Enforcement of Security and Privacy Policy Specifications in First Order Temporal Logic

APPROVED BY SUPERVISING COMMITTEE:

Jianwei Niu, Ph.D., Chair

Ram Krishnan, Ph.D.

Jeff von Ronne, Ph.D.

Accepted: Dean, Graduate School

DEDICATION

I would like to dedicate this thesis to my loving wife and my mother.

Enforcement of Security and Privacy Policy Specifications in First Order Temporal Logic

by

JARED F. BENNATT, B.S.

THESIS
Presented to the Graduate Faculty of
The University of Texas at San Antonio
in Partial Fulfillment
of the Requirements
for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
July 2015

ACKNOWLEDGEMENTS

I would like to thank my wife for her support in this endeavor as well as my three kids who motivated me to get out of the house. This thesis would not be possible without the guidance and support of Dr. Jianwei Niu and I thank her for always pushing me to acheive to my fullest potential. I would also like to thank Dr. Ram Krishnan and Dr. Jeff von Ronne for being on my Master's Thesis Committee and especially to Dr. Ram Krishnan for extensive discussions on the topic of Stale-Safety. This thesis is based upon work supported by the National Science Foundation under Grant No. CNS-0964710.

July 2015

Enforcement of Security and Privacy Policy Specifications

in First Order Temporal Logic

Jared F. Bennatt, M.Sc.
The University of Texas at San Antonio, 2015

Supervising Professor: Jianwei Niu, Ph.D.

In distributed systems, attribute staleness is inevitable which leads to stale authorization decisions. While it may not be possible to eliminate staleness, this thesis shows that it is possible to limit what type of authorization decisions are made when using stale attributes. Two stale-safety properties, one strictly stronger than the other, are introduced in the context of g-SIS (Group-Centric Secure Information Sharing): Weak Stale-Safety property and Strong Stale-Safety property. Three versions of g-SIS are modeled: a stale-unsafe version, a weakly stale-safe version, and a strongly stale-safe version. Model checking is used to formally verify which stale-safety property/properties, if any, each g-SIS model exhibits. A small model theorem is discussed to extend the results of model checking to first order temporal logic. Finally general stale-safety is discussed along with a notion of minimal stale-safety and shown to be broadly applicable to domains other than g-SIS.

TABLE OF CONTENTS

Acknowledgements . . . . **iv**

Abstract . . . . **v**

List of Tables . . . . **ix**

List of Figures . . . . **x**

Chapter 1: Introduction **1**

Chapter 2: HIPAA Privacy Policy Analysis **5**

2.1 Privacy Promises That Can Be Kept 5

2.2 Automatic Analysis of Federal Laws 8

2.2.1 Structure of Federal Laws 8

2.2.2 Getting the Data 9

2.2.3 Analyzing Meta-Data 9

2.2.4 Future Work 11

Chapter 3: g-SIS **12**

3.1 Objectives 12

3.2 Enforcement Model for g-SIS 13

Chapter 4: Staleness in g-SIS **17**

4.1 System Characterization 17

4.2 Example of Staleness 18

4.3 Formal Property Specification in g-SIS 19

4.4 Predicates 19

4.5 Access Policy Specification 20

4.6 Weak Stale-safe Security Property 21

4.7 Strong Stale-Safe Security Property 23

Chapter 5: Modeling g-SIS 25

5.1 g-SIS Class Diagrams 25

5.2 $\texttt{Authz}_{\texttt{TRM}}$ as a State Machine 27

5.2.1 Stale-Unsafe $\texttt{Authz}_{\texttt{TRM}}$ 27

5.2.2 Strongly Stale-Safe $\texttt{Authz}_{\texttt{TRM}}$ 27

5.2.3 Weakly Stale-Safe $\texttt{Authz}_{\texttt{TRM}}$ 28

Chapter 6: SMV Model of g-SIS Using Real Timestamps 30

6.1 Modeling Events in g-SIS 30

6.1.1 Simultaneous Events 31

6.2 Minimal Clock Size 33

6.3 SMV Modules 35

Chapter 7: Results of Model Checking g-SIS 38

7.1 Small and Large Enforcement Models 38

7.2 Unsafe Enforcement Model 39

7.3 Weak Stale-Safe TRM 41

7.4 Strong Stale-Safe TRM 42

Chapter 8: Verification of Unbounded g-SIS Finite Enforcement Systems 44

8.1 Informal Proof of Small Model Validity 44

Chapter 9: Discussion of General Stale-Safety 47

9.1 Minimal Stale-Safety 47

9.2 Staleness in SAAM 48

Chapter 10: Conclusion 50

10.1 Related Work 50

10.2 Final Thoughts 52

Appendix A: SMV Modules 54

A.1 MAIN.mod 54

A.2 EVENTS.mod 55

A.3 CC.mod 61

A.4 TRM_UNSAFE.mod 63

A.5 TRM_WEAK.mod 65

A.6 TRM_STRONG.mod 68

A.7 USER.mod 70

A.8 question-response.mod 71

A.9 CLOCK.mod 74

A.10 CLOCK1.mod 75

A.11 compile.sh 75

REFERENCES 77

Vita

LIST OF TABLES

Table 6.1 g-SIS System Characterization 33

Table 7.1 TRM_UNSAFE: Weakly stale-safe counterexample 41

Table 7.2 TRM_WEAK: Strongly stale-safe Counterexample 41

LIST OF FIGURES

Figure 2.1 PPE Policy 5

Figure 2.2 Policy Slicing Procedure 7

Figure 2.3 XML Snippet 9

Figure 2.4 Meta-Analysis 10

Figure 3.1 g-SIS Architecture 14

Figure 4.1 Stale-safety Violation 19

Figure 4.2 Stale-safety Formulas 22

Figure 4.3 Stale-safety Illustrations 22

Figure 5.1 Class diagrams for g-SIS with single user and single object. 25

Figure 5.2 Description of g-SIS Methods 26

Figure 5.3 Definition of `TRM.refresh(cc : CC)`. 26

Figure 5.4 Definition of `TRM.request-perform(trm : TRM)`. 27

Figure 5.5 Strong TRM 28

Figure 5.6 Weak TRM 29

Figure 6.1 34

Figure 6.2 34

Figure 9.1 Staleness in SAAM 49

Chapter 1: INTRODUCTION

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 attribute 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 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.

In general it is not practical to eliminate the use of stale attributes for access control decisions. 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 thesis. 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 will be concerned with arises when the authoritative information point knows that the employee has been dismissed but at some decision point the employee's status is still showing as active.

In a theoretical sense, some staleness is inherent in the intrinsic limit of network latencies. We are more interested in situations where staleness is at a humanly meaningful scale, say minutes, hours or days. 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 up to 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? The central contribution of this thesis is to formalize this notion of safe use of stale attributes. This thesis demonstrates specifications of systems that do and do not satisfy this requirement. I believe that the requirements for *stale-safety* identified in this article represent fundamental security properties the need for which arises in secure distributed systems in which the management and representation of authorization state are not centralized. In this sense, it is suggested that this thesis has identified and formalized a *basic security property* of distributed enforcement mechanisms, in a similar sense that non-interference [13] and safety [15] are basic security properties that are desirable in a wide range of secure systems.

Specifically, I present formal specifications of three properties, each strictly stronger than the next. 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 *would* have been authorized at some point in the recent past. The strongest 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.

I believe that the *weak stale-safety* property is a requirement for most actions (*e.g.*, read or write) in distributed access control systems. There are likely situation when the *strong stale-safety* property is (further) required of some or all actions in many applications, for instance, when deciding whether or not to allow modifications to sensitive materials. The specific application domain called group-centric secure information sharing (g-SIS) [18] is used as a running example to illustrate the properties of stale-safety. This thesis formalizes the properties in first-order linear temporal logic (FOTL), as it is a natural choice for supporting unbounded number of users, objects, and groups in an information sharing system.

The stale-safe properties require an enforcement model that may comprise an unbounded number of users, objects, and groups (we call it a large enforcement model) to ensure that any requested action is only performed if that action was authorized during a previous refresh of authorization information. Determining the validity of FOTL properties is, in general, undecidable. To alleviate this problem, a small model theorem is conceived to establish that the proof of FOTL stale-safety properties against a large enforcement model can be reduced to the proof of propositional linear temporal logic (PLTL) properties against a small enforcement model containing only one user and one object within a single group[1]. For the stale-safety properties presented, this is intuitive since any given authorization decision will depend on *only* on a single user, object, and group. This reduction allows the use of the analytical power of model checking [7], which can automatically verify whether a finite model satisfies a PLTL property. The NuSMV [6] model checker is used to obtain an automated proof for the small enforcement model.

This thesis extends the work previously done in [17]. This thesis formally specifies stale-safe security properties in terms of FOTL. A *complete* specification of three enforcement models for the g-SIS system is given: unsafe, weak, and strong. To prove the FOTL stale-safe properties it is shown that reasoning about an enforcement model involving an unbounded number of users, objects, and groups can be achieved by model checking a machine with *only* a single user, single object, and within a single group. Further, this thesis begins to discuss stale-safety in a more general way. And shows that it can be applied to other domains such as SAAM [8, 36].

The remainder of this thesis is organized as follows. Chapter 2 discusses work that I have done to analyze privacy policies, specifically the HIPAA privacy policy. Next Chapter 3 discusses the group-centric secure informationsharing problem, which will be used throughout this thesis to illustrate stale-safe properties. In Chapter 4, the stale-safe security properties are formally specified using FOTL. In Chapter 5, a model of g-SIS is presented which is the basis for the formal SMV

[1]When the carriers used in the interpretation of FOTL formulas are finite, it is possible to convert FOTL formulas into propositional formulas by replacing variables by constants and using conjunction and disjunction to represent universal and existential quantification, respectively.

model described in Chapter 6. Chapter 7 presents the results of model checking and Chapter 8 discusses why it is valid to only check the small enforcement model. Chapter 9 presents a discussion on generalizing stale-safety beyond g-SIS. Finally I discuss related work and conclude in Chapter 10.

Chapter 2: HIPAA PRIVACY POLICY ANALYSIS

This chapter discusses other projects I have been involved in. First I discuss the paper *Privacy Promises That Can Be Kept: A Policy Analysis Method With Application to the HIPAA Privacy Rule* [5] for which I was a contributing author and then I discuss my work on parsing and analyzing the HIPAA privacy policy.

2.1 Privacy Promises That Can Be Kept

Barth et al. [2] introduced the policy specification language Contextual Integrity (CI). Chowdhury et al. [5] made some modifications to CI so that it could adequately capture the HIPAA privacy policy (Health Insurance Portability and Accountability Act). It is beyond the scope of this thesis to discuss the differences between CI and the policy specification language presented in [5]. Instead, I will discuss some broad ideas from CI that are necessary to understand the motivation behind [5].

$$\begin{gathered}\Box \forall p_1, p_2, q : P.\forall m : M.\forall t : T.\mathrm{send}(p_1, p_2, m) \wedge \mathrm{contains}(m, q, t) \rightarrow \\ \bigvee_{\varphi^+ \in \mathrm{norms}^+} \varphi^+ \wedge \bigwedge_{\varphi^- \in \mathrm{norms}^-} \varphi^-\end{gathered}$$

Figure 2.1: Overall structure of the privacy policy language specified in [5].

Figure 2.1 shows the FOTL privacy policy used in [5]. The structure is essentially the same as in CI however we do put some restrictions on the form of the formulae φ^+ and φ^- that are not present in CI. The formula specifies when it is OK to send a message m from entity p_1 to entity p_2 containing attributes t about entity q. One of the key ideas presented in CI is the idea of *positive* and *negative* norms (φ^+ and φ^- respectively). Notice that disjunction is used to combine the positive norms and conjunction is used to combine the negative norms. This reflects the idea that positive norms *allow* an action and negative norms *deny* an action. Said in plain English, *all* negative norms must be satisfied and *at least one* positive norm must be satisfied (thus if any one negative norm *is not* satisfied, the send is not an allowed action).

There are two properties of interest for policies of the form in Figure 2.1: *weak compliance* (WC) and *strong compliance* (SC). An action may be denied if *either* it doesn't satisfy past requirements *or* does not satisfy future obligations. This presents a serious problem for the enforcement of such a policy because it should be clear that there is no way to enforce, at the time one is allowed to send a message, whether or not future obligations allowing that action will be met. This is where WC and SC become important for denying or allowing an action.

Definition 2.1.1 (Weak Compliance)**.** *An action is said to satisfy WC if it currently meets all past requirements and there exists a trace such that future obligations can be met.*

Definition 2.1.2 (Strong Compliance)**.** *An action is said to satisfy SC if it currently meets all past requirements and there is no trace that will prevent the future obligations from being met.*

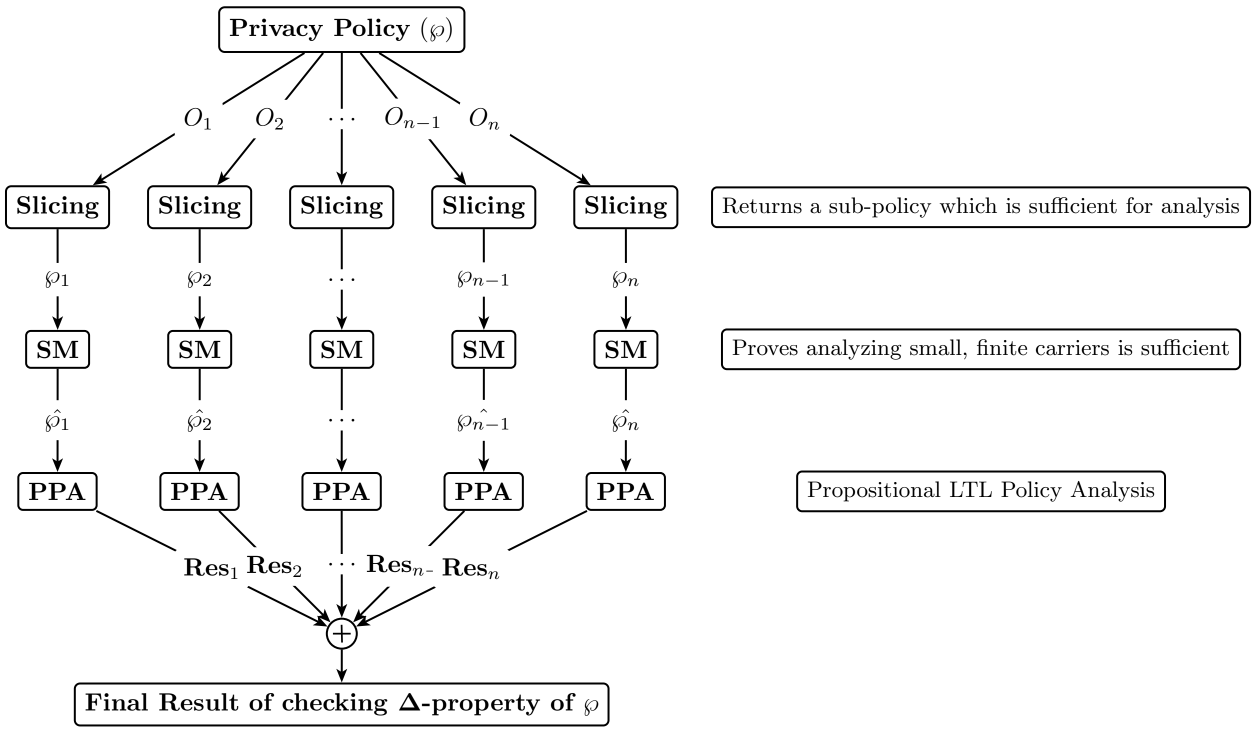
Note that WC is a minimal constraint for allowing an action. If there *does not* exist a trace such that all future obligations can be met, then we know for sure that this action will eventually violate the policy and thus should not be allowed. There is a subtle difference between WC and SC. Consider that a doctor discloses information about a patient to a third party. This action is allowed only if the doctor informs the patient about the disclosure at a later date (a future obligation). However, the doctor is *not* allowed to inform the patient of the disclosure if he feels it will harm the patient.[1]

The original disclosure in the above example satisfies WC but does not satisfy SC. This is because there *is* a way for the doctor to later inform the patient of the disclosure. However it *does not* satisfy SC because the doctor may later decide informing the patient could harm them and thus would not be allowed to inform the patient about the disclosure thus violating the previous future obligation. This is a fairly trivial example and obviously exposes a problem with the original policy. HIPAA has similar rules however instead of two separate norms it is stated as the doctor must inform the patient *unless* this future event occurs (i.e. the doctor feels informing the patient

[1]For instance if the doctor discloses information to law enforcement and later finds out that the patient has become mentally unstable.

would harm them). Now this policy satisfies both WC and SC since the non-disclosure doesn't violate a future obligation if the doctor later decides it would be harmful to the patient.

While the complexity of checking WC is PSPACE, research [3, 11] has shown that it can be checked efficiently in practice. On the other hand SC is undecidable in general. It is with this in mind that we introduced the property that WC entails SC. If a policy has this property then it means we can effectively enforce SC (otherwise we cannot). The rest of this section is devoted to giving an overview of the procedure presented by Chowdhury et al. [5] to check whether or not a policy of the form given in Figure 2.1 has this WC entails SC property.



Privacy Policy (℘)
O_1 O_2 ⋯ O_{n-1} O_n
Slicing
Returns a sub-policy which is sufficient for analysis
$\wp_1$ $\wp_2$ ⋯ $\wp_{n-1}$ $\wp_n$
SM
Proves analyzing small, finite carriers is sufficient
$\hat{\wp_1}$ $\hat{\wp_2}$ ⋯ $\hat{\wp_{n-1}}$ $\hat{\wp_n}$
PPA
Propositional LTL Policy Analysis
$\mathbf{Res}_1$ $\mathbf{Res}_2$ ⋯ $\mathbf{Res}_{n-}$ $\mathbf{Res}_n$
⊕
Final Result of checking Δ-property of ℘

Figure 2.2: Gives an overview of the procedure presented in [5] to decide whether or not a policy has the WC entails SC property.

Figure 2.2 gives a visual overview of the procedure used to determine whether or not a policy has the property WC entails SC. The first step in the procedure to slice the policy. In general privacy policies such as HIPAA are extremely large. Policy slicing breaks this large policy into many smaller policies which are independent of each other. So rather than attempting to analyze the policy as a whole, we only need to analyze each piece received from policy slicing.

Even after policy slicing, we are still left with FOTL formulae. To determine whether or not these FOTL formulae satisfy WC entails SC we must write them as finite LTL formulae. This requires a small model theorem which states that if the finite LTL formula satisfies the property WC entails SC then so too does the full FOTL formula (for each policy slice). For HIPAA, we reasoned that since there was no interaction between any two entities or any two messages, that we could analyze an LTL formula which only instantiated a single sending entity, a single receiving entity, a single subject, and a single message. In this way we were able to show that the HIPAA privacy policy indeed does have the property that WC entails SC.

2.2 Automatic Analysis of Federal Laws

This section describes work I did as an independent study to automate the generation of formal policies directly from the HIPAA text. Currently some portions of HIPAA have been translated to LTL policies by hand. These documents are stored in marked up formats (such as PDF) and are not amenable to automated analysis (to test a policy, it must be transcoded by hand into a suitable form such as SMV modules). Further, such analysis is highly prone to human error (both the initial creation of LTL policies and secondary transformations).

2.2.1 Structure of Federal Laws

All federal laws are defined by the Office of the Federal Register. There are 50 CFRs (**C**ode of **F**ederal **R**egulations) which represent broad areas subject to federal regulation. Each **CFR** is divided into many **parts** that are grouped into **subchapters** which are further divided into **subparts** and finally **rules**. Each subpart generally has two special rules, 1) *Applicability* and 2) *Definitions*, followed by rules describing the regulations (to be applied as defined in the Applicability rule).

As an example, HIPAA (Health Insurance Portability and Accountability Act) is contained in 45 CFR Parts 160, 162, and 164 (45 CFR contains regulations regarding public welfare). We are mainly interested in 45 CFR Part 164, Subpart E which specifies guidelines governing individually identifiable health information.

2.2.2 Getting the Data

All federal laws are available in both HTML and XML formats from the following website:

http://www.ecfr.gov/cgi-bin/ECFR

However, I initially gathered data for HIPAA from HTML text obtained from the following website:

http://www.hipaasurvivalguide.com/hipaa-regulations/hipaa-regulations.php

```
...
<Rule number="164.514(f)(1)">
    <Description>
    Standard: Uses and disclosures for fundraising. Subject to
    the conditions of paragraph (f)(2) of this section, a covered
    entity may use, or disclose to a business associate or to an
    institutionally related foundation, the following protected
    health information for the purpose of raising funds for its
    own benefit, without an authorization meeting the requirements
    of 164.508:
    </Description>
    <Rule number="164.514(f)(1)(i)">
        <Description>
        Demographic information relating to an individual,
        including name, address, other contact information, age,
        gender, and date of birth;
        </Description>
    </Rule>
    ...
</Rule>
...
```

Figure 2.3: Snippet from XML generated from HIPAA privacy rule.

Figure 2.3 shows a snippet from the generated XML file. Looking at the rule number, it can be seen that this contains a portion of subpart 164, rule 514. Even further, this comprises a portion of the sixth (letter `f`) paragraph from section `164.514` which itself is divided into several smaller parts.

2.2.3 Analyzing Meta-Data

The XML snippet in Figure 2.3 shows that many rules reference other rules. For example rule `164.514(f)(1)` describes *what* information may be shared while rule `164.514(f)(2)` de-

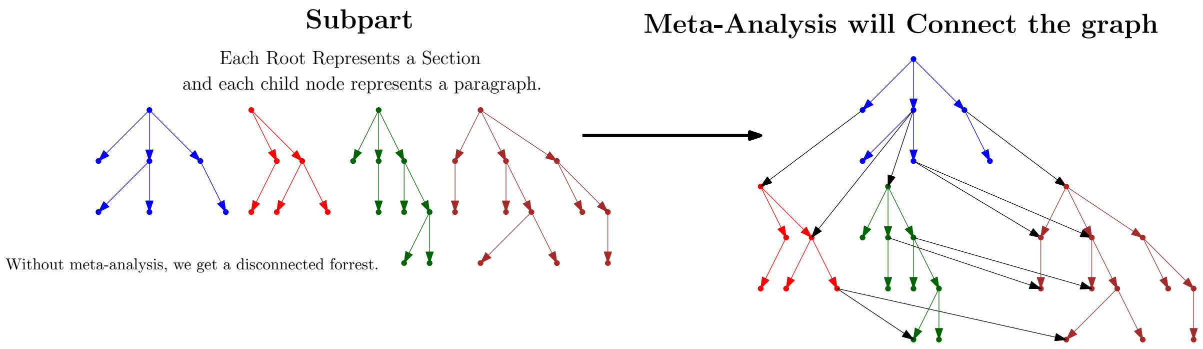

Subpart
Each Root Represents a Section
and each child node represents a paragraph.
Without meta-analysis, we get a disconnected forrest.
Meta-Analysis will Connect the graph

Figure 2.4: Figure showing hypothetical results of meta-analysis.

scribes *when* this information may be shared (i.e. "Subject to the conditions of paragraph `(f)(2)` of this section"). Furthermore rule `164.514(f)(1)` describes an exception to section `164.508` (which describes the authorization required to share various types of protected health information).

The initial goal is to create a dependency graph for every rule in HIPAA. Figure 2.4 shows a hypothetical result from such an analysis. Since each rule is uniformly labeled, a dependence can be gleaned from the mere mention of a rule. For instance, the rules found in `164.514(f)(1)` have a clear dependence on the rules found at `164.514(f)(2)` (since the description of `164.514(f)(1)` mentions `(f)(2)`).

This analysis should partition the law into several branches each applying to different scenarios. The branches would likely have some overlap and so would not be disjoint. The hope would be that this could help to generate test cases as well as isolate more complex parts of the law. These test cases could then be used to generate models suitable for formally model checking the law. It may even be possible to automate the construction of these models.

Obviously this initial analysis would incorrectly assume that rule `164.514(f)(1)` depends on section `164.508`. Refining the initial analysis requires a less naive approach. I believe a natural language parser may aid in giving a more correct analysis.

2.2.4 Future Work

The above meta-analysis does not attempt to understand the law in any way. With the help of a natural language parser it may be possible to convert the English presentation into formal privacy policies. This is likely possible because the laws are written in a very structured language to begin with so it may not require a very sophisticated language parser.

Chapter 3: g-SIS

Let us first discuss a simple distributed system that will be used in the following chapters as a first illustration of the stale-safety problem. We will consider the problem of information sharing amongst a group of authorized users. This framework will be referred to as group-centric secure information sharing or g-SIS [19].

The g-SIS model consists of groups of users and objects. A user ultimately represents a human being but the term user may be used to describe a process running on behalf of a certain human actor in the system. An object represents information. Users and objects may be added to and removed from each of the various groups. A user's authorization to access an object holds at any time depending on the relative membership states of the user and the object in question. For example, a user might be authorized to access an object only if both the user and object are current members in the group such as is the case with the traditional notion of group authorization in many operating systems. We could have an additional constraint that the user should be a current member in the group when the object is added, e.g. the authorization policy used in many secure multicast applications. Formal models for g-SIS have been studied in detail [18]. We consider an example architecture that can enforce g-SIS policies.

3.1 Objectives

The following are the characteristics of the g-SIS application.

- Group membership is expected to be dynamic. That is, a user may join and leave and an object may be added and removed multiple times.
- A server, called the Control Center (CC), facilitates operation of the system. It maintains attributes of various entities in the system such as membership status of each user and object in each group. The CC acts as a PIP.
- Authorization decisions can be made offline. That is, for every access attempt, the CC

need not be involved for making the access decision. Clearly, an appropriate client-side enforcement mechanism is required to enforce group policy. To this end, we assume a user-side Trusted Reference Monitor (TRM) to enforce group policies in a trustworthy manner. The TRM caches authorization information such as user and object attributes (e.g., user join time, leave time, etc.) locally on the user's access machine (a computer with an appropriate TRM) and refreshes them periodically with the CC server. The TRM acts as both a PDP and PEP.

- Objects are made available via super-distribution. In the super-distribution approach, protected group objects (encrypted with a group key for instance) are released into the cloud (cyber space). Users may obtain such objects from the cloud and may access them if authorized. For instance, a user may directly email a group object to another user or transfer it via a USB flash drive. Thus objects need not be downloaded from the CC for each access attempt. A group key is provisioned on users' access machines in such a manner that only a TRM may access the key to encrypt and decrypt group objects. (See the trusted computing group's initiative [35] for example.) The TRM faithfully enforces group policies based on user and object attributes.

3.2 Enforcement Model for g-SIS

Figure 3.1 shows one possible enforcement model for g-SIS and illustrates the interaction of various entities in g-SIS. A Group Administrator (GA) controls group membership. The Control Center (CC) maintains authorization information (e.g. attributes of group users and objects) on behalf of the GA.

- *User Join*: Joining a group involves obtaining authorization from the GA followed by obtaining group attributes from the CC. In step 1.1, the user contacts the GA using an access machine that has an appropriate TRM and requests authorization to join the group. The GA authorizes the join in step 1.2 (by setting AUTH to TRUE). The TRM furnishes the authorization to join the group to the CC in step 1.3 and the CC updates the users JoinTS in step

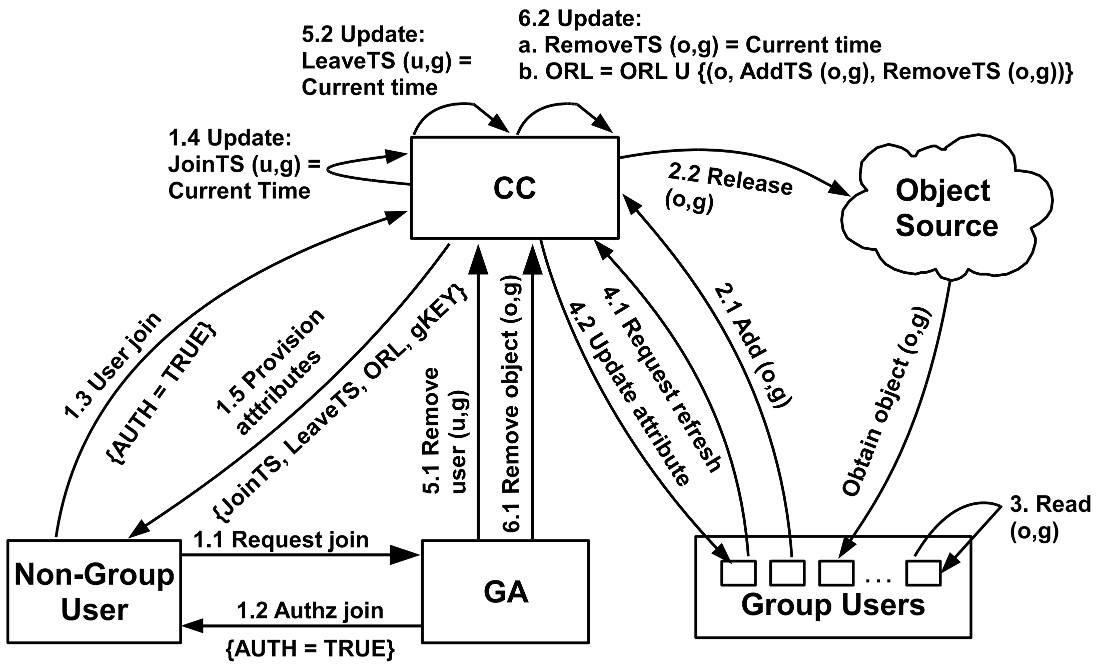


Figure 3.1: An example architecture of a g-SIS System.

1.4. In step 1.5, the CC verifies GA's authorization and issues the attributes. JoinTS is the timestamp of user join (set to a valid value), LeaveTS is the time at which a user leaves the group (initially set to time of join), gKey is the group key specifying which group objects can be decrypted, ORL is the Object Revocation List which lists the objects removed from the group. We assume that these attributes may be accessed and modified only by the TRM and not by any other entity in the user's access machine.

- *Policy Enforcement*: From here on, the user is considered a group member and may start accessing group objects (encrypted using the group key) as per the group policy and using the attributes obtained from the CC. This is locally mediated and enforced by the TRM. Since objects are available via super-distribution and because of the presence of a TRM on user's access machines, objects may be accessed offline conforming to the policy. The TRM does not have to contact the CC to make an access decision, rather, the TRM will use the authorization information cached and refreshed most recently on the local TRM to enforce the policy every time. For example, the TRM may enforce the policy that the user is allowed to access only those objects that were added after she joined the group and disallow access to objects added before her join time. Further users may lose access to all objects after leaving the group. Such decisions can be made by comparing the join and leave timestamps of user, add and remove timestamps of object. Objects may be added to the group by users by obtaining an add timestamp from the CC. The CC approves the object, sets the AddTS and releases the object into the cloud (steps 2.1 to 2.2). We assume that the AddTS attribute that reflects the time of the last add is embedded in the object itself. However, the $\mathrm{RemoveTS}$ cannot be embedded in the object because when the object is removed every copy of the object would have to be found and modified. This is not feasible due to the offline access nature of the application. Instead, an Object Revocation List (ORL) with elements of the form $(\mathrm{o}, \mathrm{AddTS}, \mathrm{RemoveTS})$ is provisioned to the access machine. Note that the ORL lists the objects removed from the group and it is required to maintain the triple since the same object could be removed and re-added.

- *Attribute Refresh*: Since users may access objects offline, the TRM needs to refresh attributes with the CC periodically (steps 4.1-4.2). How frequently this is done is a matter of policy and/or practicality. In certain scenarios, frequent refreshes in the order of milliseconds may be feasible while in others refreshes may occur only once a day.

- *Administrative Actions*: The GA may have to remove a user or an 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 setting the user's $\mathrm{LeaveTS}$ attribute in step 5.2. In the case of object removal, the ORL is updated with the object's AddTS and $\mathrm{RemoveTS}$ (steps 6.1-6.2). These attribute updates are communicated to the user's access machine during the refresh steps 4.1 and 4.2.

As one can see, there is a delay in attribute update in the access machine that is defined by the refresh window. Although a user may be removed from the group at the CC, the TRM may let users access objects until a refresh occurs. This is due to attribute staleness that is inherent to any distributed system. I discuss this topic in detail in the subsequent chapters. Note that building trusted systems to realize the architecture in figure 3.1 is well-studied in the literature[1] and is outside the scope of this article. The above system is an instantiation of a more general system in which the policy information point (in this case the CC) and policy enforcement and decision points (in this case the TRM is both the PDP and PEP) are decentralized.

[1]See related work on trusted computing (http://www.trustedcomputinggroup.org) for example.

Chapter 4: STALENESS IN g-SIS

As discussed earlier, in a distributed system, access decisions are almost always based on stale attributes which may lead to critical access violations. In practice, eliminating staleness completely may not be feasible but a realizable notion of bounding staleness can be conceived. In regards to access control decisions, the *principle of stale-safety* states that when it is necessary to rely upon stale authorization information, if the user is granted access to an object, the authorization to access that object should have *definitely held* in the recent past.

In this chapter, I discuss a scenario in which stale attributes lead to access violations in g-SIS and then informally discuss the stale-safe properties.

4.1 System Characterization

The g-SIS system consists of users and objects, trusted access machines with TRMs, a GA and a CC. Access machines maintain a local copy of user attributes which they refresh periodically with the CC. AddTS is part of the object itself. A removed object is listed in the ORL which is provided to access machines as part of a refresh. For the purpose of this illustration, let us assume that each user is tied to an access machine from which objects are accessed and there is a single GA and single CC per group. Further let us assume a group policy in which a user is allowed to access an object as long as both the user and object are current members of the group and the object was added after the user joined the group. Thus the g-SIS system can be characterized as follows:

User attributes	$\{\mathrm{JoinTS}, \mathrm{LeaveTS}\}$
Object attributes	$\{\mathrm{AddTS}, \mathrm{RemoveTS}\}$
Group attributes	$\{\mathrm{gKey}, \mathrm{ORL}\}$
Access Policy	$\mathrm{JoinTS}(\mathtt{u}, \mathtt{g}) \leq \mathrm{AddTS}(\mathtt{o}, \mathtt{g}) \wedge$
	$\mathrm{LeaveTS}(\mathtt{u}, \mathtt{g}) < \mathrm{JoinTS}(\mathtt{u}, \mathtt{g}) \wedge$
	$(\mathtt{o}, \mathrm{AddTS}(\mathtt{o}, \mathtt{g}), \mathrm{RemoveTS}(\mathtt{o}, \mathtt{g}))$
	$\notin \mathrm{ORL}(\mathtt{g})$

in which u, o, and g represent a specific user, object, and group, respectively. For simplifying the presentation we omit the parameters when it is clear from the context.

4.2 Example of Staleness

Figure 4.1 shows a timeline of events involving a single group that can lead to an access violation due to stale authorization information. 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. The first window is RW_0 (where u1 joins), 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 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. A local access decision will be made by the TRM based on the attributes obtained at the latest RT. If the TRM allows u1 to read o2, this would violate the principle of stale-safety.

The TRM's access policy would allow access to both o1 and o2 (since o2 is added after u1 joined and the TRM is not aware that u1 has left the group). Ideally, u1 should not be allowed to access *either* of o1 or o2. However note that u1 *was* authorized to access o1 in RW_3 whereas was u1 was *never* authorized to access o2 (in any refresh window).

Furthermore, from a confidentiality perspective in information sharing, granting u1 access to o1 even if u1 had left the group is relatively less of a problem than granting access to o2. This is because u1 was authorized to access o1 some time in the past and hence the TRM may assume that o2's information has already been released to u1. However, there is never a time that u1 was authorized to access o2 and allowing u1 access to o2 means that u1 may gain knowledge of information that u1 was never authorized to receive. This is a critical violation and should not be allowed. In summary, it may be OK for a user to access information that the user once had authorization to access (e.g. o1) but it is *never* OK to allow a user to access an object that the user was never authorized to access (e.g. o2).

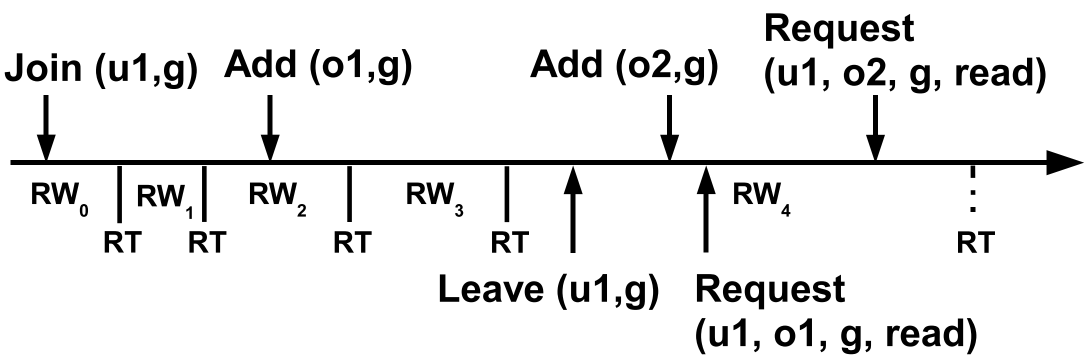


Figure 4.1: A possible system trace allowing u1 to access o2 will violate the Principle of Stale-Safety.

4.3 Formal Property Specification in g-SIS

In this section I use first-order linear temporal logic (FOTL) to specify stale-safety properties of varying strength for g-SIS. FOTL differs from the familiar propositional linear temporal logic [27] by incorporating predicates with parameters, constants, variables, and quantifiers. 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 $\vee$). The future temporal operator $\Box$ (read henceforth) represents all future states. For example, formula $\Box p$ means that p is true in all future states. Some of the past operators are $\ominus$ and $\mathcal{S}$ (read previous and since respectively) have the following semantics. Formula $\ominus p$ means that p was true in the previous state. Note that $\ominus p$ is false in the very first state. The formula $(p\,\mathcal{S}\,q)$ means that q has happened sometime in the past and p has held continuously following the last occurrence of q to the present.

4.4 Predicates

Our formalization of stale-safe properties uses the following parameterized predicates:

`request`(u,o,g,op)	User `u` requests to perform an operation `op` on object `o` in group `g`.
$\texttt{Authz}_{\texttt{CC}}$(u,o,g,op)	The central access policy (as seen by the CC) authorizing user `u` to perform an operation `op` on object `o` in group `g`.
`join`(u,g)	User `u` joins group `g`.
`leave`(u,g)	User `u` leaves group `g`.
`add`(o,g)	Object `o` is added to group `g`.
`remove`(o,g)	Object `o` is placed into the ORL for group `g`.
`perform`(u,o,g,op)	User `u` performs operation `op` on object `o` in group `g`.
`RT`(u,g)	TRM synchronizes its attributes for user `u` and group `g` with the CC.

From here on we do not explicitly write the parameters–it should be understood that the above predicates require a specific user, group, object, and operation.

4.5 Access Policy Specification

We first specify the example access policy discussed in chapter 4.1 using FOTL and label it as $\texttt{Authz}_{\texttt{CC}}$. 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 the CC at subsequent refresh times (`RT`'s). $\texttt{Authz}_{\texttt{CC}}$ is the centralized access policy that assumes instant propagation of authorization information contrasted with $\texttt{Authz}_{\texttt{E}}$, the access policy that the TRM enforces. The TRM makes a best effort to enforce $\texttt{Authz}_{\texttt{CC}}$. We argue that a minimal requirement for a best effort is to enforce stale-safety.

Figure 4.3a shows $\texttt{Authz}_{\texttt{CC}}$–a FOTL representation of a group policy (chapter 4.1) in a centralized setting. It states that user `u` is allowed to perform an operation `op` on object `o` if `o` was added to group `g` sometime in the past and both `u` and `o` have not left the group since `o` was added. It also requires that the user joined the group prior to when the object was added. As the name implies, $\texttt{Authz}_{\texttt{CC}}$ can only be enforced by the CC and not by the TRM. This is because the `leave` and `remove` events at the CC are not visible to the TRM until the next refresh. When a request is received, the TRM does not know what events have happened at the CC since the TRM's last refresh of attributes.

Next, I specify two stale-safe security properties of varying strength. The weakest of the properties 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.

4.6 Weak Stale-safe Security Property

Let us introduce two formulas, φ_1 and φ_2 (see figure 4.2), formalizing pieces of stale-safe security properties for g-SIS.

Figure 4.3b illustrates formula φ_1. Formula φ_1 has three requirements:

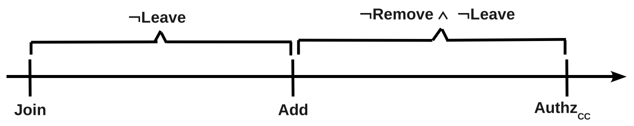
1. There is no `perform` since `request`.

2. *All* `RT`'s since `request` indicate that $\texttt{Authz}_{\texttt{CC}}$ is true.

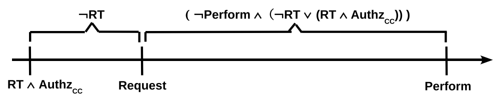
3. The last `RT` prior to `request` indicated that $\texttt{Authz}_{\texttt{CC}}$ was true.

$$\begin{aligned}
\texttt{Authz}_{\texttt{CC}}(\texttt{u},\texttt{o},\texttt{g}) \equiv & (\neg\texttt{remove} \wedge \neg\texttt{leave})\,\mathcal{S}\,(\texttt{add} \wedge (\neg\texttt{leave}\,\mathcal{S}\,\texttt{join})) \\
\varphi_1(\texttt{u},\texttt{o},\texttt{g},\texttt{op}) \equiv & \ominus((\neg\texttt{perform} \wedge (\neg\texttt{RT} \vee (\texttt{RT} \wedge \texttt{Authz}_{\texttt{CC}})))\,\mathcal{S} \\
& (\texttt{request} \wedge (\neg\texttt{RT}\,\mathcal{S}\,(\texttt{RT} \wedge \texttt{Authz}_{\texttt{CC}})))) \\
\varphi_2(\texttt{u},\texttt{o},\texttt{g},\texttt{op}) \equiv & \ominus((\neg\texttt{perform} \wedge \neg\texttt{RT})\,\mathcal{S} \\
& (\texttt{RT} \wedge \texttt{Authz}_{\texttt{CC}} \wedge \\
& ((\neg\texttt{perform} \wedge (\neg\texttt{RT} \vee (\texttt{RT} \wedge \texttt{Authz}_{\texttt{CC}})))\,\mathcal{S}\,\texttt{request})))
\end{aligned}$$

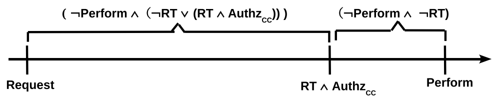
Figure 4.2: Formulas for specifying pieces of stale-safety in g-SIS as well as a definition of $\texttt{Authz}_{\texttt{CC}}$.



(a) Ideal Access Policy ($\texttt{Authz}_{\texttt{CC}}$).



(b) Formula φ_1.



(c) Formula φ_2.

Figure 4.3: Illustration of various formulas for g-SIS.

All three of these requirements must be met to satisfy φ_1. Thus even if the latest refresh happens *after* `request` *and* indicates that $\texttt{Authz}_{\texttt{CC}}$ is true, φ_1 is not satisfied if the last `RT` prior to `request` indicated that $\texttt{Authz}_{\texttt{CC}}$ was false. On the other hand, even if the last `RT` prior to `request` indicated that $\texttt{Authz}_{\texttt{CC}}$ was true, φ_1 is not satisfied if there is *any* `RT` after the request indicating that $\texttt{Authz}_{\texttt{CC}}$ is false.

Figure 4.3c illustrates formula φ_2 for g-SIS. Again, it requires that there is no `perform` since `request` and that *all* `RT`'s since `request` indicate that $\texttt{Authz}_{\texttt{CC}}$ is true. There are two distinctions between φ_1 and φ_2: 1) φ_2 *requires* at least one `RT` between `request` and `perform`[1]–φ_1 does not; and 2) φ_2 does *not* require that there was an `RT` prior to `request` indicating that $\texttt{Authz}_{\texttt{CC}}$ was true–φ_1 does.

Definition 4.6.1 (Weak stale-safety)**.** *A g-SIS enforcement model has the* weak stale-safe security

[1]Recall that φ_2 requires $\texttt{Authz}_{\texttt{CC}}$ be true at the required `RT` and all others between `request` and `perform`.

property *if it satisfies the following FOTL formula:*

$$\forall u \in \mathcal{U}, o \in \mathcal{O}, g \in \mathcal{G}, op \in \mathcal{P} :$$
$$\Box(\mathtt{perform} \rightarrow \varphi_1 \vee \varphi_2)$$

Recall that both φ_1 and φ_2 require that all `RT`'s between `request` and `perform` indicate that $\mathtt{Authz}_{\mathtt{CC}}$ is true. However, φ_1 requires an `RT` ***prior*** to `request` while φ_2 does not. Therefore by combining the two formulae in definition 4.6.1, `perform` is allowed even if there is no `RT`[2] prior to `request` so long as there exists at least one `RT` between `request` and `perform` *and* all such `RT`'s indicate that $\mathtt{Authz}_{\mathtt{CC}}$ is true. Similarly φ_2 ***requires*** an `RT` between `request` and `perform` while φ_1 does not. Thus definition 4.6.1 allows `perform` even if there are no `RT`'s between `request` and `perform` so long as the last `RT` prior to `request` indicated that $\mathtt{Authz}_{\mathtt{CC}}$ was true.

4.7 Strong Stale-Safe Security Property

This property is strictly stronger than weak stale-safety. For this reason, and because, unlike weak stale-safety, it is a reasonable requirement for higher assurance systems, it is given it a second name.

Definition 4.7.1 (Strong stale-safety)**.** *A g-SIS enforcement model has the* strong stale-safe security property *if it satisfies the following FOTL formula:*

$$\forall u \in \mathcal{U}, o \in \mathcal{O}, g \in \mathcal{G}, op \in \mathcal{P} :$$
$$\Box(\mathtt{perform} \rightarrow \varphi_2)$$

Again recall that φ_2 *requires* at least one `RT` between `request` and `perform` *and* that all such `RT`'s indicate that $\mathtt{Authz}_{\mathtt{CC}}$ is true. Thus while weak stale-safety allows authorization to

[2]Precisely, there need not be an `RT` prior to `request` indicating that $\mathtt{Authz}_{\mathtt{CC}}$ was true.

be verified either prior to or after the request, strong stale-safety mandates that authorization is verified after the request and before performing an operation.

Chapter 5: MODELING g-SIS

This chapter provides a model for g-SIS. First I give a class diagram to show how the CC, TRM, and USER interact. Then I use finite state machines to show how $\texttt{Authz}_{\texttt{TRM}}$ is defined for the unsafe, weakly safe, and strongly safe TRMs.

5.1 g-SIS Class Diagrams

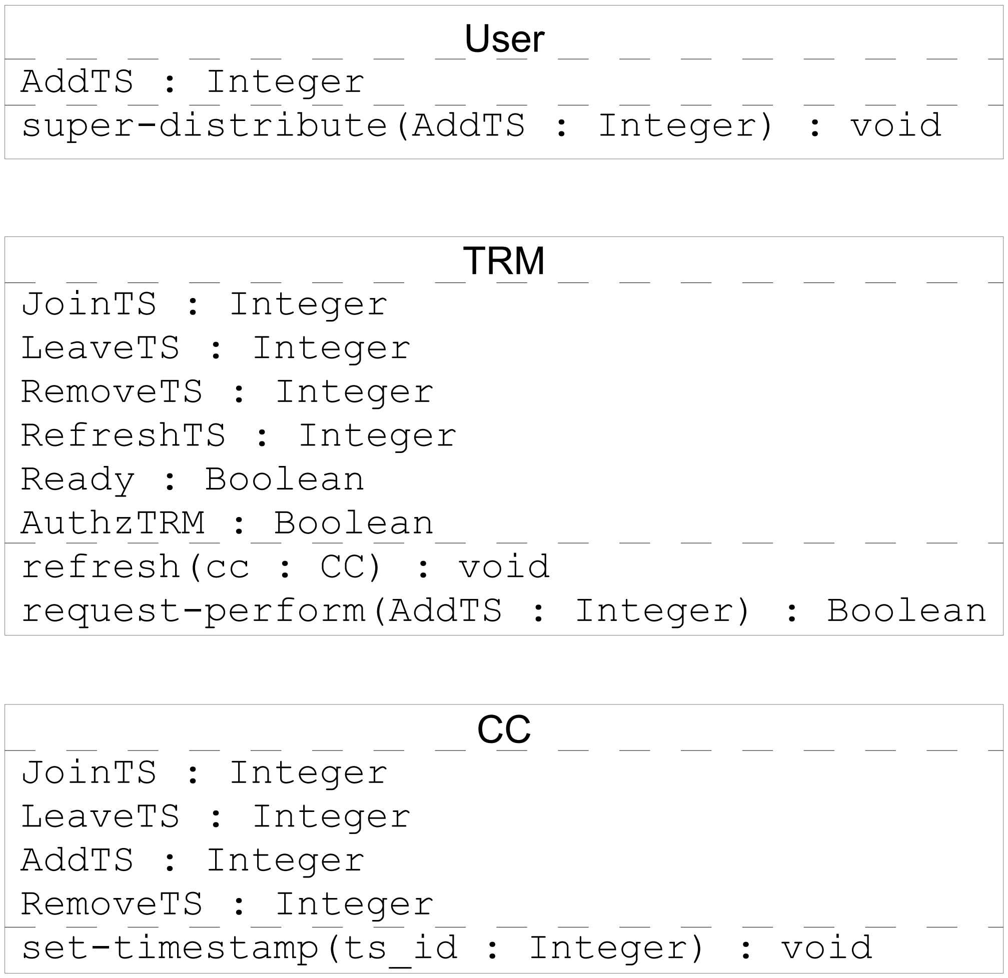


Figure 5.1: Class diagrams for g-SIS with single user and single object.

I model a total of seven events which each trigger a corresponding method call in the class diagram given in figure 5.1: 1) `user_join`, 2) `user_leave`, 3) `object_add`, 4) `object_remove`, 5) `refresh`, 6) `request_perform`, and 7) `super_distribute`. Figure 5.2 gives a brief description of the methods found in figure 5.1. The methods `set-timestamp` and `super-distribute` are very simple: a timestamp is simply set. The add and remove events for the user/object trigger calls to `set-timestamp` with different `ts_ids` depending on which event, `object_add`, `object_remove`, `user_join`, or `user_leave`, occurs. The only restriction on `super-distribrut` is that the parameter AddTS should be a value generated from the CC after some `object_add` event occurrs.

`set-timestamp`	Sets specified timestamp.
`super-distribute`	Sets `AddTS` used by the user for requests.
`refresh`	Refreshes TRMs attributes.
`request-perform`	Requests the TRM to perform and returns the decision.

Figure 5.2: Description of methods.

The last two methods are more involved. Figures 5.3 and 5.4 show how `TRM.refresh` and `TRM.request-perform` are defined, respectively. Note that the TRM relies on the CC to produce a timestamp for `RefreshTS` [1] that can be compared to other timestamps generated by the CC.

```
TRM.refresh(cc : CC) : void
1   JoinTS = cc.JoinTS
2   LeaveTS = cc.LeaveTS
3   RemoveTS = cc.RemoveTS
4   // set RefreshTS
5   cc.set-timestamp(RefreshTS_ID)
```

Figure 5.3: Definition of `TRM.refresh(cc :  CC)`.

Note that in figure 5.4, the TRM waits to respond to the request. The field `Ready` is considered a free variable and thus the TRM responds after a request is made once the event "becomes ready" occurs (at the TRM). This is a key feature of this model design for the TRM. This allows us to model that some time has elapsed between the request to perform and the actual perform (if allowed by the TRM).

[1]`RefreshTS` is set on line 5 in figure 5.3.

```
TRM.request-perform(AddTS : Integer) : Boolean
1   Ready = False
2   when Ready is True
3      return AuthzTRM
```

Figure 5.4: Definition of `TRM.request-perform(trm : TRM)`.

5.2 `Authz`$_{\mathtt{TRM}}$ as a State Machine

The previous section gave a rough overview of how the CC responds to various events and how the TRM and User interact. It is the definition of `AuthzTRM` that we are most interested in because this definition dictates whether or not this model enforces the g-SIS policy and/or whether or not it exhibits stale-safety.

5.2.1 Stale-Unsafe `Authz`$_{\mathtt{TRM}}$

In our example of a stale-unsafe TRM, `AuthzTRM` is simply the boolean evaluation inside the TRM:

$$\mathtt{Authz}_{\mathtt{E}} \equiv \mathtt{AddTS} > \mathtt{RemoveTS} \wedge \mathtt{JoinTS} > \mathtt{LeaveTS} \wedge \mathtt{AddTS} > \mathtt{JoinTS}$$

where `AddTS` is the super distributed timestamp given to the TRM when `request-perform` is called. The evaluation $\mathtt{Authz}_{\mathtt{E}}$ is used in the state machines for the weakly stale-safe and strongly stale-safe definitions of `AuthzTRM`.

5.2.2 Strongly Stale-Safe `Authz`$_{\mathtt{TRM}}$

The strongly stale-safe TRM, STRM, is very simple: there must be a refresh after the initial request. This is reflected in figure 5.5 by the fact that `AuthzTRM` is `False` if STRM is in either the state **No RT** (the initial state) or **Deny** and `AuthzTRM` is only true when STRM is in the **Allow** state. Thus, if the TRM becomes ready to respond, the value of `AuthzTRM` depends on what state STRM

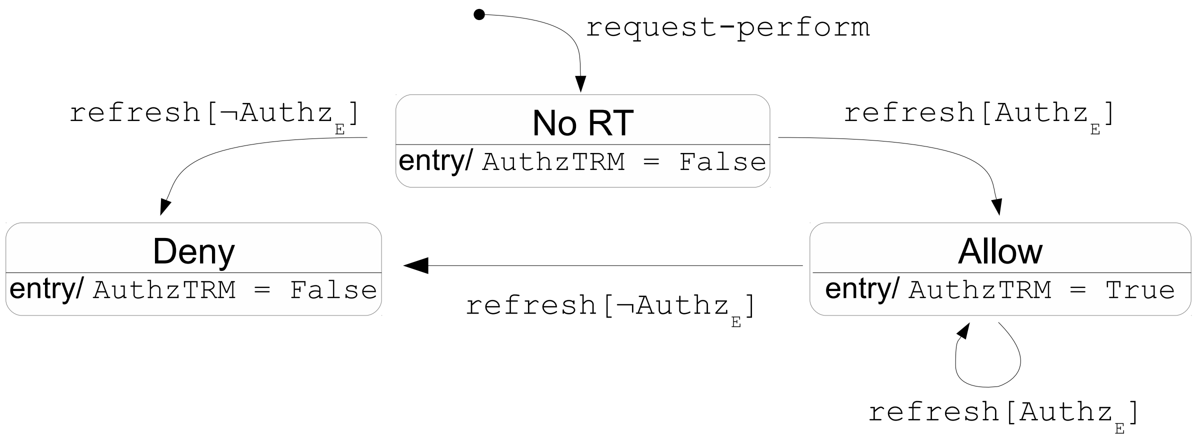


Figure 5.5: Example of a strongly stale-safe definition of AuthzTRM.

currently occupies.

5.2.3 Weakly Stale-Safe $\texttt{Authz}_{\texttt{TRM}}$

The weakly stale-safe TRM, WTRM, builds upon the strongly stale-safe TRM, STRM. The major difference between WTRM and STRM is that WTRM *can* allow a perform without an additional refresh. This splits the initial **No RT** state into two states: the **No RT/Deny** and **No RT/Allow** initial states. Otherwise WTRM is identical to STRM. The value stale can be computed from the provided AddTS and the TRMs RefreshTS value:

$$\texttt{stale} \equiv \texttt{AddTS} > \texttt{RefreshTS}$$

The condition $\neg$stale requires that the object attempting to be accessed was added prior to the last refresh. This, along with the fact that $\texttt{JoinTS} < \texttt{AddTS}$ if $\texttt{Authz}_{\texttt{E}}$ is True, ensures weak stale-safety when no subsequent refresh occurs (after the request is made). Figure 5.6 shows that if the TRM becomes ready to respond, AuthzTRM will be False if WTRM is in the **No Reply/Deny** or **Deny** state and AuthzTRM will be True if WTRM is in the **No Reply/Allow** or **Allow** state.

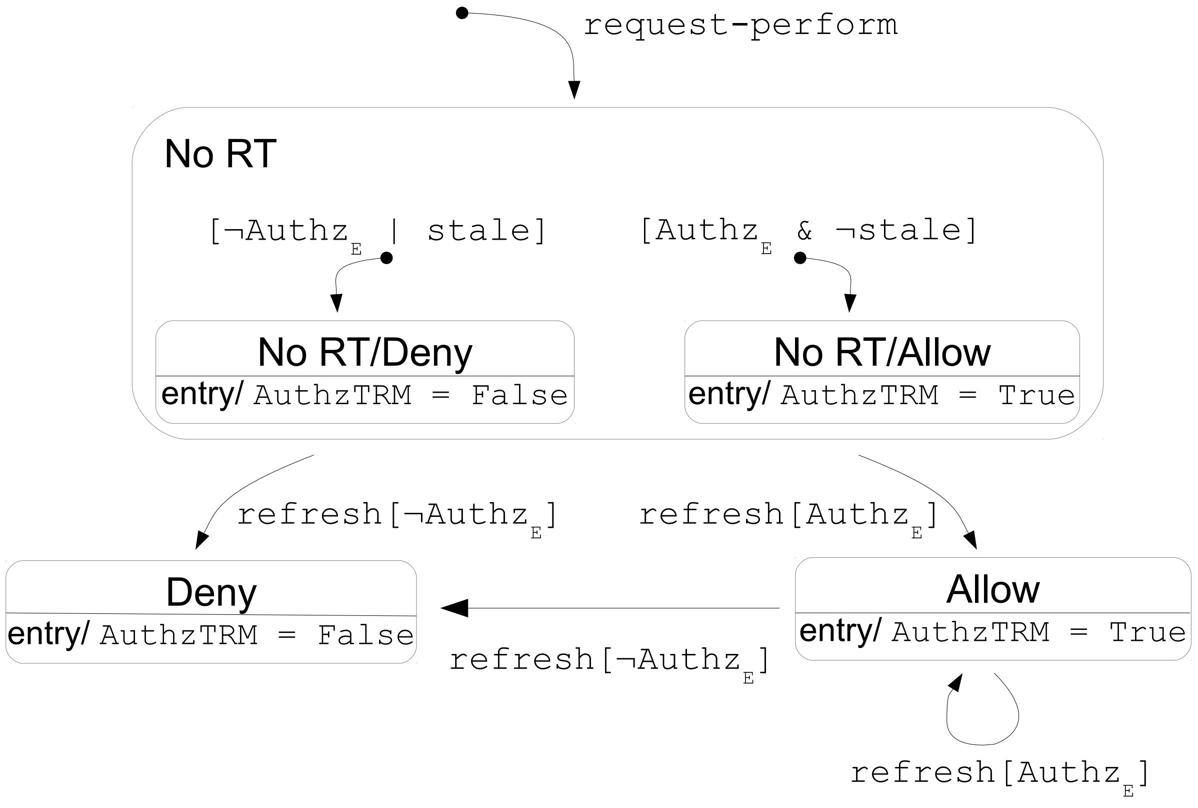


Figure 5.6: Example of a weakly stale-safe definition of `AuthzTRM`.

Chapter 6: SMV MODEL OF g-SIS USING REAL TIMESTAMPS

There at least two possible approaches for attempting to model g-SIS using SMV modules. The choice is how to keep track of timestamps.[1] We can either use *real* timestamps or *relative* timestamps. Since an SMV model must be finite, using real timestamps limits us to a finite clock; that is after it reaches its maximum value, the model stops transitioning (time essentially stops). If, instead it is really the *order* of events that we care about then we may use relative timestamps to provide an ordering. The remainder of this chapter discusses how a model using real timestamps can be realized.

6.1 Modeling Events in g-SIS

For the most part, the seven events considered, 1) `user_join`, 2) `user_leave`, 3) `object_add`, 4) `object_remove`, 5) `refresh`, 6) `request_perform`, and 7) `super_distribute`, are considered free variables. However in modeling these events, I do impose some realistic constraints. Namely a "remove" action *cannot* occur unless an "add" action previously occured and vice versa.[2] Further "add"/"remove" events are not allowed to happen simultaneously[3]–not to be confused with the fact that events modifying the user/object can happen simultaneously with events modifying the object/user. Further I restrict the event `super_distribute` such that it can *only* occur *after* an object has been added and *cannot* occur while the user is awaiting a response from the TRM; i.e. the user *cannot* change the value of the AddTS midway through the evaluation of the policy by the TRM.[4] Finally a `request_perform` cannot occur while the user is awaiting a decision from the TRM (the user cannot make multiple

[1]Another approach is to abstract away the timestamps altogether.

[2]Specifically this means that a `remove_object` event cannot occur unless a `add_object` previously occurred and, likewise, a `user_leave` event cannot occur unless a `user_join` event has occurred and vice versa.

[3]Because a user/object must initially be "unjoined", allowing such events to happen simultaneously would necessitate that when they *do* happen simultaneously that the "remove" event happened second; but this is identical to no "add" event happening in the first place.

[4]After the object has been added, `super_distribute` can occur at any time so long as the user isn't awaiting a response.

requests simultaneously). When I use real timestamps, with a finite clock, the final restriction is that *no* events which generate timestamps are allowed to occur after the clock stops. This includes all events except `request_perform`. Notice that the last restriction is the *only* restriction on the event `refresh` (i.e. the event `refresh` is nearly completely free).

6.1.1 Simultaneous Events

When events occur in the same state, the following shows the order in which they happen:

1. `user_join/leave`
2. `object_add/remove`
3. `super_distribute`
4. `request_perform`
5. `refresh`

The rest of this subsection discusses why the LTL formulas dictate this specific ordering. The very first thing to note is that the events `add_object`, `super_distribute`, and `request_perform` are allowed to happen all in the same state. There is only a single way to interpret such a state which is that the events happen in the order stated (since `super_distribute` can only occur after `add_object` and `request_perform` can only occur after `super_distribute`).

The next two interesting sets of events to consider are when `user_join` and `object_add` occur simultaneously. Consider the following sequence of events: first the event `user_leave` occurs followed by simultaneous events of `user_join` and `object_add`. This will result in LeaveTS $<$ AddTS $=$ JoinTS. Now consider what $\texttt{Authz}_{\texttt{CC}}$ should be in this state according to $\texttt{Authz}_{\texttt{CC}} = (\neg\texttt{remove} \wedge \neg\texttt{leave})\,\mathcal{S}\,(\texttt{add} \wedge (\neg\texttt{leave}\,\mathcal{S}\,\texttt{join}))$. We find that $(\texttt{add} \wedge (\neg\texttt{leave}\,\mathcal{S}\,\texttt{join}))$ is true in this state and thus $\texttt{Authz}_{\texttt{CC}}$ must be be true. Thus we find that when JoinTS $>$ LeaveTS $\wedge$ AddTS $>$ RemoveTS, if AddTS $=$ JoinTS, then $\texttt{Authz}_{\texttt{CC}}$ should be true *and* therefore that when an `object_add` and `user_join` occur in the same state, `object_add` must occur second. The order of `user_leave` and `object_remove` is less important since either ordering would result in $\texttt{Authz}_{\texttt{CC}}$ becoming false.

Indeed when $\texttt{Authz}_{\texttt{CC}}$ is defined as $\mathrm{JoinTS} > \mathrm{LeaveTS} \wedge \mathrm{AddTS} > \mathrm{RemoveTS} \wedge \mathrm{AddTS} \geq \mathrm{JoinTS}$ all three models have the following property:[5]

$$\Box\Big(\texttt{Authz}_{\texttt{CC}} \leftrightarrow \big((\neg\texttt{remove} \wedge \neg\texttt{leave})\,\mathcal{S}\,(\texttt{add} \wedge (\neg\texttt{leave}\,\mathcal{S}\,\texttt{join}))\big)\Big) \tag{6.1}$$

If, instead we prefer that when `user_join` and `object_add` occur simultaneously that the `object_add` occur first, then $\texttt{Authz}_{\texttt{CC}}$ should be false in that state and it requires a slight modification to our definition of $\texttt{Authz}_{\texttt{CC}}$: $\texttt{Authz}_{\texttt{CC}} \equiv (\neg\texttt{remove} \wedge \neg\texttt{leave})\,\mathcal{S}\,(\texttt{add} \wedge (\neg\texttt{leave}\,\mathcal{S}\,\texttt{join}) \wedge \ominus(\neg\texttt{leave}\,\mathcal{S}\,\texttt{join}))$. In this case we find that we should use $\mathrm{AddTS} > \mathrm{JoinTS}$ rather than $\mathrm{AddTS} \geq \mathrm{JoinTS}$ and indeed modeling results confirm this.[6] It is worth noting at this point that the original definition of $\texttt{Authz}_{\texttt{CC}}$ is preferable since it requires fewer states to generate counterexamples.

The next event to consider is when `refresh` occurs with other events. The event `refresh` is always considered to happen last when combined with other events. As for "add"/"remove" events this is a natural choice since `refresh` triggers the TRM to update its attributes (which presumably were just updated by the CC). Otherwise, when a `refresh` event occurs, the TRM would need to get the previous values from the CC. However, just as in the case with `user_join` and `object_add`, the definition of $\texttt{Authz}_{\texttt{CC}}$, φ_1, and φ_2 dictates that the model behave this way. Further we can see that `refresh` must be considered to have happened *after* a simultaneous `request_perform`. This is due to the definition of strong stale-safety.

[5]I have played with this definition, when I change the $\geq$ to only $>$ then $\texttt{Authz}_{\texttt{CC}}$ in the model is no longer equivalent to the LTL definition.

[6]Since I use the model's definition of $\texttt{Authz}_{\texttt{CC}}$ in the LTL specifications, so long as the definition of $\texttt{Authz}_{\texttt{E}}$ in the TRM matches the definition of $\texttt{Authz}_{\texttt{CC}}$, all of the stale-safe properties remain intact regardless of which convention I use.

6.2 Minimal Clock Size

A naive first estimate might be that the minimum clock size must be seven to match the number of possible events. As discussed in the previous section, we can allow events to occur simultaneously which will serve to greatly reduce the minimum necessary clock size.

Theorem 6.2.1 (Minimal Clock Size)**.** *The minimum clock size, capable of generating all meaningful counterexamples, is* 2*.*

In addition to the Weak and Strong Stale-Safe properties (these results are discussed in Chapter 7), there are several others properties which we expect the models to exhibit or expect the models do not exhibit. Table 6.1 summarizes these properties. The final property, minimal stale-safety, is discussed in Chapter 9.

$\Box\Big(\mathtt{Authz}_{\mathtt{CC}} \leftrightarrow \big((\neg\mathtt{remove} \wedge \neg\mathtt{leave})\,\mathcal{S}\,(\mathtt{add} \wedge (\neg\mathtt{leave}\,\mathcal{S}\,\mathtt{join}))\big)\Big)$	The model's definition of $\mathtt{Authz}_{\mathtt{CC}}$ is identical to the LTL definition.
$\Box(\mathtt{perform} \rightarrow \varphi_0)$	The model enforces the LTL definition for $\mathtt{Authz}_{\mathtt{TRM}}$.
$\Box(\mathtt{perform} \rightarrow \ominus\mathtt{Authz}_{\mathtt{CC}})$	The model enforces $\mathtt{Authz}_{\mathtt{CC}}$ (none of the models should have this property).
$\Box(\mathtt{perform} \rightarrow \Diamond\!\!\!-\,\mathtt{Authz}_{\mathtt{CC}})$	The model exhibits minimal stale-safety. STRM and WTRM should have this property and the stale-unsafe TRM should not.

Table 6.1: Properties which g-SIS models either should or should not exhibit.

The first two properties in Table 6.1, that check the model's definition of $\mathtt{Authz}_{\mathtt{CC}}$ and that each model enforces $\mathtt{Authz}_{\mathtt{TRM}}$, do not influence the minimal clock size. These two properties are inherent in any g-SIS model (otherwise we wouldn't be modeling g-SIS rather something different). Figure 6.1 shows a possible trace which shows that even STRM can be found to not enforce $\mathtt{Authz}_{\mathtt{CC}}$ with only two clock ticks. In the first state, the user joins, the object is added then distributed, the user then requests, and finally the TRM refreshes. In this first state both $\mathtt{Authz}_{\mathtt{E}}$ and $\mathtt{Authz}_{\mathtt{CC}}$ are true. In the second state, the user leaves and the object is removed and the TRM responds to the user's request to perform from the first state. $\mathtt{Authz}_{\mathtt{CC}}$ is now false while

$\texttt{Authz}_{\texttt{E}}$ remains true (since there is no refresh in the second state). The user then is allowed to perform in the third state even though $\texttt{Authz}_{\texttt{CC}}$ was previously false.

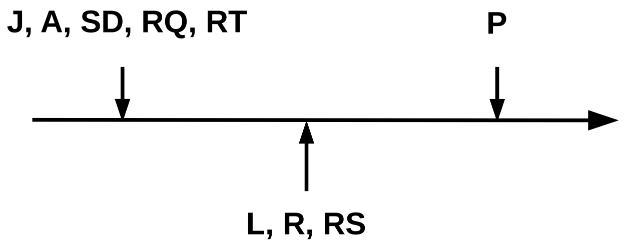


Figure 6.1: Possible trace showing that the TRMs will not always enforce $\texttt{Authz}_{\texttt{CC}}$. Abbreviations: **J**-`user_join`, **A**-`object_add`, **SD**-`super_distribute`, **RQ**-`request_perform`, **RT**-`refresh`, **L**-`user_leave`, **R**-`object_remove`, **RS**-TRM responds, and **P**-`perform`.

We expect that both WTRM and STRM satisfy minimal stale-safety but the unsafe TRM should not. Therefore we should be able to find a counterexample with a clock size of two. Figure 6.2 shows such a trace. In the first state the user joins and the TRM refreshes. Note that both $\texttt{Authz}_{\texttt{CC}}$ and $\texttt{Authz}_{\texttt{E}}$ are false in this first state. In the second state the user leaves, the object is added then distributed, the user requests to perform, and the TRM immediately responds to this request. The TRM is unaware that the user left thus when it naively checks $\texttt{Authz}_{\texttt{E}}$, the TRM finds that $\text{AddTS} > \text{JoinTS}$ and the user is allowed to perform in the third state even though $\texttt{Authz}_{\texttt{CC}}$ was never true. On the other hand WTRM would not allow this perform because $\text{AddTS} > \texttt{RT}$ therefore the predicate $\texttt{stale} \equiv \text{AddTS} > \texttt{RT}$ is true and $\texttt{Authz}_{\texttt{E}}$ is considered stale.

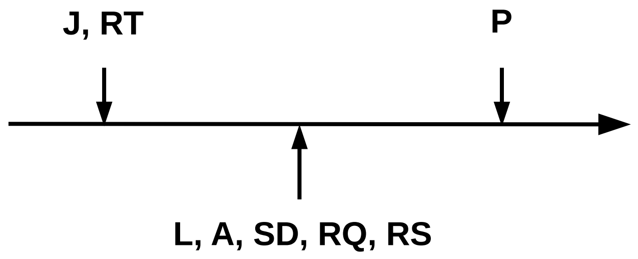


Figure 6.2: Possible trace showing that the unsafe TRM may allow a perform even though $\texttt{Authz}_{\texttt{CC}}$ is never true. Abbreviations: **J**-`user_join`, **A**-`object_add`, **SD**-`super_distribute`, **RQ**-`request_perform`, **RT**-`refresh`, **L**-`user_leave`, **R**-`object_remove`, **RS**-TRM responds, and **P**-`perform`.

There are two main reasons that a clock of size two suffices. The first reason is that $\texttt{Authz}_{\texttt{CC}}$

is a boolean value, thus we need two states (two clock ticks) to allow it to take on its two different values. In the case that we require `add_object` to occur *before* `user_join`, we would need *three* clock ticks because it would not be possible for $\texttt{Authz}_{\texttt{CC}}$ to be true in the first state (the soonest $\texttt{Authz}_{\texttt{CC}}$ can become true is the second state). The second reason is that perform is allowed to happen in the *third* state (so in a sense while we only need two clock ticks, we need *three* states). We need to allow $\texttt{Authz}_{\texttt{CC}}$ to become true and become false (two ticks) and *then* `perform` to occur. This behavior is inherent in all of the properties since `perform` always implies something about the previous state.

The question remains whether or not we could get counterexamples with fewer clock ticks. In fact, with a clock size of one, both the unsafe TRM and WTRM fail to enforce Strong Stale-Safety! *However*, with a clock size of one, all three TRMs will now enforce $\texttt{Authz}_{\texttt{CC}}$ and Weak Stale-Safety. It may seem counterintuitive that the unsafe TRM and WTRM enforce $\texttt{Authz}_{\texttt{CC}}$ yet fail to enforce Strong Stale-Safety. Recall that the clock only restricts how many "add"/"remove" events can occur. The user can continue to request indefinitely. Therefore it's possible for $\texttt{Authz}_{\texttt{CC}}$ to become true in the first state and remain true (because the user/object cannot be removed after this initial state). Meanwhile the user may make a request later for which there is no `refresh` event thus violating Strong Stale-Safety. This reiterates why the minimum size of the clock is two: there must be one state to allow $\texttt{Authz}_{\texttt{CC}}$ to become true and another to allow $\texttt{Authz}_{\texttt{CC}}$ to then become false.

6.3 SMV Modules

I chose to use the open-source model checker NuSMV [6] because it supports past temporal operators and it is a BDD-based, highly-optimized model checking tool that can handle a relatively large state space. NuSMV requires as input a model written in the SMV modeling language. NuSMV supports a modular design through the use of Modules[7]. Every SMV model requires a MAIN

[7]I view the SMV Modules as very similar to objects and used an object-oriented approach when designing these models.

module which acts as the entry point.

The SMV modules I created mirror very closely the classes presented in Chapter 5 so there are separate modules for the **CC**, **TRM**, and **User**. In addition, there are three helper modules: **CLOCK**, **EVENTS**, and **question-response**. The module **CLOCK** is responsible for creating timestamps and stopping after the clock reaches a maximum value (which can be set through a script). The module **EVENTS** is responsible for enforcing the constraints presented in Section 6.1 (the events are largely left free). Finally the module **question-response** is used to model the interaction between the user and the TRM when the user makes a request. When a request is made, the **question-response** will eventually trigger the TRM to respond, which could happen in the same state as the request or an arbitrary number of states later.[8]

The full SMV code can be found in Appendix A. I only present the definition of $\texttt{Authz}_{\texttt{TRM}}$ here for each of the three TRMs created. First, all three TRMs have the macro `authzE`:

```
authzE := join_ts > leave_ts & user.add_ts >= join_ts & user.add_ts > remove_ts;
```

The unsafe TRM merely defines `authzTRM` to be equal to `authzE`. Just as in Chapter 5, I will begin with STRM's definition of `authzTRM`:

```
init(authzTRM) := request & refresh & authzE;

next(authzTRM) := case
   -- initialize to FALSE if there is no refresh or authzE is FALSE
   next(request) : next(refresh) & next(authzE);
   -- a refresh always just sets authzTRM to authzE.
   next(refresh) : next(authzE);
   TRUE : authzTRM;
esac;
```

Note that `authzTRM` does not change unless a request or refresh occurs. When a request occurs, `authzTRM` should become FALSE if there is no refresh or if there is a refresh but `authzE` is FALSE. We see that if, after a request, `authzTRM` is FALSE, then the only way for it to become TRUE is if a refresh occurs that shows `authzE` to be TRUE.

[8]Figure 6.1 shows an example where the TRM responds in the state following the request and Figure 6.2 shows an example where the TRM responds in the same state as the request.

Next I present `authzTRM` for WTRM:

```
init(authzTRM) := request & authzE & !stale;

next(authzTRM) := case
   -- initialize to FALSE if authzE is false or if stale is TRUE.
   next(request) : next(authzE) & !next(stale);
   -- a refresh always just sets authzTRM to authzE.
   next(refresh) : next(authzE);
   TRUE : authzTRM;
esac;
```

This definition is very similar to `authzTRM` for STRM. However, now a refresh isn't required when a request is made; instead we must check the condition `stale`. Again, if `authzTRM` is initially FALSE after a request, the only way for `authzTRM` to become TRUE is for a refresh to occur showing `authzE` to be TRUE (just as with STRM).

Chapter 7: RESULTS OF MODEL CHECKING g-SIS

In this section, I present the results of model checking the three following systems: Δ_0, Δ_1, Δ_2 which represent stale-unsafe, weakly stale-safe, and strongly stale-safe enforcement models, respectively (the only difference in the three models is the different TRMs used).

7.1 Small and Large Enforcement Models

I denote the small enforcement model containing a single instance (one user and one object in one group) of the enforcement module which uses the unsafe TRM as Δ_0-system.

Definition 7.1.1 (Δ_0-system)**.** *Let Δ_0 represent a small g-SIS enforcement model defined as below:*

$$\begin{aligned}\Delta_0(u,o,g) \equiv CC(u,o,g))\ |parallel| \\ (USER(u,o,g)\ |rendez| \\ TRM\_UNSAFE(u,o,g))\end{aligned}$$

In which, CC, USER, and TRM_UNSAFE rendezvous (rendez) respectively, and then are composed via parallel composition.

An unbounded finite (UF) enforcement model (or a large model) contains any number of instances of a Δ-system (Δ_0, Δ_1, or Δ_2), ranging over all users, objects, and groups, executing in parallel, formally,

$$\Delta_i^{UF} \equiv \underset{u\in \mathrm{U}.o\in \mathrm{O}.g\in G}{||} \Delta_i(u,o,g)$$

where i takes on a value from $\{0,1,2\}$.

In the following subsections, I present several theorems concerning the behavior of each of the three Δ systems. In each case, I only model checked the small model. I later argue, in Chapter 8, that this is sufficient for proving each unbounded finite system also models the FOTL formulas presented.

7.2 Unsafe Enforcement Model

$\texttt{Authz}_{\texttt{CC}}$ is not enforceable locally at the TRM, therefore we need to formulate another version that is enforceable at the TRM. $\texttt{Authz}_{\texttt{TRM}}$, below, shows the re-formulation of $\texttt{Authz}_{\texttt{CC}}$ as enforceable by the TRM.

$$\begin{aligned}
&\forall u : \mathrm{U}.\forall o : \mathrm{O}.\forall g : \mathrm{G}.\forall op : \mathrm{P}.\\
&\Box(\texttt{Authz}_{\texttt{TRM}}(u,o,g,op) \leftrightarrow\ (\neg\texttt{RT}(u,g)\,\mathcal{S}\,(\texttt{add}(o,g) \wedge (\neg\texttt{RT}(u,g)\,\mathcal{S}\,(\texttt{RT}(u,g)\wedge\\
&(\neg\texttt{leave}(u,g)\,\mathcal{S}\,\texttt{join}(u,g)))))) \ \vee (\neg\texttt{RT}(u,g)\,\mathcal{S}\,(\texttt{RT}(u,g) \wedge \texttt{Authz}_{\texttt{CC}}(u,o,g,op))))
\end{aligned}$$

The occurrence of `join`, `leave` and `remove` are ascertained at `RT`. However, `add` is not subject to this constraint and its occurrence is ascertained independently of `RT`. $\texttt{Authz}_{\texttt{TRM}}$ is a disjunction of two cases. The first part addresses the scenario in which the requested object was added after the most recent `RT`. The second part handles the situation where the object was added before the most recent `RT`.

Let us introduce formula φ_0 to represent the TRM authorization of u performs an operation op of o in g:

$$\begin{aligned}
\varphi_0(u,o,g,op) \equiv \ominus((&\neg\texttt{perform}(u,o,g,op) \wedge (\neg\texttt{RT} \vee (\texttt{RT} \wedge \texttt{Authz}_{\texttt{CC}}(u,o,g,op))))\,\mathcal{S}\\
&(\texttt{Authz}_{\texttt{TRM}}(u,o,g,op) \wedge (\neg\texttt{request}(u,o,g,op)\wedge\\
&\neg\texttt{perform}(u,o,g,op))\,\mathcal{S}\,\texttt{request}(u,o,g,op)))
\end{aligned}$$

We proved through model checking the following formula for the Δ_0 system:

$$\Delta_0(u,o,g) \models \Box(\texttt{perform}(u,o,g) \rightarrow \varphi_0(u,o,g))$$

In Chapter 8 I argue that this implies that the unbounded finite system also models this formula.

Theorem 7.2.1 (Enforceability Theorem)**.** *The Δ_0^{UF}-system enforces $\texttt{Authz}_{\textit{TRM}}$. That is:*

$$\Delta_0^{UF} \models \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\textit{perform}(u,o,g) \rightarrow \varphi_0(u,o,g))$$

This theorem states that $\texttt{Authz}_{\text{TRM}}$ is enforceable by the Δ_0-system. Recall that $\Box(\texttt{perform} \rightarrow \varphi_0)$ ensures that the user can perform an action only if $\texttt{Authz}_{\text{TRM}}$ is satisfied (section 4.5).

Theorem 7.2.2 (Weak Unsafe TRM Theorem)**.** *The Δ_0-system does not satisfy the Weak Stale-Safety property. That is:*

$$\Delta_0^{UF} \not\models \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\textit{perform}(u,o,g) \rightarrow (\varphi_1(u,o,g) \vee \varphi_2(u,o,g)))$$

This theorem states that the Δ_0-system is not stale-safe. Specifically, it fails the weak stale-safety security property. Recall that this was illustrated earlier in figure 4.1.

Table 7.1 shows the counterexample provided by NuSMV; this shows that the Δ_0-system fails the weak stale-safety property. In state 1.1 the user is joined and a refresh occurs, but since the object has not been added yet, both $\texttt{Authz}_{\text{CC}}$ and $\texttt{Authz}_{\text{CC}}$ remain FALSE. In state 1.2, the object is added, super distribution occurs, and the user requests to perform on the object. This makes both $\texttt{Authz}_{\text{TRM}}$ and $\texttt{Authz}_{\text{CC}}$ now TRUE. Finallly, in state 1.3, the user performs. This fails weak stale-safety because the refresh occurs before the object is added. Although the trace generated by NuSMV didn't show it, the user could have left in the second state which would have made $\texttt{Authz}_{\text{CC}}$ FALSE but $\texttt{Authz}_{\text{TRM}}$ would have still become TRUE and a perform would have occurred for which the user was never authorized.

Table 7.1: Counterexample trace generated by NuSMV showing that the unsafe TRM (Δ_0) does not satisfy weak stale-safety (the event `super_distribute` is abbreviated SD).

VAR	1.1	1.2	1.3
leave	false	false	false
join	TRUE	false	false
add	false	TRUE	false
remove	false	false	false
SD	false	TRUE	false
request	false	TRUE	false
rt	TRUE	false	TRUE
authzcc	false	TRUE	TRUE
authztrm	false	TRUE	TRUE
perform	false	false	TRUE
t	1	2	2

Table 7.2: Counterexample trace generated by NuSMV showing that the weak TRM (Δ_1) does not satisfy strong stale-safety. (note that state 2.1 is not shown because all variables are FALSE in the initial state)

VAR	2.1	2.2	2.3	2.4
leave	false	false	false	false
join	TRUE	false	false	false
add	TRUE	false	false	false
remove	false	TRUE	false	false
SD	TRUE	false	false	false
request	TRUE	false	TRUE	false
rt	TRUE	false	false	TRUE
authzcc	TRUE	false	false	false
authztrm	TRUE	TRUE	TRUE	false
perform	false	false	TRUE	TRUE
t	1	2	2	2

Evidently, if Δ_0-system fails the weak property, it would also fail the strong property. The same counter-example generated by NuSMV for theorem 7.2.2 applies here.

7.3 Weak Stale-Safe TRM

Figure 5.6 the state machine used to define TRM_WEAK.

Definition 7.3.1 (Δ_1-system)**.** *Let Δ_1 represent a small g-SIS enforcement model defined as below.*

$$\begin{aligned}\Delta_1(u,o,g) \equiv CC(u,o,g)) \; &|parallel| \\ &(USER(u,o,g) \; |rendez| \\ &\text{TRM\_WEAK}(u,o,g))\end{aligned}$$

where CC, USER, and TRM_WEAK *rendezvous respectively, and then execute concurrently via parallel composition.*

Theorem 7.3.2 (Weak Stale-Safe TRM Theorem). *Δ_1^{UF} satisfies the Weak Stale-Safe Security Property.*

$$\Delta_1^{UF} \models \forall u : \mathrm{U}.\forall o : \mathrm{O}.\forall g : \mathrm{G}.\Box(\mathtt{perform}(u,o,g) \rightarrow (\varphi_1(u,o,g) \vee \varphi_2(u,o,g)))$$

NuSMV successfully verifies that the Δ_1-system satisfies the weak stale-safe security property. Obviously, the Δ_1-system does not satisfy the strong stale-safety property.

$$\Delta_1^{UF} \not\models \forall u : \mathrm{U}.\forall o : \mathrm{O}.\forall g : \mathrm{G}.\Box(\mathtt{perform}(u,o,g) \rightarrow \varphi_2(u,o,g))$$

As expected, NuSMV generates a counter-example showing that the Δ_1-system does not satisfy strong stale-safety.

Table 7.2 shows the counterexample trace generated by NuSMV. Note that the user performs twice and that it is the second perform that violates strongl stale-safety. In the first state, the user joins, the object is added, super distribution occurs, the user requests to perform, and a refresh occurs. This makes both $\mathtt{Authz}_{\mathtt{TRM}}$ and $\mathtt{Authz}_{\mathtt{CC}}$ to be TRUE. In the second state, the object is removed which causes $\mathtt{Authz}_{\mathtt{CC}}$ to become FALSE but $\mathtt{Authz}_{\mathtt{TRM}}$ remains TRUE. In the 3rd state, the user performs and also requests to perform again. Finally in the 4th state a refresh occurs and the user performs. This violates strong stale-safety because there is no refresh after the second request and before the second perform (recall that refresh events are always considered to happen last when events occur simultaneously in the same state).

7.4 Strong Stale-Safe TRM

Figure 5.5 shows a our model TRM_STRONG.

Definition 7.4.1 (Δ_2-system). *Let Δ_2 represent a small g-SIS enforcement model defined as below.*

$$\begin{aligned}\Delta_2 \equiv CC(u,o,g))\ |parallel| \\ (USER(u,o,g)\ |rendez| \\ \text{TRM\_STRONG}(u,o,g))\end{aligned}$$

where CC, User, and TRM_STRONG *rendezvous respectively, and then execute concurrently via parallel composition*

Theorem 7.4.2 (Strong Stale-Safe TRM Theorem)**.** Δ_2 *satisfies the Strong Stale-Safe Security Property.*

$$\Delta_2^{UF} \models \forall u : \text{U}.\forall o : \text{O}.\forall g : \text{G}.\Box(\texttt{perform}(u,o,g) \rightarrow \varphi_2(u,o,g))$$

NuSMV successfully verifies that the Δ_2-system satisfies strong stale-safety.

Chapter 8: VERIFICATION OF UNBOUNDED g-SIS FINITE ENFORCEMENT SYSTEMS

In Chapter 7 I presented Δ systems describing three different enforcement models (through their differing TRM HTS's). The model checking results for stale-safety regarding these systems rely on small models (containing a single user, single object, and single group). In this section I argue that the results in Chapter 7 are valid over multiple or even an unbounded number of users, objects, and groups.

When model checking the small carrier Δ systems, there is nothing special about the choice of user, object, and group–they are anonymous. So intuitively, it should be the case that a system composed of many Δ systems would still model the same theorems as the single Δ systems for all of its users, objects, and groups.

8.1 Informal Proof of Small Model Validity

First consider a first-order policy of the following form: $\forall p \in \mathcal{P} : \Box\psi(p)$, where $\psi(p)$ is an LTL formula which takes the variable p as a parameter. The only restriction on ψ is that it does *not* recognize any subgroups of $\mathcal{P}$. This generally means that p does not contain any predicates or functions of p–only LTL operators and propositional expressions. This forbids some members of $p \in \mathcal{P}$ from being treated differently in the policy. Contrast the above policy with $\forall p_1, p_2 \in \mathcal{P} : \Box\psi(p_1, p_2)$. It's not clear how p_1 and p_2 interact inside this formula and thus verification may require instantiating *several* p's.[1]

I use the HIPAA privacy policy as an example which violates the form of the above policy. The HIPAA privacy policy has a predicate *inrole* which takes two parameters: a person/entity and a particular role (e.g. patient, covered entity, business associate, law enforcement, etc.). This predicate serves to distinguish actions each role may take and thus treats elements from the sort of

[1]Note that the formula $\psi(p_1, p_2)$ could be something trival like: $\psi(p_1, p_2) = \phi(p_1) \wedge \phi(p_2)$–this formula is redundant because of the universal quantifier over the domain $\mathcal{P}$ and thus could be verified with only a *single* instantiation from $\mathcal{P}$.

all entities differently because the predicate *inrole* cannot ever be true for some entities (e.g. some patients are not doctors) and may not be true, under some circumstances, even for entities which sometimes *can* take on the particular role (e.g. a patient that is also a doctor but, under the current circumstance, is acting as a patient).

Just as we must constrain the types of policies this small model theorem applies to, we must put some constraints on the types of models for which it is valid. Informally, the constraint is that each instance of the model behaves the "same". The constraint is similar to the constraint we place on the types of policies: the model must not treat any instances differently. We say that instances A and B behave the same if they produce identical traces for the same set of input events. Since different instances are not required to receive identical input (e.g. if user A joins, this does not mean that user B joins), the instances will not, in general produce identical traces.

Picture that a large enforcement model consists of many smaller models (instances), labeled 1 through n (each with their own instantiation of the variable p in the policy above). To check whether the first-order formula is satisfied amounts to individually checking each model. We can think of the instance labeled as 1 as representative of all of the n instances. If we find a trace such that some other model, labeled i, (other than the first) falsified the formula $\psi(p_i)$, then since all of the models behave the same way, we know that we can recreate the set of events that caused instance i to falsify the formula and instead of directing them towards instance i, direct them towards the first instance. Now the first instance will falsify the formula. That is, if it is possible for *any* of the instances to falsify the policy then it is possible to falsify the policy with only the first instance (a single instance).

The policies in section 4.3 have the above form. The fact that p (in section 4.3) is a tuple is not important. In fact, this illustrates the fact that the policy may have *many* universally quantified variables and yet still can be modeled by a single instance of a parameterized system so long as each of those quantified variables are from semanticaly disjoint domains (i.e. user, object, and group in our case). To clarify, this reasoning would not hold in general if, for instance, we were analyzing a policy quantified over doctors and patients since it may be possible for a person to be both a doctor sometimes and a patient in other cases. Thus it would not hold that we *only* need a single doctor and a single patient. As an example, the policy may specify that doctor `d1` cannot be the patient of doctor `d2` for whom `d2` is also a patient of `d1`'s. In the case of such a policy, a possible small model theorem would need to instantiate, at a minimum, two doctors and a patient who is never a doctor (so at least three people).

Chapter 9: DISCUSSION OF GENERAL Stale-Safety

The previous chapters presented g-SIS as an example for studying stale-safety. This section provides a discussion of stale-safety in a more general context. In g-SIS, stale-safety specifically deals with access decisions (authorizations). To generalize stale-safety, we will discuss *inferences*; not just access decisions. In the example of g-SIS, we may incorrectly *infer* that a user is allowed access to a certain object by using stale attributes contained at the TRM. When using stale information, there can never be a guarantee that an inference will be correct. The weakest requirement for stale-safety is that when we make an inference, it should *at least* have been valid at some point. With this in mind, I propose a more general definition of stale-safety which can be applied to many different use cases. In section 9.2, I show how this definition can be used to identify staleness violations in SAAM [8,36].

9.1 Minimal Stale-Safety

We consider two different predicates:

$\texttt{allow}(\texttt{i})$	Inference $\texttt{i}$ is allowed.
$\texttt{sound}(\texttt{i})$	The inference $\texttt{i}$ is sound.

Note that it's neither the case that in general $\texttt{allow}(\texttt{i}) \rightarrow \texttt{sound}(\texttt{i})$ nor that $\texttt{sound}(\texttt{i}) \rightarrow \texttt{allow}(\texttt{i})$. The first statement is acknowledging staleness–when we allow an inference we do so with stale attributes and thus cannot be assured that the inference is still sound. The second statement is acknowledging that 1) an enforcement model is under no obligation to allow all sound inferences and 2) it might not even be possible for an enforcement model to *know* that an inference has become sound (e.g. with periodic refreshes like g-SIS).

Definition 9.1.1 (Minimal Stale-Safety)**.** *An enforcement model* M*, which defines* $\texttt{allow}(i)$*, is said to be minimally stale-safe if it models the following FOTL property:*

$$\forall i : \mathrm{I}.\Box\big(\texttt{allow}(i) \rightarrow \diamondminus \texttt{sound}(i)\big)$$

Where I *is the set of all inferences that* M *can make.*

The $\diamondminus$ operator (read once) has the following semantics: $\diamondminus p$ means that p was true at least one time in the past. Clearly definition 9.1.1 captures the spirit of stale-safety by requiring that all allowed inferences be valid at some point in the past.

Definition 9.1.1 is a fairly weak requirement. In practice, one can define a more strict stale-safe property just as I presented for g-SIS. The pattern used in g-SIS is straightforward: once it is ascertained that an inference is unsound, it is presumed to remain unsound until evidence is presented otherwise. In g-SIS this is done through periodic refreshes of attributes. If the latest refresh of attributes determines that user `u` in group `g` does *not* have authorization to perform operation `op` on object `o`, then `u` is not allowed to perform `op` on `o` even if `u` was previously authorized to do so. Our definitions of weak stale-safety and strong stale-safety for g-SIS (definitions 4.6.1 and 4.7.1, respectively) imply minimal stale-safety as defined in definition 9.1.1.

9.2 Staleness in SAAM

SAAM (***S****econdary and* ***A****pproximate* ***A****uthorization* ***M****odel*) is a conceptual framework introduced by Crampton et al. [8] SAAM uses historical (stale) information to infer an approximate (incomplete) access policy. The logic required to instantiate a SAAM depends on the class of access control policy used. Crampton et al. [8] and Wei et al. [36] present SAAMs for BLP (*Bell-LaPadula*) [4, 21] and RBAC (*role-based access control*) [9, 30], respectively.

The SAAMs from [8] and [36] do not consider a dynamic policy. They both assume the policy never changes in order to infer the *exact* decision (they use the term *safe decision*[1]) from previously made ones. As an example consider the following scenario describing a BLP access control policy. There are two subjects s and s'; object o; and it has been previously observed that both $(s', o, \texttt{append})$ and $(s, o, \texttt{read})$ were allowed. We may infer 1) that s' has a lower (or equal) clearance level than o, 2) that s has a higher (or equal) clearance level than o, and therefore that 3) s has a higher (or equal) clearance level than s'. Using the labeling function notation this is

[1]There is no relation between our use of the term *stale-safety* and the use of the term *safe decision* in [8, 36].

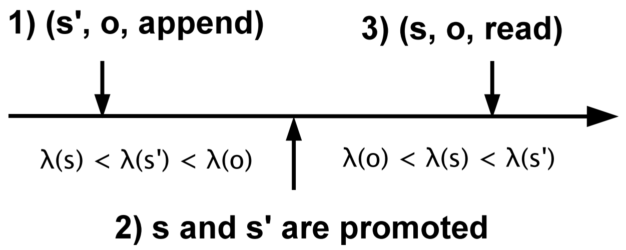


Figure 9.1: A timeline giving an example of SAAM$_{BLP}$ violating minimal stale-safety.

equivalent to 1) $\lambda(s') \leq \lambda(o)$, 2) $\lambda(s) \geq \lambda(o)$, and 3) $\lambda(s) \geq \lambda(s')$.

Figure 9.1 shows an example of events where the inference $\lambda(s) \geq \lambda(s')$, in the above example, is *never* true. It shows that between the time when $(s', o, \texttt{append})$ and $(s, o, \texttt{read})$ are *both* allowed, that s and s' are promoted, but remain in the same initial order. They were both initially at a lower security clearance than o and thus were both allowed to append o. After they are both promoted, they would then both be allowed to read from o, but *not* append it anymore. This shows a possible way that the inference made by SAAM$_{BLP}$, $\lambda(s) \geq \lambda(s')$, is never true. Further this can lead to security leaks contrary to the spirit of a BLP access policy. If we introduce a second object o' and observe that $(s, o', \texttt{append})$ is allowed (after the events in figure 9.1), then we might mistakenly infer, based on the inference $\lambda(s) \geq \lambda(s')$, that $(s', o', \texttt{append})$ should also be allowed. Assuming that $\lambda(s) < \lambda(o') < \lambda(s')$ (after s and s' are promoted) this would allow subject s' to propagate information from a higher clearance level to a lower one.

One possible way to prevent this type of stale-safety violation is to attach time stamps to the attributes just as in g-SIS. Without timestamps in g-SIS, we could not determine whether or not an object and user have overlapping membership in a group. In the above BLP example, simply knowing how long each subject and object have maintained their current security clearance level can help avoid stale-safety violations. E.g. in figure 9.1 when $(s, o, \texttt{read})$ is allowed, if we know that s, s', and o all maintained their clearance level since $(s', o, \texttt{append})$ was allowed[2] then $\lambda(s) \geq \lambda(s')$ *is* a stale-safe inference.

[2]Which would imply that event 2) in figure 9.1 could not have happened.

Chapter 10: CONCLUSION

10.1 Related Work

To the best of my knowledge, this is the first effort towards a formalization of the notion of stale-safety and proof-of-concept verification in distributed systems. The work of Lee et al [22,23] is the closest to this thesis that I have seen in the literature, but focuses exclusively on the use of attribute certificates, called credentials, for assertion of attribute values in trust negotiation systems. Lee et al focus on the need to obtain fresh information about the revocation status of credentials to avoid staleness and propose different levels of consistency amongst the credentials used to make a decision. Our formalism is based on the notion of a "refresh time," that is the time when an attribute value was known to be accurate. We believe the notion of refresh time is central to formulation of stale-safe properties. Because Lee et al admit only attribute certificates as carriers of attribute information there is no notion of refresh time in their framework. Further, formal specification of stale-safe properties using temporal logic allows designers to ensure stale-safety in their enforcement models using automated verification techniques such as model checking.

Furthermore, as [22] note, there is a rich body of literature on achieving consistency in distributed systems (see [34] for a survey). The fundamental problem addressed in this domain is to achieve consistency in data replication systems and balance it with performance and availability. For example, [37] discuss a continuous consistency model that explores the range of consistency levels between strong consistency (where consistency requirement is absolute) and optimistic consistency (where consistency requirement is not critical). In contrast, stale-safety is concerned about making safe authorization decisions at an enforcement point using attribute values that may be out-of-date. We envision that stale-safe security properties identified in this article would be used for making authorization decisions in replicated systems.

Another body of related work is the problem of Time-Of-Check- To-Time-Of-Use (TOCTTOU). TOCTTOU is a type of race condition in software that can be exploited to gain unauthorized access to privileged resources. For example, consider a code fragment that checks for user's write access to a file name in step 1 (the file name is typically a full path to a specific resource in UNIX type operating systems) and if the check succeeds, allows the user to write to the file in step 2. An attacker can exploit this by symbolically linking the file name to a more privileged file during the time between steps 1 and 2, thereby gaining access to an unauthorized file. The weak stale-safety property in contrast is not concerned about changes in the resource that is being accessed. Instead, it is concerned about making an authorization decision based on the authorization information obtained recently. Specifically, weak stale-safety proposes to disallow certain actions even if the authorization information confirms access. Thus a system that satisfies the weak or the strong stale-safety properties is still susceptible to TOCTTOU attacks. For instance, the same attack described above can be launched against a g-SIS system that does not address TOCTTOU issues. In an abstract sense, the TOCTTOU problem is *similar* to stale-safety. Because TOCTTOU has not been studied formally or systematically, it is difficult to comment on its differences with respect to the formal notion of stale-safety.

The work of Schneider ([32]) characterizes security policies enforceable using a security automata by monitoring system execution. Specifically, it ensures that a program (or the HTS model in our case) in fact enforces the security policy and terminates the program in case of failure to do so. In comparison, our work strengthens this by ensuring that the enforced security policies are stale-safe. While precise enforcement of security policy is critical in high-assurance and safety-critical systems [20], stale-safety allows a designer to make trade-offs in systems where availability is more or less equally desirable.

The use of model checking in automating analysis of security policies, properties, and protocols has attracted a lot of research attention. There has been fruitful research in model checking security protocols since Gavin Lowe's seminal work [25, 26] using model checker FDR for CSP to reveal a subtle attack of the Needham-Schroeder authentication protocol and cryptographic protocols. Recently, model checking has been increasingly employed to reason about security properties, especially in RBAC and trust management systems. Sistla and Zhou [33] propose a model-checking framework for security analysis of RT policies [24]. Jha and Reps [16] verify such properties as authorization, availability, and shared access of the SPKI/SDSI policy language using model checking. Fisler et.al. [10] analyze the impact of policy changes on RBAC systems and verify the separation of duty properties using their own model checking tool called Margrave. Schaad et. al. [31] verify separation of duty properties in RBAC systems through NuSMV [6]. Hansen et. al. [14] utilize an explicit model checking tool called Spin to verify various static and dynamic separation of duty properties in RBAC. Additionally, other works [1, 12, 28, 38] also leverage formal techniques to verify properties of security and privacy policy specifications.

In extending the small carrier (single) model to the large carrier (multiple entities), I used similar reasoning as in [29] to argue that our model checking results can be extended to FOTL. There are two major differences in my approach to that of [29]. The first is that the parameterized model differs from their definition of a BDS (Bounded-discrete data system). The BDS they present has a specific transition relation which makes the many processes transition in an interleaved fashion. This is evident by the fact that their transition relation has an existential quantifier, meaning that at least one process transitions (it can be many, but only one *must* transition). Contrasting with our model, our transition relation involves a universal quantifier, indicating that all processes transition at once, in every state. The second major difference is that our processes cannot modify any values from the other processes. Ultimately the approach in finding a Small Model Theorem was similar to [29]–that is I found a counterexample in a large system then showed that this counterexample could be generated by a smaller system (in this case a system of size 1).

10.2 Final Thoughts

Attribute staleness is inherent to any distributed system due to physical distribution of user and object attributes. While it is not possible to eliminate staleness entirely, we can still manage and limit its impact. In this thesis, I proposed three stale-safe security properties: Weak Stale-Safety, Strong Stale-Safety, and Minimal Stale-Safety. This formalization not only enabled me to precisely state the properties but also allowed systems to be formally verified. With model checking I proved the small model satisfies the properties and, do to the nature of our particular system, extend those results to FOTL.

Appendix A: SMV MODULES

The full code of the SMV Modules as well as the compiler script are presented in this Appendix. Each section contains the code for each of the individual modules and the final section gives the script for generating a full SMV model that is suitable for NuSMV.

A.1 MAIN.mod

```
MODULE main()
VAR

   events : EVENTS(trm, clock);
clock : CLOCK();
cc : CC(events.user_join, events.user_leave, events.object_add,
   events.object_remove, clock);
user : USER(events.super_distro, cc);
trm : TRM(events, user, cc, clock);

DEFINE
respond := trm.respond;
perform := trm.perform;
rt := events.refresh;
authztrm := trm.authzTRM;
authzcc := cc.join_ts > cc.leave_ts & cc.add_ts > cc.remove_ts
   & cc.add_ts >= cc.join_ts;
request := events.request;
remove := events.object_remove;
leave := events.user_leave;
add := events.object_add;
join := events.user_join;

-- check that authzcc satisfies the formula for authzcc
```

```
LTLSPEC G(authzcc <-> ((!remove & !leave) S (add & (!leave S join))))

--perform -> authz_TRM
LTLSPEC G(
perform -> Y(
!perform S ((!rt S (add & (!rt S (rt & (!leave S join ))))) |
(!rt S (rt & ((!remove & !leave) S add) & (!leave S join))) &
((!request & !perform) S request))
)
);

-- perfrom -> authzcc, all three should fail this
LTLSPEC G(perform -> Y(authzcc))

-- minimal stalesafety
LTLSPEC G(perform -> O authzcc)



--weak-stale safety
--unsafe perform should fail, weak and strong should pass
LTLSPEC G(perform -> (Y((!perform & (!rt | (rt & authzcc))) S
(request & (!rt S (rt &authzcc)))) | Y((!perform & !rt) S
(rt & authzcc & ((!perform & (!rt | (rt & authzcc))) S request)))))


--strong-stale safety
   LTLSPEC G(perform -> Y((!perform & !rt) S(rt & authzcc & ((!perform &
   (!rt | (rt & authzcc))) S request))))
```

A.2 EVENTS.mod

```
MODULE EVENTS(trm, clock)
```

```
VAR
   user_join : boolean;
   user_leave : boolean;
   object_add : boolean;
   object_remove : boolean;
   super_distro : boolean;
   request : boolean;
   refresh : boolean; -- refresh is completely free

   user_joined : boolean;
   object_added : boolean;
   object_requested : boolean;

   can_super_distro : boolean;
   can_request : boolean;


ASSIGN

   -- the "add" events are allowed to occur in the first state

   init(user_join) := {TRUE, FALSE};
   init(object_add) := {TRUE, FALSE};

      -- leave events cannot occur in the same state as add events (nor until
      --    an add event has occured) so they cannot happen in the initial
      --    state.

   init(user_leave) := FALSE;
   init(object_remove) := FALSE;


      -- Init can_super_distro and can_request (needed before initting
      --    super_distro and request).

      init(can_super_distro) := case
         object_add : TRUE;
         TRUE : FALSE;
      esac;

      init(super_distro) := case
         can_super_distro : {TRUE, FALSE};
         TRUE : FALSE;
      esac;

      init(can_request) := case
         super_distro : TRUE;
         TRUE : FALSE;
      esac;

   init(request) := case
      can_request : {TRUE, FALSE};
      TRUE : FALSE;
   esac;

      -- init user_joined, object_added, and object_requested

      init(user_joined) := case
         user_join : TRUE;
         TRUE : FALSE;
      esac;

      init(object_added) := case
         object_add : TRUE;
         TRUE : FALSE;
      esac;
```

```
init(object_requested) := case
   request : TRUE;
   TRUE : FALSE;
esac;

-- Define user_joined, object_added, and object_requested

next(user_joined) := case
   next(user_join) : TRUE;
   next(user_leave) : FALSE;
   TRUE : user_joined;
esac;

next(object_added) := case
   next(object_add) : TRUE;
   next(user_leave) : FALSE;
   TRUE : object_added;
esac;

next(object_requested) := case
   next(request) : TRUE;
   next(trm.respond) : FALSE;
   TRUE : object_requested;
esac;



-- define when add/remove events are allowed to occur
-- NONE of these events are allowed after the clock stops

next(user_join) := case
   !clock.CLOCK_STOPPED : case
      -- can only join if user is NOT joined in previous state.
      user_joined : FALSE;
      TRUE : {TRUE, FALSE};
   esac;
   TRUE : FALSE;
esac;

next(user_leave) := case
   !clock.CLOCK_STOPPED : case
      -- can only leave if user is joined in previous state.
      user_joined : {TRUE, FALSE};
      TRUE : FALSE;
   esac;
   TRUE : FALSE;
esac;

next(object_add) := case
   !clock.CLOCK_STOPPED : case
      -- can only add if object is NOT added in the previous state
      --    (object_added) is initially false (unless object_add
      --    occurs in the first state).
      object_added : FALSE;
      TRUE : {TRUE, FALSE};
   esac;
   TRUE : FALSE;
esac;

next(object_remove) := case
   !clock.CLOCK_STOPPED : case
      object_added : {TRUE, FALSE};
      TRUE : FALSE;
   esac;
   TRUE : FALSE;
esac;
```

```
-- Define can_super_distro, super_distro, can_request, and request.
-- Generally super_distro may become true anytime outside of a request
-- (so not while object_requested remains true).  The same goes for
-- request.

-- Becomes (and stays) true the first time object_add is true (an object
--    is added).
next(can_super_distro) := case
   next(object_add) : TRUE;
   TRUE : can_super_distro;
esac;

next(super_distro) := case
   -- can_super_distro eventually becomes true and then stays true
   --    forever, so eventually this case is really what is evaluated.
   next(can_super_distro) : case
      -- don’t super distribute while waiting for a response.
      object_requested : FALSE;
      TRUE : {TRUE, FALSE};
   esac;

   -- if you can’t super distribute, then don’t
   TRUE : FALSE;
esac;



-- Becomes (and stays) true the first time super_distro becomes true
--    (super distribution occurs).
next(can_request) := case
   next(super_distro) : TRUE;
   TRUE : can_request;
```

```
      esac;

      next(request) := case
         next(can_request) : case
            -- don't request while waiting for response
            object_requested : FALSE;
            TRUE : {TRUE, FALSE};
         esac;

         -- if you can't request, then don't
         TRUE : FALSE;
      esac;
```

A.3 CC.mod

```
MODULE CC(user_join, user_leave, object_add, object_remove, clock)
VAR
   -- the CC timestamps go back one further than the clock to setup an
   -- initial state (where authzCC is false).
add_ts : 0..MAX_TIME;
remove_ts : 0..MAX_TIME;
join_ts : 0..MAX_TIME;
leave_ts : 0..MAX_TIME;

ASSIGN
init(add_ts) := case
   object_add : clock.t;
   TRUE : 0;
esac;

init(remove_ts) := case
   object_add : 0;
   TRUE : clock.t;
```

```
esac;

init(join_ts) := case
   user_join : clock.t;
   TRUE : 0;
esac;

init(leave_ts) := case
   user_join : 0;
   TRUE : clock.t;
esac;



next(add_ts) := case
next(object_add) : next(clock.t);
TRUE : add_ts;
esac;

next(remove_ts) := case
next(object_remove) : next(clock.t);
TRUE : remove_ts;
esac;

next(join_ts) := case
next(user_join) : next(clock.t);
TRUE : join_ts;
esac;

next(leave_ts) := case
next(user_leave) : next(clock.t);
TRUE : leave_ts;
esac;
```

A.4 TRM_UNSAFE.mod

```
MODULE TRM(events, user, cc, clock)
VAR
   -- the CC timestamps go back one further than the clock to setup an
   -- initial state (where authzCC is false).

remove_ts : 0..MAX_TIME;
join_ts : 0..MAX_TIME;
leave_ts : 0..MAX_TIME;

refresh_ts : 0..MAX_TIME;

authzTRM : boolean;

perform : boolean;

qa : question-response(request, TRUE, FALSE);

ASSIGN

   -- put timestamps in a state where AuthzTRM is false.
init(remove_ts) := case
   refresh : cc.remove_ts;
   TRUE : 0;
esac;

init(join_ts) := case
   refresh : cc.join_ts;
   TRUE : 0;
esac;

init(leave_ts) := case
```

```
   refresh : cc.leave_ts;
   TRUE : clock.t;
esac;

init(refresh_ts) := case
   refresh : clock.t;
   TRUE : 0;
   esac;

next(remove_ts) := case
next(refresh) : next(cc.remove_ts);
TRUE : remove_ts;
esac;

next(join_ts) := case
next(refresh) : next(cc.join_ts);
TRUE : join_ts;
esac;

next(leave_ts) := case
next(refresh) : next(cc.leave_ts);
TRUE : leave_ts;
esac;

next(refresh_ts) := case
   next(refresh) : next(clock.t);
   TRUE : refresh_ts;
esac;

init(perform) := FALSE;

next(perform) := case
   qa.respond & authzTRM : TRUE;
   TRUE : FALSE;
esac;

authzTRM := authzE;

   DEFINE
      add_ts := user.add_ts;
      authzE := join_ts > leave_ts & user.add_ts >= join_ts &
         user.add_ts > remove_ts;
      stale := user.add_ts > refresh_ts;
      respond := qa.respond;
      request := events.request;
      refresh := events.refresh;
```

A.5 TRM_WEAK.mod

```
MODULE TRM(events, user, cc, clock)
VAR
   -- the CC timestamps go back one further than the clock to setup an
   -- initial state (where authzCC is false).

remove_ts : 0..MAX_TIME;
join_ts : 0..MAX_TIME;
leave_ts : 0..MAX_TIME;

refresh_ts : 0..MAX_TIME;

authzTRM : boolean;

perform : boolean;

qa : question-response(request, TRUE, FALSE);
```

```
ASSIGN


   -- put timestamps in a state where AuthzTRM is false.
init(remove_ts) := case
   refresh : cc.remove_ts;
   TRUE : 0;
esac;

init(join_ts) := case
   refresh : cc.join_ts;
   TRUE : 0;
esac;

init(leave_ts) := case
   refresh : cc.leave_ts;
   TRUE : clock.t;
esac;

init(refresh_ts) := case
   refresh : clock.t;
   TRUE : 0;
   esac;

next(remove_ts) := case
next(refresh) : next(cc.remove_ts);
TRUE : remove_ts;
esac;

next(join_ts) := case
next(refresh) : next(cc.join_ts);
TRUE : join_ts;
esac;
```

```
next(leave_ts) := case
next(refresh) : next(cc.leave_ts);
TRUE : leave_ts;
esac;

next(refresh_ts) := case
   next(refresh) : next(clock.t);
   TRUE : refresh_ts;
esac;


init(perform) := FALSE;

next(perform) := case
   qa.respond & authzTRM : TRUE;
   TRUE : FALSE;
esac;

      init(authzTRM) := request & authzE & !stale;

      next(authzTRM) := case
         -- initialize to FALSE if authzE is false or if stale is TRUE.
         next(request) : next(authzE) & !next(stale);
         -- a refresh always just sets authzTRM to authzE.
         next(refresh) : next(authzE);
         TRUE : authzTRM;
      esac;

   DEFINE
      add_ts := user.add_ts;
      authzE := join_ts > leave_ts & user.add_ts >= join_ts &
         user.add_ts > remove_ts;
      stale := user.add_ts > refresh_ts;
      respond := qa.respond;
      request := events.request;
      refresh := events.refresh;
```

A.6 TRM_STRONG.mod

```
MODULE TRM(events, user, cc, clock)
VAR
   -- the CC timestamps go back one further than the clock to setup an
   -- initial state (where authzCC is false).

remove_ts : 0..MAX_TIME;
join_ts : 0..MAX_TIME;
leave_ts : 0..MAX_TIME;

refresh_ts : 0..MAX_TIME;

authzTRM : boolean;

perform : boolean;

qa : question-response(request, TRUE, FALSE);

ASSIGN


   -- put timestamps in a state where AuthzTRM is false.
init(remove_ts) := case
   refresh : cc.remove_ts;
   TRUE : 0;
esac;
```

```
init(join_ts) := case
   refresh : cc.join_ts;
   TRUE : 0;
esac;

init(leave_ts) := case
   refresh : cc.leave_ts;
   TRUE : clock.t;
esac;

init(refresh_ts) := case
   refresh : clock.t;
   TRUE : 0;
   esac;

next(remove_ts) := case
next(refresh) : next(cc.remove_ts);
TRUE : remove_ts;
esac;

next(join_ts) := case
next(refresh) : next(cc.join_ts);
TRUE : join_ts;
esac;

next(leave_ts) := case
next(refresh) : next(cc.leave_ts);
TRUE : leave_ts;
esac;

next(refresh_ts) := case
   next(refresh) : next(clock.t);
   TRUE : refresh_ts;
```

```
esac;


init(perform) := FALSE;

next(perform) := case
   qa.respond & authzTRM : TRUE;
   TRUE : FALSE;
esac;

      init(authzTRM) := request & refresh & authzE;

      next(authzTRM) := case
         -- initialize to FALSE if there is no refresh or authzE is FALSE
         next(request) : next(refresh) & next(authzE);
         -- a refresh always just sets authzTRM to authzE.
         next(refresh) : next(authzE);
         TRUE : authzTRM;
      esac;

   DEFINE
      add_ts := user.add_ts;
      authzE := join_ts > leave_ts & user.add_ts >= join_ts &
         user.add_ts > remove_ts;
      stale := user.add_ts > refresh_ts;
      respond := qa.respond;
      request := events.request;
      refresh := events.refresh;
```

A.7 USER.mod

```
MODULE USER(super_distro, cc)
VAR
```

```
      add_ts : 0..MAX_TIME;

ASSIGN

      init(add_ts) := case
         super_distro : cc.add_ts;
         TRUE : 0;
      esac;

      next(add_ts) := case
         next(super_distro) : next(cc.add_ts);
         TRUE : add_ts;
      esac;
```

A.8 question-response.mod

```
-----------------------------------------------------------------------
--This models the events required for a question and response.  The
--action for a question and response is as follows: a question
--is asked, then answered.  There are three essential parts to
--this action: the state when the question is recieved, the
--state when the question is answered, and then the state/states
--between the question and the answer.  This module is NOT
--concerned with the specific question.  It provides a
--mechanism to automate asking a question, such that the
--internal values can be used to query when the answerer is
--responding and in which state the answerer actually answers.
--
--Model: When [ask] becomes true, a question is asked and a response
--may be given (it also may never be given).  It is unclear
--whether asking before a response is given throws away the old
--question, or if the new question is discarded.  Therefore
--both models can be accomodated, however, ONE of the two
```

```
--questions MUST be discarded.  It is up to the implementation
--to make this decision.  It will based on the value of the
--response given @ the time when this question is asked and how
--that response is calculate.
--
--If a question is being asked (and has not been answered yet),
--then a response can ONLY be given if [ready] is true (meaning
--the answerer is "ready" to give a response) or when
--[force-response] is true (meaning the answerer is "consciously"
--answering in this state).  During states when the answerer is
--"ready", a response will be given at random, so being "ready"
--does NOT mean you will definitely respond.
--
--If [force-response] is ALWAYS false, this simulates an answerer
--that gives a response randomly while they are ready.  If, in
--addition, [ready] is ALWAYS true, then this simulates an
--answerer who responds at random time intervals after a question
--is asked.
--
--If [force-response] is ALWAYS true, then the answerer ALWAYS
--answers as soon as they are ready.  If BOTH [force-response]
--AND [ready] are ALWAYS true, then this models an answerer that
--ALWAYS answers the question immediately
-----------------------------------------------------------------------
MODULE question-response(ask, ready, force_response)
VAR
--these are described in the ASSIGN block
responding : boolean;
respond : boolean;
ASSIGN
--[responding] can ONLY change when a question is asked
-- OR when a response is given.
```

```
--responding is true EVERYTIME you ask a question and
--remains true up to and including the state in
--which you respond.
init(responding) := ask;

next(responding) := case
--no matter what, if the next value of ask
-- is true, then the next value of responding
-- HAS to be true.
next(ask) : TRUE;

--If you don’t ask another question in the
-- next state AND you responded in this state
-- then next value of responding should be
-- FALSE.
respond : FALSE;

--else remain the same
TRUE : responding;
esac;

--Only respond while responding (i.e. you ARE in the
-- process of responding) and when ready, and ALWAYS
-- respond when forced to do so (but still ready).

respond := case

--respond is FALSE ALWAYS while NOT
-- responding
responding & ready: case

--force-response is "greedy",
-- i.e. it ALWAYS forces a response.
```

```
force_response : TRUE;

       --Randomly respond while responding AND
       -- answerer is ready.  Notice that it IS
       -- possible to respond in the same state as the
       -- question.  But [responding] MUST be set
       -- first, and thus the question MUST happen
       -- "before" the response.  Code-wise, because
       -- the value of [respond] depends on the
       -- current value of responding, the value of
       -- [responding] MUST be set before [respond].
       -- Since [responding] is set by the current
       -- value of [ask], the event models by the
       -- boolean [ask] happens before the event
       -- modeled by the event [respond].  The other
       -- two ways that respond is set is by the
       -- PREVIOUS values of either [respond] or
   -- [responding].
TRUE : {TRUE, FALSE};
      esac;

--NEVER respond while NOT responding.
TRUE : FALSE;
esac;
------------------------------------------------------------------------
```

A.9 CLOCK.mod

```
MODULE CLOCK()
VAR
t : 1..MAX_TIME;

ASSIGN
```

```
init(t) := 1;
next(t) := case
t < MAX_TIME : t + 1;
TRUE : t;
esac;

   DEFINE
      CLOCK_STOPPED := t = MAX_TIME;
```

A.10 CLOCK1.mod

```
MODULE CLOCK()
   DEFINE
      t := 1;
      CLOCK_STOPPED := TRUE;
```

A.11 compile.sh

```
#!/bin/bash

temp_file="ohgiuweqrbhawefhiu.bak"

if [ $# -ne 3 ]; then
   echo "Usage: ./compile.sh <n> <TRM_FILE> <OUTPUT_FILE>"
   echo "Example: ./compile.sh 2 TRM_UNSAFE.mod main_unsafe.smv"
   exit 1
fi # else continue

if [[ $1 =~ [1-9][0-9]* ]]; then
   # go ahead and create SMV file
   if [ $1 -ne 1 ]; then
      echo "cat MAIN.mod EVENTS.mod CLOCK.mod CC.mod $2 USER.mod > $3"
      cat MAIN.mod EVENTS.mod CLOCK.mod CC.mod USER.mod question-response.mod \
         "$2" > "$3"
   else
      echo "cat MAIN.mod EVENTS.mod CLOCK1.mod CC.mod $2 USER.mod > $3"
      cat MAIN.mod EVENTS.mod CLOCK1.mod CC.mod USER.mod question-response.mod \
         "$2" > "$3"
   fi
else
   echo "Usage: ./compile.sh <n> <TRM_FILE> <OUTPUT_FILE>"
   echo "Example: ./compile.sh 2 TRM_UNSAFE.mod main_unsafe.smv"
   exit 2
fi

# SMV file is already created, now substitute argument given for MAX_TIME
#  everywhere
# Use sed to substitute MAX_TIME with first argument

echo "sed 's/MAX_TIME/$1/g' < $$3 > $temp_file"
sed s/MAX_TIME/$1/g < "$3" > "$temp_file"

echo "mv $temp_file $3"
mv "$temp_file" "$3"

echo "$3 SUCCESSFULLY CREATED!!!"
exit 0
```

REFERENCES

[1] Arosha K. Bandara, Emil C. Lupu, and Alessandra Russo. Using event calculus to formalise policy specification and analysis. In *POLICY*, pages 26–39, 2003.

[2] Adam Barth, Anupam Datta, John C. Mitchell, and Helen Nissenbaum. Privacy and contextual integrity: Framework and applications. *IEEE Symposium on Security and Privacy*.

[3] David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In *ACM SACMAT '10*.

[4] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[5] Omar Chowdhury, Andreas Gampe, Jianwei Niu, Jeffery von Ronne, Jared Bennatt, Anupam Datta, Limin Jia, and William H. Winsborough. Privacy promises that can be kept: a policy analysis method with application to the HIPAA privacy rule. In *18th ACM Symposium on Access Control Models and Technologies, SACMAT '13, Amsterdam, The Netherlands, June 12-14, 2013*, pages 3–14, 2013.

[6] A. Cimatti, E.M.Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[7] Edmund M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Language and Systems (TOPLAS)*, 8(2):244–263, 1986.

[8] Jason Crampton, Wing Leung, and Konstantin Beznosov. The secondary and approximate authorization model and its application to bell-lapadula policies. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*, SACMAT '06, pages 111–120, New York, NY, USA, 2006. ACM.

[9] David Ferraiolo and Richard Kuhn. Role-based access control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[10] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael C. Tshchantz. Verification and change-impact analysis of access-control policies. In *ICSE*, pages 196–205. ACM Press, 2005.

[11] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: theory, implementation and applications. In *ACM CCS '11*.

[12] D. Gilliam, J. Powell, and M. Bishop. Application of lightweight formal methods to software security. In *Proceedings of the 14th IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE 2005)*, 2005.

[13] J.A. Goguen and J. Meseguer. Security policies and security models. *IEEE Symposium on Security and Privacy*, 12, 1982.

[14] Frode Hansen and Vladimir Oleshchuk. Conformance checking of RBAC policy and its implementation. In *Information Security Practice and Experience*, volume 3439 of *LNCS*, pages 144–155. Springer Berlin/Heidelberg, 2005.

[15] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Comm. of the ACM*, pages 461–471, August 1976.

[16] S. Jha and T. Reps. Model checking SPKI/SDSI. volume 12, pages 317–353, 2004.

[17] Ram Krishnan, Jianwei Niu, Ravi Sandhu, and William H. Winsborough. Stale-safe security properties for group-based secure information sharing. In *the 6th ACM workshop on Formal methods in security engineering FMSE*, pages 53–62, 2008.

[18] Ram Krishnan, Jianwei Niu, Ravi Sandhu, and William H. Winsborough. Group-centric secure information sharing models for isolated groups. *ACM Trans. Infor. Syst. Secur.*, 14(3):23:1–23:29, 2011.

[19] Ram Krishnan, Ravi Sandhu, Jianwei Niu, and William H. Winsborough. A conceptual framework for group-centric secure information sharing. In *Proc. of the 4th ACM International Symposium on Information, Computer, and Communications Security*, pages 384–387, 2009.

[20] L. Lamport. Logical foundation, distributed systems-methods and tools for specification. *Lecture Notes in Computer Science*, 190, 1985.

[21] L. J. LaPadula and D. E. Bell. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, 1973. Reprinted in *J. of Computer Security*, vol. 4, no. 2–3, pp. 239–263, 1996.

[22] A.J. Lee, K. Minami, and M. Winslett. Lightweight cnsistency enforcement schemes for distributed proofs with hidden subtrees. *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 101–110, 2007.

[23] A.J. Lee and M. Winslett. Safety and consistency in policy-based authorization systems. *Proceedings of the 13th ACM conference on Computer and communications security*, pages 124–133, 2006.

[24] Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. *Security and Privacy, IEEE Symposium on*, 0:114, 2002.

[25] Gavin Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996.

[26] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *10th IEEE workshop on Computer Security Foundations*, pages 18–30, 1997.

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

[28] Michael J. May, Carl A. Gunter, and Insup Lee. Privacy APIs: Access control techniques to analyze and verify legal privacy policies. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop*, 2006.

[29] Amir Pnueli, Sitvanit Ruah, , and Lenore Zuck. Automatic deductive verification with invisible invariants. In *TACAS: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, pages 82–97, 2001.

[30] Ravi Sandhu, Edward Coyne, Hal Feinstein, and Charles Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feburary 1996.

[31] Andreas Schaad, Volkmar Lotz, and Karsten Sohr. A model checking approach to analysis organizational controls. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT06)*, pages 139–149, 2006.

[32] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[33] A. Prasad Sistla and Min Zhou. Analysis of dynamic policies. *Information and Computation*, 206(2-4):185–212, 2008.

[34] A.S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2007.

[35] TCG. TCG specification architecture overview. *http://www.trustedcomputinggroup.org*, 2011.

[36] Qiang Wei, Jason Crampton, Konstantin Beznosov, and Matei Ripeanu. Authorization recycling in hierarchical rbac systems. *ACM Trans. Inf. Syst. Secur.*, 14(1):3:1–3:29, June 2011.

[37] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.

[38] Nan Zhang, Mark Ryan, and Dimitar P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security*, 16(1):1–61, 2008.

VITA

Jared F. Bennatt grew up about five miles outside of Vanderbilt, TX, a small town near the city of Victoria, TX. In 2005 he received his Bachelor of Science in Physics from the University of Texas at Austin. Jared F. Bennatt re-entered college in the fall of 2008 at the University of Texas at San Antonio where he pursued a degree in Computer Science. The following spring he began working with Dr. Jianwei Niu in the research area of security policy enforcement and analysis. He entered the graduate program in the fall 2010. During the following years he was able to contribute to research projects and co-authored a research paper on privacy policy enforcement (specifically analyzing the HIPAA privacy). In July 2015, he completed all of the necessary coursework for an M.S. in Computer Science as required by the Graduate School of the University of Texas at San Antonio. Jared F. Bennatt has accepted employment as a high school math teacher in the Victoria, TX area and is hoping to apply the skills learned while at UTSA to aid instruction.