Stale-Safe Security Properties for Group-Based Secure Information Sharing

August 12, 2008
Technical Report CS-TR-2008-012
Department of Computer Science
The University of Texas, San Antonio, TX 78249.

Ram Krishnan
George Mason University
Fairfax, VA, USA
rkrishna@gmu.edu

Jianwei Niu
Univ of Texas at San Antonio
San Antonio, TX, USA
niu@cs.utsa.edu

Ravi Sandhu
Univ of Texas at San Antonio
San Antonio, TX, USA
ravi.sandhu@utsa.edu

William H. Winsborough
Univ of Texas at San Antonio
San Antonio, TX, USA
wwinsborough@acm.org

ABSTRACT

Attribute staleness arises due to the physical distribution of authorization information, decision and enforcement points. This is a fundamental problem in virtually any secure distributed system in which the management and representation of authorization state is not centralized. This problem is so intrinsic, it is inevitable that access control will be based on attribute values that are stale. While it may not be practical to eliminate staleness, we can *limit* unsafe access decisions made based on stale subject and object attributes. In this paper, we propose and formally specify four stale-safe security properties of varying strength which limit such incorrect access decisions. We use Linear Temporal Logic (LTL) to formalize these properties making them suitable to be verified by using model checking. We show how these properties can be applied in the specific context of group-based Secure Information Sharing (g-SIS) as defined in this paper. We specify the authorization decision/enforcement points of the g-SIS system as a Finite State Machine (FSM) and show how this FSM can be modified so as to satisfy one of the stale-safe properties. We formally verify that this FSM satisfies the stale-safe property using a mature model checker called Symbolic Model Verifier (SMV).

Keywords

Attribute Staleness, Security Properties, Model Checking, Secure Information Sharing, Trusted Computing

.

1. INTRODUCTION

The concept of a stale-safe security property is based on the following intuition. In a distributed system authoritative information about subject 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 subject and object attributes at one or more authorization decision points, and are enforced at one or more authorization enforcement points. Because of the physical distribution of authorization information, decision and enforcement points, and consequent inherent network latencies, it is inevitable that access control will be based on attributes values that are stale (i.e., not the latest and freshest values). In a highly connected high-speed network these latencies may be in milliseconds, so security issues arising out of use of stale attributes can be effectively ignored. In a practical real-world network however, these latencies will more typically be in the range of seconds, minutes and even days and weeks. For example, consider a virtual private overlay network on the internet which may have intermittently disconnected components that remain disconnected for sizable time periods. In such cases, use of stale attributes for access control decisions is a real possibility and has security implications.

We believe that, in general, it is not practical to eliminate the use of stale attributes for access control decisions.[1] In a theoretical sense, some staleness is inherent in the intrinsic limit of network latencies, of the order of milliseconds

[1]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 paper. For example, if an employee is dismissed there may be a lag between the time that action takes effect and when it is recorded in cyberspace. The lag we are concerned with arises when the authoritative information point knows that the employee has been dismissed but at some decision point the employee's status is still showing as active.

in modern networks. We are more interested in situations where staleness is at a humanly meaningful scale, say minutes, hours or days. In principle, with some degree of clock synchronization amongst the authorization information, decision and enforcement points, it should be possible to determine and bound the staleness of attribute values and access control decisions. For example, a SAML assertion produced by an authorization decision point includes a statement of timeliness, i.e., start time and duration for the validity of the assertion. It is upto the access enforcement point to decide whether or not to rely on this assertion or seek a more timely one. Likewise a signed attribute certificate will have an expiry time and an access decision point can decide whether or not to seek updated revocation status from an authorization information point.

Given that the use of stale attributes is inevitable, the question is how do we safely use stale attributes for access control decisions and enforcement? The central contribution of this paper is to formalize this notion of "safe use of a stale property" in the specific context of group-based secure information sharing (g-SIS) as defined in this paper. We also demonstrate specifications of systems that provably do and do not satisfy this requirement as revealed by model checking of the specifications. We believe this formalism can be extended to more general contexts beyond the group-based secure information sharing considered in this paper, but this is beyond the current scope. We believe that the requirements for "safe use of a stale property" identified in this paper represent fundamental security properties the need for which arises in virtually any secure distributed systems in which the management and representation of authorization state is not centralized. In this sense, we suggest that we have identified and formalized a *basic security property*, in the same sense that non-interference [21] and safety [11] are basic security properties that are desirable in a wide range of secure systems.[2]

Specifically, we present formal specifications of four "stale-safe" properties. The most basic and fundamental requirement we consider deals with ensuring that while authorization data cannot be propagated instantaneously throughout the system, it is highly desirable to ensure that a requested action was definitely authorized at some point in the recent past. With staleness we may allow the authorization to hold for longer than it should have, but there is no doubt that the access was authorized in the past. Two additional requirements can be added to obtain properties that are stronger with respect to when it is required that the action be authorized. The first is that, to be permitted, it must be confirmed that a requested action is authorized at a point in time after the request and before the action is performed. The second requirement bounds the elapsed time between the point at which the authorization is confirmed and the point at which the action is performed. Because both of these requirements can be added singly or in combination, we obtain four different stale-safe properties.

We formalize these four properties in Linear Temporal Logic (LTL), making them suitable to be verified by using model checking. We show how these properties can be applied in the specific application domain of group-based secure information sharing (g-SIS). We specify one component of a g-SIS system as a finite state machine (FSM). We present two FSM's—one that does not and one that does satisfy the weakest of our state-safe properties. We formally verify the correct specification using model checking. We also use the model checker to obtain an execution trace of the incorrect FSM, which could be used by a designer to correct that FSM.

In section 2, we discuss the group-based Secure Information Sharing problem which will be used throughout the paper to illustrate the stale-safe properties. In section 3, we formalize the stale-safe security properties using Linear Temporal Logic. We specify a weak and strong version of the properties each of which is further restricted with a notion of elapsed time between the time at the which the operation is authorized and performed. In section 4, we construct a Finite State Machine (FSM) that enforces a specific policy for g-SIS. We formally verify the FSM against the weak property stated in section 3 using Model Checking. We also specify an FSM that appears to be a natural candidate for g-SIS but fails this property. In section 5, we list related work and we conclude in section 6.

2. GROUP-BASED SECURE INFORMATION SHARING (g-SIS)

Secure Information Sharing (SIS) or sharing information *while* protecting it is one of the earliest problems to be recognized in computer security, and yet remains a challenging problem to solve. A detailed discussion of SIS problem motivation and solution approaches can be found in [14]. The central problem is that copies of digital information are easily made and controls on the original typically do not carry over to the copies. One approach tried in the past has been to tie access control to each copy also so that copies are as tightly controlled as the original. The most common form of this approach is so-called mandatory or lattice-based access control [28] where copies are also labeled to reflect security sensitivity of the original. More recently, an alternate approach has emerged wherein plaintext unprotected copies are prohibited, while encrypted protected copies can be freely made. This implies that access controls need to be enforced on the client machines where the content is decrypted and displayed, so as to ensure that only authorized users get to see the content and that they are unable to make plaintext unprotected copies. There has been considerable interest in this approach, initially driven by the forces of digital rights management for entertainment content seeking to protect revenue but more generally seeking to protect content for its sensitivity.

2.1 Objectives

The group-based SIS (g-SIS) problem [14] is motivated by the need to share sensitive information amongst a group of authorized users. For simplicity we only consider the case of read access to the objects in the group. Every member of

[2]The work of Lee et al [15, 16] is the closest to ours that we have seen in the literature, but focuses exclusively on the use of attribute certificates, called credentials, for assertion of attribute values. Lee et al focus on the need to obtain fresh information about the revocation status of credentials to avoid staleness. As we will see 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.

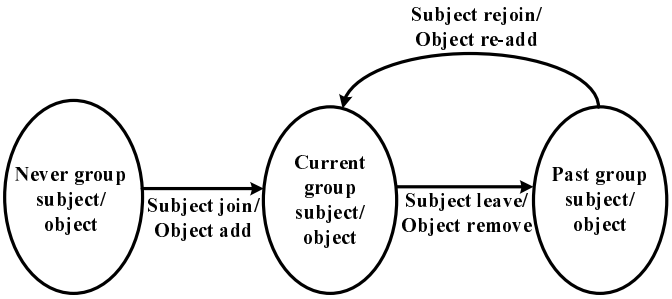

Figure 1: Subject and object membership states.

the group is authorized to read group objects. For purpose of this paper, we specify the following objectives for the g-SIS problem. For brevity, the terms subjects and objects refer to subjects and objects that belong to the group.

1. Objects are always protected (encrypted) and never exists in plain text except when viewed.

2. Objects are assumed to be available via super-distribution. This simply means that the objects are protected once and subjects may access them when authorized—objects are not individually prepared for each subject. We limit super-distribution to occur within a group.

3. Subjects can access objects off-line without involving the server using trusted access machines. The degree of trust required on the access machines may vary depending on the application and policy. In one case, the access machines may be implicitly trusted because of its physical location (e.g. access machines in an organization). In a completely distributed setting, a Trusted Reference Monitor (TRM) needs to be present on the access machines that can verify the integrity of the system and enforce the authorization policies in a trustworthy manner [27, 20]. This can be achieved using integrity measurements, remote attestation and other features enabled by Trusted Computing Technology [2] or software analogs of this technology.[3]

4. Each group has a Group Administrator (GA) who controls group membership and policies. The GA can add or remove subjects and objects from the group. We do not specify how an admin is appointed. The admin may or may not be a member of the group. Each group also has a Control Center (CC), a server that maintains authoritative subject and object attributes and provides group credentials to new members. Changes in subject and object attributes or group policies are updated by the GA at the CC and this change will eventually be propagated to the subject's access machines (discussed later in detail).

We now digress briefly to compare g-SIS with a related problem –broadcast/multicast encryption. Member management in g-SIS scenario sharply differs from Secure Internet Multicast. In multicast, as users join and leave a group, remaining members go through a re-key process thereby refreshing the group key [24]. However, for secure information sharing, such a requirement is extremely un-friendly because members need not be always connected to a server to access the objects. Thus, continuing our list of objectives, we have the following.

5. When a subject joins or leaves the group, remaining members should not be affected. In other words, join and leave of a subject should be completely oblivious to other subjects. Remaining subjects should not be forced to be online or go through a re-key process[4].

6. Secure multicast focuses on maintaining forward and backward secrecy of data [24]. Forward-secrecy requires that a leaving subject should not be able to read data that will subsequently be exchanged in the future. Backward-secrecy requires that a joining subject should not be able to read data exchanged amongst the remaining subjects in the past. However, *information* sharing may not be limited to forward and backward secrecy. For g-SIS flexible membership policies may be required. When a new user joins the group, whether he can access any group objects created prior to his membership is policy-dependant. Objects created after he joins the group are accessible. When a member leaves the group, whether he can continue to access objects created during his membership period is policy-dependant. However, he cannot access any object exchanged in the future within the group.

2.2 Group Management and Policy Enforcement

Subjects and objects in a group go through various states as shown in figure 1. Different access policies are possible depending on the relative state of subjects and objects . For example, a current subject could be allowed access only to current objects or also to objects created before the subject joined the group. Similarly, a past subject may lose access to all objects or retain access to objects created during his membership period. When a subject rejoins the group, he may either gain access to objects created during his past membership or simply join the group as a new subject. Similarly, many different object policies are possible. Detailed discussions can be found in [14]. Each group may thus pick a specific set of group-level access policies for subjects and objects.

Figure 2 shows one possible enforcement model for the g-SIS problem and illustrates the interaction of various components in g-SIS. The Group Administrator (GA) controls group membership and policies. The Control Center (CC) is responsible for maintaining authoritative group credentials and attributes of group subjects and objects on behalf of the GA.

[3]It is generally accepted that software-only solutions will provide a lower degree of assurance than solutions with a hardware root of trust. The issues discussed in this paper are orthogonal to assurance so will apply to both software and hardware based solutions.

[4]Members should not be asked to re-key or contact a server to get a new key. Note that re-keying is not an efficient solution in SIS as the member needs to keep track of which document was encrypted with which key. As users join and leave a group, the remaining members will need to go through a re-key process resulting in encrypting documents with different keys along the time line. One cannot discard the old key (as done in multicast) as disseminated documents encrypted with the old key continue to persist.

- *Subject Join*: Joining a group involves obtaining authorization from the GA followed by obtaining group credentials from the CC. In step 1.1, the subject contacts the GA using an access machine and requests authorization to join a group. The GA verifies that the subject is not already a member and authorizes the subject in step 1.2 (by setting AUTH to TRUE). The subject furnishes the authorization to join the group and the evidence that the access machine is in a good software state to the CC in step 1.3. The CC remotely verifies GA's authorization, if the subject's access machine is trustworthy (using the evidence) and has a known Trusted Reference Monitor (TRM) that is responsible for enforcing policies. In step 1.4, the CC provisions the attributes. sid is the Subject Id, Join_TS is the time-stamp of subject join (set to a non-NULL value), Leave_TS is the time at which a subject leaves the group (initially set to NULL), gKey is the group key using which group objects can be decrypted, Policy is the group's access policy, ORL is the Object Revocation List which lists the objects removed from the group.

- *Policy Enforcement*: From here on, the subject is considered a group member and may start accessing group objects (encrypted using the group key) as per the group policy and using the credentials obtained from the CC. This is locally mediated and enforced by the TRM. Note that the objects are available via super-distribution and because of the presence of a TRM on subject's access machines, objects may be accessed offline conforming to the policy. For example, the TRM on an access machine may allow the subject to access objects added after subject joined the group and disallow access to objects added before he/she joined the group. Such decisions can be made by using the join and leave time-stamps of subject, add and remove time-stamps of object and comparing their relative values. Objects may be added to the group by subjects by obtaining an add time-stamp (setting an Add_TS attribute for the object) from the CC. We assume object attributes are embedded in the object itself. Note that due to super-distribution, the remove time-stamps for objects cannot be embedded in the object (since there could be many copies of the same object). Instead, an Object Revocation List (with the remove time-stamps of object ids) is provisioned on the access machine.

- *Attribute Refresh*: Since subjects may access objects offline, the access machines need to connect to the CC and refresh subject attributes periodically. How this is done is a matter of policy and/or practicality. For example, a refresh could take effect in an access machine based on time or a usage count. Offline access to secure clock may be impractical in many circumstances. Usage count is a practical approach when using Trusted Computing Technology. A discussion on using the monotonic counters in the Trusted Platform Module can be found in [30]. The usage count limits the number of times the credentials may be used to access group objects (like consumable rights). Thus objects may be accessed until the usage count is exhausted and the access machine will be required to refresh attributes in step 3.1 and 3.2 before any further access can be granted. Attributes RT and N represent the refresh time-stamp and usage count of the subject respectively.

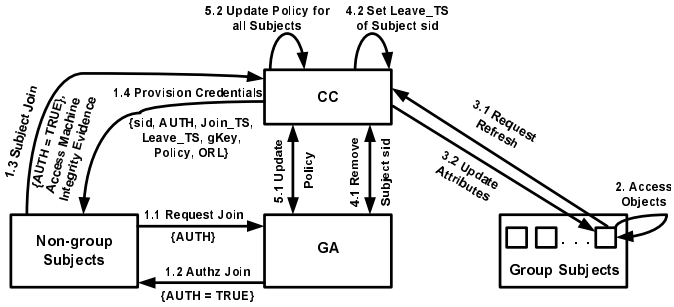

Figure 2: g-SIS System.

- *Administrative Actions*: The GA may have to remove a subject or object from the group or update group policy. In step 4.1, the GA instructs the CC to remove a subject. The CC in turn marks the subject for removal by setting the subject's Leave_TS attribute in step 4.2. This attribute update is communicated to the subject's access machine during the refresh step 3.1 and 3.2. In the case of object removal, the ORL is updated with the object's id and Remove_TS. Policy updates (or any other update for that matter) are handled in a similar manner as shown in step 5.1 and 5.2.

As you can see, there is a delay in attribute update in the access machine that is defined by the refresh window. Although a subject may be removed from the group at the CC, the access machines will let subjects access group objects until the subject attributes are refreshed at the next refresh step. This access violation is due to attribute staleness that is inherent to any distributed system however short the refresh window is. We discuss this topic in detail in the subsequent sections. This paper does not focus on building trusted systems to realize the architecture in figure 2 and it is a work in progress. This is an area of active work and we direct interested readers to [27] and [20], to cite a few.

3. STALE SAFE SECURITY PROPERTIES FOR g-SIS

As discussed earlier, in distributed systems access decisions are almost always based on stale-attributes and stale-attributes lead to critical access violations. In this section we propose Stale-safe Security Properties that limit such access violations. Note that it is impossible to completely eliminate staleness in practice and thus our intension here is best effort. We first discuss a few scenarios were stale attributes lead to access violations using the g-SIS example and informally discuss the stale-safe properties. We formalize them next.

3.1 System Characterization

The g-SIS system consists of subjects and objects, trusted access machines (using which objects are accessed), a GA and a CC. Access machines maintain a local copy of subject attributes which they refresh periodically with the CC. Object attributes are part of the object itself. A removed

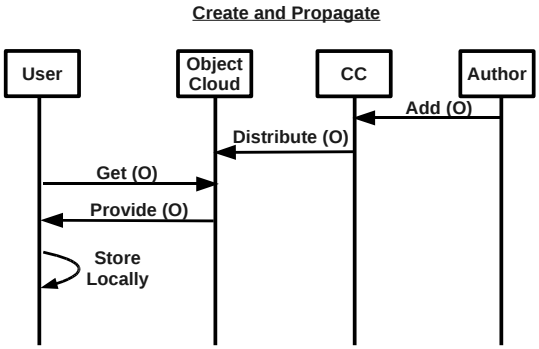

Figure 3: Super-distribution.

object is listed in the Object Revocation List (ORL), which are provided to access machines as part of refresh. To easily illustrate the properties, we assume that each subject is tied to an access machine from which objects are accessed and there is a single GA and single CC per group. Also, we assume that the refresh is based on usage count. Suppose a policy that a subject is allowed to access an object as long as both the subject and object are current members of the group and the object was added after the subject joined the group. Thus the g-SIS system can be characterized as follows:

Subject attributes	{id, Join_TS, Leave_TS, ORL, gKey, RT, N}
Object attributes	{id, Add_TS}.
Refresh Time (RT)	Access machine contacts CC to refresh subject attributes and ORL.
Refresh Window (RW)	Time interval between two RT's (depends on how quick the usage count is exhausted).
Access Policy	$\text{Authz}_\text{P}(S, O, OP) \rightarrow O \notin \text{ORL}(S) \wedge$ $\text{Leave\_TS}(S) = \text{NULL} \wedge$ $\text{Join\_TS}(S) \leq \text{Add\_TS}(O)$.

Figure 3 illustrates super-distribution. An Author (a group subject) creates an object, encrypts the object using the group key (mediated by TRM) and sends it to the CC for approval and distribution. The CC (or possibly a GA) approves the object, time-stamps object add and releases this protected object for distribution. Since the object is protected, it is not necessarily guarded by the CC. Instead it is made available to subjects by distribution through networks such as WWW, email, etc. This infospace is called the Object Cloud in figure 3. The User (another group subject) can obtain these encrypted objects and store them locally in his/her access machine. The sequence diagram in Figure 4 illustrates the staleness problem. The User and the TRM interacts with the GA and CC in steps 1 to 5 to join the group. The TRM refreshes attributes with the CC in steps 6 and 7. Briefly after the refresh, the GA removes this subject by setting his/her Leave_TS attribute at the CC (a non-null value). Note that this step is not visible to the TRM until the next refresh steps 11 and 12. In the mean time, the User may request access to objects the were obtained via super-distribution (step 9). "Create and Propagate" refers to the scenario in figure 3. At this point, the TRM evaluates the policy based on the attributes that it maintains. This should be successful and the object is displayed to the user in step 10. Note the difference in Leave_TS values between the CC and TRM. Only after the following refresh (steps 11 and 12) does the TRM notice that the subject has been removed from the group and denies any further access (steps 13 and 14).

Figure 5 shows a timeline of events involving a single group. Subject $S1$ joins the group and the attributes are refreshed with the CC periodically. RT represents the time at which refreshes happen. The time period between any two RT's is a Refresh Window, denoted RW_i. After join, RW_0 is the first window, RW_1 is the next and so on. Suppose RW_4 is the current Refresh Window. Objects $O1$ and $O2$ were added to the group by *some* group subject (or the GA) during RW_2 and RW_4 respectively and they are available to $S1$ via super-distribution. In RW_4, $S1$ requests access to $O1$ and $O2$. An access decision will be made by the TRM in the access machine as per the attributes obtained at the latest RT.

As you can see, our access policy will allow access to both $O1$ and $O2$. However it is possible that S1 was removed by the GA right after the last RT and before $Request(S1, O1, access)$ in RW_4 (see figure 4). Ideally, S1 should not be allowed to access both $O1$ and $O2$.

From a confidentiality perspective in information sharing, granting $S1$ access to $O1$ is relatively less of a problem than granting access to $O2$. This is because the CC or the GA can assume that $S1$ was always authorized access to $O1$ and hence information has already been released to $S1$. In the worst case, $S1$ continues to access the same information ($O1$) until the next RT. However, $S1$ never had an authorization to access $O2$ and letting $S1$ access $O2$ means that $S1$ has gained knowledge of new information. This is a critical violation and should not be allowed. Such scenarios are what our stale-safe security properties address. A subject cannot access an object if it was added to the group after the last refresh time even if the authorization policy allows access. This can be achieved by comparing the Add_TS (O) with the most recent refresh time-stamp (RT_recent). Thus the access decision for a stale-safe g-SIS system should be made as follows:

Access Policy	$\text{Authz}_\text{P}(S, O, OP) \rightarrow O \notin \text{ORL}(S) \wedge$ $\text{Leave\_TS}(S) = \text{NULL} \wedge$ $\text{Join\_TS}(S) \leq \text{Add\_TS}(O)$.
Stale-safe Property	$\text{SafeP}\ (S, O) \rightarrow \text{Add\_TS}\ (O) < \text{RT}_\text{recent}$
Stale-safe Access Policy	$\text{Authz}_\text{P}(S, O, OP) \wedge \text{SafeP}(S, O)$

The property we discussed considers attributes to be stale if it is time-stamped later than the last refresh time-stamp of the access machine. A more strict property may require the access machine to refresh attributes before granting any access. That is, when $S1$ requests access to $O1$, the stricter version of the stale-safe property mandates that the access machine refreshes the subject attributes before making an authorization decision. Further, it is natural to consider elapsed time since the last refresh to be an important issue in limiting staleness of authorization data. We formalize these notions in the following subsection.

3.2 Formal Property Specification

In this section we use Linear Temporal Logic (LTL) [18] to

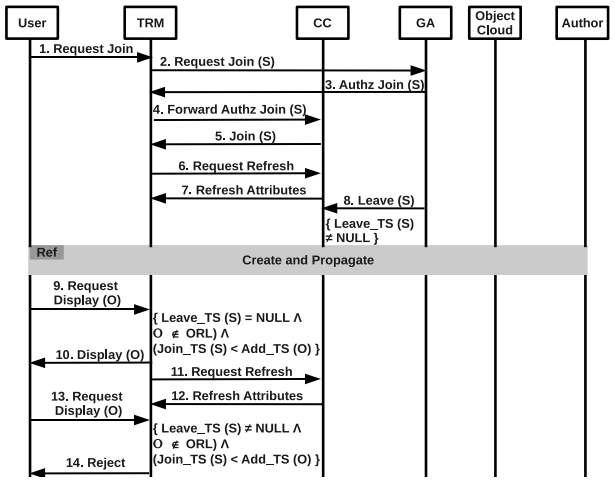

Figure 4: Staleness Illustration.

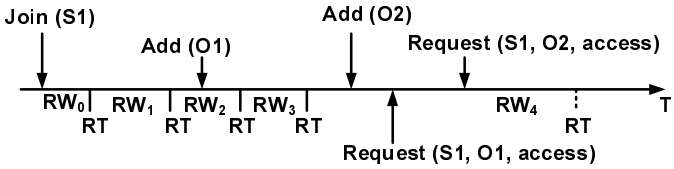

Figure 5: Events on a time line to illustrate stale-safe properties in g-SIS.

specify four different formal stale-safety properties of varying strength. Temporal logic is a specification language for expressing properties related to a sequence of states in terms of temporal logic operators and logic connectives (e.g., $\wedge$ and $\vee$). Temporal logic operators are of two types: Past and Future. The past operators $\ominus$ and Since (read previous and since respectively) have the following semantics. $\ominus\, p$ means that the formula p was true in the previous state. Note that $\ominus\, p$ is false in the very first state. p Since q means that q has happened sometime in the past and p has held continuously following the last occurrence of q to the present. The future operators $\bigcirc$, $\diamond$, and $\Box$ represent next state, some future state, and all future states respectively. For example, $\Box p$ means that formula p is true in all future states. Also, the formula p *until* q (read p until q) means that q will occur sometime in the future and p will remain true at least until the first occurrence of q.

Our formalization uses the following predicates:

request (S, O, OP)	The subject requests to perform an action OP on an object.
Authz_P (S, O, OP)	S is authorized to perform an action OP on O.
Join (S) and Leave (S)	Subject Join and Leave events.
Add (O) and Remove (O)	Object Add and Remove events.
perform (S, O, OP)	S performs an action OP on O in the current state.
RT (S)	The TRM contacts the CC to update subject attributes.

In the forthcoming formulae (φ_0, φ_1 and φ_2) and throughout this paper, we drop the corresponding parameters S, O and OP in these predicates for clarity. They should however be interpreted with the respective semantics described above.

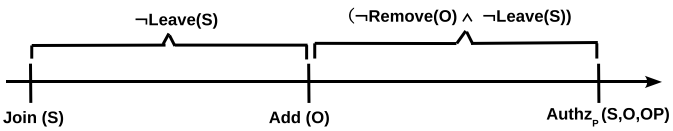

Figure 6: Access Policy (Authz_P).

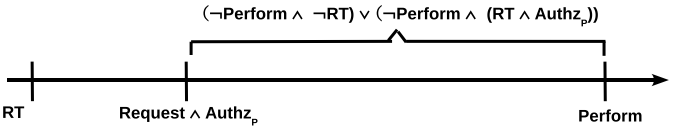

Figure 7: Formula φ_0.

3.2.1 *Stale-unsafe Access Decision*

We first formalize a stale-unsafe access decision using the access policy discussed in section 3.1 as an example. Authz_P below is the same policy represented using LTL. Formula φ_0 formalizes an access decision that is stale-unsafe.

$$\text{Authz}_\text{P} \equiv (\neg\text{Remove} \wedge \neg\text{Leave}) \text{ Since } ((\text{Add} \wedge \neg\text{Leave}) \text{ Since Join})$$

$$\varphi_0 \equiv \ominus\, (\neg\text{perform} \wedge (\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_\text{P}))) \text{ Since } (\text{request} \wedge \text{Authz}_\text{P})$$

Figure 6 illustrates Authz_P. Authz_P says that S is allowed to perform an action OP on O if prior to the current state the object was added to the group and both the subject and object have not left the group since. Also, the subject joined the group prior to the time the at which the object was added to the group and has not left the group ever since.

Figure 7 illustrates formula φ_0. φ_0 says that the operation was authorized at the time of request. Prior to the current state, the operation has not been performed since it was requested. Also since it was requested, any refreshes that may have occurred indicated that the operation was authorized ($\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_\text{P})$).

Definition 3.1 (Staleness Unaware). *A Finite State Machine (FSM) is* staleness unaware *if it satisfies the following LTL formula:*

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

Observe that in a *Staleness Unaware* FSM, verifying that Authz_P holds at the time of request will allow the subject access objects that were added during the time between RT and (request $\wedge$ Authz_P) in figure 7. This can be clearly seen by converging Authz_P in figure 6 with that in figure 7. As discussed earlier, it is unsafe to let group subjects access these objects before a refresh can confirm the validity of their group membership.

We now specify stale-safe security properties of varying strength. The weakest of the properties we specify requires that a requested action be performed only if a refresh of subject attributes and ORLs 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

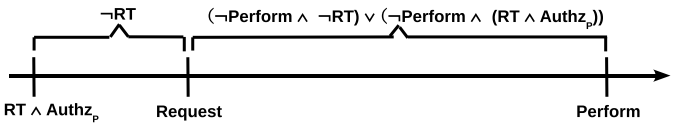

Figure 8: Formula φ_1.

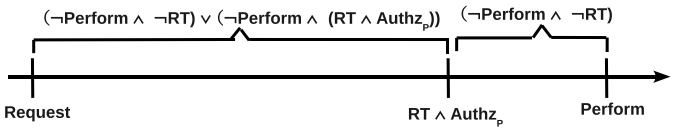

Figure 9: Formula φ_2.

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.

3.2.2 Weak Stale-safe Security Property

Let us introduce two formulas formalizing pieces of stale-safe security properties. Intuitively, φ_1 can be satisfied only if authorization was confirmed prior to the request being made. On the other hand, φ_2 can be satisfied only if authorization was confirmed after the request. Note that weak stale safety is satisfied if either of these is satisfied prior to a requested action being performed.

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

$$\varphi_2 \equiv \ominus\,(\neg\text{perform} \wedge \neg\text{RT})\ \text{Since}\,(\text{RT} \wedge \text{Authz}_\text{P} \wedge \\ ((\neg\text{perform} \wedge (\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_\text{P}))) \text{ Since request}))$$

Figure 8 illustrates formula φ_1. φ_1 says that prior to the current state, the operation has not been performed since it was requested. Also since it was requested, any refreshes that may have occurred indicated that the operation was authorized ($\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_\text{P})$). Finally, a refresh must have occurred prior to the request and the last time a refresh was performed prior to the request, the operation was authorized.

Observe that formula φ_1 mainly differs from φ_0 on the point at which Authz_P is evaluated. Referring to figure 8, evaluating Authz_P at the latest RT guarantees that requests to access any object that may be added during the following refresh window will be denied.

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

Figure 9 illustrates formula φ_2. φ_2 does not require that there was a refresh prior to the request. Instead it requires that a refresh occurred between the request and now. It further requires that the operation has not been performed since it was requested and that every time a refresh has occurred since the request, the operation was authorized.

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

We call perform $\rightarrow \varphi_1$ *backward-looking stale safety*, as it does not require that a confirmation of authorization occur after the request has been received. We call perform $\rightarrow \varphi_2$ *forward-looking stale safety*, as it requires that confirmation of authorization is obtained after the request, before the action is performed.

Definition 3.2 (Weak stale safety). *An FSM has the* weak stale-safe security property *if it satisfies the following LTL formula:*

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

3.2.3 Strong Stale-safe Security Property

Forward-looking stale safety is strictly stronger than weak stale safety. For this reason, and because, unlike backward-looking stale safety, it is a reasonable requirement for controlling many operations, we give it a second name.

Definition 3.3 (Strong stale safety). *An FSM has the* strong stale-safe security property *if it satisfies the following LTL formula:*

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

3.2.4 Quantifying "Freshness" of Authorization

Let us now consider how to model requirements that constrain the actual time at which actions such as attribute refresh occur. For this we introduce a sequence of propositions $\{P_i\}_{0 \le i \le n}$ that model n time intervals (owing to the propositional nature of LTL, we can model only a finite number of time intervals.). These propositions partition each trace into contiguous state subsequences that lie within a single time interval, with each proposition becoming true immediately when its predecessor becomes false. They can be axiomatized as follows:

$$P_1 \text{ Until } (\Box\neg P_1 \wedge \\ (P_2 \text{ Until } (\Box\neg P_2 \wedge \\ (P_3 \text{ Until } (... \\ \text{Until } (\Box\neg P_{n-1} \wedge \Box P_n)...)))))$$

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

constants.

$$\varphi_1(k) \equiv \ominus(\neg\text{perform} \wedge (\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_\text{P}))) \text{ Since } (\text{request} \wedge \bigvee_{max(0,k-\ell_2)\le i\le k} P_i \wedge (\neg\text{RT Since } (\text{RT} \wedge \text{Authz}_\text{P} \wedge \bigvee_{max(0,k-\ell_1)\le i\le k} P_i)))$$

$$\varphi_2(k) \equiv \ominus(\neg\text{perform} \wedge \neg\text{RT}) \text{ Since } (\text{RT} \wedge \text{Authz}_\text{P} \wedge \bigvee_{max(0,k-\ell_1)\le i\le k} P_i \wedge ((\neg\text{perform} \wedge (\neg\text{RT} \vee (\text{RT} \wedge \text{Authz}_\text{P}))) \text{ Since } (\text{request} \wedge \bigvee_{max(0,k-\ell_2)\le i\le k} P_i))$$

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

Definition 3.4 (Timely, weak stale safety). *An FSM has the* timely, weak stale-safe security property *if it satisfies the following LTL formula:*

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

Definition 3.5 (Timely, strong stale safety). *An FSM has the* timely, strong stale-safe security property *if it satisfies the following LTL formula:*

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

3.3 Stale-safe Systems

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

3.4 Extensions

In the earlier section, we discussed the most fundamental stale-safe properties. We now consider stale-safety in the context of a truly distributed system with multiple access machines for a subject, multiple CC'S and GA's and multiple group memberships of subjects and objects. As you can see, this is a broad and complex problem that requires in-depth research and is beyond the scope of this paper. However, we informally articulate the staleness problem and desirable properties in such a scenario.

- *Multiple Access Machines*: Consider a scenario where a group subject may access group objects from multiple access machines. Each machine maintains a local set of attributes and they are not only stale with respect to the CC but also with respect to other access machines. If not careful, a subject may maintain various membership states across multiple access machines. For example, the same subject could be current member in one machine, but could leave the group and rejoin from another machine. As per our access policy earlier, the subject ends up accessing two sets of objects from two different machines –one as a current member and the other as a rejoined member at the same time[5]. Such violations may be serious in the context of mutual exclusion. Thus the stale-safe property for multiple access machines should make sure that the subject's membership state is consistent across all the machines. When a subject attempts to rejoin a group from one access machine, the CC should instruct the TRM's on subject's other access machines to revoke all access until the subject leaves and rejoins on all machines.

- *Multiple CCs*: Recall that the GA updates subject attributes at the CC which in turn is updated at the subject's access machine during a refresh. This property deals with attribute staleness with respect to GA's updates in the case of multiple CCs. A GA may update attributes at one CC and the attributes at other CCs remain stale until this update is propagated across all the CCs. In such a scenario, a refresh from one of the CCs by the access machine may turn out to be stale. The stale-safe property for multiple CCs should make sure that the access machine will be allowed to update attributes only if the attributes at the CC are more recent than that of the access machine. Suppose the CC maintains a subject attribute LU_TS that is the time-stamp of the last update received from the GA for that subject. And the access machine maintains an attribute LSR_TS that is the time-stamp of the lastest refresh when that refresh actually resulted in a real attribute update. Then an access machine can refresh subject attributes only if the LU_TS (S) > LSR_TS (S).

- *Multiple Groups (non-hierarchical)*: In the case of multiple groups, an object from one group could be shared with another. We call the group that owns the object source group. If the object is removed from the source group, attribute staleness could let other group subjects retain access.

[5]Note that this may not be a problem if the policy lets subjects retain access to past objects.

A. If an object is removed from the source group, it should also be removed from other groups with which the object is shared.
B. If a subject is revoked access to an object from one group, he should also be revoked access to that object from any other membership group with which it is shared.

- *Multiple Groups (hierarchical)*: Consider hierarchical groups where subjects in higher level groups have access to objects at or below its hierarchical level. Thus the subjects in the leaf groups have access only to a single group and subjects at the root groups have access to objects of all group in the hierarchy. Suppose that each level has its own CC. In order to limit staleness, an access policy should use the appropriate subject and object attributes from respective groups in question:
A. An access decision for the subject should be made based on the object attributes from the source group and subject attributes from the subject's highest hierarchical membership group.
B. When a subject leaves a group and joins any group at the lower level of the hierarchy, he/she should be revoked access to any object that he/she retains access from past group[6].

4. MODEL CHECKING g-SIS

Model checking [6, 7] is an automated verification technique that analyzes a finite model of a system (i.e., a finite state machine (FSM) that produces computation traces consisting of infinite sequences of states) and exhaustively explores the state space of the model to determine whether desired properties hold in the model. In the case that a property is false, a model checker produces a counterexample consisting of a trace that violates the property, which can be used to correct the model or modify the property specification.

SMV [22, 5] is a family of model checking tools based on binary decision diagrams (BDDs). BDDs represent states very compactly. In SMV, models are represented by using variables that are assigned values in each step of the FSM. Properties to be checked are specified by temporal logic [23] formulas. SMV provides built-in finite data types, such as boolean, enumerated type, integer range, arrays, and bit vectors. In SMV, the initial state is defined by assigning initial values to state variables. State transitions are specified by assigning values to be assumed by each state variable x in the next state, which is denoted by next(x). Each such value is given by an expression over variables in either the current or the next state. The assignments are effectively performed simultaneously to obtain the subsequent state. SMV allows nondeterministic assignment, i.e., the value of variable is chosen arbitrarily from the set of possible values.

The set of next assignments execute concurrently in a step to determine the next state of the model. SMV allows nondeterministic assignment, i.e., the value of variable is chosen arbitrarily from the set of possible values. SMV supports macros, which are replaced by their definitions, so they do not increase the system's state space.

The model checker we use in this work supports only future temporal operators ("in the next state," "in all future states," "in some future state," and "until"), so the formulas expressing stale safety in section 3.2 have to be reformulated in this restricted form.

In this section, we use model checking (with SMV) to verify the weak stale-safe property for g-SIS. Model checking the entire g-SIS system with CCs, GAs and multiple groups is out of scope for this paper (please see discussions in Future Work). Instead, we model check the Trusted Reference Monitor (TRM) that is responsible for enforcing the access policies in the subject's access machine. Please see the Appendix for code, the properties that are verified and the results obtained from SMV.

4.1 Formal Verification of the Trusted Reference Monitor

In modeling the TRM, one of the first things to decide is how a refresh of subject attributes is forced so that the TRM periodically updates attributes with CC. Recall that we discussed various approaches: refresh based on timeout, usage count, etc. We could further use rate limits or a combination of these approaches. The TRM we consider models refresh based on usage count[7]. The CC/GA determines a usage count for each subject that specifies the number of times the group credentials (e.g. group key) may be used by the TRM to access objects off-line before a refresh is required. Suppose N is the usage count. Every time a subject accesses an object, N is decremented by the TRM. Once N reaches zero for that subject, the TRM denies access to any object until the attributes are refreshed by the access machine with the CC. As part of this refresh, N is reset to the initial value.

The TRM includes one FSM for each object available at the access machine. We discuss the formal verification of an object machine, $\text{FSM}_{\text{object}}$, against the Weak Stale Safety property (Definition 3.2). Figure 10 shows one possible design of $\text{FSM}_{\text{object}}$ to enforce an authorization policy given by Authz_{E}. Staleness is not considered in this machine. The predicate Authz_{E} is $(\neg\text{Remove\_TS}(O) \wedge \neg\text{Leave\_TS}(S) \wedge (\text{Join\_TS}(S) \leq \text{Add\_TS}(O)))$, indicating that the object O has not been removed from the group, subject S has not left the group, and the subject S joined the group before the object O was added. This is the same as the LTL formula Authz_{P} discussed in section 3 except that we now use attributes that can be directly coded in SMV[8]. We label state transitions using the format $e[C]/A$, in which e is the event, C is the condition that has to be satisfied to enable the transition, and A represents actions that need to be performed when the transition is taken.

The $\text{FSM}_{\text{object}}$ is responsible for mediating request from the subject to access the object to which it corresponds. It remains in the idle state until a request to access the object arrives from the subject. At this point, $\text{FSM}_{\text{object}}$ checks the authorization policy (Authz_{E}) to decide whether the subject can access the requested object. There are then

[6]Note that this property may not be applicable to all group policies.

[7]This is chosen due to the lack of availability of any practical solution for a secure off-line source of time today. The Trusted Platform Module [2] provides a monotonic counter and hence we choose to develop and model check a usage count based TRM that can be later implemented. Our discussions remain valid irrespective of the approach we take.

[8]Note that the check for the presence of the object in ORL is simplified and simulated as an event in SMV. This event results in setting the boolean attribute Remove_TS(O) to TRUE.

three possible paths the FSM can take, depending on which condition is satisfied:

Request[¬Authz$_E$]: If Authz$_E$ fails, FSM$_{object}$ rejects the request and remains in the idle state. This is the request transition that starts and ends at the idle state. The Refresh transition captures attribute updates received from the CC triggered by other instances of FSM$_{object}$ running on behalf of the same subject.

Request[Authz$_E$ ∧ $N = 0$]: If Authz$_E$ succeeds, but the usage count is exhausted, the machine is required to refresh attributes before any access can be granted. A refresh request action (Refresh$_{REQ}$) is initiated in this case. This creates a synchronizing event (transitions labeled Refresh in the idle, authorized and refreshed states) for all the FSM$_{object}$ instances in the local TRM, which has the effect of updating all subject attributes, as well as the ORL, with the values that are current at the CC. The synchronous event simply ensures that the update is atomic with respect to transitions at every FSM$_{object}$. After the refresh, the FSM$_{object}$ then enters the refreshed state. It again checks the authorization policy to see whether the subject is now allowed the requested access in light of the updated attribute values. If Authz$_E$ holds, the FSM directly enters the authorized state from which it transitions to idle while decrementing the usage count. If Authz$_E$ does not hold, the FSM denies access and immediately returns to idle.

Request[Authz$_E$ ∧ $N > 0$]: If Authz$_E$ succeeds and the usage count N is not exhausted, the machine enters the authorized state and waits for the subject to access the object. The requested action is performed only after re-checking the policy Authz$_E$ and decrementing the usage count. This re-checking is critical because Authz$_E$ checked earlier may no more hold due to updated attributes received from the Refresh transition which could possibly be triggered by another instance of FSM$_{object}$. We discuss this in more detail in the following paragraph. FSM$_{object}$ thereafter returns to the idle state.

Consider the self-transitions labeled Refresh in idle, authorized and refreshed states. It is needed to allow refreshes initiated by other FSM$_{object}$'s to occur atomically with respect to other transitions. A subject could request access to multiple objects; a separate instance of FSM$_{object}$ runs for each such object. A problem may arise due to a possible lag between the time at which the access was authorized (after which the FSM$_{object}$ is in the authorized state) and the time at which the subject actually performs the access. Suppose $S1$ is allowed to access object $O1$ and that $O1$'s FSM$_{object}$ enters and remains in the authorized state until $S1$ performs the action. In the meantime, $S1$ requests and performs access on multiple objects and exhausts the usage count. Finally when an access request for an object $O2$ is initiated, the FSM$_{object}$ of which forces a refresh because the usage count ran out. This illustrates that fact that multiple refreshes can occur when FSM$_{object}$ is in the authorized state for $O1$. When $S1$ actually performs the access of $O1$, it may no longer be authorized due to updated attributes. Such refreshes are permitted by the self-transition from the authorized state back to itself. When a refresh response is received by any instance of FSM$_{object}$, it is broadcasted to all other instances, and the attributes are updated in every machine. In this way, when a perform is generated, Authz$_E$ uses the latest attributes to verify policy. Thus when the FSM$_{object}$ instance for $O2$ has updated attributes, it sends those updated attribute values

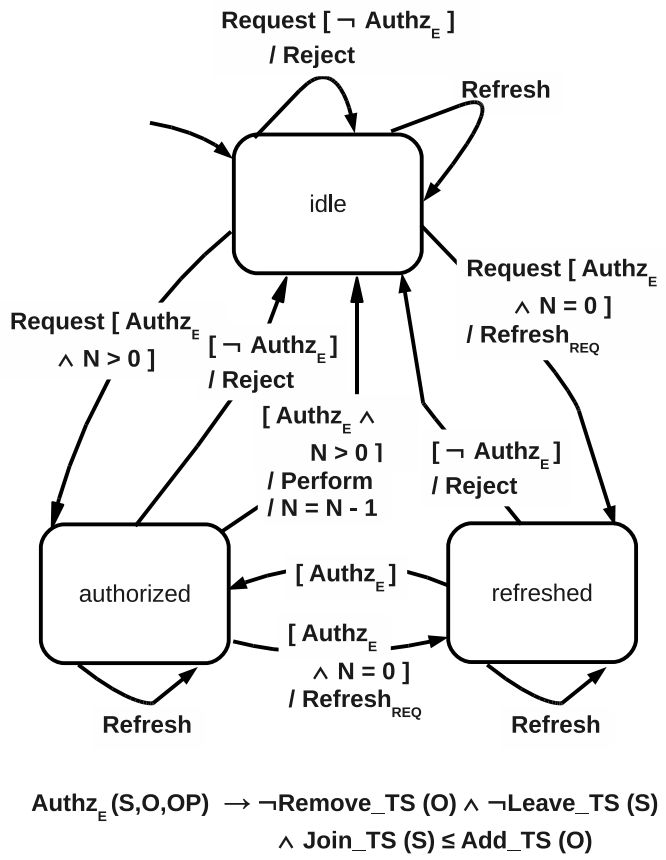

Figure 10: Stale-*unsafe* FSM$_{object}$.

to all running instances of FSM$_{object}$. Consequently, when $S1$ performs the action, it may not be correct to allow access to $O1$, due to updated attributes and failed Authz$_E$. Thus checking the policy Authz$_E$ at both request and perform time is critical for the correctness of FSM$_{object}$.

Code listing B in appendix A.2 shows the construction of FSM$_{object}$ using SMV and properties that are verified against it. As shown, the backward-looking stale safety property (formula φ_1, section 3.2) does not hold. The model checker immediately detects and reports the problem with a counter example. Note that the property is re-formulated using future temporal operators only which can be model checked using SMV. The FSM$_{object}$ in figure 10 is not stale-safe because it allows access to objects that were added after the last refresh time. This problem is fixed in figure 11 which we discuss in the following subsection.

4.1.1 Stale-safe TRM

Figure 11 is one of many approaches to build a stale-safe FSM$_{object}$. As you can see, the only difference from figure 10 is the extra check for stale attributes (in our case *Stale* is RT_TS ≤ Add_TS(O)). The transition from idle to authorized is enabled if the authorization policy succeeds, the usage count is still available *and* the attributes are not stale (i.e., Add_TS(O) < RT_TS). The transition from idle to refreshed is enabled if the authorization policy is successful but either the attributes are stale or the off-line usage limit is reached. From refreshed, FSM$_{object}$ en-

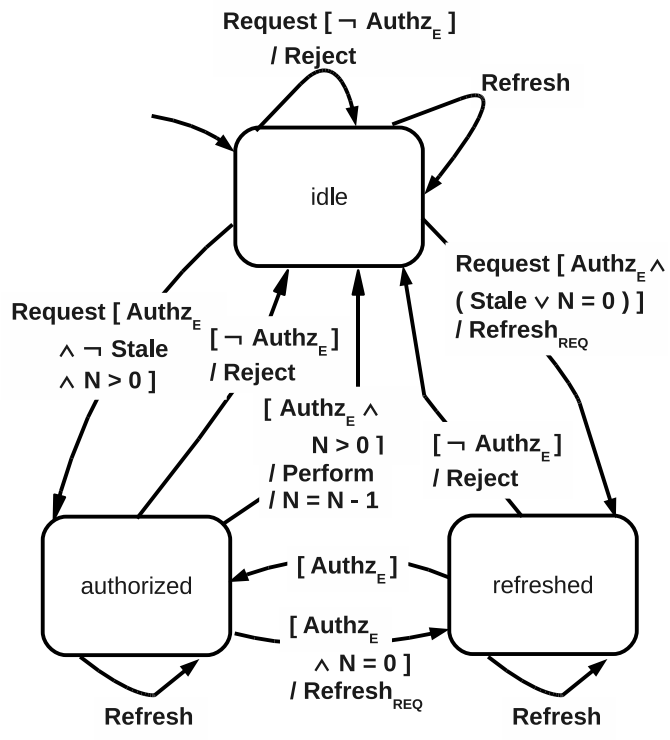

Figure 11: Stale-*safe* $\text{FSM}_{\text{object}}$.

ters authorized state if Authz_E still holds with the refreshed attributes else it returns to idle. Thus this machine satisfies formula φ_1 (backward-looking stale-safety) when the attributes are not stale, and it satisfies formula φ_2 (forward-looking stale-safety) when the attributes are stale.

Code Listing A in appendix A.2 shows an SMV implementation of this machine. The result for this machine, which we have verified by using the model checker, is discussed in appendix A.1. It is also possible to construct machines that will satisfy the other properties we discussed earlier.

5. RELATED WORK

Security requirements express the goals for protecting the confidentiality, integrity, and availability of cyber systems. There has been substantial work on developing models and policy languages for addressing these security concerns. Access control lies at the heart of system security [29]. Formal specification and verification techniques and tools, such as model checking, have been increasingly leveraged to verify security properties of access control systems [8, 13, 32, 3, 9, 19, 31, 34]

Zhang et. al. [34] developed a model checking approach to examine the access right of a group of principles. The access control is modeled in the RW language, which is a propositional logic-based policy language to express reading and writing access [10]. May et. al. [19] formalize the rules of Health Insurance Portability and Accountability Act into an extended access control matrix, which can be analyzed by model checker SPIN.

Security analysts of access control systems and policies have increasingly leveraged automated tool support to verify properties in support of security objectives. Jha et al. [13] verify such properties as authorization, availability, and shared access of the SPKI/SDSI policy language through the use of a language containment type of model checking.

Sistla et. al. [32] provides a framework for reasoning about security analysis of dynamic RT policies. Of significant value is their proof of a tight EXPTIME complexity for role containment queries. Additionally, they describe a structure to verify security properties using an explicit model checking approach.

Fisler et al. [8] analyzes the impact of policy changes on role-based access control (RBAC) systems using their Margrave tool. Such policies are represented as multi-terminal BDD's for efficient storage and manipulation. They successfully verify the separation of duty properties in RBAC system.

Schaad et al. [31] also verifies separation of duty properties in RBAC systems, but uses a mature model checking tool called NuSMV.

In addition, security analysis that answers the question whether security stake-holders can cause the authorization system to enter a state, in which certain queries (e.g., safety or liveness properties) hold or fail to hold, has been automatically performed [12, 17, 33, 25, 26], via the SMV family of model checkers.

6. CONCLUSIONS AND FUTURE WORK

Attribute staleness is inherent to any distributed system and can result in serious access violations. In this paper, we proposed stale-safe security properties using the group-based Secure Information Sharing problem as an example. We formalized four stale-safe properties of varying strengths using Linear Temporal Logic amenable to formal verification using Model Checking. Model Checking is a powerful and flexible approach to verify security properties of large and complex systems such as g-SIS. We designed and verified the Trusted Reference Monitor resident in access machines that satisfies the weak stale-safe property. We believe that these properties can be generalized to any distributed applications using Attribute-based Access Control with minor extensions/modifications if any. Our next steps are along three exciting areas:

In section 3, we identified staleness problem in the context of multiple CCs, multiple groups and multiple access machines and proposed extensions. We believe studying and formalizing these extensions is valuable to build systems with flexible stale-safe properties.

Verifying the complete g-SIS system is a major future work. This is a complex problem which is composed of multiple FSMs for TRM, CC and GA. All these machines need to handle various operations such as membership management of subjects and objects, provisioning group credentials, multiple group memberships, etc.

Implementation of g-SIS is a work in progress and many approaches are possible. The access machines need to have a Trusted Computing Base (TCB) that has a hardware and software component. The TPM (although not the only trusted hardware infrastructure required) provides the hardware root of trust. The software component comprises of a trustworthy kernel (possibly a microkernel like L4 [1] or a VMM [4]) and the TRM.

7. REFERENCES

[1] The L4 microkernel family. *http://os.inf.tu-dresden.de/L4/*.

[2] TCG specification architecture overview. *http://www.trustedcomputinggroup.org*.

[3] A. K. Babdara, E. C. Lupu, and A. Russo. Using event calculus to formalise policy specification and analysis. In *Proceedings of the 4th International Workshop on Policires for Distrubuted Systems and Networks (POLICY 2003)*, pages 26–39, 2003.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. L. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, 2003.

[5] 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.

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

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

[8] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tshchantz. Verification and change-impact analysis of access-control policies. In *ICSE*, pages 196–205. ACM Press, 2005.

[9] 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.

[10] D. P. Guelev, M. Ryan, and P. Y. Schobbens. Model checking access control policies. In *Proceedings of the 7th Information Security Conference*, volume 3225 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

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

[12] S. Jha, N. Li, M. Tripunitara, Q. Wang, and W. Winsborough. Towards formal verification of role-based access control policies. To appear in IEEE Transactions on Dependable and Secure Computing (TDSC), 2008.

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

[14] R. Krishnan, R. Sandhu, and K. Ranganathan. PEI models towards scalable, usable and high-assurance information sharing. *Proc. of the 12th ACM Symposium on Access Control Models and Technologies*, pages 145–150, 2007.

[15] A. 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.

[16] A. 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.

[17] N. Li and M. V. Tripunitara. Security analysis in role-based access control. *ACM Transactions on Information and System Security*, 9(4):391–420, November 2006.

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

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

[20] J. M. McCune, T. Jaeger, S. Berger, R. Caceres, and R. Sailer. Shamon: A system for distributed mandatory access control. *Proc. of the 22nd Annual Computer Security Applications Conference*, pages 23–32, 2006.

[21] J. McLean. Security models. *Encyclopedia of Software Engineering. Wiley & Sons*, 1994.

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

[23] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, volume 526, pages 46–67, 1977.

[24] S. Rafaeli and D. Hutchison. A survey of key management for secure group communication. *ACM Computing Surveys*, pages 309–329, September 2003.

[25] M. Reith, J. Niu, and W. Winsborough. Model checking to security analysis in trust management. In *ICDE, Workshop on Security Technologies for Next Generation Collaborative Business Applications (SECOBAP'07)*, 2007.

[26] M. Reith, J. Niu, and W. Winsborough. Role-based trust management security policy analysis and correction environment (RT-SPACE). In *International Conference on Software Engineering (ICSE) Research Demonstration*, 2008.

[27] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. *Proc. of the 13th conference on USENIX Security Symposium*, 13:16–16, 2004.

[28] R. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, November 1993.

[29] R. Sandhu. Rationale for the RBAC96 family of access control models. In *RBAC '95: Proceedings of the first ACM Workshop on Role-based access control*, page 9, New York, NY, USA, 1996. ACM Press.

[30] L. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a tpm without a trusted os. *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 27–42, 2006.

[31] A. Schaad, V. Lotz, and K. 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] A. P. Sistla and M. Zhou. Analysis of dynamic policies. In *Proceedings of Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis*, pages 233–262, 2006.

[33] S. D. Stoller, P. Yang, C. R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *CCS'07*, pages 445–455, 2007.

[34] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *Proceedings of the 8th Information Security Conference*, volume 3650 of *LNCS*, pages 446–460. Springer-Verlag, 2005.

APPENDIX

A. MODEL CHECKING USING NUSMV

Expression operators !, &, |, and -> represent logical operators "not", "and", "or" and "implies", respectively. Comments follow the symbol "- -". Further, the symbols F, X and U represent the future temporal operators *eventually*, *next*, and *until* respectively (please refer section 3.2).

The set of next assignments execute concurrently in a step to determine the next state of the model. SMV allows non-deterministic assignment, i.e., the value of variable is chosen arbitrarily from the set of possible values. SMV supports macros, which are replaced by their definitions, so they do not increase the system's state space.

A.1 Stale-safe TRM

Code listing A is an SMV implementation of the stale-safe $\text{FSM}_{\text{object}}$ shown in figure 11. Note that the property that is verified is φ_1 (stated as LTLSPEC towards the end). It is re-formulated using the future temporal operators. As reported by SMV, figure 11 satisfies the Weak Stale Safety property. The second property simply makes an additional check that it is always the case, if a subject is able to perform an action on an object then that object was added before the last refresh time. SMV confirms that this is indeed the case.

Code Listing A

```
MODULE main
VAR
--declare attributes
r_ts : 0..100;
leave_ts : boolean;
remove_ts : boolean;
join_ts : {2,18};
--usage count
N : 0..5;
--clock ticks
ticks : 1..10;
--declare events
--input event from subject
request_event : boolean;
--the latched request_event
request : boolean;
--refresh event received from the CC
refresh : boolean;
--action perform
perform : boolean;
--input event representing if subject has left
leave : boolean;
--input event representing if object has been removed
remove : boolean;
-- declare states
idle : boolean;
authorized : boolean;
refreshed : boolean;

DEFINE
add_ts := 10;
stale := r_ts <= add_ts;
authzE := (add_ts > join_ts) &
(!leave_ts) & (!remove_ts);
authzSS := authzE & !stale & (N>0);

ASSIGN
init(join_ts) := {2,18};
next(join_ts) := join_ts;

init(leave_ts) := 0;
next(leave_ts) := case
idle & refresh & leave : 1;
idle & request & !refresh &
authzE & (stale | N=0) & leave : 1;
authorized & refresh & leave : 1;
refreshed & refresh & leave : 1;
1 : leave_ts;
esac;

init(remove_ts) := 0;
next(remove_ts) := case
idle & refresh & remove : 1;
idle & request & !refresh &
authzE & (stale | N=0) & remove: 1;
authorized & refresh & remove : 1;
refreshed & refresh & remove : 1;
1 : remove_ts;
esac;

init(r_ts) := join_ts;
next(r_ts) := case
idle & refresh & (r_ts <= 90) : r_ts + ticks;
idle & request & !refresh & authzE &
(stale | N=0) & (r_ts <= 90): r_ts + ticks;
authorized & refresh & (r_ts <= 90) : r_ts + ticks;
refreshed & refresh & (r_ts <= 90) : r_ts + ticks;
1 : r_ts;
esac;

init(N) := 5;
next(N) := case
idle & refresh : 5;
idle & request & !refresh &
authzE & (stale | N=0): 5;
authorized & refresh : 5;
refreshed & refresh : 5;
--perform
authorized & !refresh & authzE & (N>0) : N - 1;
1 : N;
esac;

init(request) := 0;
next(request) := case
```

```
idle & request_event & !refresh & authzE: 1;
authorized & !refresh & (!authzE | authzE): 0;
1: request;
esac;

init(idle):= 1;
next(idle):= case
idle & refresh : 1;
idle & request & !refresh & !authzE : 1;
idle & request & !refresh & authzSS : 0;
idle & request & !refresh &
authzE & (stale | N=0) : 0;
authorized & !refresh & !authzE : 1;
authorized & !refresh & authzE & (N>0): 1;
refreshed & !authzE : 1;
1: idle;
esac;

init(authorized):= 0;
next(authorized):= case
idle & request & !refresh & authzSS : 1;
refreshed & authzSS : 1;
authorized & refresh : 1;
authorized & !refresh & !authzE : 0;
authorized & !refresh & authzE & N=0: 0;
1 : authorized;
esac;

init(refreshed):= 0;
next(refreshed):= case
idle & request & !refresh &
authzE & (stale | N=0) : 1;
refreshed & !authzE : 0;
refreshed & authzSS : 0;
authorized & !refresh & authzE & N=0 :1;
1 : refreshed;
esac;

init(perform) := 0;
next(perform) := case
authorized & !refresh & authzE : 1;
1 : 0;
esac;

---formula phi1 for WEAK STALE SAFETY
LTLSPEC G ( (refresh & authzE & F request) ->
(X((!refresh | (refresh & authzE) & !request) U
((request & F perform) -> (!perform &
(!refresh | (refresh & authzE)) U perform)))) )

LTLSPEC G ( perform -> add_ts < r_ts )

[root@localhost TRMobject.smv]# NuSMV
trm_object_safe.smv
*** This is NuSMV 2.4.3 (compiled on Mon May
5 02:33:40 UTC 2008)
*** For more information on NuSMV see
<http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv@irst.itc.it>.

-- specification
G ( (refresh & authzE & F request) ->
(X((!refresh | (refresh & authzE) & !request) U
((request & F perform) -> (!perform &
(!refresh | (refresh & authzE))
U perform)))) ) is true
-- specification
G (perform -> add_ts < r_ts)  is true
[root@localhost TRMobject.smv]# NuSMV -int
trm_object_safe.smv
NuSMV > go
NuSMV > print_reachable_states
################################################
system diameter: 19
reachable states: 1.12752e+06 (2^20.1047) out of
2.48218e+07 (2^24.5651)
################################################
NuSMV >
```

A.2 Stale-unsafe TRM

Code Listing B is an SMV implementation of stale-unsafe $\text{FSM}_{\text{object}}$ shown in figure 10. For brevity, we only show the lines that differ from Code Listing A. One can construct this unsafe machine by replacing the corresponding lines in listing A with the ones specified in listing B. A significant change is the missing check for stale-safety. All occurrences of the variable $authzSS$ has been replaced with $authzE$. authzE is just the access policy and authzSS is the stale-safe version of authzE. Also, checks for the variable *stale* are removed. The property that is specified here is formula φ_0 (Staleness Unaware) which is satisfied by this machine. In order to see the time-stamps of objects that are accessible using this machine, we obtain a counter-example by specifying the second property that checks if the objects being accessed were added after the last refresh time. As you can see in the trace, in state 1.4, the subject performs an action on an object whose *add_ts* is 10. But however the last refresh time-stamp *r_ts* at this point for the subject is 2. This is clearly a stale-unsafe access. Note that we use the -bmc option (for Bounded Model Checking) to get a counter-example of minimal length.

Code Listing B

```
ASSIGN

init(leave_ts) := 0;
next(leave_ts) := case
idle & request & !refresh &
authzE & (N=0) & leave : 1;
esac;

init(remove_ts) := 0;
next(remove_ts) := case
idle & request & !refresh &
authzE & (N=0) & remove: 1;
1 : remove_ts;
esac;

init(r_ts) := join_ts;
next(r_ts) := case
idle & request & !refresh & authzE &
(N=0) & (r_ts <= 90): r_ts + ticks;
esac;

init(N) := 5;
next(N) := case
```

```
idle & request & !refresh &
authzE & (N=0): 5;
esac;

init(idle):= 1;
next(idle):= case
idle & request & !refresh & authzE : 0;
idle & request & !refresh &
authzE & (N=0) : 0;
esac;

init(authorized):= 0;
next(authorized):= case
idle & request & !refresh & authzE : 1;
refreshed & authzE : 1;
esac;

init(refreshed):= 0;
next(refreshed):= case
idle & request & !refresh &
authzE & (N=0) : 1;
refreshed & authzE : 0;
esac;

--formula phi0, STALENESS UNAWARE
LTLSPEC G( (refresh & F(request & authzE)) ->
(!request U ((request & authzE & F perform) ->
((!perform & (!refresh |
(refresh & authzE))) U perform))))

LTLSPEC G( perform -> add_ts < r_ts)

[root@localhost TRMobject.smv]# NuSMV
trm_object_unsafe.smv
-- specification
G( (refresh & F(request & authzE)) ->
(!request U ((request & authzE & F perform) ->
((!perform & (!refresh |
(refresh & authzE))) U perform)))) is true
-- specification
G (perform -> add_ts < r_ts)  is false
-- as demonstrated by the following execution sequence
Trace Description: LTL Counterexample
Trace Type: Counterexample
.
.
.
[root@localhost TRMobject.smv]# NuSMV -bmc
trm_object_unsafe.smv
-- no counterexample found with bound 0
-- no counterexample found with bound 1
.
.
.
-- no counterexample found with bound 10

-- no counterexample found with bound 0
-- no counterexample found with bound 1
-- no counterexample found with bound 2
-- specification  G (perform -> add_ts < r_ts) is false
-- as demonstrated by the following execution sequence
Trace Description: BMC Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  r_ts = 2
  leave_ts = 0
  remove_ts = 0
  join_ts = 2
  N = 5
  ticks = 1
  request_event = 1
  request = 0
  refresh = 0
  perform = 0
  leave = 0
  remove = 0
  idle = 1
  authorized = 0
  refreshed = 0
  authzE = 1
  add_ts = 10
-> Input: 1.2 <-
-> State: 1.2 <-
  request_event = 0
  request = 1
-> Input: 1.3 <-
-> State: 1.3 <-
  idle = 0
  authorized = 1
-> Input: 1.4 <-
-> State: 1.4 <-
  N = 4
  request = 0
  perform = 1
  idle = 1
[root@localhost TRMobject.smv]# NuSMV -int
trm_object_unsafe.smv
NuSMV > go
NuSMV > print_reachable_states
################################################
system diameter: 20
reachable states: 1.02864e+06 (2^19.9723) out of
2.48218e+07 (2^24.5651)
################################################
NuSMV >
```