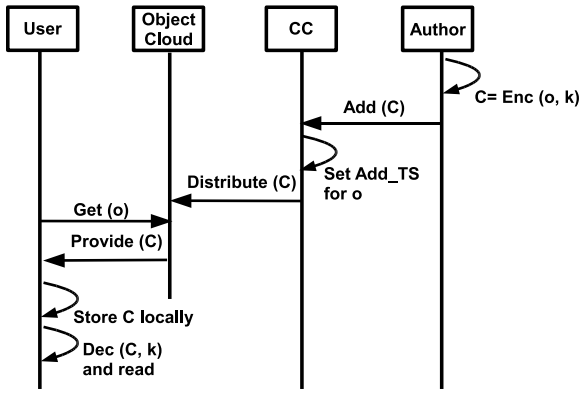

Figure 4.2: Super-distribution in g-SIS.

and MD in g-SIS. For simplicity, we assume that a symmetric key is used for encrypting and decrypting objects in both SD and MD. Our discussions below apply equally well to SD and MD approaches using asymmetric keys.

In SD based approach all group users share a single group key for encrypting and decrypting group objects. Thus in SD (figure 4.2), an Author (a group user) creates an object, encrypts the object using the group key (mediated by TRM) and sends it to the CC for approval and distribution. The CC verifies the object, time-stamps object add and releases this protected object into the infospace (referred to as Object Cloud) through conventional networks such as WWW, Email, etc. or sneakernet such as USB flash drives. These are steps 2.1 and 2.2 in figure 4.1 for SD approach. Other group users can obtain such encrypted objects and store them locally in their access machines for later offline access. Note that group objects need not be obtained from CC or a specific Object Source in SD. Since the group key is shared with all group users, the TRM can decrypt the object and display to the user offline without involving the CC every time.

In MD (figure 4.3), the CC shares a unique key with each group user. In contrast, in SD the same key is shared with all users in the group. The Author (a group user) creates an object, encrypts the object using the author's key shared with the CC (mediated by TRM) and sends it to the CC for approval and distribution. The CC approves the object, time-stamps object add and saves it locally. These are steps 2.1 and 2.2 in figure 4.1 for the MD

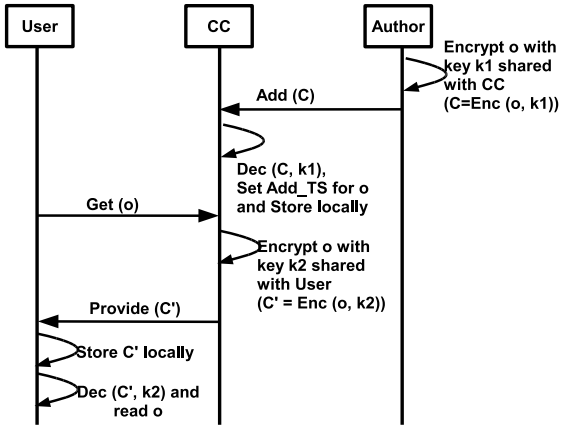

Figure 4.3: Micro-distribution in g-SIS.

based approach. When a User requests access to the object, the CC specifically encrypts the object with the key shared with that User. This is a critical difference from SD. Since objects need to be individually encrypted/prepared for each group user, the scalability of the system is gravely affected (here scalability refers to performance in the context of increasing number of objects shared amongst group users). Since a large number of objects may be shared within the group, the requirement that the CC decrypt and then custom encrypt each object for each group user affects the scalability of MD based enforcement model. In SD, the object is encrypted once and all authorized group users are able to access them without the intervention of the CC. We discuss how MD differs from SD in g-SIS.

- User Join: In SD, the CC provisions the group key for each joining user. In MD, the CC needs to create and share a new key with each joining user. This requires maintenance of a large number of keys.

- Object Add: In MD, the TRM would send the object to the CC. The CC needs to encrypt this object with every other user's key on demand so that it is accessible to them. This suffers from scalability and performance issues.

- Object Access: In MD, the first time a user needs to access an object, it needs to be obtained from the CC where it is custom encrypted for that user. It can thereafter

be accessed offline whenever authorized. Note that every group object needs to be initially obtained from CC in MD. In contrast, the objects could be obtained through any means in SD since all objects are encrypted with the group key.

- Attribute Refresh, User and Object Remove: There is no difference between SD and MD approach for g-SIS.

- Assurance and Recourse: In SD, if any one of the group user's access machine is compromised, the group key can be exposed and all group objects can be read in plaintext. A new group key can be provisioned after recovering from the compromise—although this can only guarantee secrecy of new objects that will be created. Due to this single point of failure problem, the sensitivity of information that be can be distributed using SD model may be limited. In MD, if any one access machine is compromised, only objects that were encrypted for the specific user using that access machine will be compromised. Other group objects remain safe because they were encrypted with different keys. In summary, in SD a compromise can result in large-scale access violation while in MD the violation is limited. This is the trade-off between scalability and assurance of each model.

Clearly, the SD based architecture has important scalability and performance benefits of simplified key management (single key per group) and the usability and convenience of offline access and "encrypt once and access where authorized" where group users can quickly share objects with other users. The MD based architecture on the other hand has important failsafe benefits such as limited damage in case of access machine compromise. Note that MD suffers from key management issues regardless of whether symmetric or asymmetric key cryptography is used. For the later, a public-private key pair needs to be shared by the CC with each group user. A more useful and practical architecture should combine the benefits of SD and MD based architecture (thereby minimizing the disadvantages of respective approaches). The hybrid architecture that we discuss in the following section attempts to achieve this by using split-key RSA.

Table 4.1: Comparison of SD, MD and Hybrid approach in g-SIS architecture.

Aspect	SD	MD	Hybrid
Key type	Symmetric/Asymmetric	Symmetric/Asymmetric	Asymmetric
Number of encryption keys per group	One	One per group user shared with CC	One
Number of decryption keys per group	One	One per group user shared with CC	One split variant for the same RSA exponent per group user
Add object	Encrypt with group key	Encrypt with key shared with CC	Encrypt with private key
Read object	Decrypt with group key and read (encrypt once and access where authorized). Offline access enabled throughout.	First time, CC custom encrypts the requested object for the user (encrypt differently for each user). Subsequent reads can be carried out offline.	First time, CC decrypts the object with its split decryption key. Subsequent reads can be carried out offline (encrypt once and access where authorized).
Usability (with respect to users)	Very high (offline access, no CC participation).	Medium (To add object, need to encrypt with the key shared with the CC. The CC in turn decrypts and custom encrypts for each user.).	High (Encryption is performed with a uniform encryption key).
Performance (with respect to CC)	Very high (CC never participates in encryption/decryption).	Medium (CC participates in decrypting and custom encrypting each object for each group user).	High (CC does not have to decrypt and custom encrypt the object like in MD. Instead, it performs a one time split key decryption operation per document).
Assurance	Low (compromising one user's access machine exposes group key thereby potentially exposing all group objects).	High (Only objects in the compromised access machine are exposed)	High (Only objects in the compromised access machine exposed).

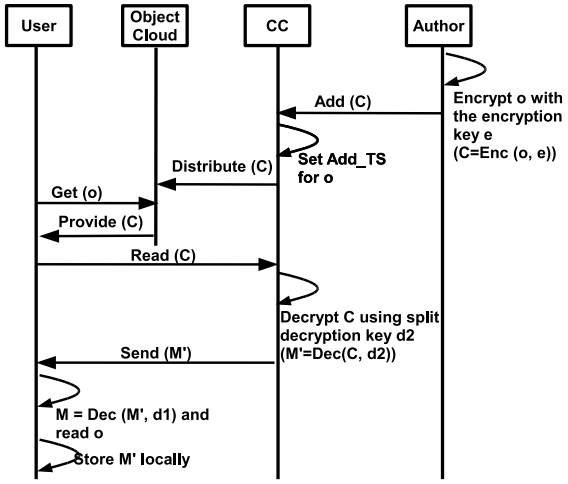

Figure 4.4: Hybrid approach in g-SIS.

4.1.4 Hybrid Approach Using Split-Key RSA

In this section, we present a hybrid approach that addresses the drawbacks of SD and MD based architecture thereby achieving the benefits of each approach.

Split-Key RSA

We provide a brief overview of split-key RSA in this section. Detailed discussions of the algorithm and proofs can be found in [44–47]. In split-key RSA, the decryption key is comprised of multiple parts each held by various parties (or users) involved in the decryption process. Thus if e and d denote encryption (public) and decryption (private) keys respectively, d can be split into n parts—$d1$, $d2$, ..., dn and shared with possibly n different parties. Thus a message encrypted with e can be decrypted only if all n parties participate.

Without the loss of generality, let us consider only two splits (and thus two parties).

$$e * d = 1 \; mod \; \varphi(n) \tag{4.1}$$

$$d1 * d2 = d \; mod \; \varphi(n) \tag{4.2}$$

$$C = M^e \; mod \; n \tag{4.3}$$

$$(M)^{d1^{d2}} \; mod \; n = \tag{4.4}$$

$$(M)^{d2^{d1}} \; mod \; n =$$

$$(M)^{d1*d2} \; mod \; n =$$

$$M^d \; mod \; n$$

In classical RSA [48], the encryption (e) and decryption (d) keys for a given n are related by equation (4.1). In split-key RSA with two splits, d can be split into two portions as guided by equation (4.2). A message M is encrypted using a single operation as shown in equation (4.3). Finally, the fundamental operation of exponentiation in RSA is given by equation (4.4). Thus decrypting a message M using the split keys $d1$ and $d2$ in any order in equivalent to decrypting the message using d. Also, note that d can be split into two parts in any number of ways, there by yielding pairs of different splits such as $d1$ and $d2$, $d3$ and $d4$, etc. Thus a message encrypted with e can be decrypted using $d1$ and $d2$ or $d3$ and $d4$, etc.

Hybrid Approach in g-SIS

Figure 4.4 illustrates the hybrid approach. Split-RSA keys are created for each group. When a user joins a group, the CC creates a unique split decryption key pair (d1 and d2 for this user), keeps one split (d2) and shares the other decryption split key (d1) with the joining user. The CC also shares the same encryption key (e) with every joining user. Thus,

in figure 4.4, the Author (a group user) adds an object by encrypting it with e and sending it to the CC. The CC approves the object, sets the add time-stamp and releases it into the object cloud similar to SD. The first time other group users need to access the object, they send a request to the CC. The CC performs the decryption on the object using its split decryption key d2 for that user and sends it back to the user. The user then decrypts this blob using his/her split-key d1 (mediated by TRM) to get the final plain-text object. The blob from CC can be stored locally and future read accesses can be performed completely offline. Note that, in practice, a symmetric object key would be used and the split-key decryption operation will be carried out on the object key instead of the entire object to minimize performance penalty of asymmetric key operation. Table 4.1 summarizes and compares g-SIS architectures based on SD, MD and hybrid approaches. Clearly, the hybrid approach for g-SIS architecture combines the advantages of both SD and MD approach and minimizes their disadvantages.

4.2 Stale-safe Security Properties

The concept of a stale-safe security property is based on the following intuition. In a distributed system authoritative information about user and object attributes used for access control is maintained at one or more secure authorization information points. Access control decisions are made by collecting relevant user and object attributes at one or more authorization decision points, and are enforced at one or more authorization enforcement points. Because of the physical distribution of authorization information, decision and enforcement points, and consequent inherent network latencies, it is inevitable that access control will be based on attributes values that are stale (i.e., not the latest and freshest values). In a highly connected high-speed network these latencies may be in milliseconds, so security issues arising out of use of stale attributes can be effectively ignored. In a practical real-world network however, these latencies will more typically be in the range of seconds, minutes and even days and weeks. For example, consider a virtual private overlay network

on the internet which may have intermittently disconnected components that remain disconnected for sizable time periods. In such cases, use of stale attributes for access control decisions is a real possibility and has security implications.

We believe that, in general, it is not practical to eliminate the use of stale attributes for access control decisions.[2] In a theoretical sense, some staleness is inherent in the intrinsic limit of network latencies, of the order of milliseconds in modern networks. We are more interested in situations where staleness is at a humanly meaningful scale, say minutes, hours or days. In principle, with some degree of clock synchronization amongst the authorization information, decision and enforcement points, it should be possible to determine and bound the staleness of attribute values and access control decisions. For example, a SAML (Security Assertion Markup Language) assertion produced by an authorization decision point includes a statement of timeliness, i.e., start time and duration for the validity of the assertion. It is upto the access enforcement point to decide whether or not to rely on this assertion or seek a more timely one. Likewise a signed attribute certificate will have an expiry time and an access decision point can decide whether or not to seek updated revocation status from an authorization information point.

Given that the use of stale attributes is inevitable, the question is how do we safely use stale attributes for access control decisions and enforcement? Our central contribution is to formalize this notion of "stale-safety" in the specific context of group-centric secure information sharing (g-SIS) as defined in this dissertation. We also demonstrate specifications of systems that do and do not satisfy this requirement. We believe this formalism can be extended to more general contexts beyond g-SIS, but this is beyond the current scope. We believe that the requirements for "stale safety" identified here represent fundamental security properties the need for which arises in virtually any secure distributed systems in which the management and representation of authorization state is decentralized in any degree.

[2]Staleness of attributes as known to the authoritative information points due to delays in entry of real-world data is beyond the scope of this dissertation. For example, if an employee is dismissed there may be a lag between the time that action takes effect and when it is recorded in cyberspace. The lag we are concerned with arises when the authoritative information point knows that the employee has been dismissed but at some other decision point the employee's status is still showing as active.

In this sense, we suggest that we have identified and formalized a *basic security property* of distributed enforcement mechanisms, in a similar sense that non-interference [49] and safety [50] are basic security properties that are desirable in a wide range of secure systems.

Specifically, we present formal specifications of two *stale-safe* properties, one strictly stronger than the other. The most basic and fundamental requirement we consider deals with ensuring that while authorization data cannot be propagated instantaneously throughout the system, in many applications it is necessary that a request should be granted only if it can be verified that it was authorized at some point in the recent past. The second, stronger property says that to be granted, the requested action must have been authorized at a point in time after the request and before the action is performed. We believe that the first property, *weak stale-safety*, is a requirement for most actions (*e.g.*, read or write) in distributed access control systems. We also believe that the second property, *strong stale-safety*, is (further) required of some or all actions in many applications (we will later discuss why strong property is desirable in system involving write actions).

We further show how these two properties can be strengthened to bound the acceptable level of staleness in terms of time elapsed between the point at which the request was last known to have been authorized and the point at which the action is performed. Thus, including these two strengthened versions, we have a total of four stale-safe properties.

4.2.1 Stale-safety

The sequence diagram in Figure 4.5 illustrates the staleness problem. The User and the TRM interacts with the GA and CC in steps 1 to 5 to join the group. The TRM refreshes attributes with the CC in steps 6 and 7. Shortly after the refresh, the GA removes this user (step 8) by setting his/her Leave_TS attribute at the CC (a non-null value). Note that this step is not visible to the TRM until the next refresh steps 11 and 12. In the mean time, the User may request access to objects the were obtained via super-distribution (step 9). At this point, the TRM evaluates the policy based on the attributes that it maintains. This will be successful by policy in section 4.1.1 and the object is displayed to the user in

step 10. Note the difference in Leave_TS values between the CC and TRM. Only after the following refresh (steps 11 and 12) does the TRM notice that the user has been removed from the group and denies any further access (steps 13 and 14).

Figure 4.6 shows a timeline of events involving a single group. User $u1$ joins the group and the attributes are refreshed with the CC periodically. RT represents the time at which refreshes happen. The time period between any two RT's is a Refresh Window, denoted RW_i. After join, RW_0 is the first window, RW_1 is the next and so on. Suppose RW_4 is the current Refresh Window. Objects $o1$ and $o2$ were added to the group by *some* group user (or the GA) during RW_2 and RW_4 respectively and they are available to $u1$ via super-distribution. In RW_4, $u1$ requests access to $o1$ and $o2$. An access decision will be made by the TRM in the access machine as per the attributes obtained at the latest RT.

Clearly, our example access policy discussed in section 4.1.1 will allow access to both $o1$ and $o2$. However it is possible that $u1$ was removed by the GA right after the last RT and before Request($u1$, $o1$, access) in RW_4 (see figure 4.5). Ideally, $u1$ should not be allowed to access either $o1$ or $o2$.

From a confidentiality perspective in information sharing, granting $u1$ access to $o1$ is relatively a lesser problem than granting access to $o2$. This is because the CC or the GA authorized $u1$ to read $o1$ in the past and hence assume that information has already been released to $u1$. In the worst case, $u1$ continues to access the same information ($o1$) until the next RT. However, $u1$ never had an authorization to access $o2$ and letting $u1$ access $o2$ means that $u1$ has gained knowledge of new information to which $u1$ was never authorized. This is a critical violation and should not be allowed. Such scenarios are what our stale-safe security properties address. A user cannot access an object if it was added to the group after the last refresh time even if the authorization policy allows access.

The property we discussed considers attributes to be stale if it is time-stamped later than the last refresh time-stamp of the access machine. A more strict property may require the access machine to refresh attributes before granting any access. That is, when $u1$ requests access to $o1$, the stricter version of the stale-safe property mandates that the access machine

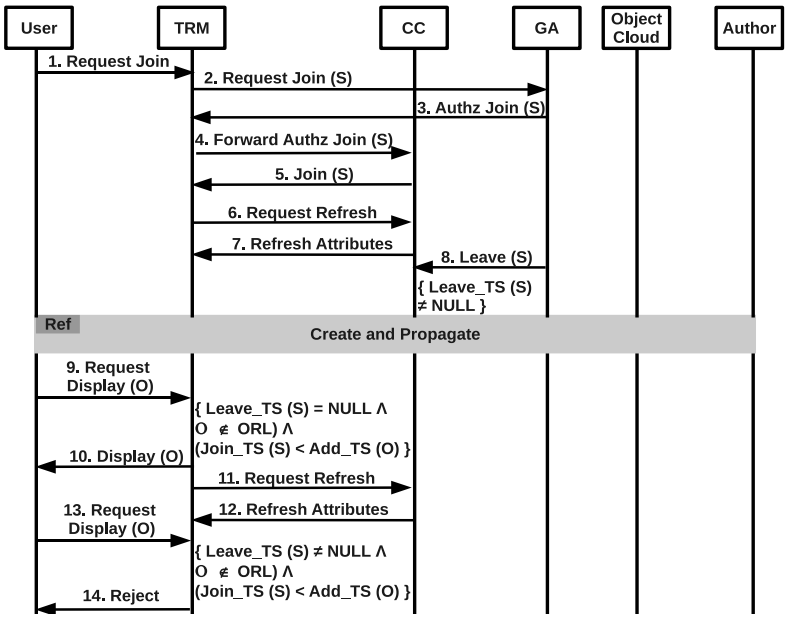

Figure 4.5: Staleness Illustration.

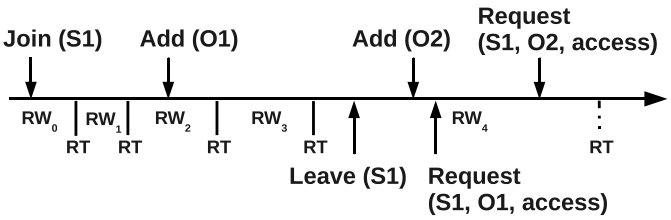

Figure 4.6: Events on a time line illustrating staleness leading to access violation.

refreshes the user attributes before making an authorization decision. Further, it is natural to consider elapsed time since the last refresh to be an important issue in limiting staleness of authorization data. We formalize these notions in the following subsection.

4.2.2 Formal Property Specification

In this section we use Linear Temporal Logic (LTL) [29] to specify four different formal stale-safety properties of varying strength. Our formalization uses the following predicates:

request (u,o,op)	u requests to perform an action op on o.
Authz (u,o,op)	u is authorized to perform an action op on o.
Join (u) and Leave (u)	Join & Leave events of u.
Add (o) and Remove (o)	Add & Remove events of o.
perform (u,o,op)	u performs op on o.
RT (u)	The TRM contacts the CC to update user attributes.

In the forthcoming formulae (φ_0, φ_1 and φ_2) and subsequent sections, we drop the corresponding parameters u, o and op in these predicates for convenience. They should however be interpreted with the respective semantics described above. Further, we assume that the very first join event of a user is equivalent to RT (since attributes are set at join time).

Access Policy Specification

We first formalize the access policy discussed in section 4.1.1 as an example. Note that, in distributed systems such as g-SIS, events such as Remove and Leave cannot be instantaneously observed by the TRM. Such information (that a user or an object is no longer a group member) can only be obtained from CC at subsequent refresh times (RT's). Thus, we have a notion of ideal or desirable policy that assumes instant propagation of authorization

information (like that of a centralized system). This is enforceable only at the CC. However, while designing the TRM (that is, in a distributed setting), one has to re-formulate this ideal policy using available authorization information so that it is enforceable locally by the TRM. We call the former Authz_{CC} and the latter $\text{Authz}_{\text{TRM}}$.

Authz_{CC} below is the same policy in section 4.1.1 represented using LTL. Figure 4.7 illustrates Authz_{CC}. Authz_{CC} says that u is allowed to perform an action op on o if the object was added to the group and both the user and object have not left the group since. Also, the user joined the group prior to the time at which the object was added to the group and has not left the group ever since. As the name implies Authz_{CC} can be enforced only by the CC and not the TRM. This is because the Leave and Remove events at CC are not visible to the TRM until the next refresh. Thus when a request is received, the TRM cannot ensure that no Leave occurred since Join or no Remove occurred since Add.

$$\text{Authz}_{\text{CC}} \equiv \ ((\neg\text{Remove} \wedge \neg\text{Leave})\ \mathcal{S}\ (\text{Add} \wedge (\neg\text{Leave}\ \mathcal{S}\ \text{Join})))$$

Let us now re-formulate Authz_{CC} so that it is enforceable locally at TRM. Recall from section 4.1 that once a user joins the group, authorization information such as object add time is available instantaneously to the TRM via super-distribution. Thus whether an object Add event occurred can be instantaneously verified by the TRM without contacting the CC. However, verification of whether a user join or leave event or an object remove event occurred can be ascertained only at refresh time (RT) with the CC. $\text{Authz}_{\text{TRM}}$ shows the re-formulation of Authz_{CC} as enforceable by the TRM. As shown, the occurrence of Join, Leave and Remove are ascertained at RT. However, Add is not subject to this constraint and its occurrence is ascertained freely.

$$\begin{aligned}\text{Authz}_{\text{TRM}} \equiv\ & (\neg\text{RT}\ \mathcal{S}\ (\text{Add} \wedge (\neg\text{RT}\ \mathcal{S}\ (\text{RT} \wedge (\neg\text{Leave}\ \mathcal{S}\ \text{Join}))))) \vee \\ & (\neg\text{RT}\ \mathcal{S}\ (\text{RT} \wedge ((\neg\text{Remove} \wedge \neg\text{Leave})\ \mathcal{S}\ \text{Add}) \wedge (\neg\text{Leave}\ \mathcal{S}\ \text{Join})))\end{aligned}$$

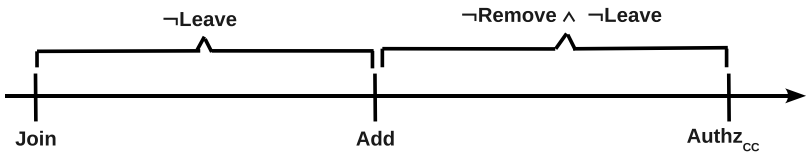

Figure 4.7: Ideal Access Policy (Authz_{CC}).

$\text{Authz}_{\text{TRM}}$ is a disjunction of two cases. The first part takes care of the case in which the requested object is added after the most recent RT. This is illustrated in case (a) of figure 4.8 where we are only able to verify that the user was still a member at RT. Since the object was not added prior to that point, we are unable to do a similar check for the object. The second part handles the situation where the object is added before the most recent RT. This is illustrated in case (b) of figure 4.8 where we are able to verify that at RT both the user and object are current members. Note that in both cases (a) and (b), our evaluation of policy is based on authorization information (except Add) available at RT.

$$\varphi_0 \equiv \ominus\ ((\neg\text{perform} \wedge (\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_{\text{CC}}))\ \mathcal{S}$$

$$(\text{Authz}_{\text{TRM}} \wedge (\neg\text{request} \wedge \neg\text{perform})\ \mathcal{S}\ \text{request}))$$

$$\varphi_0' \equiv \ominus\ ((\neg\text{perform} \wedge (\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_{\text{CC}}))\ \mathcal{S}\ (\text{request} \wedge \text{Authz}_{\text{CC}}))$$

Figure 4.9 is a pictorial representation of formula φ_0. It illustrates how a reference monitor reacts to a request to access an object. When a request arrives from a user, the TRM subsequently verifies if the policy ($\text{Authz}_{\text{TRM}}$) holds. If successful, the TRM allows the user to perform the requested action. Note that if an RT occurs in the meantime, the TRM re-evaluates the policy with the updated attributes. Thus, φ_0 says that the operation was authorized sometime between request and perform. Clearly, the formula $\Box$ (perform $\rightarrow$ φ_0) reflects this behavior of the TRM. In contrast, the formula $\Box$(perform $\rightarrow$ φ_0') is not enforceable as argued earlier since Authz_{CC} cannot be verified at request time. However, observe that verifying that $\text{Authz}_{\text{TRM}}$ holds at the time of request will allow the user access

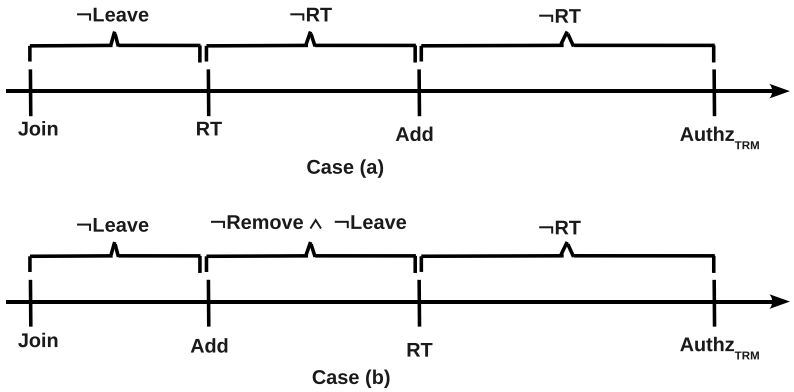

Figure 4.8: Approximate Access Policy ($\text{Authz}_{\text{TRM}}$).

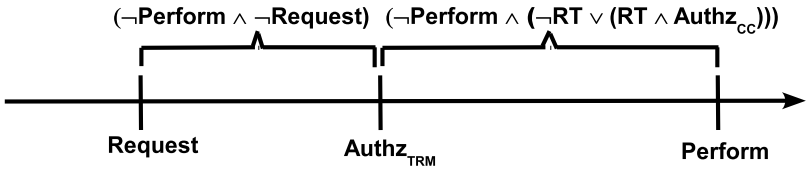

Figure 4.9: Formula φ_0.

to objects that were added during the time between RT and request$\wedge\text{Authz}_{\text{TRM}}$ in figure 4.9. We illustrated this in case (a) of figure 4.8. As discussed earlier, it is unsafe to let users access these objects before a refresh can confirm the validity of their group membership.

We next specify stale-safe security properties of varying strength. The weakest of the properties we specify requires that a requested action be performed only if a refresh of authorization information has shown that the action was authorized at that time. This refresh is permitted to have taken place either before or after the request was made. The last refresh must have indicated that the action was authorized and all refreshes performed since the request, if any, must also have indicated the action was authorized. This is the *weak stale-safe security property*. By contrast, the *strong stale-safe security property* requires that the confirmation of authorization occur after the request and before the action is performed.

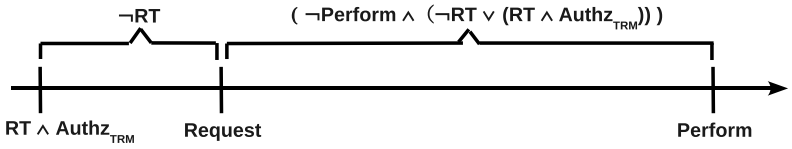

Figure 4.10: Formula φ_1.

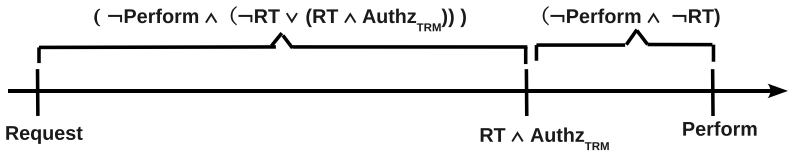

Figure 4.11: Formula φ_2.

Weak Stale-safe Security Property

Let us introduce two formulas formalizing pieces of stale-safe security properties. Intuitively, φ_1 can be satisfied only if authorization was confirmed prior to the request being made. On the other hand, φ_2 can be satisfied only if authorization was confirmed after the request.

$$\varphi_1 \equiv \ominus\,((\neg \text{perform} \wedge (\neg \text{RT} \vee (\text{RT} \wedge \text{Authz}_{\text{CC}})))$$

$$\mathcal{S}\ (\text{request} \wedge (\neg \text{RT}\ \mathcal{S}\ (\text{RT} \wedge \text{Authz}_{\text{CC}}))))$$

$$\varphi_2 \equiv \ominus\,((\neg \text{perform} \wedge \neg \text{RT})\ \mathcal{S}\ (\text{RT} \wedge \text{Authz}_{\text{CC}} \wedge$$

$$((\neg \text{perform} \wedge (\neg \text{RT} \vee (\text{RT} \wedge \text{Authz}_{\text{CC}})))\ \mathcal{S}\ \text{request})))$$

Figure 4.10 illustrates formula φ_1. φ_1 says that prior to the current state, the operation has not been performed since it was requested. Also since it was requested, any refreshes that may have occurred indicated that the operation was authorized ($\neg \text{RT} \vee (\text{RT} \wedge \text{Authz}_{\text{CC}})$). Finally, a refresh must have occurred prior to the request and the last time a refresh was performed prior to the request, the operation was authorized.

Observe that formula φ_1 mainly differs from φ_0 on the point at which Authz_{CC} is evaluated. Referring to figure 4.10, evaluating Authz_{CC} at the latest RT guarantees that requests to access any object that may be added during the following refresh window will be denied.

Note that φ_1 is satisfied if there is no refresh between the request and the perform. It requires that any refresh that happens to occur during that interval indicate that the action remains authorized. In our g-SIS application, this could preclude an action being performed, for instance, if the user leaves the group, a refresh occurs, indicating that the action is not authorized, the user rejoins the group, and another refresh indicates that the action is again authorized. For some applications, this might be considered unnecessarily strict.

Figure 4.11 illustrates formula φ_2. φ_2 does not require that there was a refresh prior to the request. Instead it requires that a refresh occurred between the request and now. It further requires that the operation has not been performed since it was requested and that every time a refresh has occurred since the request, the operation was authorized.

Note that φ_2 can be satisfied without an authorizing refresh having occurred prior to the request, whereas φ_1 cannot. Thus, though φ_2 ensures fresher information is used to make access decisions, it does not always logically entail φ_1 as it is many times satisfied by traces that do not satisfy φ_1.

Thus the formula perform $\rightarrow \varphi_1$ requires that a confirmation of authorization occur after the request has been received. The formula perform $\rightarrow \varphi_2$ requires that confirmation of authorization is obtained after the request, before the action is performed.

Definition 4.1 (Weak stale safety)**.** An enforcement model has the *weak stale-safe security property* if it satisfies the following LTL formula:

$$\tau_0 = \Box\ (\text{perform} \rightarrow (\varphi_1 \vee \varphi_2))$$

Strong Stale-safe Security Property

This property is strictly stronger than weak stale safety and unlike weak stale safety, it is a reasonable requirement for higher assurance systems.

Definition 4.2 (Strong stale safety)**.** An enforcement model has the *strong stale-safe security property* if it satisfies the following LTL formula:

$$\tau_1 = \Box\,(\text{perform} \rightarrow \varphi_2)$$

Note that the formulas (φ_0, φ_1 and φ_2) were concerned about the temporal placement of refresh time (RT) with respect to the time at which the request came from the user, the time at which the requested action is performed and the verification if authorization held. This separation of request and perform is important to differentiate weak-stale safety from strong-stale safety property. In weak-stale safety, although authorization held at RT prior to request, it is possible for an RT to occur between request and perform. If fresh attributes are available, it is important to re-check if authorization holds in light of this update. Formula φ_1 requires that authorization continue to hold at such occurrences. On the other hand, strong-stale safety mandates that after request, the action cannot be performed until authorization is verified with up-to-date attributes.

"Freshness" of Authorization

Let us now consider how to express requirements that bound acceptable elapsed time between the point at which attribute refresh occurs and the point at which a requested action is performed. We refer to this elapsed time as the degree of *freshness*. For this we introduce a sequence of propositions $\{P_i\}_{0\leq i\leq n}$ that model n time intervals. These propositions partition each trace into contiguous state subsequences that lie within a single time interval, with each proposition becoming true immediately when its predecessor becomes false. They can be axiomatized as $P_1\,\mathcal{U}\,(\Box\neg P_1 \wedge (P_2\,\mathcal{U}\,(\Box\neg P_2 \wedge (P_3\,\mathcal{U}\,(\ldots\,\mathcal{U}$

$(\Box\neg P_{n-1} \wedge \Box P_n)...)))))$. This partially defines correct behavior of a clock, given by a component of the FSM. Note that is it not possible to express in LTL that the clock transits from P_i to P_{i+1} at regular intervals of elapsed time. The current time can be interrogated by the other FSM components with which it is composed. It can also be referred to in the variant stale-safe properties presented in the following paragraphs. If the clock is accurate with respect to transiting from P_i to P_{i+1} at regular intervals, the enforcement machine obeying these variant properties will enforce freshness requirements correctly.

We now formulate variants of φ_1 and φ_2 that take a parameter k indexing the current time interval. These formulas use two constants, ℓ_1 and ℓ_2 which represent the number of time intervals since the authorization and the request, respectively, that is considered acceptable to elapse prior to performing the requested action. The formulas prohibit performing the action if either the authorization or the request occurred further in the past than permitted by these constants.

$$\varphi_1(k) \equiv \ominus\, ((\neg\text{perform} \wedge (\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_{\text{CC}}))) \,\mathcal{S}$$

$$(\text{request} \wedge \bigvee_{max(0,k-\ell_2)\leq i\leq k} P_i \wedge$$

$$(\neg\text{RT}\, \mathcal{S}\, (\text{RT} \wedge \text{Authz}_{\text{CC}} \wedge \bigvee_{max(0,k-\ell_1)\leq i\leq k} P_i))))$$

$$\varphi_2(k) \equiv \ominus\, (\neg\text{perform} \wedge \neg\text{RT})\, \mathcal{S}$$

$$(\text{RT} \wedge \text{Authz}_{\text{CC}} \wedge \bigvee_{max(0,k-\ell_1)\leq i\leq k} P_i \wedge$$

$$((\neg\text{perform} \wedge (\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_{\text{CC}}))) \,\mathcal{S}$$

$$(\text{request} \wedge \bigvee_{max(0,k-\ell_2)\leq i\leq k} P_i))$$

With these formulas, we are now able to state variants of weak and strong stale safety that require timeliness, as defined by the parameters ℓ_1 and ℓ_2.

Definition 4.3 (Timely, weak stale safety)**.** An FSM has the *timely, weak stale-safe security property* if it satisfies the following LTL formula:

$$\Box \Big(\bigwedge_{0 \leq k \leq n} (\text{perform} \wedge P_k) \rightarrow (\varphi_1(k) \vee \varphi_2(k)) \Big)$$

Definition 4.4 (Timely, strong stale safety)**.** An FSM has the *timely, strong stale-safe security property* if it satisfies the following LTL formula:

$$\Box \Big(\bigwedge_{0 \leq k \leq n} (\text{perform} \wedge P_k) \rightarrow \varphi_2(k) \Big)$$

4.2.3 Stale-safe Systems

We discuss the significance of the weak and strong stale-safe properties in the context of stale-safe systems designed for confidentiality or integrity. Confidentiality is concerned about information release while integrity is concerned about information modification. Both weak and strong properties are applicable to confidentiality –the main trade-off between weak and strong here is usability. Weak allows users to read objects when they are off-line while strong forces users to refresh attributes with the server before access can be granted. Depending on the security and functional requirements of the system under consideration, the designer has the flexibility to choose between weak and strong to achieve stale-safety. In the case of integrity, the weak property can be risky in many circumstances –the strong property is more desirable. This is because objects modified by unauthorized users may be used/consumed by other users before the modification can be undone by the server. For instance, in g-SIS, a malicious unauthorized user (i.e. a malicious user who has been revoked group membership but is still allowed to modify objects for a time period due to stale attributes) may inject bad code and share it with the group. Other unsuspecting

users who may have the privilege to execute this code may do so and cause significant damage. In another scenario, a malicious user may inject incorrect information into the group and other users may perform certain critical actions based on faulty information. Thus, although both weak and strong properties may be applicable to confidentiality and integrity, integrity systems satisfying only the weak property may have certain undesirable characteristics as discussed in this section.

4.3 Formal Verification of Stale-Safety

In this section, we use model checking to formally verify and build stale-safe systems. We first build a finite state machine hierarchy [51–53] to represent how the three components, CC, GA, and TRM, interact to enforce the g-SIS policy specified in section 4.1.1 (also see figure 4.1). We show that such a system is not stale-safe. Next, we show how to modify such a system to satisfy the weak or strong stale-safe security property.

4.3.1 Notations and Conventions

The enforcement model is composed of multiple hierarchical transition systems (HTS) via a collection of composition operators. An HTS is an extended state machine that consists of transitions and a hierarchical set of states. Formally, a hierarchical transition system (HTS) is an 8-tuple,$< S, H, I, F, E, V, V_I, T >$, where S is a finite set of states, H is the state hierarchy, and each state $s \in S$ is either a basic state or a super state that contains other states. Each super state must have a default state. $I \subseteq S$ is the non-empty set of initial states. $F \subseteq S$ is the set of final basic states. E is a finite set of events. V is a finite set of typed variables. $V - I$ is a predicate describing the initial values of variables. T is a finite set of transitions. A transition label can include a triggering event, a condition on the triggering event, and an action, which can be assignments to typed variables or event generation. The elements on the transition label are optional. A transition executes only if it is enabled (i.e., the HTS is currently in the source states, the positive triggering event

occurs and the negative event is absent, and the condition evaluates to true). If more than one transition is enabled, the transition whose source states have the lowest rank (i.e., are closest to the basic states) is chosen to execute. If multiple transitions, whose source states have the same rank, are enabled, the ones labeled with triggering events have priority. A state may have an *entry* transition, which executes whenever the state is entered and has the highest priority.

A specification of a system is a hierarchical composition of HTSs; and concurrency, synchronization, communication are introduced via composition operators, such as parallel, rendezvous, and interrupt. A component in the composition can be an HTS or a composed HTS (CHTS). The composition operators control when the CHTSs execute and how the CHTSs share data (e.g., generated events). In the parallel composition of multiple components, all the component execute together if they are enabled simultaneously. Otherwise, the enabled components execute in isolation. (NuSMVs synchronous composition matches this definition of parallel composition.) In binary rendezvous composition, exactly one transition in the sending component generates a rendezvous event that triggers exactly one transition in the receiving component, and both transitions execute together. If only one component is enabled by (or generates) a synchronization event in the same step, then the first component is forced to wait until the other component is ready. Otherwise, the behavior of the components is interleaved, executing transitions that do not involve rendezvous events. The interrupt composition allows control to pass between components via a set of interrupt transitions. These transitions may have sources and destinations that are sub-states of the components rather than the components' root states. In the interrupt composition, either the source, the destination, or an interrupt transition is chosen to execute. In the context of this paper, it is always the case that the interrupt transition has the lowest priority.

4.3.2 Modeling g-SIS

In specifying the g-SIS enforcement architecture in figure 4.1, we consider the interaction between the GA, CC and the TRM in the context of a single user and a single object.

To verify the stale-safe properties against the g-SIS model with any number of users and objects, it suffices to prove that the properties hold in the case of one user and one object. This is because, in g-SIS, authorization for a user on an object has no implication on the authorization for the same user on any other object. This is evident from the formulation of both Authz_{CC} and Authz_{E}. Similarly, a user's authorization has no implication on the authorizations of any other user. Henceforth, verifying stale-safety for one user-object pair for the entirety of both user and object membership life-cycle (i.e., join, leave and re-join for the user and add, remove and re-add for the object), proves that the properties hold for all possible pairs of users and objects.

In the enforcement model (Figure 4.12), the GA, CC, and TRM are specified as CHTSs. The GA is modeled as a parallel composition of two HTSs: GA_User models the behavior when a user joins and leaves a group, GA_object models the behavior when an object leaves the group. The GA_User and GA_Object HTSs keep track of the user and object membership states respectively. For simplicity, we omit the two step join process discussed in figure 4.1 and let the GA join a user to the group by sending a request directly to the CC. Furthermore, we assume that only the GA may add/remove the object to/from the group.

The CC is modeled as a CHTS containing a parallel composition of CC_User and CC_object, and an interrupt transition. CC_User models how to set the join or leave time stamps when a user joins or leaves a group. CC_Object models how to set the add or remove time stamps when an object is added or removed. The GA and CC are composed using the rendezvous synchronization operator. Consider the interaction between GA_User and CC_User machines in figure 4.12. In GA_User, when join_user occurs in userNonMember state, a req_join_TS event is generated and userMember state is entered. This event is captured by the CC_User machine and the Join_TS is set for the user. The time-stamp is obtained from the Clock[3]. Similarly, when leave_user occurs in the userMember state, a req_leave_TS event is generated and userNonMember state is entered. Note that the user's

[3]Since we need a finite model for model checking, in our code we actually limit the number of ticks to a finite value such as 10.

membership state is always synchronized between the GA and CC machines. As shown, the GA_Object machine handles object membership states and behaves similar to the GA_User machine. However, in the case of CC_Object, when req_add_TS is received in the objNon-Member state from GA_Object, Add_TS is set and objectMem state is entered. objectMem is a super-state that models super-distribution of the added object. In more complex scenarios, super-distribution may be modeled by a separate FSM but a super-state is sufficient for our purpose here. In this state, the superDistr event models the fact that the protected object is distributed into the cloud. Recall that the Add_TS is embedded in the object itself and thus the TRM can obtain it directly from the object when an access decision needs to be made. When req_remove_TS is received from GA_User, the Add_TS is set, the ORL is updated and CC_Object enters objectNonMem state.

The CC CHTS also responds to refresh event and send to TRM machine, TRM_0, up to date attributes (Join_TS, Leave_TS and ORL). Note that Add_TS is not updated as part of the refresh due to super-distribution. There could any number of copies of the object in the cloud. If an object needs to be re-added after being removed, a new Add_TS is set on another copy of the same object and super-distributed. When the user requests access to a copy of an object, the TRM can easily detect if the object has been removed (or re-added) by comparing the Add_TS embedded in the object and the latest (Add_TS, Remove_TS) specified in the ORL.

The user machine Group_User represents the interface to a user in the system. If a request to perform an action on an object is received (reqPerform), such as to read an object, Group_User sends a request to the TRM machine (TRM_0) and waits in the waiting state. TRM_0 decides whether the perform is allowed or denied and accordingly the user may or may not be allowed to perform the requested action.

The TRM machine, TRM_0, has three possible states. In the idle state, the machine receives request from Group_User. The TRM and Group_user CHTSs are composed using the rendezvous synchronization operator. The authorized state represents the fact that the requested action is authorized to be performed (i.e., $Authz_E$ is true). The refreshed state

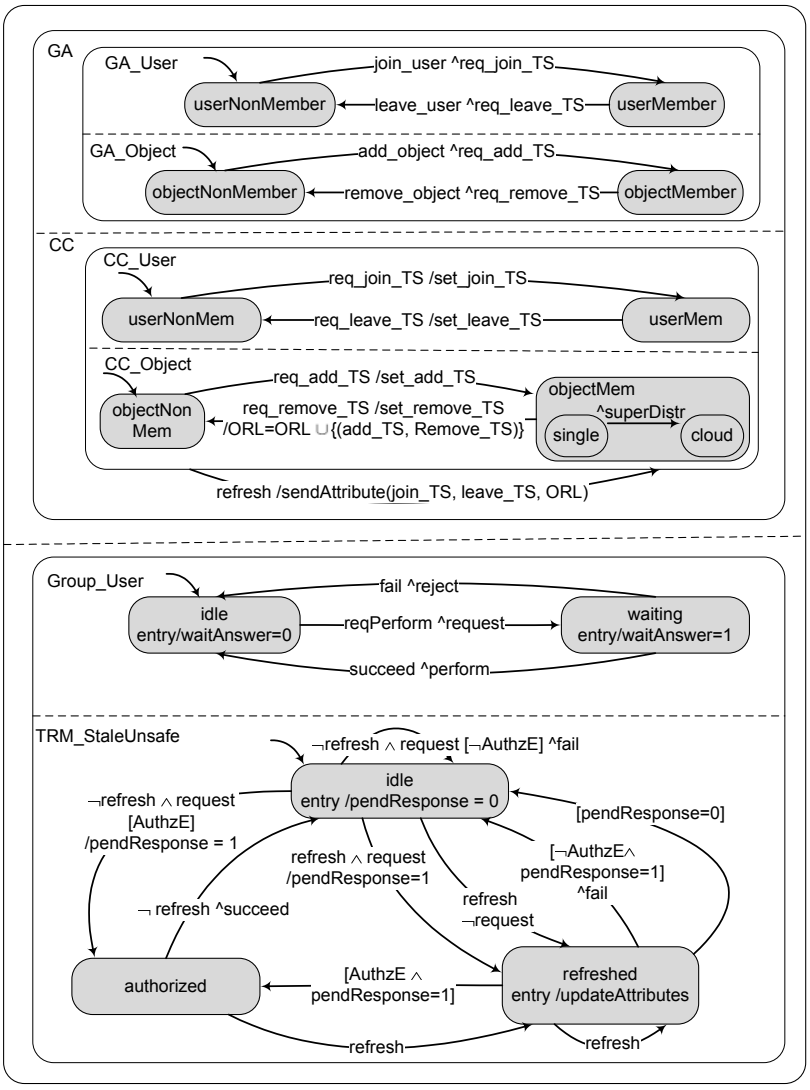

Figure 4.12: Independent FSMs modeling the g-SIS architecture. The TRM machine, TRM_0, is not stale-safe.

represents the fact that the TRM copy of attributes have been refreshed with CC. In the idle state in TRM_0, there are three possible scenarios: a request occurs, a refresh occurs or both request and refresh occur simultaneously. Whenever a refresh event occurs in the idle state, the machine enters the refreshed state and updates the attributes by obtaining them from CC. If a request is received and $Authz_E$ is not satisfied, a fail response is sent back to Group_User. But if $Authz_E$ holds, the machine enters the authorized state and generates a succeed event and transitions back to the idle state. As shown, the succeed event is captured by Group_User which subsequently allows the user to perform the requested action. However, if a refresh occurs in the authorized state, TRM_0 transitions to the refreshed state, updates the attributes and re-checks if $Authz_E$ holds in light of the updated attribute values. This ensures that if the machine is in authorized state and new attribute values are available, $Authz_E$ is re-evaluated based on the new attribute values. In the idle state, if both a request and refresh are received, the request is cached by TRM_0. It enters the refreshed state and updates the attributes. It then processes the cached request by checking if $Authz_E$ holds. If true, it enters the authorized state, otherwise it returns to the idle state directly by rejecting the requested action. Note that the variable pendResponse keeps track of whether a request has been processed or not.

We now prove a sequence of theorems. These theorems are concerned about various properties that we discussed in this section. We utilize the open-source model checker called NuSMV [54] to prove the theorems. We encode our models in the NuSMV language and specify the properties we need to verify in LTL. The model checker verifies if the LTL property is satisfied by the model. The proof for these theorems such as the counter examples generated by NuSMV is given in appendix C.2. Furthermore, due to space limitations, we provide the complete code and results generated by NuSMV in [55].

Definition 4.5 (Δ_0-system)**.** Let Δ_0 represent a g-SIS enforcement system, which is a composition of GA, CC, Group_User, and TRM_0, where GA and CC, and Group_User and TRM_0 rendezvous respectively, and then execute concurrently via parallel composition.

Theorem 4.1 (Unenforceablity Theorem)**.** The Δ_0-system does not enforce Authz_{CC}. That is:

$$\Delta_0 \nvDash \Box(\text{perform} \rightarrow \varphi_0')$$

This theorem states that Authz_{CC} is not enforced at TRM_0. This is self-evident because, as discussed earlier, certain events such as user leave and object remove are not immediately visible to TRM_0. Thus there will be instances in which TRM_0 will allow a user to perform even when Authz_{CC} is false. The model checker produces a counter-example as explained in appendix C.2.1.

Note that this does not prove that no system can enforce TRM_0. One can argue that it is possible to design alternative TRMs which may enforce Authz_{CC}. However, we believe that the Δ_0-system as characterized is a general representation of a typical distributed system where authoritative authorization information used for authorization decisions is not synchronized between the clients and server.

Theorem 4.2 (Enforceablity Theorem)**.** The Δ_0-system enforces $\text{Authz}_{\text{TRM}}$. That is:

$$\Delta_0 \vDash \Box(\text{perform} \rightarrow \varphi_0)$$

This theorem states that the Δ_0-system enforces $\text{Authz}_{\text{TRM}}$. Recall that $\Box(\text{perform} \rightarrow \varphi_0)$ ensures that the user can perform if $\text{Authz}_{\text{TRM}}$ is satisfied. This is successfully verified as shown in appendix C.2.1.

Theorem 4.3 (Weak Unsafe TRM Theorem)**.** The Δ_0-system does not satisfy the Weak Stale-Safety property. That is:

$$\Delta_0 \nvDash \Box(\text{perform} \rightarrow (\varphi_1 \vee \varphi_2))$$

This theorem states that the Δ_0-system is not stale-safe. Specifically, it fails the weak stale-safety security property. The counter-example generated by NuSMV is explained in appendix C.2.1.

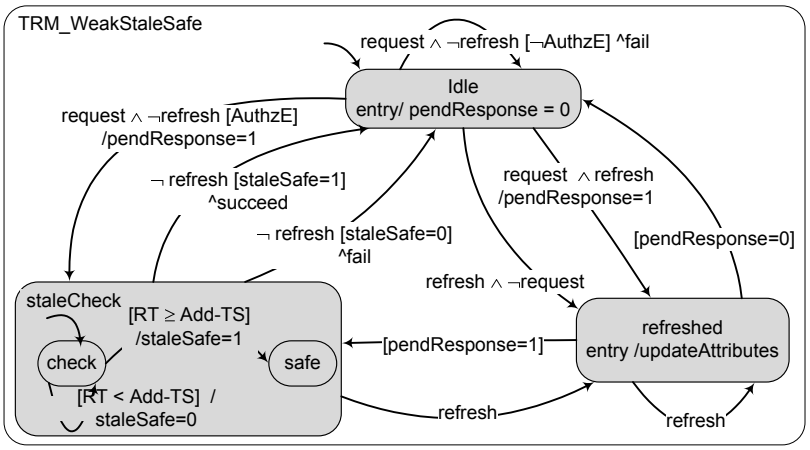

Figure 4.13: TRM_1: This TRM machine satisfies weak stale-safety.

Corollary 1 (Strong Unsafe TRM Theorem)**.** The Δ_0-system does not satisfy the Strong Stale-Safety property. That is:

$$\Delta_0 \not\models \Box(\text{perform} \rightarrow \varphi_2)$$

Evidently, if Δ_0-system fails the the weak property, it would also fail the strong property. The same counter-example generated by NuSMV for theorem 4.3 applies here.

4.3.3 Weak Stale-Safe TRM

Figure 4.13 shows one possible design for TRM that satisfies weak stale-safety. This TRM machine, TRM_1, has a super-state called authorizing that ensures that any authorization decision made is weak stale-safe. When the request is received, the machine enters the super-state and first checks if Authz_E holds based on local attributes of the TRM. Next, if Authz_E holds, it checks if the decision is weak stale-safe. This is achieved by verifying that the object that is being requested access to was added before the most recent refresh time—that is, RT $\geq$ Add_TS. Here RT is a variable that maintains the time of most recent refresh. This ensures that any object that is received via super-distribution and added after the TRM's last refresh time is unsafe to access. As illustrated in figure 4.6, the user could

potentially leave the group between the most recent RT and the object add time but the TRM is not aware of this until next refresh. This could result in a situation where a user is granted access to an object to which he/she was never authorized. We now prove that TRM_1 satisfies Weak Stale-Safety property.

Definition 4.6 (Δ_1-system)**.** Let Δ_1 represent a g-SIS enforcement system, which is a composition of GA, CC, Group_User, and TRM_1, where GA and CC, and Group_User and TRM_1 rendezvous respectively, and then execute concurrently via parallel composition.

Theorem 4.4 (Weak Stale-Safe TRM Theorem)**.** Δ_1 satisfies the Weak Stale-Safe Security Property.

$$\Delta_1 \vDash \Box(\text{perform} \rightarrow (\varphi_1 \vee \varphi_2))$$

NuSMV successfully verifies that the Δ_1-system satisfies the weak stale-safe security property as shown in appendix C.2.2. Obviously, the Δ_1-system does not satisfy the strong stale-safety property.

$$\Delta_1 \nvDash \Box(\text{perform} \rightarrow \varphi_2)$$

As expected, NuSMV generates a counter-example showing that the Δ_1-system does not satisfy strong stale-safety. The explanation of the generated counter-example is given in C.2.2.

4.3.4 Strong Stale-Safe TRM

Figure 4.14 shows a straight forward way to satisfy strong stale-safety. Note that TRM_2 refreshes the attributes with CC every time a request from the user is received. If after refresh Authz_E holds, the user is allowed to perform, else the request is rejected. Clearly, this should satisfy strong stale-safety because formula φ_2 requires that a refresh be performed after request before verifying if Authz_E holds, which is consistent with the model in figure 4.14.

Definition 4.7 (Δ_2-system)**.** Let Δ_2 represent the g-SIS system represented by the collection of FSMs: GA_User, GA_Object, CC_User, CC_Object, CC_Refresh, Group_User and TRM_2.

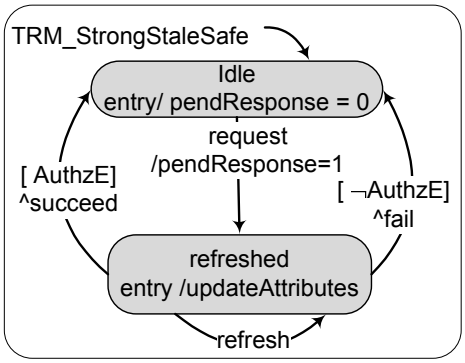

Figure 4.14: TRM_2: This TRM machine satisfies strong stale-safety.

Theorem 4.5 (Strong Stale-Safe TRM Theorem)**.** Δ_2 satisfies the Strong Stale-Safe Security Property.

$$\Delta_2 \vDash \Box(\text{perform} \rightarrow \varphi_2)$$

NuSMV successfully verifies that the Δ_2-system satisfies strong stale-safety as shown in the result it generated in appendix C.2.3.

Chapter 5: Implementation Model for g-SIS

We specify protocols for each step in the architecture in figure 4.1. We intentionally omit some system level details in these protocols for clarity. For example, we assume that the messages in the presented protocols all carry a MAC (Message Authentication Code) and are protected against replay appropriately using well-known techniques. The focus here is on how the TPM and TRM play a role in enforcing access policies offline by preventing or detecting tamper of group credentials by a malicious user. The TPM features a monotonic counter, a hardware counter, intended to reflect freshness of any value. For the purpose of this implementation model, we assume that a running version of the monotonic counter module presented in many approaches in the literature (see for example [42, 43]) is part of the TRM.

Mutual Authentication A mutual authentication protocol (such as Authenticated Diffie-Hellman [56]) is required for any two-party communication. In g-SIS, CC and GA are identified using certificates Cert_{CC} and Cert_{GA} respectively, issued by a trusted Certificate Authority (CA). For simplicity, we assume that a subject is tied to an access machine and hence is identified using the id of the TRM. A TRM is identified by an Attestation Identity Key (AIK) certificate. At the end of a mutual authentication protocol, the parties should have authenticated each other and share two session keys K_s and K_m used for encryption and MAC respectively.

Notations $X||Y$ refers to item X concatenated with item Y. Key operations are represented using an underscore and multiple items are enclosed within curly braces. Thus $\{X||Y\}\_\text{K}_\text{s}$ means that item $X||Y$ is encrypted using K_s. If AK is an asymmetric key, then

$\{X\}\_AK$ represents encryption of X using the public part of AK and $\{X\}\_Sign_{AK}$ represents a signature on X using the private part of AK. Finally, $\{X\}\_\mathrm{K_m}, \mathrm{K_s}$ means that item X has been MAC'ed and encrypted appropriately using keys $\mathrm{K_m}$ and $\mathrm{K_s}$ respectively. We use labels to refer to long cryptographic items for convenience. For example, $P : \{X||Y||Z\}$ and a subsequent usage of P such as $\{P\}\_\mathrm{K_s}$ denotes $\{X||Y||Z\}\_\mathrm{K_s}$. A Mutual Authentication (MA) protocol run between entities A and B is denoted MA(A:$\mathrm{id_A}$, B:$\mathrm{id_B}$) where $\mathrm{id_A}$ and $\mathrm{id_B}$ are the identities of A and B respectively. For example, MA(TRM:$\mathrm{AIK_{TRM}}$, GA:$\mathrm{Cert_{GA}}$) denotes a protocol between TRM and GA.

5.1 TPM Based Protocols

In this section, we present TPM protocols supporting SD and Hybrid object distribution approaches for the g-SIS enforcement model presented in section 4.1.

5.1.1 Protocols for SD Model

Join Protocol Figure 5.1 shows the protocol for a new user join (steps 1.1-1.4 in figure 4.1). In the authorization step, the GA verifies that the user is not a current member and returns a signature on $\mathrm{AIK_{TRM}}$. "JoinAUTH" is simply a label that communicates the semantics that the user with the id $\mathrm{AIK_{TRM}}$ is allowed to join the group. In the provisioning step, the TRM contacts the CC to obtain the group credentials. The TRM needs to attest its platform software and hardware state to the CC before the credentials can be provisioned. First, the TRM obtains the current virtual monotonic counter value from its counter module. The nonces used in the mutual authentication can be reused as a nonce for this operation ($g^x||g^y$ represents Diffie-Hellman style exponents used as nonces). Next, the TRM requests the TPM to create a non-migratable key-pair that will be owned by the TRM. The TPM, in response, returns $\mathrm{TRM_{pub}}, \{\mathrm{TRM_{priv}}||PCR\}\_SRK$. As discussed earlier, this message implies that the private part of the created key-pair, $\mathrm{TRM_{priv}}$, is sealed

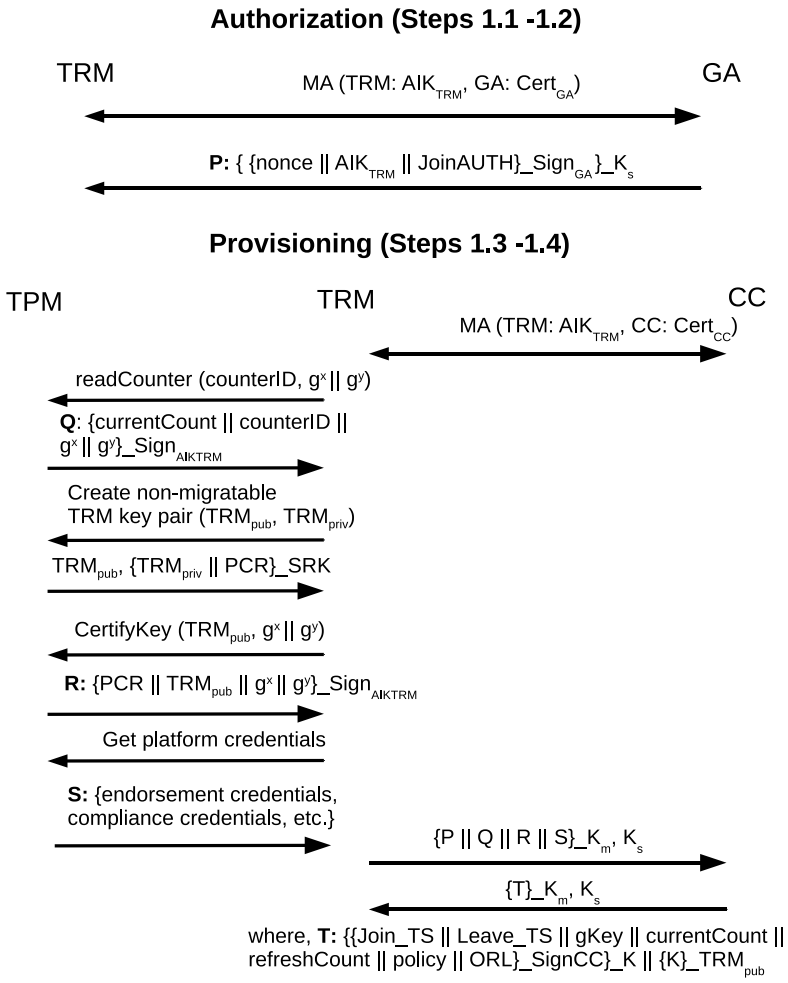

Figure 5.1: Join (steps 1.1-1.4 in figure 4.1).

to a software state of PCR. Thus $\mathrm{TRM_{priv}}$ can be unsealed in the future by the TRM only if the software state at unseal time matches the one specified in the PCR at seal time. If the seal-time PCR represents a trustworthy software state, $\mathrm{TRM_{priv}}$ will be accessible to TRM whenever the platform is in the same trustworthy state in the future.

These semantics can be communicated to a challenger (CC in this case) using the TPM's CertifyKey capability. The CertifyKey command takes the $\mathrm{TRM_{pub}}$ key and the private part's associated PCR and a nonce and signs them using the $\mathrm{AIK_{TRM}}$. Again, note that the nonce used here is the same as the ones used for mutual authentication which reflects the

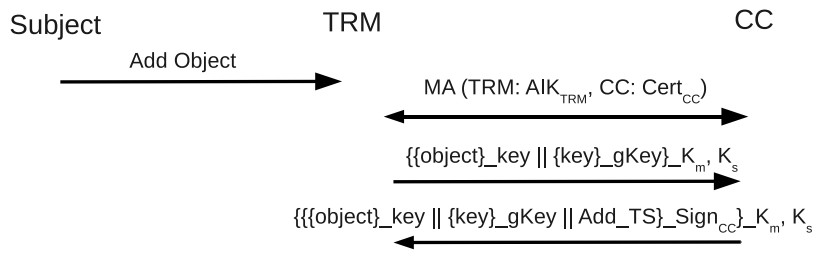

Figure 5.2: Add (steps 2.1-2.2 in figure 4.1).

freshness of the certified key blob. The TPM will sign TRM_{pub} using AIK_{TRM} (which is a key of type AIK) only if TRM_{pub} is non-migratable. That is, the TRM key-pair can never be accessed using any TPM other than the TPM that created the key-pair. Thus the CC can get the following assurance by looking at the certified key (message labeled as R). The private counter-part of TRM_{pub} can be accessed only if the software state of the platform is as represented by PCR. If the CC knows the hash-value of a well-known trusted platform state, it can verify this value against the reported PCR and decide to trust the TRM or not. The TRM further gathers the platform credentials (which reflects the trustworthiness of the hardware). Finally, the TRM sends the GA's join authorization (P), current virtual monotonic counter value (Q), the certified TRM key (R) and the platform credentials (S) to the CC. The CC verifies these values and returns the group credentials encrypted with a symmetric key K which in turn is encrypted with TRM_{pub}. As one can see, the group credentials (e.g. gKey) can be accessed only with TRM_{priv}. But TRM_{priv} is accessible to the TRM only if the platform is in the same software state as specified at seal-time and only in the same platform it was created in. If all is well, the TRM can access the group key and decrypt objects as per group policy.

Object Add and Offline Access Protocol The object to be added is first sent to the CC for approval (figure 5.2). The CC sets the add time-stamp, signs it and the object is ready for sharing with other users.

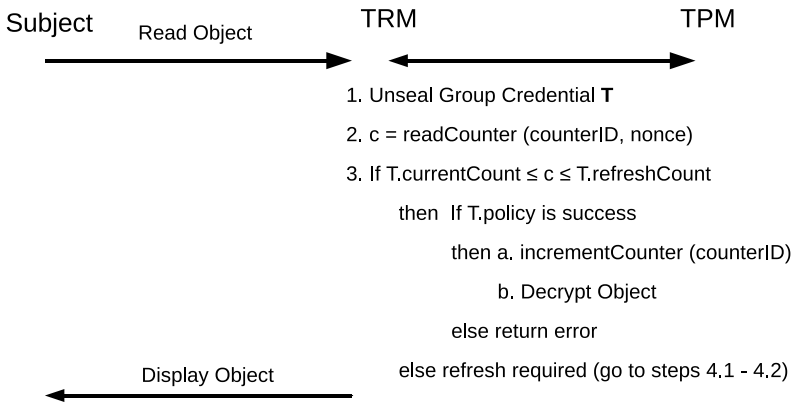

Figure 5.3: Object Read (step 3 in figure 4.1).

We assume that the objects that need to be read (step 3 in figure 4.1) are available locally in the user's access machine via super-distribution. When a user requests access to an object (figure 5.3), the TRM sends a request to unseal the group credentials to the TPM. Recall that the group credentials were sealed to a trusted platform state at join time. Hence the TPM will unseal it only if the current platform state is still trustworthy. The TRM then reads the current counter value and verifies that it is greater than or equal to currentCount specified in the unsealed group credential. This check prevents a replay of old credentials that could be launched by the user or other malware. Since the counter will be incremented on every use of the group credential and the counter being monotonic, the currentCount value in the older group credentials will be less than the value that was read. The TRM also checks that the current counter value is lesser than or equal to the refreshCount specified in the unsealed credential. This check verifies that the usage count on the group credential has not yet been exhausted. Thus if the policy allows access to the object (see section 4.1.1), the TRM increments the counter, decrypts the object and allows the user to read the object. Note that the user or malware can never make or hijack copies of plaintext object. The TRM allows the user to read the object only under a protected memory section that it controls. If the usage count is exhausted (i.e., refreshCount reached), the group credential needs to be refreshed before any further access will be allowed.

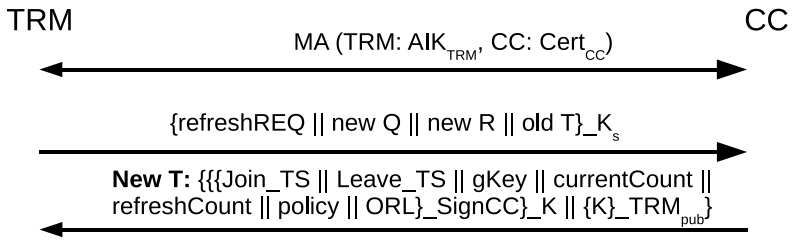

Figure 5.4: Refresh (Steps 4.1-4.2 in figure 4.1).

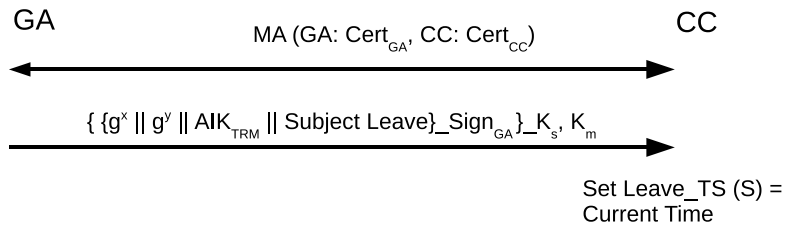

Figure 5.5: Leave (steps 5.1-5.2 in figure 4.1).

Refresh Protocol To refresh (figure 5.4), the TRM sends the current counter value (Q) and a fresh certified TRM key (R) along with refresh request (refreshREQ) to the CC (similar to join protocol). A new credential (T) is created with updated refreshCount. If the user or object were removed, corresponding attributes are updated in new T.

Leave and Remove Protocols In figure 5.5, GA instructs the CC to remove a user (AIK_{TRM}). The CC sets the user's Leave_TS to the current time that its maintains. Similarly, the ORL is updated in the case of object remove (figure 5.6). Note that the object id could be its hash value and the updates are propagated to the TRM during refresh.

5.1.2 Protocols for Hybrid Model

W now present the protocols that are different for the Hybrid model from that of the Super-Distribution based model. The protocols that are not discussed here (refresh, leave and remove protocols) are identical to that of those specified earlier for the SD model.

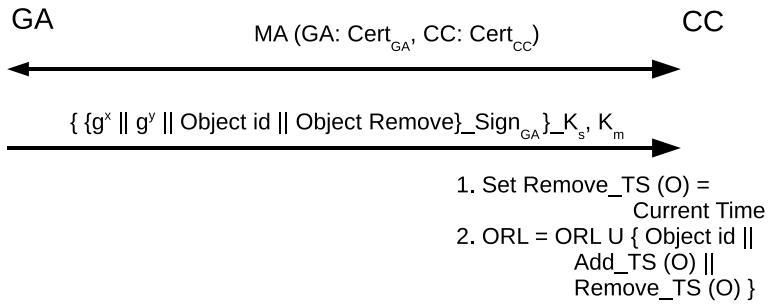

Figure 5.6: Remove (steps 6.1-6.2 in figure 4.1).

Join Protocol The join protocol for the hybrid model shown in figure 5.7 is almost identical to that of the join protocol in the SD model. The difference is in the last step of the protocol where the group key is provisioned. Here the CC generates a custom decryption split key for the TRM (gDecKeyUser) and encrypts it with other group credentials using the TRM_{pub} key. Note that the encryption key (gEncKey) remains the same for all group users. As one can see, the group credentials (e.g. gEncKey and gDecKeyUser) can be accessed only by using TRM_{priv}. But TRM_{priv} is accessible to the TRM only if the platform is in the same software state as specified at seal-time and only in the same platform it was created in. If all is well, the TRM can access the group credentials and access objects as per group policy.

Object Add and Offline Access Protocol The object to be added is first sent to the CC for approval (figure 5.8). The CC sets the add time-stamp, signs it and the object is ready for sharing with other users.

We assume that the objects that need to be read (step 3 in figure 4.1) are available locally in the user's access machine via super-distribution. In the hybrid approach, the first time the user requests to access an object (figure 5.9), the TRM requests the CC to do a split key decryption operation on the object. The CC checks if the user is authorized to read the object, fetches the split decryption key shared with the requesting user, performs a decryption operation using its split key and returns the blob to the TRM. The TRM can

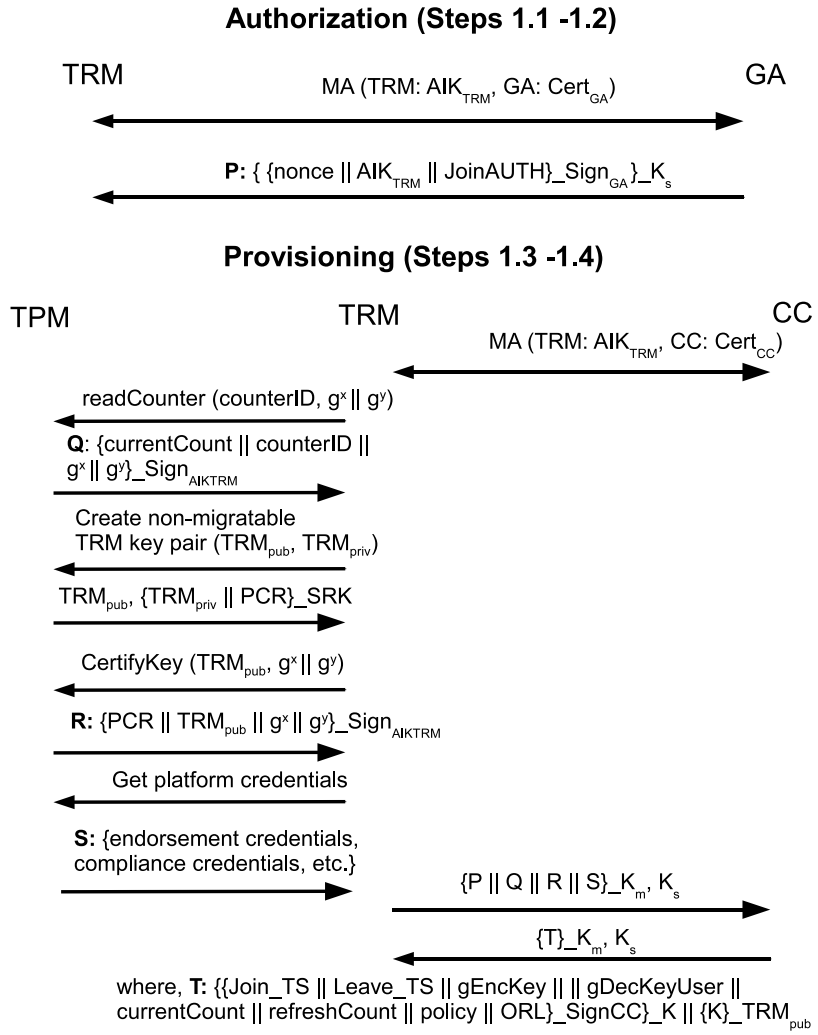

Figure 5.7: Join (steps 1.1-1.4 in figure 4.1).

then perform decryption using its portion of the split key and display the object to the user. The TRM may also encrypt the object using a symmetric object key and store locally for future accesses.

Subsequently, when a user requests access to an object (figure 5.10), the TRM sends a request to unseal the group credentials to the TPM. Recall that the group credentials were sealed to a trusted platform state at join time. Hence the TPM will unseal it only if the current platform state is still trustworthy. The TRM then reads the current counter value and verifies that it is greater than or equal to currentCount specified in the unsealed group

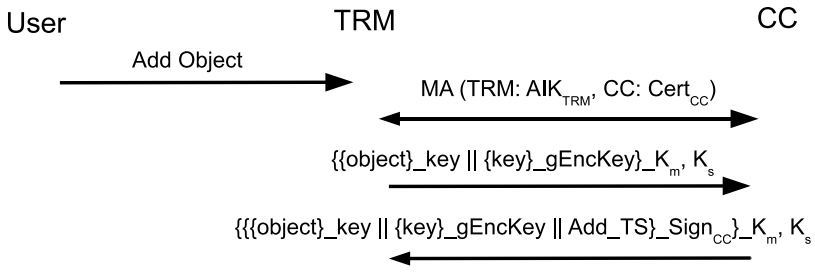

Figure 5.8: Add (steps 2.1-2.2 in figure 4.1).

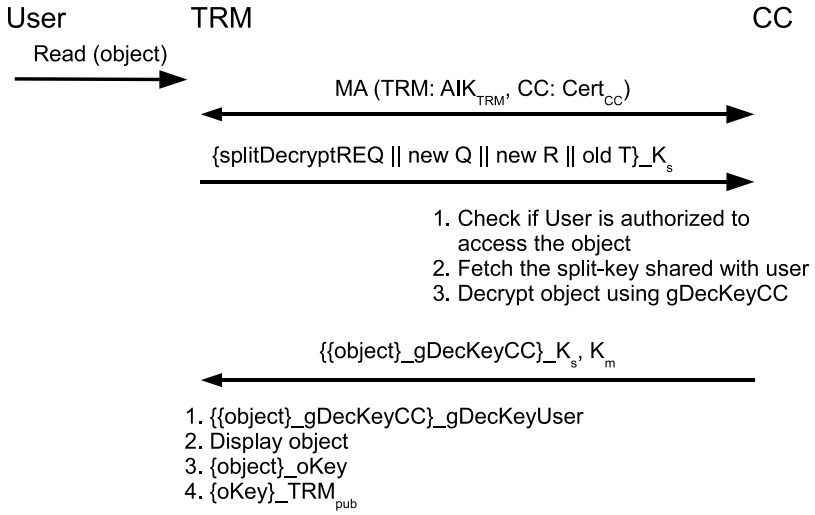

Figure 5.9: Object Read (first time).

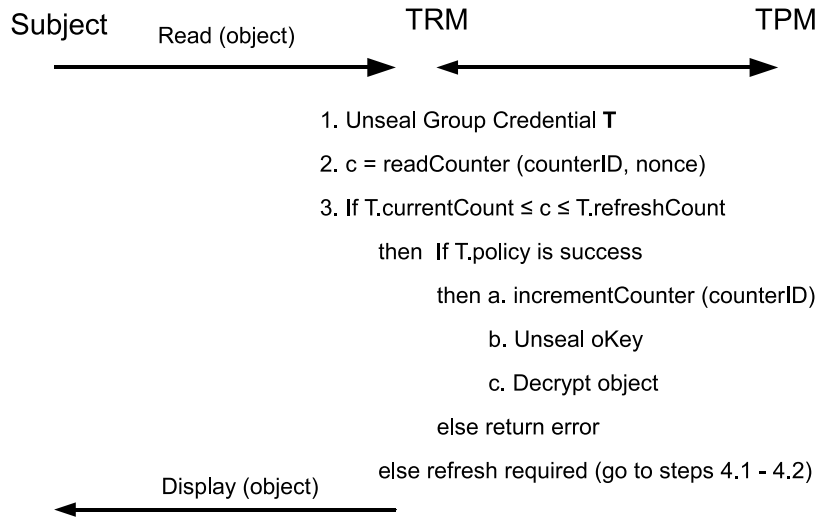

Figure 5.10: Object Read (subsequent accesses).

credential. This check prevents a replay of old credentials that could be launched by the user or other malware. Since the counter will be incremented on every use of the group credential and the counter being monotonic, the currentCount value in the older group credentials will be less than the value that was read. The TRM also checks that the current counter value is lesser than or equal to the refreshCount specified in the unsealed credential. This check verifies that the usage count on the group credential has not yet been exhausted. Thus if the policy allows access to the object (see section 4.1.1), the TRM increments the counter, decrypts the object and allows the user to read the object. Note that the user or malware can never make or hijack copies of plaintext object. The TRM allows the user to read the object only under a protected memory section that it controls. If the usage count is exhausted (i.e., refreshCount reached), the group credential needs to be refreshed before any further access will be allowed.

5.2 Implementation Model Overview

We now provide an overview of the g-SIS Implementation Model and a supporting access control framework.

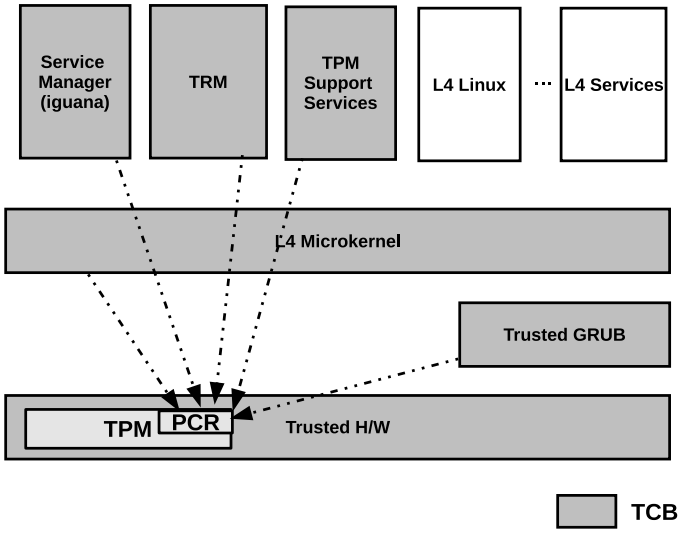

Figure 5.11: g-SIS implementation model.

5.2.1 g-SIS Trusted Execution Environment

The TRM requires a trusted execution environment to run and hence a Trusted Computing Base (TCB) needs to be established on the access machines. Many approaches have been discussed in the literature and figure 5.11 shows one such approach using the L4 microkernel [57]. L4 is a light-weight, message-based and secure microkernel. L4 ensures that each service running in its context is isolated from each other. A Service Manager (SM) is responsible for managing the L4 services. The TRM and TPM support services run as L4 services each with its own protected address space. The SM owns the TPM device and hence only it can access the TPM primitives. The grayed blocks form the TCB. We assume that the BIOS acts as the Core Root of Trust for Measurement (CRTM). The CRTM is an entity that can be trusted to measure the first piece of software in the boot chain. CRTM initially measures TrustedGRUB [43], an extension of the GRUB bootloader, that can measure the integrity of various files and store it in TPM's PCR during the boot process. After the boot process is complete, the TPM PCR value reflects the integrity of the entire system. This PCR value is reported to CC in order to attest to the platform's integrity.

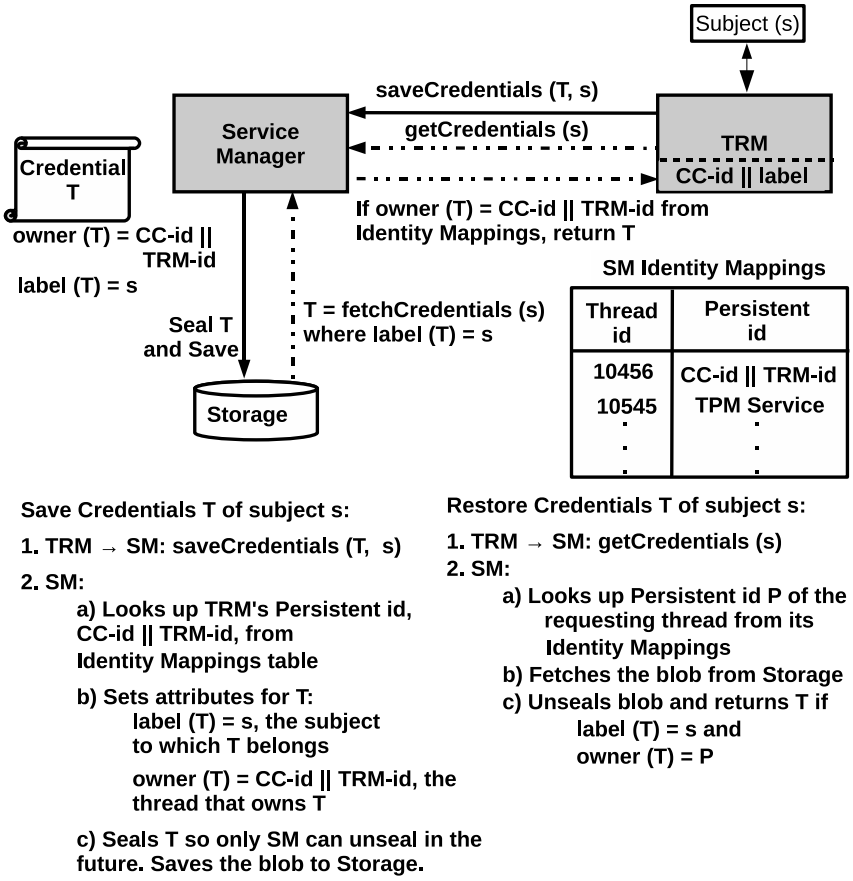

Figure 5.12: g-SIS Credentials Access Control.

5.2.2 Access Control For Group Credentials

Although the group credentials are sealed, the TPM cannot determine which software can unseal the credentials. It is critical that L4 ensures only the TRM and not any other service gets access to the group credentials like gKey. Our approach is to maintain a local identity infrastructure for L4 threads such as the TRM and ensure that the identity will be preserved across boot cycles. That is, the TRM should be uniquely identified across boot cycles.

Figure 5.12 shows the access control framework necessary for this purpose. When the TRM is initially provisioned, a persistent local id is stored as an attribute of the TRM's ELF (Executable and Linkable Format) file. At boot time, when SM loads TRM, it reads the persistent id from the ELF and maps it to the TRM's assigned thread id for that session.

The SM maintains such mappings for all threads in an Identity Mapping table. This allows SM to identify the TRM even if the thread id changes across boot cycles. Note that in figure 5.12 the persistent id includes the identity of the CC which issued the credentials.

When SM receives a request to save a user's credentials T from TRM (solid lines in figure), it uses the Identity Mappings table to look up the TRM's persistent id corresponding to the requesting thread id. Note that the saveCredentials method indicates the identity of the user (s) to which the credentials belongs—in the event multiple users use a single machine. Next, it sets two attributes of T for later retrieval. The 'label' attribute is set to the user to which the credentials belongs and the 'owner' attribute is set to the TRM's persistent id. SM then seals T so that only it can unseal T in the future and saves the sealed blob.

When SM receives a request to retrieve the credential of s from the TRM (dotted lines in figure), it looks up the requesting thread's persistent id from the Identity Mappings table. Next, it fetches the blob from storage that belongs to the user s using the 'label' attribute. It then unseals and verifies that the set 'owner' attribute matches the persistent id that it looked up earlier. The TRM can now use the credentials to serve user requests.

5.2.3 Proof-of-Concept

A complete implementation of the g-SIS model is beyond the scope of this dissertation. However, we discuss a simpler version of the protocol in figure 5.1 that was implemented as a Proof-of-Concept (PoC). This protocol was chosen because it is concerned about secure provisioning of group credentials on the group user's access machine which we believe is a critical part of the implementation model.

For this PoC, we assume that a Trusted Computing Base (TCB) as shown in figure 5.11 has been established. That is, the TRM is executing in a Trusted Execution Environment or TEE (which includes trusted hardware). The software state of the TEE is reflected by the PCR value in the TPM after boot time and this PCR value can be reported to remote entities which can verify that the TRM is executing in a trustworthy state.

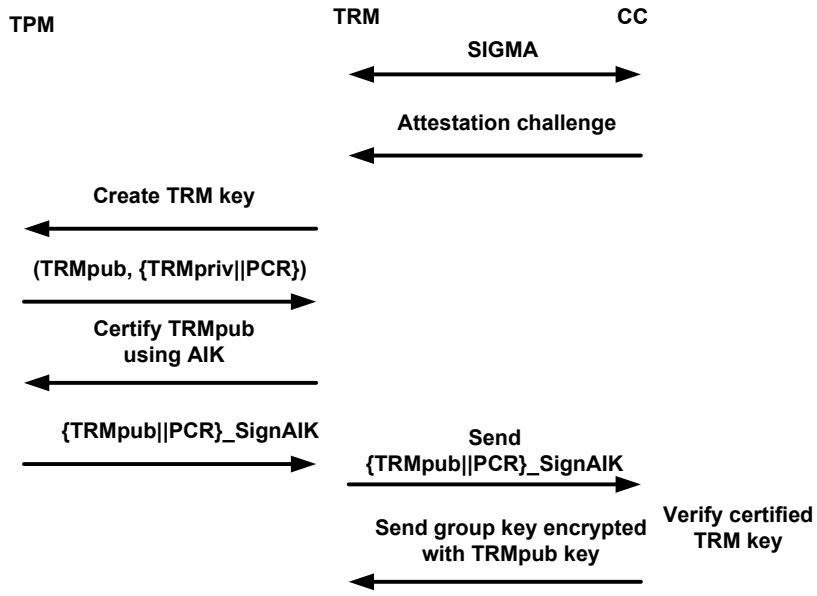

Figure 5.13: Proof-of-Concept: Provisioning protocol overview.

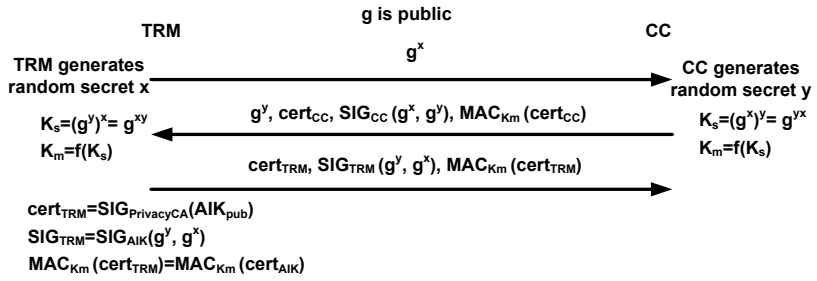

Figure 5.14: SIGMA protocol for authentication.

For this PoC, we focus on secure provisioning of the group key on the user's machine. Figure 5.13 illustrates the abstract steps involved in the PoC. This interaction is between CC and the TRM. The CC needs to remotely verify that the TRM is executing in a trustworthy state and provision the group key to the TRM in such a manner that the key is accessible to the TRM in the future only under the same trustworthy state in which it was initially provisioned.

The protocol begins with the SIGMA protocol [58] in which the CC authenticates the TRM. The SIGMA protocol is illustrated in figure 5.14. The TRM uses the AIK certificate

issued by a Privacy CA to authenticate itself to the CC. The TRM and CC each create their own random secret and arrive at a session and mac key. Note that the certificate used by the TRM is the AIK certificate issued by a Privacy CA. The AIK certificate serves to identify the machine in which the TRM is executing. The Trusted Computing Group [36] has a well-defined specification for the Privacy CA to follow in order to issue the AIK identity certificate to any TPM enabled machine. Subsequently, he CC sends the attestation challenge asking the TRM to prove that it is running in a TEE. The TRM requests the TPM to create a non-migratable key that is bound to the hash value in the TPM's PCR which reflects the trustworthiness of its platform. The TPM creates a key pair and binds the private part to the platform PCR. This in effect means that, in the future, if any data is encrypted with the public key, it can be decrypted only when the platform is in the same trustworthy state as reflected by the current PCR value. This is because the private part of the TRM key will be "unwrapped" by the TPM only if the current value in its PCR matches the one specified at the create time of this key pair. The following data structure records the attributes of the TRM key created by the TPM.

```
typedef struct tdTPM_KEY{
    TPM_STRUCT_VER ver;
    TPM_KEY_USAGE keyUsage;
    TPM_KEY_FLAGS keyFlags;
    TPM_AUTH_DATA_USAGE authDataUsage;
    TPM_KEY_PARMS algorithmParms;
    UINT32 PCRInfoSize;
    BYTE* PCRInfo;
    TPM_STORE_PUBKEY pubKey;
    UINT32 encDataSize;
    [size_is(encDataSize)] BYTE* encData;
} TPM_KEY;
```

The keyFlags member of the structure specifies whether the key as indicated by the pubKey member of the structure is migratable or not. The PCRInfo structure specifies the set of PCR numbers selected (pcrSelection) and the required composite hash values of the selected PCRs in order to use the private part of the key. This is indicated by the digestAtRelease structure member. encData points to a data structure that stores the confidential attributes of the key including the encrypted private part of the TRM key.

```
typedef struct tdTPM_PCR_INFO{
    TPM_PCR_SELECTION pcrSelection;
    TPM_COMPOSITE_HASH digestAtRelease;
    TPM_COMPOSITE_HASH digestAtCreation;
} TPM_PCR_INFO;
```

As mentioned earlier in the join protocol in section 5.1.1, the TPM provides a "Tspi_Key_Certify_Key" protected capability that can be used to communicate to any remote entity that the key pair was created and can be used only under such a trusted platform state. Tspi_Key_Certify_Key uses a certifying key to certify the TRM public key. If AIK is used as the certifying key in Tspi_Key_Certify_Key, the TPM additionally requires that the TRM key be non-migratable. This means that the TRM key can never be used in any other platform other than the platform in which it was created. The Tspi_Key_Certify_Key capability uses the AIK to sign the following TPM_CERTIFY_INFO data structure and returns it to the TRM.

```
typedef struct tdTPM_CERTIFY_INFO{
    TPM_STRUCT_VER version;
    TPM_KEY_USAGE keyUsage;
    TPM_KEY_FLAGS keyFlags;
    TPM_AUTH_DATA_USAGE authDataUsage;
    TPM_KEY_PARMS algorithmParms;
    TPM_DIGEST pubkeyDigest;
    TPM_NONCE data;
```

```
    BOOL parentPCRStatus;
    UINT32 PCRInfoSize;
    [size_is(pcrInfoSize)] BYTE* PCRInfo;
} TPM_CERTIFY_INFO;
```

The pubkeyDigest specifies the digest of the TRM public key that is bound to the PCR digest value specified by the PCRInfo structure. This indicates that the TRM private key can be used only if the PCR value at that time matches the value pointed by PCRInfo. Since the AIK is used to sign TPM_CERTIFY_INFO, a remote entity such as the CC can ascertain that the TRM key is not migratable. This is an in-built behavior of the Tspi_Key_Certify_Key capability as enforced by the TRM. This can further be confirmed by examining the keyFlags value and ensuring that it says the key is non-migratable.

At this point, the signed TPM_CERTIFY_INFO structure is sent to the CC. The CC verifies that the structure was signed using the platform AIK certified by a known Privacy CA. This ensures that the platform (hardware) is in a trustworthy state. The CC then examines the structure and ensures that the structure members hold as per the expected and pre-determined values. Subsequently, the CC encrypts the g-SIS group key using the TRM public key and sends the blob to the TRM. The TRM can decrypt the group key only if the platform and the software are in the same state as indicated by PCRInfo. The fact that the PCR values are as expected implies that *only* the TRM can access the group key and that the TRM's execution environment is isolated from other programs executing in the system.

Code Design

The PoC was implemented in a system with TPM v1.2 running Fedora Core 8 with Linux Kernel 2.6.26.8-57.fc8. This kernel is pre-complied with the TPM driver. The TrouSerS 0.3.2-1 software stack [59] was used to access various TPM capabilities. TrouSerS is a CPL (Common Public License) licensed Trusted Computing Software Stack primarily implemented and maintained by IBM. It provides high-level APIs to various underlying TPM

functionalities exposed by the TPM via the driver. Due to space limitations, we provide the complete code in [55].

TRM Design: Table 5.1 shows a high-level design of TRM code which is composed of 3 C program files: trm.c, trm_helper.c and quote.c. In trm.c, the sendFile, recvFile modules help to send and receive any arbitrary information to and from the CC respectively. For example, the certified TRM key can be sent to the CC using sendFile and the group key can be received from the CC using recvFile. Similarly, the sendString and recvString modules are used to send and receive small amount of data in terms of strings to and from the CC respectively. The main module in trm.c consists of 4 major steps. Steps 1 through 3 are used for mutual authentication between the TRM and CC using the SIGMA protocol. At the end of step 3, the TRM and CC share a secret session key and a mac key (for verifying integrity of messages exchanged subsequently).

Step 4 is a critical module that implements the remaining protocol runs after SIGMA as illustrated in figure 5.13. As shown in table 5.1, the step4 module creates a non-migratable TRM key pair, uses the AIK to certify the TRM key, sends the certified TRM key and the key's attributes to the CC, receives the encrypted group key from CC and successfully decrypts the group key as long as the platform remains in the trusted state. The createKey module takes two parameters that specifies the id with which the created TRM key will be registered in the user's machine. The id is required in order to use the TRM key in the future. The certifyKey module takes a nonce that will be included in the certified key information in order to prevent replays in the future. For simplicity, the exponents used in SIGMA is re-used as a nonce. This also has the benefit of tying the entire protocol to a single session. For convenience, the attributes of the certified TRM key are extracted and sent separately from the certified key to the CC. Thus two files are sent. The first one contains the certified TRM key. The second file contains the properties of the certified TRM key.

Table 5.1: Summary of TRM design

File	Summary
trm.c	int sendFile(char *filePath, int clientSocket) int recvFile(char *fpath,unsigned char *fdata, FILE *fp) int sendstring(char *string,int size,int sockfd)) int recvstring(char *string,FILE *fp) int step4(int clientSocket, FILE* fp, char* key, char* gxgy, char* id1, char* id2) Create TRM key (createKey(char*, char*)) Certify TRM key using AIK (certifyKey(char* gxgy)) Send certified key to CC Send certified key attributes to CC Receive group key Decrypt group key (unBind (int)) int main(int argc, char *argv[]) Steps 1-3: SIGMA protocol, establish session and mac key Step 4: Receive attestation challenge step4(clientSocket,fp, key, gxgy, argv[3], argv[4])
trm_helper.c	TSS_RESULT createKey(char* id1, char* id2) TSS_RESULT certifyKey(char* gxgy) TSS_RESULT unBind(int size)
quote.c	int _quote_Main(char* nonce_, int nonce_length, unsigned char* retData) Used to get the TPM to sign protocol nonce gx||gy using AIK.
trm_helper.h, quote.h	TRM configuration files

Following this, the TRM receives the group key from the CC that is encrypted with the TRM key. The TRM uses the unBind module to successfully decrypt the group key. Note that this will succeed only if the TPM's PCR value (reflecting trustworthiness of the TRM platform) matches the one specified at the creation time of the TRM key. As shown, the trm_helper.c is the main file that implements all the TPM code. The _quote_main module in quote.c implements anti-replay in the SIGMA protocol by getting the TPM to sign the nonce using the AIK. Finally, the header file trm_helper.h and quote.h are used to configure the TRM. For example, we specify the key ids of the storage root key of the TPM and TRM key, the PCR number(s) to which the TRM key should be bounded at creation time, etc.

CC Design: The CC is primarily implemented in three C program files: cc.c, cc_helper.c and quote_verify.c. The code design for CC is illustrated in table 5.2. Like the TRM, the CC implements the sendFile, recvFile, sendString and recvString modules to send and receives files and strings to and from the TRM respectively. For instance, the CC uses the recvFile module to receive the attestation response from the TRM and the sendFile module to send the group key to the TRM.

The main CC module implements the networking components using C sockets. After initiation, the CC waits for connection requests from clients such as the TRM. On receipt, the CC engages in a SIGMA mutual authentication with the TRM. It receives the TRM's AIK certificate issued by a Privacy CA. For the purpose of this implementation, we simulate the privacy CA and identity creation. An AIK is created locally in the TRM's machine and certified using the privacy CA's private key. We assume that the public of the privacy CA is well-known. The CC uses the privacy CA's public key certificate to verify the AIK certificate and thereby authenticates the TRM. This steps confirms that the CC trusts the privacy CA to have verified the platform credentials of the TRM's machines (that is the TPM was created by a trusted manufacturer and the platform hardware is in a trustworthy state). Steps 1 to 3 in the main module implements the SIGMA protocol. Thus at the end of step 3, the CC and TRM share a session and mac key.

Table 5.2: Summary of CC design

File	Summary
cc.c	int sendFile(char *filePath,int slaveSocket) int recvFile(char *fpath,unsigned char *fdata, FILE *fp) int sendstring(char *string,int size,int sockfd)) int recvstring(char *string,FILE *fp) TSS_RESULT step4(int slaveSocket, FILE *fp, char* key, char* gxgy) Send attestation challenge to TRM Receive attestation response from TRM: certified key and certified key attributes Verify integrity of received info Verify attestation info Send group key int main(int argc, char *argv[]) Wait for connection request from TRM Steps 1-3: SIGMA protocol, establish session and mac key Step 4: step4(slaveSocket, fp, key, buffer)
cc_helper.c	int verifyAttestaion(int vd_size, char* gxgy) Verify signature on certified TRM key (verifyCertifiedKey(vd, vd_size)) Verify properties of certified TRM key (verifyCertifyInfo(tci, vd, gxgy)) Encrypt Group key (Bind()) TSS_RESULT verifyCertifiedKey(BYTE* vd, int vd_size) TSS_RESULT verifyCertifyInfo(TCPA_CERTIFY_INFO* tci, BYTE* vd, char* gxgy) Verify pub key digest (verifyPubKeyDigest(tci-¿pubkeyDigest.digest)) Verify PCR info of TRM key (verify if PCR value is trustworthy) Verify if TRM is non-migratable TSS_RESULT verifyPubKeyDigest(BYTE* pubKeyDigest)
quote_verify.c	TSS_RESULT quote_verify(unsigned char* nonce_, int nonce_length, char* gxgy, int gxgyLength, char* fPath, int fsize) Verify the signatures are fresh using gx||gy as the nonce.
cc_helper.h, quote_verify.h	CC configuration files

The step4 module in main is the critical module that sends attestation challenge, receives the attestation response from TRM, verifies the integrity of all the received messages, verifies the contents of the received attestation and sends the group key if all of these steps succeed. As shown in table 5.2, the cc_helper.c file implements all these modules. After sending the attestation challenge and receiving the response from the TRM, the verifyAttestation module checks the AIK signature on the received data using the verifyCertifiedKey module. The vd parameter points to the received attestation verification data. The verifyCertifiedKey module uses the AIK public key certificate issued by the privacy CA to verify the signature on vd. Next, the verifyCertifyInfo module verifies the actual content of the received certified data. It checks the integrity of the received TRM public key, verifies that the key is bound to a trustworthy PCR on the TRM's machine and that the TRM key is not migratable to another machine. It also performs a number additional checks (e.g. the algorithm used to create TRM key, etc) not listed in table 5.2.

The quote_verify module is used by the CC to prevent against replay during the SIGMA protocol. Specifically, it checks that the session nonce is part of all the exchanged messages with the TRM. Finally, the header files cc_helper.h and quote_verify.h are used for configuring the CC. For instance, cc_helper.h specifies the trustworthy PCR numbers and their values to which the TRM key should be bound, the key usage properties (e.g. non-migratable), etc.

Chapter 6: Conclusion and Future Work

In this dissertation, we presented Policy, Enforcement, and Implementation models for Group-Centric Secure Information Sharing. In the policy layer, we developed the foundations for g-SIS by specifying a core set of properties that are required of all g-SIS specifications. We proved that the core properties are satisfiable and independent. We identified various temporal semantics of group operations and developed a π-system as a candidate g-SIS model and proved that it satisfies the core g-SIS properties. We also extended the core properties to read-write g-SIS models with and without object versioning. Further, we discussed a case-study built around an inter-organizational collaboration scenario. We presented a complete model involving administrative and operational components.

In the enforcement layer, we proposed a g-SIS enforcement model based on super, micro and hybrid object distribution models. We identified novel enforcement level security properties for stale-safety in distributed systems. We proposed stale-safe properties of varying strength and conducted formal analysis of g-SIS enforcement model specified as finite state machines. We showed that the initial model is not stale-safe and showed how it can be easily modified to satisfy the weak and subsequently the strong stale-safe security properties. We verified the model using a mature open-source model checker called NuSMV.

In the implementation layer, we proposed a trusted computing technology based implementation model for g-SIS. We specified trusted computing based protocols to realize the super, micro and hybrid object distribution based g-SIS enforcement models. As a proof-of-concept we implemented one of the critical protocols, called the provisioning protocol, using standard platforms with the Trusted Platform Module. At the end of the protocol, the CC provisions group credentials in the user's platform in such a manner that the TRM running in a trusted execution environment can access the credentials in the future only if the platform remains in a well-known trustworthy platform state.

The theory as presented in this dissertation is a first step towards studying g-SIS and serves to identify several directions in which it can be extended in the future. The future work can be categorized into Policy, Enforcement and Implementation layers of g-SIS as discussed below.

Policy Layer: In the policy layer, a major area for future research is to study inter-group relationships. For instance, a group may be set up between two organizations for a major collaboration project. One can envision additional sub-groups that may be established for a special purpose. Certain relationships between the primary group and sub-groups may be set up for administrative convenience. Examples of such relationships include subordination, conditional membership, mutual exclusion, etc.

In subordination, membership in a parent group group allows users to create subjects and access objects in one or more subordinate groups. Subordination can be Read-only and Read-Write. In read-only subordination, the user in a parent group can only read objects in subordinate group using subjects. Read-Write subordination allows the user to both read and write objects in subordinate groups.

In conditional membership, a user's membership in a conditional group is contingent upon her membership in an enabling group. In mutual exclusion, a user may not be allowed to be a member of mutually conflicting groups at the same time. In dynamic exclusion, the user may be allowed to be a member of mutually exclusive groups, but would not be allowed to create subjects in conflicting groups at the same time.

We briefly discuss the application of subordination and conditional group membership relations for flexible sharing across different lattices in the Bell-LaPadula (BLP) model. We use the following assumptions and rules for this construction using groups:

1. A group represents a clearance level in BLP.

2. Users have a specific clearance in an organization but may belong to multiple groups as allowed by the mutual exclusion relationship between groups. For instance, suppose A_TS and A_S represent groups with top secret and secret users and objects

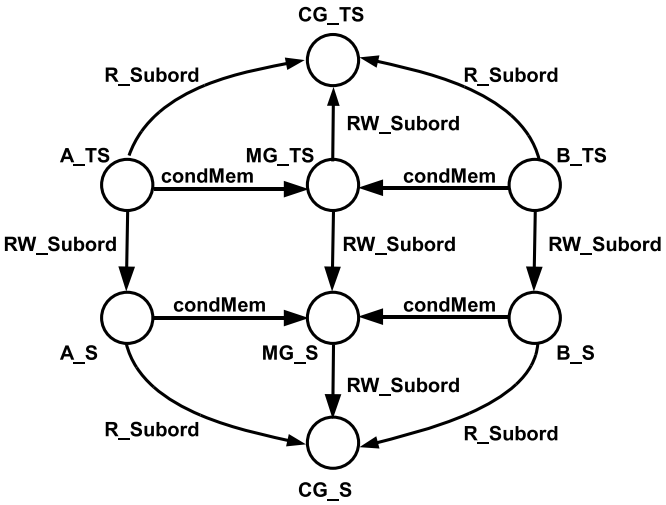

Figure 6.1: BLP in MGF.

respectively in an organization A. Users are not allowed to be members of both A_TS and A_S groups. However, the user may belong to groups in other organizations. For example, the user may belong to both A_TS and B_TS, where B is another organization.

3. Objects have a single classification and thus belong to only one group.

4. Users can create subjects in any group that is Read-only or Read-Write Subordinate to the user's group. Thus, subjects are associated with a specific group.

5. A subject can read objects from any group that is Read-only or Read-Write Subordinate to the subject's group (subjects can only read at the same and lower levels).

6. A subject can write only to those objects in the subject's group. Note that a user can create and write to an object in a group only if RW Subordinate path exists between the user's and the object's group. and groups to which the subject's group is Read-Write Subordinate to. That is subjects may write at the same and higher levels.

Figure 6.1 illustrates collaboration between two military organizations A and B. The four groups in the middle column is referred to as a Mission Group Framework (MGF) established

for collaboration between organizations A and B. MGF facilitates this collaboration by setting up Mission Groups (MS's) and Common Groups (CG's) for Top Secret (TS) and Secret (S) users in A and B. In figure 6.1, we only explain the relationship between groups in organization A (A_TS and A_S) and their relationship with MGF. A similar relationship exists between groups in organization B and MGF.

A_TS is the group containing users with TS clearance and objects with TS classification in organization A. Similarly, A_S is the group containing users with Secret (S) clearance and objects with S classification in organization A. We define A_S to be RW Subordinate to A_TS. This defines the domination relationship in BLP and allows A_TS users to create subjects and read and write objects in A_TS and A_S as defined by rules 4, 5 and 6.

MG_TS contains TS users and objects from both A_TS and B_TS. Similarly, MG_S contains S users and objects from both A_S and B_S. TS objects in A_TS and B_TS are shared with MG_TS. We define a conditional membership relationship between A_TS and MG_TS and B_TS and MG_TS respectively. This ensures that user's membership in MG_TS is contingent upon his/her membership in A_TS or B_TS. Similarly, user's membership in MG_S is contingent upon his/her membership in A_S or B_S due to the condMem relationship defined between MG_S and A_S and MG_S and B_S respectively. Note that MG_S is made subordinate to MG_TS and this allows MG_TS users to read and write to MG_S objects. Users in MGF can send out periodic updates to their source organizations using the common CG groups. This is achieved by making CG_TS and CG_S RW Subordinate to MG_TS and MG_S respectively. Note that this prevents subjects in MG_TS to read from MG_TS and write to CG_TS. A user can create a subject in CG_TS and write information in CG_TS about the status of mission. Further, an administrator can also send updates about the mission status to CG_TS by directly publishing information in CG_TS. If the users are trusted in MGF, the RW Subordinate definition between MG_TS and CG_TS allows for controlled mission updates to reach A and B without the intervention of an administrator during the course of the mission. A publish operation may facilitate consolidated information to be moved from MG_TS to CG_TS towards the end of the

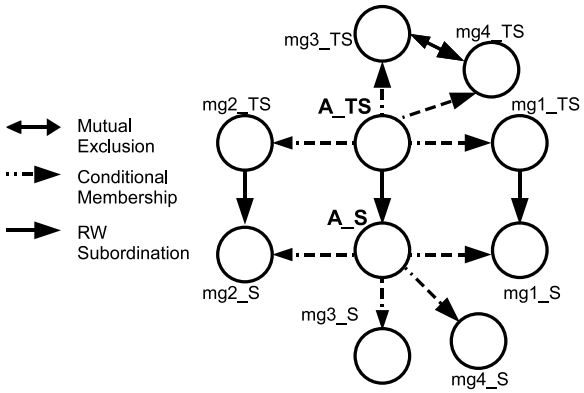

Figure 6.2: Mission Groups within organization A.

mission. Finally, note that CG_TS is R Subordinate to A_TS. This way A_TS users cannot write information to CG_TS. They may only read objects from CG_TS and possibly copy to A_TS. Note that both CG_TS and CG_S should not have any users. Otherwise, users in CG_TS, for example, can accidentally corrupt information in org A by writing to objects in A_TS (assuming R Subordinate allows write up).

Moreover mission groups can be established within a single organization. Users may be specifically added to mission groups as allowed by their clearance level. This is illustrated in figure 6.2.

Enforcement Layer: In the Enforcement layer, a major research work is on generalization of stale-safety to multiple authorization information sources. For example, if authorization information is obtained from sources A, B and C and a policy decision is made based on the obtained attribute values, how do we ensure that when authorization decision is made all the attributes held at the same time? If this cannot be ascertained, authorization decision will be based on attributes that never held at the same time. For instance, suppose that a user is allowed to enter a room only if she is member of the group and has the Top Secret clearance. Suppose the Policy Decision Point first obtains the user's group membership attribute and then her clearance attribute. In this scenario, the group membership and clearance attributes may dynamically change at the time of evaluation. Thus

it is important to ascertain that there existed some time at which both the user's attributes held and an authorization was made based on those values.

Another important future work is on generalization of stale-safety to Attribute-Based Access Control (ABAC). ABAC is generally preferred for specifying policies in distributed systems due to its flexibility and ease of use. We believe that generalizing stale-safety so that any policy specified using ABAC will be stale-safe would be a contribution with real-world impact.

Implementation Layer: At the implementation layer, we implemented a protocol for provisioning a group on the user's machine. This however assumed that the user's machine is in a trustworthy state. Ascertaining the trustworthiness of user's machine is a major future research area. Platform trustworthiness includes the hardware and software state of the system. TCG provides a means to measure the software at boot time and store in the TPM's PCR so that it can be reported to remote entities and the trustworthiness of the software state of the system can thereby be verified. However, this only confirms the software state of the system at boot time. Clearly, in any system, the software state changes with time and it if often desirable to estimate the software state at the given time. Measuring the run time integrity of a system is complex problem and is a potential area of major future research.

Appendix A: Proof of Consistency and Independence Theorems

The NuSMV models below describe how variables can be modified in each step of a system execution. A model is specified using NuSMV. Here, the NuSMV model is expressed in terms of Join, Leave, Add and Remove events (declared as boolean variables) that are allowed to occur concurrently in a non-deterministic manner. The theorem is expressed as an implication having the well-formed traces (the τ's) in the antecedent and the conjunction of core properties as the consequent. NuSMV takes the model and the LTL formula and determines whether the formula holds.

In the NuSMV model, the group events and authorization are declared as boolean variables in the VAR section. The DEFINE section consists of macro definitions to improve the readability of the code. The LTL formula to be checked are listed in the LTLSPEC section. Comments follow the symbols - -. The logic operators $\vee$, $\wedge$, and $\neg$ are represented as |, &, and !, respectively. The temporal logic operators $\bigcirc$, $\Box$, $\mathcal{U}$, $\mathcal{W}$, $\ominus$, $\blacklozenge$, and $\mathcal{S}$ are represented as X, G, U, W, Y, O and S respectively in the LTL formulas.

A.1 Consistency Theorem

The LTLSPEC towards the end of the following code negates the the whole consistency theorem. This forces NuSMV to find and generate a counter-example in which all the core properties hold.

```
MODULE main


VAR
authz: boolean;
    Join: boolean;
    Leave: boolean;
```

```
    Add: boolean;
    Remove: boolean;

---------------------------------------

LTLSPEC
!(
--well-formed trace constraints
    (
        G !(Add & Remove) & G !(Join & Leave) &
      G (Join  -> X ((!Join U Leave) | G !Join)) &
     G (Leave -> X((!Leave U Join) | G!Leave)) &
     G (Remove -> X((!Remove U Add) | G!Remove)) &
     G (Add -> X((!Add U Remove) | G!Add)) &
     G (Leave -> O Join) & G (Remove -> O Add)
    )
    ->
--Core Properties
    G((authz -> ((authz U (Join | Leave | Add | Remove)) | G authz)))
    &
    G((!authz -> ((!authz U (Join | Leave | Add | Remove)) | G !authz)))
    &
    ((!authz U (authz & (!Leave S Join) & (!Remove S Add))) | G !authz)
    &
    G((Leave & !authz) -> ((!authz U Join) | G !authz))
    &
    G((Remove & !authz) -> ((!authz U Add) | G !authz))
    &
```

```
    G(Join -> (((Add -> ((authz U (Leave | Remove)) | G authz)) U
        (Leave | Remove)) | G(Add -> ((authz U (Leave | Remove)) | G authz))))

)
```

Running NuSMV generates a counter-example in which all the core properties hold thereby proving the Consistency Theorem.

```
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification !(((((((((((( G !(Add & Remove) &  G !(Join & Leave)) &
G (Join ->  X ((!Join U Leave) |  G !Join))) &  G (Leave ->  X ((!Leave U
Join) |  G !Leave))) &  G (Remove ->  X ((!Remove U Add) |  G !Remove))) &
G (Add ->  X ((!Add U Remove) |  G !Add))) &  G (Leave ->  O Join)) &
G (Remove ->  O Add)) &  G (authz -> ((authz U (((Join | Leave)
| Add) | Remove)) |  G authz))) &  G (!authz -> ((!authz U (((Join | Leave)
| Add) | Remove)) |  G !authz))) & ((!authz U ((authz & (!Leave S Join)) &
(!Remove S Add))) |  G !authz)) & G ((Leave & !authz) -> ((!authz U Join) |
G !authz))) &  G ((Remove & !authz) -> ((!authz U Add) | G !authz))) &
G (Join -> (((Add -> ((authz U (Leave | Remove)) |  G authz))
U (Leave | Remove)) | G (Add -> ((authz U (Leave | Remove)) |  G authz)))))
is false
-- as demonstrated by the following execution sequence
```

```
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  authz = 0
  Join = 1
  Leave = 0
  Add = 0
  Remove = 0
-> Input: 1.2 <-
-> State: 1.2 <-
  Join = 0
  Leave = 1
-> Input: 1.3 <-
-> State: 1.3 <-
  Leave = 0
-> Input: 1.4 <-
-> State: 1.4 <-
  Join = 1
-> Input: 1.5 <-
-> State: 1.5 <-
  authz = 1
  Join = 0
  Add = 1
-> Input: 1.6 <-
-> State: 1.6 <-
  authz = 0
  Leave = 1
```

```
  Add = 0
-> Input: 1.7 <-
-> State: 1.7 <-
  authz = 1
  Join = 1
  Leave = 0
-> Input: 1.8 <-
-- Loop starts here
-> State: 1.8 <-
  Join = 0
-> Input: 1.9 <-
-> State: 1.9 <-
  Leave = 1
-> Input: 1.10 <-
-- Loop starts here
-> State: 1.10 <-
  Leave = 0
-> Input: 1.11 <-
-> State: 1.11 <-
```

A.2 Proof of Independence Theorem

Due to space constraints, the complete code listing and NuSMV output for the proof of this theorem is given in [55]. The following code serves to illustrate the verification. In the two LTLSPEC's below, we specify 5 of the 6 core properties in the antecedent and the other property (the Provenance property in this case) in the consequent. The first LTLSPEC asserts that the Provenance core property in the consequent can be proved from the remaining 5 properties in the antecedent. The second LTLSPEC asserts that the Provenance property in the consequent cannot be proved from the remaining 5 properties

in the antecedent. NuSMV produces a counter example for both the assertions proving that the Provenance property in the consequent is independent of the remaining 5 properties. Complete verification and independence proof with counter-examples for the remaining properties can be found in [55].

```
MODULE main


VAR
authz: boolean;
Join: boolean;
Leave: boolean;
Add: boolean;
Remove: boolean;


---------------------------------------


LTLSPEC
(
--well-formed trace constraints
    (
        G !(Add & Remove) & G !(Join & Leave) &
      G (Join  -> X ((!Join U Leave) | G !Join)) &
     G (Leave -> X((!Leave U Join) | G!Leave)) &
     G (Remove -> X((!Remove U Add) | G!Remove)) &
     G (Add -> X((!Add U Remove) | G!Add)) &
     G (Leave -> O Join) & G (Remove -> O Add)
    )
    &
```

```
    G((authz -> ((authz U (Join | Leave | Add | Remove)) | G authz)))
    &
    G((!authz -> ((!authz U (Join | Leave | Add | Remove)) | G !authz)))
    &
    G((Leave & !authz) -> ((!authz U Join) | G !authz))
    &
    G((Remove & !authz) -> ((!authz U Add) | G !authz))
    &
    G(Join -> (((Add -> ((authz U (Leave | Remove)) | G authz)) U
    (Leave | Remove)) | G(Add -> ((authz U (Leave | Remove)) | G authz))))
)
    ->
--Provenance Property
    ((!authz U (authz & (!Leave S Join) & (!Remove S Add))) | G !authz)

LTLSPEC
(
--well-formed trace constraints
    (
        G !(Add & Remove) & G !(Join & Leave) &
      G (Join  -> X ((!Join U Leave) | G !Join)) &
     G (Leave -> X((!Leave U Join) | G!Leave)) &
     G (Remove -> X((!Remove U Add) | G!Remove)) &
     G (Add -> X((!Add U Remove) | G!Add)) &
     G (Leave -> O Join) & G (Remove -> O Add)
    )
    &
```

```
    G((authz -> ((authz U (Join | Leave | Add | Remove)) | G authz)))
    &
    G((!authz -> ((!authz U (Join | Leave | Add | Remove)) | G !authz)))
    &
    G((Leave & !authz) -> ((!authz U Join) | G !authz))
    &
    G((Remove & !authz) -> ((!authz U Add) | G !authz))
    &
    G(Join -> (((Add -> ((authz U (Leave | Remove)) | G authz))
    U (Leave | Remove)) | G(Add -> ((authz U (Leave | Remove)) | G authz))))
)
    ->
-- Negation of Provenance Property
    ! ((!authz U (authz & (!Leave S Join) & (!Remove S Add))) | G !authz)
```

Running NuSMV generates a counter-example proving the independence of the Provenance Property from the remaining five core properties.

```
*** This is NuSMV 2.4.3 (compiled on Tue May 22 14:08:54 UTC 2007)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson

-- specification ((((((((((((( G !(Add & Remove) &  G !(Join & Leave)) &
G (Join ->  X ((!Join U Leave) |  G !Join))) &  G (Leave ->
X ((!Leave U Join) |  G !Leave))) &  G (Remove ->  X ((!Remove U Add)
```

```
|  G !Remove))) &  G (Add ->  X ((!Add U Remove) |  G !Add))) &
G (Leave ->  O Join)) &  G (Remove ->  O Add)) &  G (authz ->
((authz U (((Join | Leave) | Add) | Remove)) |  G authz))) &
G (!authz -> ((!authz U (((Join | Leave) | Add) | Remove)) |
G !authz))) &  G ((Leave & !authz) -> ((!authz U Join) |
G !authz))) &  G ((Remove & !authz) -> ((!authz U Add) |
G !authz))) &  G (Join -> (((Add -> ((authz U (Leave |
Remove)) |  G authz)) U (Leave | Remove)) |  G (Add ->
((authz U (Leave | Remove)) |  G authz))))) -> ((!authz
U ((authz & (!Leave S Join)) & (!Remove S Add))) |
G !authz))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  authz = 1
  Join = 1
  Leave = 0
  Add = 0
  Remove = 0
-> Input: 1.2 <-
-> State: 1.2 <-
  authz = 0
  Join = 0
  Leave = 1
  Add = 1
-> Input: 1.3 <-
-> State: 1.3 <-
```

```
  Leave = 0
  Add = 0
-> Input: 1.4 <-
-> State: 1.4 <-
  Join = 1
-> Input: 1.5 <-
-- Loop starts here
-> State: 1.5 <-
  Join = 0
  Leave = 1
-> Input: 1.6 <-
-> State: 1.6 <-
  Leave = 0
-> Input: 1.7 <-
-> State: 1.7 <-
  Join = 1
-> Input: 1.8 <-
-> State: 1.8 <-
  Join = 0
  Leave = 1
-- specification ((((((((((((( G !(Add & Remove) &  G !(Join & Leave))
&  G (Join ->  X ((!Join U Leave) |  G !Join))) &  G (Leave ->
X ((!Leave U Join) |  G !Leave))) &  G (Remove ->
X ((!Remove U Add) |  G !Remove))) &  G (Add ->
X ((!Add U Remove) |  G !Add))) &  G (Leave ->  O Join))
&  G (Remove ->  O Add)) &  G (authz -> ((authz U
(((Join | Leave) | Add) | Remove)) |  G authz))) &
G (!authz -> ((!authz U (((Join | Leave) | Add) | Remove))
```

```
|  G !authz))) &  G ((Leave & !authz) -> ((!authz U Join)
|  G !authz))) &  G ((Remove & !authz) -> ((!authz U Add)
|  G !authz))) &  G (Join -> (((Add -> ((authz U (Leave
| Remove)) |  G authz)) U (Leave | Remove)) |  G (Add ->
((authz U (Leave | Remove)) |  G authz))))) -> !((!authz
U ((authz & (!Leave S Join)) & (!Remove S Add))) |
G !authz))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
-> State: 2.1 <-
  authz = 1
  Join = 1
  Leave = 0
  Add = 1
  Remove = 0
-> Input: 2.2 <-
-> State: 2.2 <-
  authz = 0
  Join = 0
  Leave = 1
  Add = 0
  Remove = 1
-> Input: 2.3 <-
-> State: 2.3 <-
  authz = 1
  Join = 1
  Leave = 0
```

```
  Add = 1
  Remove = 0
-> Input: 2.4 <-
-- Loop starts here
-> State: 2.4 <-
  Join = 0
  Add = 0
-> Input: 2.5 <-
-- Loop starts here
-> State: 2.5 <-
-> Input: 2.6 <-
-> State: 2.6 <-
  Remove = 1
-> Input: 2.7 <-
-- Loop starts here
-> State: 2.7 <-
  Remove = 0
-> Input: 2.8 <-
-> State: 2.8 <-
```

Appendix B: Proof of Entailment Theorems

B.1 Proof of Mixed Operations and Membership Renewal Semantics Entailment Theorems

The following NuSMV model in the code listing simply allows group events such as SJ, LJ, SR, etc. to occur concurrently in a non-deterministic manner. This model produces all possible traces in terms of these group events. We concern ourselves only with the legal mixed g-SIS traces by formulating the π-system g-SIS specification as the antecedent and the core and membership renewal properties as the consequent of an implication. NuSMV verifies that the LTLSPEC holds thereby proving both the theorems.

Code Listing A

```
MODULE main
VAR
SL: boolean;
LL: boolean;
    SA: boolean;
LA: boolean;
SJ: boolean;
LJ: boolean;
SR: boolean;
LR: boolean;
    authz: boolean;

DEFINE
Join := SJ | LJ;
    Leave := SL | LL;
```

```
    Add := SA | LA;
Remove := SR | LR;


----------------------------------------
LTLSPEC
(
--well-formed trace constraints
      G !(Add & Remove) & G !(Join & Leave) &
      G (Join  -> X ((!Join U Leave) | G !Join)) &
     G (Leave -> X((!Leave U Join) | G!Leave)) &
     G (Remove -> X((!Remove U Add) | G!Remove)) &
     G (Add -> X((!Add U Remove) | G!Add)) &
     G (Leave -> O Join) & G (Remove -> O Add) &
     G(!(SJ & LJ) & !(SL & LL) & !(SA & LA) & !(SR & LR)) &
--Pi-system g-SIS specification
     G(authz <-> (((!SL & !SR) S ((SA | LA) & ((!LL & !SL)
        S (SJ | LJ)))) | ((!SL & !SR) S (LJ & ((!SR & !LR) S LA)))))
    )
->
--Core Properties
(
    G((authz -> ((authz U (Join | Leave | Add | Remove)) | G authz)))
    &
    G((!authz -> ((!authz U (Join | Leave | Add | Remove)) | G !authz)))
    &
    ((!authz U (authz & (!Leave S Join) & (!Remove S Add))) | G !authz)
    &
    G((Leave & !authz) -> ((!authz U Join) | G !authz))
```

```
    &
    G((Remove & !authz) -> ((!authz U Add) | G !authz))
    &
    G(Join -> (((Add -> ((authz U (Leave | Remove)) | G authz))
    U (Leave | Remove)) | G(Add -> ((authz U (Leave | Remove)) | G authz))))
)
&
--Membership Renewal Properties
--Lossless Join
G(
((Join & !Remove & Y authz) -> authz)
) &
--Gainless Leave
G(
((Leave & (!Join U (authz & !Join))) ->
Y ((!authz & !Join) S (authz & (!Join S Join))))
) &
--Non-Restorative Leave
G(
((Leave & authz) -> Y authz)
)
```

B.2 Proof of Most Restrictive Entailment Theorem

In the LTLSPEC below, the π-system g-SIS specification is replaced by the fixed operation (SJ, SL, SA, SR) specification. Furthermore, we specify the corresponding membership and membership renewal properties along with the core properties in the consequent. NuSMV successfully verifies that the theorem is true.

Code Listing B

```
MODULE main


VAR


SL: boolean;
   SA: boolean;
SJ: boolean;
SR: boolean;


authz: boolean;


DEFINE
Join := SJ;
    Leave := SL;
    Add := SA;
Remove := SR;


---------------------------------------
LTLSPEC
(
--well-formed trace constraints
      G !(Add & Remove) & G !(Join & Leave) &
      G (Join  -> X ((!Join U Leave) | G !Join)) &
     G (Leave -> X((!Leave U Join) | G!Leave)) &
     G (Remove -> X((!Remove U Add) | G!Remove)) &
     G (Add -> X((!Add U Remove) | G!Add)) &
     G (Leave -> O Join) & G (Remove -> O Add) &
```

```
--Most Restrictive Mixed g-SIS Specification
G(authz <-> ((!SL & !SR) S (SA & (!SL S SJ))))
)
->
--Core Properties
(
    G((authz -> ((authz U (Join | Leave | Add | Remove)) | G authz)))
    &
    G((!authz -> ((!authz U (Join | Leave | Add | Remove)) | G !authz)))
    &
    ((!authz U (authz & (!Leave S Join) & (!Remove S Add))) | G !authz)
    &
    G((Leave & !authz) -> ((!authz U Join) | G !authz))
    &
    G((Remove & !authz) -> ((!authz U Add) | G !authz))
    &
    G(Join -> (((Add -> ((authz U (Leave | Remove)) | G authz)) U
    (Leave | Remove)) | G(Add -> ((authz U (Leave | Remove)) | G authz))))
)
&
--Membership Renewal Properties
--Lossless Join: beta0
G(
((Join & !Remove & Y authz) -> authz)
) &
--Gainless Leave: beta2
G(
((Leave & (!Join U (authz & !Join))) ->
```

```
Y ((!authz & !Join) S (authz & (!Join S Join))))
) &
--Non-Restorative Leave: beta 3
G(
((Leave & authz) -> Y authz)
)
&
--Membership Properties
--Strict Join: alpha0
G(
(authz -> O (SA & (!SL S SJ)))
) &
--Strict Leave: alpha1
G(
(authz -> (!SL S SJ))
) &
--Strict Add: alpha2
G(
(SA -> (!O SJ -> ((!authz U SA) | G !authz)))
) &
--Strict Remove: alpha3
G(
(SR -> ((!authz U SA) | G !authz))
)
```

Appendix C: Stale-Safety Verification

C.1 Description of Code

There are four NuSMV modules that represent this model. They are the User, GA, TRM and CC module. The GA essentially only talks to the CC (to notify the CC to change its time stamps). The TRM talks to the CC (to get timestamps) and to the user (to notify the user that it can perform or deny perform).

User Module:
This Module models a user that requests a perform action on an anonymous object. The user is unaware of whether or not the object should be accessed (performed on) and relies on the local TRM (modeled by the TRM module) to determine whether or not they can perform (on the object). This model relies on a random decision by the user to access the object.

TRM Module:
Each TRM (i.e., stale unsafe, weak and strong stale-safe TRMs) is based on a similar model. The TRM randomly refreshes (with no restrictions) at which time the timestamps are set to those of the CC. When the user makes a request this initiates a question-response event. The TRM will only respond once it becomes ready to do so, at which point it will give an answer to the request (i.e. allow perform or deny perform). The three models differ based on when the TRM becomes ready. Also, the unsafe TRM gets its response to the request different from the weak and strong TRM's.

TRM Unsafe: The unsafe TRM is always ready (to respond). The response is the evaluation of the policy with TRM's (possibly stale) timestamps.

TRM Weak: This TRM is essentially always ready, and only becomes not ready when staleness occurs. This happens when, in the previous state, the policy did not hold, but it

does hold in this state (without a refresh). This will make the TRM become not ready. The value of the TRM's response (to requests) is the value of the policy during the last refresh.

TRM Strong: This TRM only becomes ready when a refresh happens during the question-response process (i.e. after a request is received). Again, the TRM's response is calculated from the latest refresh.

CC Module:

The CC maintains the most current time stamps. It does this by accepting add/remove actions from the GA module. The TRM gets all of its timestamps from the CC during a refresh. Every time an object is added by the GA the CC goes into a super distributed state and the TRM will immediately pick up the new object_add_ts.

GA Module:

The GA is similar to the user in that it randomly performs 4 basic actions (the user only performs one basic action). These actions are add/remove object and join/leave user. The only restriction on the randomness is that the GA is smart in performing these actions. Because the CC maintains up-to-date timestamp information, the GA can query the CC as to whether or not a user/object is part of the group. In this way, the GA only performs an add action when the user/object is not part of the group and only performs remove actions when the user/object is part of the group.

Other Modules:

There is also a Clock module and a question-response module. The clock is straight forward, it gives a sequential index to each state. It starts at zero and advances by one in each state and stops when a maximum value is reached. The question-response module is the workhorse since it models the interaction between the user and the TRM. This module is only concerned with the timing of events: a question is asked then, some time later, answered. It captures this interaction with 2 internal variables: responding and respond.

Responding just means that you are in the middle of a question, and respond signifies that you are giving a response in this state. The question and the value of the response are not part of this module; essentially this module can be used to determine when it is correct to grab the response from the answerer. There are three external inputs given: ask, ready, and force-response. These variables define the behavior of the questioner (by defining ask) and the responder (by defining when the responder is ready and when the responder will definitely respond).

C.2 Counter-Examples

C.2.1 Verification against Δ_0-system

Verification of Unenforceablity Theorem

```
*** This is NuSMV 2.4.3 (compiled on Mon Feb 23 20:34:39 UTC 2009)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification  G (perform ->  Y (!perform S
(authzcc & ((!request & !perform) S request))))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample

State 1:
trm.object_remove_ts = 0
trm.user_leave_ts = 0
authzcc = 0
authztrm = 0
```

```
State 2:
ga.action_join_user = 1
trm.action_refresh = 1
trm.user_join_ts = 1

authzcc = 0
authztrm = 0

State 3:
user.request_perform = 1
ga.action_add_object = 1
ga.action_leave_user = 1
trm.object_add_ts = 2
trm.question.respond = 1
authztrm = 1
authzcc = 0

State 4:
user.action_perform = 1
authzcc = 0
```

As shown in the counter-example above, NuSMV successfully proves that Authz_{CC} is not enforceable at TRM_0. That is, $\Delta_0 \not\models \Box(\text{perform} \rightarrow \varphi_0')$. In this trace it should be clear that [authzcc] is always false (value of 0). The user is joined in the 2nd state, where a refresh also happens. This sets the user_join_ts for the TRM. In this state both authztrm and authzcc are false. In the 3rd state, the user requests, the ga adds an object which triggers object_add_ts to be set for the TRM, which changes the value of authztrm to true, and, finally, the TRM responds to the user request in this state with a response of true

(authztrm). This fails the property because there is no point between the request and perform where authzcc was true.

Verification of Enforceability Theorem

```
*** This is NuSMV 2.4.3 (compiled on Mon Feb 23 20:34:39 UTC 2009)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification G(perform -> Y ((!perform & (!rt | (rt &
authzcc))) S (authztrm & ((!request & !perform) S request))) is true
```

NuSMV successfully verifies $\Box(\text{perform} \rightarrow \varphi_0)$ against the Δ_0-system as shown above.

Verification of Weak Stale-Safety against Δ_0-system

```
*** This is NuSMV 2.4.3 (compiled on Mon Feb 23 20:34:39 UTC 2009)
*** For more information on NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification  G (perform -> ( Y ((!perform & (!rt |
(rt & authzcc))) S (request & (!rt S (rt & authzcc)))) |
Y ((!perform & !rt) S ((rt & authzcc) & (!perform S request)))))
is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample

State 1:
```

```
trm.object_remove_ts = 0
trm.user_leave_ts = 0

authzcc = 0
authztrm = 0

State 2:
ga.action_join_user = 1
trm.action_refresh = 1
trm.user_join_ts = 1

authzcc = 0
authztrm = 0

State 3:
user.request_perform = 1
ga.action_leave_user = 1
ga.action_add_object = 1
trm.object_add_ts = 2
trm.question.respond = 1

authzcc = 0
authztrm = 1

State 4:
user.action_perform = 1
```

The counter-example generated by NuSMV clearly shows that the Δ_0-system fails the weak stale-safety security property. The user is joined in State 2 and the TRM refreshes. This

causes the TRM's user_join_ts to be set, but both authzcc and authztrm are still false. In the next (3rd state) the user requests, an object is added, the user is removed (leave), and the TRM responds to the user's request. Because the object is added, the CC will go into a super distributed state and the TRM will instantly get the object_add_ts, which causes the TRM's policy to now evaluate to true (authztrm). Therefore the user performs in the next state (4th state) with stale attributes. This fails the weak staleness property because there is no refresh where authzcc held and yet there is a perform. Note: authzcc could have held in State 3 if the ga had not removed the user, but the attributes in the TRM would still be considered stale because it is never locally verifies that the object and user were part of the group at the same time through a refresh before it allows a perform.

C.2.2 Verification against Δ_1-system

Weak Stale-Safety

```
-- specification  G (perform -> ( Y ((!perform & (!rt | (rt & authzcc)))
S (request & (!rt S (rt & authzcc)))) |  Y ((!perform & !rt) S
((rt & authzcc) & (!perform S request)))))  is true
```

NuSMV successfully verifies that the Δ_1-system satisfies the weak stale-safe security property.

Strong Stale-Safety

```
-- specification  G (perform ->  Y ((!perform & !rt) S
((rt & authzcc) & (!perform S request))))  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample


State 1:
```

```
trm.user_leave_ts = 0
trm.object_remove_ts = 0
authzcc = 0
authztrm = 0

State 2:
ga.action_add_object = 1
ga.action_join_user = 1
trm.action_refresh = 1
trm.object_add_ts = 1
trm.user_join_ts = 1
authzcc = 1
authztrm = 1

State 3:
user.request_perform = 1
ga.action_remove_object = 1
ga.action_leave_user = 1
trm.question.respond = 1
authzcc = 0
authztrm = 1

State 4:
user.action_perform = 1
```

As shown in the counter-example generated by NuSMV above, the Δ_1-system fails strong stale-safety. In State 2 the GA adds the object and joins the user and the TRM refreshes. The refresh causes both user_join_ts and object_add_ts to be updated at the TRM which makes authztrm true and also authzcc is true. In State 3, the user requests, the ga removes

the object and user, and the TRM responds to the users request. Because there is no refresh, authztrm remains true (which is the value sent to the user) and the user is then allowed to perform in State 4. Again, it's not necessary that authzcc becomes false in this state, just that between this request and the perform in State 4 there was no refresh by the TRM.

C.2.3 Verification against Δ_2-system

Strong Stale-Safety

```
-- specification  G (perform ->  Y ((!perform & !rt) S ((rt & authzcc)
& (!perform S request)))) is true
```

As shown in the NuSMV report above, the Δ_2-system satisfies strong stale-safety.

Bibliography

Bibliography

[1] S. Rafaeli and D. Hutchison, "A survey of key management for secure group communication," *ACM Computing Surveys*, vol. 35, no. 3, pp. 309–329, 2003.

[2] G. S. Graham and P. J. Denning, "Protection: principles and practice," in *AFIPS '71 (Fall): Proceedings of the November 16-18, 1971 fall joint computer conference*. New York, NY, USA: ACM, 1971, pp. 417–429.

[3] B. W. Lampson, "Protection," *SIGOPS Operating Systems Review*, vol. 8, no. 1, pp. 18–24, 1974.

[4] *Trusted Computer System Evaluation Criteria*, DoD National Computer Security Center (DoD 5200.28-STD), December 1985.

[5] R. Sandhu, "Lattice-based access control models," *IEEE Computer*, vol. 26, no. 11, pp. 9–19, 1993.

[6] D. Bell and L. LaPadula, "Secure computer systems: Mathematical foundations," MITRE Technical Report MTR-2547, Tech. Rep., 1973.

[7] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman, "Role-Based Access Control Models," *IEEE Computer*, pp. 38–47, 1996.

[8] S. Osborn, R. Sandhu, and Q. Munawer, "Configuring role-based access control to enforce mandatory and discretionary access control policies," *ACM Transactions on Information and Systems Security*, vol. 3, no. 2, pp. 85–106, 2000.

[9] J. Park and R. Sandhu, "The UCON$_{ABC}$ usage control model," *ACM Transactions on Information and Systems Security*, vol. 7, no. 1, pp. 128–174, 2004.

[10] M. C. Mont, S. Pearson, and P. Bramhall, "Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services," in *DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications*. Washington, DC, USA: IEEE Computer Society, 2003, p. 377.

[11] S. Bandhakavi, C. C. Zhang, and M. Winslett, "Super-sticky and declassifiable release policies for flexible information dissemination control," in *WPES '06: Proceedings of the 5th ACM workshop on privacy in electronic society*. New York, NY, USA: ACM, 2006, pp. 51–58.

[12] D. W. Chadwick and S. F. Lievens, "Enforcing "sticky" security policies throughout a distributed application," in *MidSec '08: Proceedings of the 2008 workshop on middleware security*. New York, NY, USA: ACM, 2008, pp. 1–6.

[13] R. Graubart, "On the need for a third form of access control," in *Proceedings of the 12th National Computer Security Conference*, 1989, pp. 296–304.

[14] C. J. McCollum, J. R. Messing, and L. Notargiacomo, "Beyond the Pale of MAC and DAC–Defining New Forms of Access Control," *IEEE Symposium on Security and Privacy*, vol. 0, p. 190, 1990.

[15] M. Abrams, J. Heaney, O. King, L. LaPadula, M. Lazear, and I. Olson, "Generalized framework for access control: Towards prototyping the ORGCON policy," in *Proceedings of the 14th National Computer Security Conference*, 1991, pp. 257–266.

[16] J. Park and R. Sandhu, "Originator control in usage control," in *POLICY '02: Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2002, p. 60.

[17] "eXtensible rights Markup Language," *www.xrml.org*.

[18] "The Open Digital Rights Language Initiative," *www.odrl.net*.

[19] C. Farkas and S. Jajodia, "The inference problem: a survey," *SIGKDD Explor. Newsl.*, vol. 4, no. 2, pp. 6–11, 2002.

[20] E. Bertino, P. A. Bonatti, and E. Ferrari, "Trbac: A temporal role-based access control model," *ACM Transactions on Information and Systems Security*, vol. 4, no. 3, pp. 191–233, 2001.

[21] J. Joshi, E. Bertino, U. Latif, and A. Ghafoor, "A generalized temporal role-based access control model," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 1, pp. 4–23, 2005.

[22] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, no. 8, pp. 461–471, 1976.

[23] R. J. Lipton and L. Snyder, "A linear time algorithm for deciding subject security," *Journal of the ACM*, vol. 24, no. 3, pp. 455–464, 1977.

[24] R. Sandhu, "The schematic protection model: its definition and analysis for acyclic attenuating schemes," *Journal of the ACM*, vol. 35, no. 2, pp. 404–432, 1988.

[25] R. S. Sandhu, "The typed access matrix model," in *Proceedings of the 1992 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 1992, p. 122.

[26] R. Sandhu, V. Bhamidipati, and Q. Munawer, "The arbac97 model for role-based administration of roles," *ACM Transactions on Information and Systems Security*, vol. 2, no. 1, pp. 105–135, 1999.

[27] T. Jaeger and J. E. Tidswell, "Practical safety in flexible access control models," *ACM Transactions on Information and Systems Security*, vol. 4, no. 2, pp. 158–190, 2001.

[28] R. Sandhu, K. Ranganathan, and X. Zhang, "Secure information sharing enabled by trusted computing and PEI models," in *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*. ACM, 2006, p. 12.

[29] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Heidelberg, Germany: Springer-Verlag, 1992.

[30] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/mechanism separation in Hydra," *ACM SIGOPS Operating Systems Review*, vol. 9, no. 5, pp. 132–140, 1975.

[31] E. M. Clarke, E. A. Emerson, and A. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Language and Systems (TOPLAS)*, vol. 8, no. 2, pp. 244–263, 1986.

[32] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachusetts, USA: The MIT Press, 1999.

[33] K. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[34] A. Cimatti, E.M.Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: A new symbolic model checker," *Journal on Software Tools for Tech. Transfer*, pp. 410–425, 2000.

[35] A. Pnueli, "The temporal logic of programs," in *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, vol. 526, 1977, pp. 46–67.

[36] "TCG Specification Architecture Overview," *http://www.trustedcomputinggroup.org*.

[37] A. P. Sistla, "Safety, liveness and fairness in temporal logic," in *Formal Aspect of Computing*, 1994, pp. 495–511.

[38] E. Barka and R. Sandhu, "Framework for role-based delegation models," in *Proc. of the 16th Annual Computer Security Applications Conference*. Washington, DC, USA: IEEE Computer Society, 2000, p. 168.

[39] X. Zhang, S. Oh, and R. Sandhu, "PBDM: a flexible delegation model in RBAC," in *Proceedings of the eighth ACM symposium on Access control models and technologies*. ACM, 2003, p. 157.

[40] E. Barka and R. Sandhu, "Role-based delegation model/hierarchical roles (RBDM1)," in *Proceedings of the 20th Annual Computer Security Applications Conference*. Tucson, Arizona, USA, IEEE Computer Society, 2004, pp. 396–404.

[41] A. Osterhues, A. R. Sadeghi, M. Wolf, C. Stüble, and N. Asokan, "Securing Peer-to-peer Distributions for Mobile Devices," in *4th Information Security Practice and Experience Conference*, 2008.

[42] A. Sadeghi, M. Scheibel, C. Stüble, and M. Wolf, "Play it once again, Sam-Enforcing stateful licenses on open platforms," in *2nd Workshop on Advances in Trusted Computing*, 2006.

[43] U. Kühn, M. Selhorst, and C. Stüble, "Realizing property-based attestation and sealing with commonly available hard- and software," in *Proc. of ACM workshop on Scalable trusted computing*, 2007.

[44] C. Boyd, "Digital multisignatures," *In Cryptography and Coding*, pp. 241–246, Oxford University Press, 1989.

[45] R. Ganesan, "Yaksha: Augmenting Kerberos with public key cryptography," in *Proc. of the Symp. on Network and Dist. Syst. Security*, 1995.

[46] R. GanesaN, "Yaksha security system," *Communications of the ACM*, vol. 39, no. 3, pp. 55–60, 1996.

[47] R. Sandhu, M. Bellare, and R. Ganesan, "Password-enabled PKI: Virtual smartcards versus virtual soft tokens," in *Proceedings of the 1st Annual PKI Research Workshop*, 2002.

[48] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[49] J. Goguen and J. Meseguer, "Security policies and security models," *IEEE Symposium on Security and Privacy*, vol. 12, 1982.

[50] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Comm. of the ACM*, pp. 461–471, August 1976.

[51] J. Niu, J. M. Atlee, and N. A. Day, "Template semantics for model-based notations," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 866–882, October 2003.

[52] E. Mikk, Y. Lakhnech, and M. Siegel, "Hierarchical automata as model for statecharts," in *ASIAN '97: Proceedings of the Third Asian Computing Science Conference on Advances in Computing Science*. London, UK: Springer-Verlag, 1997, pp. 181–196.

[53] M. Yannakakis, "Hierarchical State Machines," *In Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS, LNCS*, vol. 1872, pp. 315–330, 2000.

[54] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," *Lecture Notes in Computer Science*, pp. 359–364, 2002.

[55] R. Krishnan, "Group-Centric Secure Information Sharing Models: Proofs of theorems and counter-examples from model checking." *http://www.profsandhu.com/ram_krishnan/*, 2009.

[56] W. Diffie, P. Oorschot, and M. Wiener, "Authentication and authenticated key exchanges," *Designs, Codes and Cryptography*, vol. 2, no. 2, pp. 107–125, 1992.

[57] J. Liedtke, "Toward real microkernels," *Communications of the ACM*, vol. 39, no. 9, pp. 70–77, 1996.

[58] H. Krawczyk, "SIGMA: The SIGn-and-MAcapproach to authenticated Diffie-Hellman and its use in the IKE protocols," in *Advances in Cryptography*. Springer, 2003.

[59] Trousers, "A TSS implementation for Linux," *IBM. Retrieve from http://trousers.sourceforge.net*, 2005.

Curriculum Vitae

Ram Narayan Krishnan was born in 1979 in India. He received his Bachelor of Technology degree in Computer Science and Engineering from Pondichery University in 2002 and Master of Science degree in Computer Engineering from New Jersey Institute of Technology in 2003.