Apply Model Checking to Security Analysis in Trust Management

Mark Reith[*], Jianwei Niu, William H. Winsborough
University of Texas at San Antonio
One UTSA Circle
San Antonio, Texas, USA, 78249
$\{$*niu,mreith*$\}$*@cs.utsa.edu, wwinsborough@acm.org*

Abstract

Trust management is a form of access control that uses delegation to achieve scalability beyond a single organization or federation. However, delegation can be difficult to control. A resource owner that delegates some authority is naturally concerned not only about who has access today, but also who will have access after others make changes to the global policy state. They need tools to help answer such questions. This problem has been studied in the case of a trust management language called RT, where, for simple questions concerning specific individuals, polynomial time algorithms are known. However, more useful questions, like "Could anyone who is not an employee ever get access?" are in general intractable. This paper concerns our efforts to build practical tools that answer such questions in many cases nevertheless by using a lightweight approach that leverages a mature model checking tool called SMV. Model checking is an automated technique that checks if desired properties hold in the model. Our experience, reported here, suggests that in our problem domain, such a tool may often be able to identify delegations that are unsafe with respect to security questions like the one mentioned above. We explain our translation from a RT policy and containment query to an SMV model and specification as well as demonstrate the feasibility of our approach with a case study.

1. Introduction

Analyzing security policies is a critical step towards engineering secure software systems. Secure software systems are often designed to separate security policy from security mechanism [1] in order to provide the flexibility to address dynamic changes in security requirements over the course of a software system's lifecycle. The software components that comprise critical systems often endure extensive testing if not more rigorous measures such as formal methods. However, poor policy design is an equally grave hazard. It is critical to verify that the policy as stated (or at least the parts of it under one's control) meets ones intended policy objectives.

Trust management (TM) is a type of security policy concerned with reasoning about delegation of access control in software systems [1, 9]. Delegation is the key aspect of TM as it addresses the problem of scalability in traditional access control systems. In a traditional system, access control is maintained by a centralized authority. In a TM system access rights are derived as a consequence of a global policy state that consists of delegation statements authored by principals in the system, some of whom own resources. The global policy state changes whenever a principal in the system adds or removes one of his policy statements. Whereas this approach provides a high degree of scalability by obviating the need for centralized authority, it also introduces management problems of its own, since portions of the policy state are authored by untrusted entities. Policy authors need analysis tools that can determine whether critical policy requirements can be compromised by untrusted and semi-trusted principals in the system.

Whereas many formal methods may be too costly for this purpose, we seek lightweight approaches [7]. Such approaches work with existing notations and are highly automated, often with established tool support. In particular, here we explore the use of model checking.

Several recent papers examine the applicability of model checking as a means to verify security properties in policies of centralized systems [3, 6, 10]. Our study is based on the security analysis problem introduced and formalized in [9], which considers whether queried properties hold in all states that are reachable through changes made by untrusted and semi-trusted principals. It is based on the role-based trust management language, RT [8]. We present a translation of RT policy to an SMV model and report on the

[*]The views expressed in this article are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

feasibility of verifying security properties on this model.

The structure of this paper is as follows. Section 2 describes the RT language and the expensive complexity of role containment analysis. Section 3 provides a brief background of model checking. Section 4 outlines the translation of an RT policy to an SMV model and provides justification for reduction techniques used to scope the size of the resulting model. Section 5 illustrates this technique in a case study. Section 6 comments on related and future work, and we conclude in Section 7.

2. The RT Policy Language

The role-based trust management policy language RT was designed to support highly decentralized attribute-based access control [8]. It enables resource providers to make authorization decisions about resource requesters of whom they have no prior knowledge. This is achieved by delegating authority for characterizing principals in the system to other entities that are in a better position to provide the characterization. For instance, to grant discounted service to students, a resource provider might delegate to universities the authority to identify students and delegate to accrediting boards the authority to identify universities.

A significant problem that policy authors face in this context is that of determining the extent of their exposure through delegation to untrusted or semi-trusted principals. The security analysis problem [9] in this context consists of determining whether changes made by principals that are not fully trusted could cause certain policy objectives to become violated. One example of the problem would ask whether anyone outside the organization could, because of changes made by principals outside the inner circle, gain access to the organization's sensitive data. In this section, we summarize RT and the security analysis of it.

2.1. Brief Review of RT Syntax & Semantics

In RT, all principals are able to define their own roles and to assign other principals to them. A role owner can do this by issuing cryptographically verifiable, role-defining statements of a few different types. To her own roles, she can add a specific principal or she can add the members of another role. In the latter case, she is delegating authority to the owner of the other role. Delegating authority to another owner can occur in two ways. First she can identify a specific principal as a delegate. Secondly, she can identify a collection of principals as delegates such that these principals are grouped by a role. Set intersection and union are also both available for role definition.

The RT language consists of two primary objects called roles and principals. A principal is an entity such as a person or software agent. Each role can be described as a set of principals and is of the form "principal.role_name". One interpretation of this role is that the principal considers the members (also principals) of this role to have an attributed denoted by the role name. For example, *Alice.friend* may be a role that contains the principals whom Alice considers friends.

Type	Syntax	Description
Type I	$A.r \leftarrow D$	Simple Member
Type II	$A.r \leftarrow B.r_1$	Simple Inclusion
Type III	$A.r \leftarrow B.r_1.r_2$	Linking Inclusion
Type IV	$A.r \leftarrow B.r_1 \cap C.r_2$	Intersection Inclusion

Figure 1. RT Statements

The basic RT language consists of four types of statements as shown by Figure 1 [9]. Type I statements directly introduce individual principals to roles. For example, *Alice.friend* $\leftarrow$ *Bob* identifies Bob as a friend of Alice. A given principal must appear in a Type I statement if it is to be contained by any role Type II statements express a form of delegation that describes the implication that if principals are in one role, then they are in another role as well. For example, the statement *Alice.friend* $\leftarrow$ *Bob.friend* describes the situation in which if a principal is a friend of Bob, then they are also a friend of Alice. Type III statements provide a mechanism to delegate to all members of a role. For example, the statement *Alice.friend* $\leftarrow$ *Bob.friend.friend* says that any friend of Bob's friends is also a friend of Alice. It does not imply that Alice's friends include Bob's friends. Finally, Type IV statements introduce intersection such that a principal must be in two roles in order to be included. For example, *Alice.friend* $\leftarrow$ *Bob.friend* $\cap$ *Carl.friend* says that only those principals who are both Bob's friends and Carl's friends are introduced into the set of Alice's friends. Note that disjunction is provided through multiple statements defining the same role.

We call the left hand side of the arrow the *defined role*. In Type III statements, the *base-linked role* is the role that contains the principals upon which the linking role is applied and the *sub-linked roles* are the roles produced by the linking role. For example, in *A.r* $\leftarrow B.r_1.r_2$, $B.r_1$ is the base-linked role and every role of the form $X.r_2$ in which X is a member of $B.r_1$ is a sub-linked role. A *policy state* is a set of statements.

2.2. RT Policy Analysis

Policy analysis [9] as we consider it here examines whether specified relationships between roles hold in all reachable policy states. We explain reachable policy states below. The relationships are set containments and take the form $\varrho \sqsupseteq \lambda$ in which ϱ and λ are each either roles or explicit (constant) sets of principals. For instance, $A.r \sqsupseteq X.u$ holds if every member of $X.u$ is a member of $A.r$ in every reachable state. Relationships of this form can be used to express many important security properties such as availability, safety, liveness and mutual exclusion. For instance a safty property might be that everyone in the role that has access to the secret database is in the employee role.

In general, any policy state can evolve into any other policy state by having principals issue new policy statements and revoke old ones. In security analysis we ask whether security properties hold only in policy states that differ from a given current policy state only by changes to roles outside some trusted set. Intuitively, this corresponds to the fact that we expect certain principals to cooperate with us in our goal of preserving certain security properties, particularly with respect to certain roles that they control. Other roles they control or that are controlled by strangers should not be assumed to be managed in cooperation with our goals. This intuition leads us [9] to define two sets of roles that are used to determine the reachable policy states, the set of *growth-restricted* roles and the set of *shrink-restricted* roles.

Growth-restricted roles are not allowed to have new statements defining them added to the state. Shrink-restricted roles are not allowed to have statements defining them removed. It is important to note that these restrictions are not actually enforced. They are simply assumptions under which the analysis is performed. Their presence enables the analysis to provide us with reassurances of things like, "So long as the people I trust do not make policy changes without first running the analysis, only company employees will be able to access the secret database."

Restricted forms of security properties can be analyzed and verified in polynomial time. These include properties in which at least one of ϱ and λ is an explicit set. They also include situations in which only Type I and Type II statements are allowed in the policy state. However, when both ϱ and λ are roles and all forms of statements are allowed, the decision problem is in general intractable.[1] This is unfortunate because such properties are extremely useful. For instance, without them one can determine whether candidates to get access to the secret database are in a certain fixed set of principals. However, employee turnover may make this false. When both ϱ and λ can be roles, we can determine whether only employees will have access, which may remain true after new employees are added.

The goal of the current work is to determine the extent to which model checking can be used to provide useful information about whether security properties hold that are in general expensive to evaluate. For instance, it may be possible to find reachable states in which the security property is violated without exploring all reachable states.

[1]Li et al. [9] proved an upper bound of co-NEXP on the time required to answer this query and that doing so was PSPACE-hard.

3. Model Checking

Model checking [2] is an automated verification technique that constructs a finite model of a system and exhaustively explores the model's state space to determine if desired properties hold. In the case that a property is false, a counterexample will be produced to show an error trace, which can be used to fix the model or the property specification. The model is composed of state and transition relations. The state of the system is determined by the values of all state variables. The transition relation is defined by the set of next assignments which execute concurrently in a step to determine the next state of the model. The properties we want to check are called specifications, which are expressed in temporal logic [12] formulas. Temporal logic is a 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 operators X, F, and G represent next state, some future state, and all future states, respectively. For example, Gp means that property p is always true in all possible states.

Consider a simple example of a set of two light switches such that each switch controls a separate light in a room. Assuming that each switch is either in the on or off position and that there are no other sources of light in the room, let us test the property that the room is never without lighting. The combinations of switch positions represent the state space, and the model checker exhaustively checks each state for the desired property. In this case, it would produce the counterexample of both switches off to refute the property. If the model checker had not found a counterexample after the exhaustive search, then this fact serves as proof that the property always holds.

SMV [11] is a mature, symbolic model checking tool. Although conceptually similar to the aforementioned example, it checks a set of states at a time rather than each explicit state. The compression of the state space allows analysis of much larger models compared to the explicit approach. In addition, SMV allows nondeterministic assignments, i.e., the value of a variable is chosen arbitrarily from the set of possible values. SMV also supports derived statements (macros), which are replaced by their definitions, so they do not increase a system's state space. In this paper, macros are intensively used in the SMV models resulting from our translation.

4. Modeling RT Policies

RT policies are allowed to change over time as statements are added or removed according to growth and shrink restrictions. This dynamic behavior can be described as transitions from one policy state to another, where each state is defined by its policy statements. We are interested in examining security properties at each state in order to prove that these properties hold throughout the changes. Model checking is appropriate for our goal as it can quickly find counterexamples when properties fail to hold. The following sections describe the pre-processing and translation necessary to verify RT policies using a model checking approach.

4.1. Maximum Relevant Policy Set

Given an initial policy and a set of restrictions, it is difficult to predict what additional statements containing roles and principals may be added to the policy in the future. Whereas an RT policy is not constrained by the number of statements, roles or principals it can contain, model checking requires a finite state space. This is achieved by defining a maximum relevant policy set (MRPS). The MRPS is the maximum set of policy statements that may contribute to the outcome of a particular query given an initial policy. Thus the MRPS is built with respect to a query and contains all initial policy statements as well as additional Type I statements. The Type I statements are necessary in order to introduce additional principals into the roles. Intuitively, any statement added to a policy can be re-written as a (possibly empty) set of Type I statements, as demonstrated in [9]. These principals introduced by the Type I statements are representative of all possible principals and a certain number of them are necessary in order to verify role containment. As previously shown [9], if the containment property does not hold, then the counterexample state will have at most $M = 2^{|S|}$ principals over $O(M^2N)$ statements, where *S* is the set of significant roles, $|S|$ is the number of significant roles, and *N* is the number of initial policy statements. A significant role is defined as one of the following:

1. The superset role in a role containment query.
2. The base-linked role of a Type III statement.
3. Both intersected roles on the RHS of a Type IV statement.

To construct the MRPS, we first place all the principals on the RHS of Type I statements from the initial policy into set *Princ*. Then we calculate how many additional principals are needed using the upper bound described by *M*. We place this number of additional principals into *Princ*. Next we build the set of roles *Roles* to include all of the roles from the initial policy and query as well as those roles constructed from the cross product of principals *Princ* and link role names. Finally, we construct new Type I statements from the cross product of *Roles* and *Princ*. These statements along with all of the initial policy statements constitute the MRPS. In addition it is useful to identify the Minimum Relevant Policy Set as the set of non-removable initial policy statements. It identifies which statements are permanently included in our model.

Initial Policy	MRPS	
$A.r \leftarrow B.r$	$A.r \leftarrow B.r$	$E.s \leftarrow F$
$A.r \leftarrow C.r.s$	$A.r \leftarrow C.r.s$	$E.s \leftarrow G$
$A.r \leftarrow B.r \cap C.r$	$A.r \leftarrow B.r \cap C.r$	$E.s \leftarrow H$
	$A.r \leftarrow E$	$F.s \leftarrow E$
	$A.r \leftarrow F$	$F.s \leftarrow F$
	$A.r \leftarrow G$	$F.s \leftarrow G$
	$A.r \leftarrow H$	$F.s \leftarrow H$
	$B.r \leftarrow E$	$G.s \leftarrow E$
	$B.r \leftarrow F$	$G.s \leftarrow F$
	$B.r \leftarrow G$	$G.s \leftarrow G$
	$B.r \leftarrow H$	$G.s \leftarrow H$
	$C.r \leftarrow E$	$H.s \leftarrow E$
	$C.r \leftarrow F$	$H.s \leftarrow F$
	$C.r \leftarrow G$	$H.s \leftarrow G$
	$C.r \leftarrow H$	$H.s \leftarrow H$
	$E.s \leftarrow E$	

Figure 2. Initial Policy (no restrictions) & Query: $B.r \sqsupseteq A.r$ vs. MRPS

Consider the example in Figure 2. There are two significant roles $B.r, C.r$ and thus at most 2^2 principals are necessary to reveal the counter-example. Two important observations are made. First, although the number of Type I statements that needed to be added seems large compared to the original policy, growth restrictions often reduce the size of the MRPS because we will not be able to add new principals to certain roles. In this way, growth restrictions are accounted for in the model. Statements defining growth restricted roles (other than those in the initial policy) are simply not included into the MRPS. Shrink restrictions are accounted for in next state relations in Section 4.2.3. The second observation is that the number of principals introduced is calculated assuming the worst case policy statement configuration. By configuration, we mean that some set of Type III and IV policy statements require the aforementioned upper bound of principals in order to expose the counterexample [9]. It is intuitive that in some configurations we can work with fewer principals to reveal the same counterexample. The actual number of principals necessary and the particular configurations where such technique may be used is the topic of future work.

4.2. RT to SMV Translation Rules

The translation from an RT policy, restrictions, and query to an SMV model is comprised of five steps described in the following subsections.

4.2.1. Build MRPS & SMV Model Header

Preprocessing the initial policy into the MRPS is the first step of this process as it provides a finite number of statements and principals to be translated into the SMV model. We detail the MRPS in comments at the head of the file for easy indexing reference. This reference provide readers with a quick understanding of what each bit position represents. Information in the header should include the original policy, restrictions, the query as well as a list of all roles and all principals considered in this model.

4.2.2. Build SMV Data Structures

Each model contains one bit vector representing all of the statements in the MRPS and additional role bit vectors representing each role. The size of the statement bit vector is the size of the MRPS and the size of each role bit vector is equivalent to the number of principals considered. For example, in Figure 2, the number of principals considered is four and thus every role will have four bits as in Figure 3. We keep the naming convention from the RT policy and reuse the original role, linking role and principal names with the exception that we remove the dot (.) since in SMV this operator has a specific and unrelated function.

```
-- bit for each statement
statement :  array 0..33 of boolean;

-- bit for each principal per role
Ar :  array 0..3 of boolean;
Br :  array 0..3 of boolean;
Cr :  array 0..3 of boolean;
Es :  array 0..3 of boolean;
Fs :  array 0..3 of boolean;
Gs :  array 0..3 of boolean;
Hs :  array 0..3 of boolean;
```

Figure 3. Example SMV Data Structures from Fig. 2 MRPS

4.2.3. Initialization & Next State Relations of Statement Bit Vector

Initialization of state variables reflects the initial policy state. As such, each bit in the statement bit vector is initialized to true if its corresponding policy statement can be found in the initial policy. Otherwise the bit is set to false indicating its corresponding statement is not included in the initial policy. A special case exists when a statement is shrink-restricted (the defined role is non-removable) and included in the initial policy. In these cases, the bit is defined as permanent indicating that the policy statement it indexes cannot be removed from any policy state. Permanent bits do not contribute to the state space. Transitions from one state to another are accomplished by leaving non-permanent statement bits to remain unbound. By unbound, we mean that the bit can be nondeterministically assigned either true or false, allowing the model checker to find a state of bits such that the property does not hold. An example of initialization and next state relations is Figure 4. While this strategy is sufficient to look for counterexamples, certain optimizations can be used to reduce the state space depending on the structure of the policy. We discuss these in Section 4.6.

```
init(statement[0]) := 0;
init(statement[1]) := 0;
statement[2] := 1;
...
next(statement[0]) := {0,1};
next(statement[1]) := {0,1};
...
```

Figure 4. Example SMV Initialization & Next State Relations

4.2.4. Build Role Derived Statements

Roles are defined in terms of policy statements and other roles. When modeling this relationship, we use derived variables since the state of the policy is defined in terms of only policy statements, not roles. A derived variable in SMV is a function of state variables and other derived variables. The translation is summarized in Figure 5. The following describes the translation of each type of statement. We model a role with multiple statements by taking the logical or of the definitions below.

Type I policy statements are expressed in the model as direct associations between roles and statements. Thus the RT statement $A.r \leftarrow B$ that is indexed as statement 0 in our MRPS is expressed as $Ar[1] := statement[0]$; in SMV where bit position 1 in all roles corresponds to B. If statement 0 exists in a policy state, then B is a member of A.r.

Type II policy statements are expressed as a relation between two roles. Thus the statement $A.r \leftarrow B.r$ that is indexed as statement 1 in our MRPS is expressed as $Ar[i] := statement[1] \,\&\, Br[i]$; for $i = 0 \ldots n$ principals. In many cases, we can use the shorthand notation $Ar := statement[1] \,\&\, Br$ which is equivalent.

Type III policy statements are more complex since they require testing each of the sub-linked roles. Given the statement $A.r \leftarrow B.r.s$ that is indexed as statement 2 in our MRPS, we express it as $Ar[i] := statement[2] \,\&\, ((Br[0] \,\&\, As[i]) \mid \ldots \mid (Br[j] \,\&\, j^{th}\ role[i]))$; for $i = 0 \ldots n$ principals and j sub-roles.

Finally, Type IV policy statements are expressed as a relation between three roles. The statement $A.r \leftarrow B.r \cap C.r$ that is indexed as statement 3 in our MRPS is expressed as $Ar[i] := statement[3] \,\&\, Br[i] \,\&\, Cr[i]$; for $i = 0 \ldots n$ principals. Again, we may use the shorthand notation $Ar := statement[3] \,\&\, Br \,\&\, Cr$; where applicable.

4.2.5. Build Specification

In model checking, the specification is the property we wish to test. The security analysis of RT may wish to test such properties as availability, safety, role containment and mu-

Type	RT	MRPS Index	SMV
I	$A.r \leftarrow B$	0	$Ar[1] := statement[0];$
II	$A.r \leftarrow B.r$	1	$Ar := statement[1]\ \&\ Br;$
III	$A.r \leftarrow B.r.s$	2	$Ar := statement[2]\ \&\ ((Br[0]\ \&\ As) \mid (Br[1]\ \&\ Bs) \mid \ldots \mid (Br[j]\ \&\ j^{th}roles));$
IV	$A.r \leftarrow B.r \cap C.r$	3	$Ar := statement[3]\ \&\ (Br\ \&\ Cr);$

Figure 5. RT Statement to SMV Statement

tual exclusion. Our approach allows each of these properties to be tested and provides counterexamples if the property does not hold. Consider the example in Figure 6 where A.r and B.r are two roles and the MRPS considers principals C, D and F.

Note that the linear temporal logic (LTL) operator G is used to signify that all states are required to hold this property. Existential properties can also be tested using the negation of G or through the LTL operator F.

4.3. State Space

The state of the policy is defined by the combination of policy statements from the MRPS. Intuitively, the transition from one policy state to another is defined as the addition or removal of policy statements. This is achieved by allowing the model checker to freely assign flip bits in the statement bit vector. While this statement bit vector encodes the state of the policy, it alone is not sufficient to test the containment query since both role memberships must be computed based on which statements are included in a given state. Computing role membership can be performed in polynomial time (that is $O(p^3)$, where p is the number of policy statements) [9] as a separate function, however this may be expensive considering the number of states that this function needs to be applied. A more efficient approach is to encode the roles as derived variables (again bit vectors) in the model such that as the state of the policy changes, the membership of all the roles are updated. The derived variables represent role membership where each element position represent whether or not a principal is included in that role. The membership of the roles is then used to test for role containment. Although these derived variables may seemingly increase the state space, they in fact have no effect on it because they are not left unbound for the model checker to manipulate. It should be noted that it is possible to create an MRPS with a state space so large that role containment cannot be verified in any reasonable amount of time. In model checking terms, this is called the state explosion problem. Thus in cases where the role containment property holds, it is possible that the specification cannot be verified in an amount of time that is useful to the policy author. However, the redeeming feature of a model checking approach is that if the property does not hold, then it may be found in a short amount of time.

4.4. Role Dependency Graph

A role dependency graph (RDG) is a useful tool for visually depicting and analyzing role-to-role and role-to-principal relationships. The RDG also provides a means of detecting circular dependencies. It is a directed graph where each node represents a role, a linked role, the conjunction of two roles, or a principal. Each edge represents a specific policy statement and is labeled by its index in the MRPS. An edge is understood to mean the source node is dependent on the destination node, and its label is the condition of the edge's existence. Nodes representing roles may have many edges, each representing a different definition of that role.

Type I statements are always illustrated as an edge between a role node and a principal node. Principal nodes are always leaves in the RDG because they cannot contain anything. Type II statements are represented by an edge between two role nodes.

Type III and IV statements are expressed with unique structures. Type III statements use an edge representing a policy statement from role node to a linked role node, but then also use a dashed edge from the linked role node to other role nodes representing sub-linked roles. The purpose of the dashed edge is to visually identify the condition that a principal is in the base-linked role. These edges are labeled with the principal's name. Figure 7 illustrates this structure.

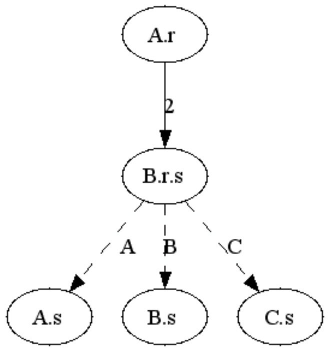

Figure 7. Type III: $A.r \leftarrow B.r.s$

Type IV statements use an edge representing a policy statement from a role node to the conjunction of two roles, but then uses an intermediate edge to show the relationship

Property	RT Query	SMV Specification	Notes
Availability	$A.r \sqsupseteq \{C, D\}$ Always	assert G $(Ar[0]$ & $Ar[1])$	C and D in A.r
Safety	$\{C, D\} \sqsupseteq A.r$ Always	assert G $(\sim Ar[2])$	E not in A.r
Containment	$A.r \sqsupseteq B.r$ Always	assert G $(Ar \mid Br = Ar)$	Nothing new in B.r
Mutual Exclusion	$A.r \otimes B.r$ Always	assert G $(Ar$ & $Br = 0)$	No intersection

Figure 6. RT Queries to SMV Specifications

between the conjunctive node and the roles from which it is composed. These edges are labeled as *it* for intermediate, do not represent policy statements and always exist. Figure 8 demonstrates this structure.

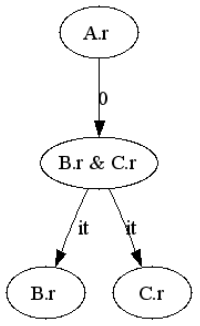

Figure 8. Type IV: $A.r \leftarrow B.r \cap C.r$

While the role dependency graph can be used to determine membership of roles, it can also provide some insight into the role containment query. For example, if a path of non-removable edges exists from a superset to a subset, then we can guarantee that the containment relationship is always true. This can be described as a "structural" relationship. However, containment can also occur through other means that we describe as an "ad hoc" relationship. A good example of this is the situation where two roles are tested for containment, but they exist in separate, unconnected graphs. These ad hoc relationships are often the real challenge of answering containment queries.

4.5. Circular Dependencies

The role dependency graph is necessary to detect circular dependencies in an RT policy. The RT language places no restrictions on self referencing statements such as $A.r \leftarrow A.r$ or circular referencing such as $A.r \leftarrow B.r$, $B.r \leftarrow A.r$. In the latter case, this is interpreted as $A.r$ is equivalent to $B.r$ if and only if both statements are non-removable. All circular references must be removed before translation to a model since SMV cannot handle circular definitions.

4.5.1. Detecting Circular Dependencies

Since there are situations where circular dependencies cause significant problems, a means to detect them is necessary. The two approaches used are well-formed syntax checks and graph cycle detection. The first approach detects cycles using syntax check as each policy statement is processed.

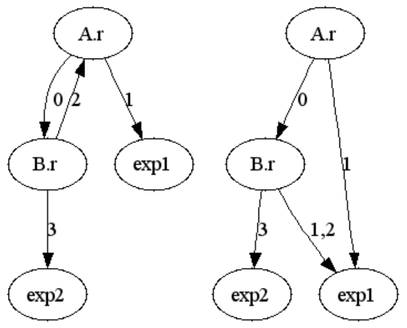

Figure 9. Circular Dependency with Type II Statements

For example, if a role is defined by itself, then we can safely remove this statement since it doesn't contribute anything to the query. It is easy to perform, however it only catches self-referencing cycles. The second and more general approach detects cycles across any number of statements using traditional depth first search. In these cases, it is not sufficient to simply remove a statement. In these cases we must perform dependency unrolling.

4.5.2. Unrolling Circular Dependencies

The relationship among roles can be equivalently represented using conditions of policy statements. Policy statements are neither modified nor added or removed in order to accomplish this. To understand how this works, let us consider how to unroll the previous example involving *A.r* and *B.r*. The left graph of Figure 9 illustrates how two Type II statements might form a circular dependency. The right graph in the figure demonstrates the unrolled version. Edges represent conditions upon which the dependency relies. Thus the role *B.r* will include *exp1* if and only if statements 1 and 2 are included in the policy state.

Circular dependencies involving Type III statements can occur frequently. Two cases involving Type III statements may cause a circular dependency. The first occurs in an explicitly recursive statement such as when the base-linked role is any parent to the linked role in the RDG. These cases require extensive unrolling and for brevity are not shown here. The second occurs when any of the sub-linked roles are any parent to the linked role. Here the circular dependency can be removed using the unrolling approach described above, as illustrated by Figure 10. In this case, the conditionals noted on the edges include not only policy

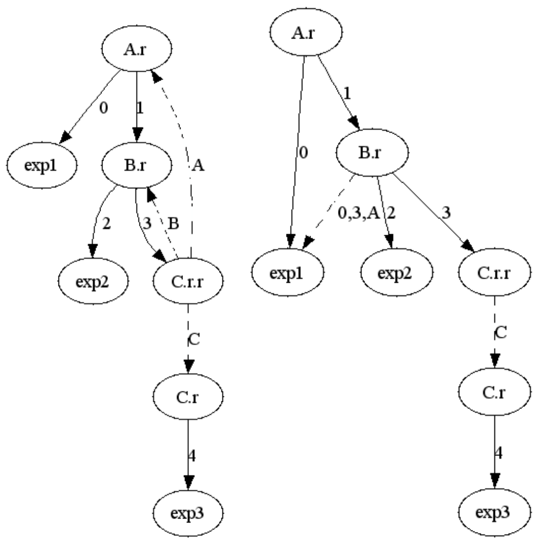

Figure 10. Circular Dependency with Type III Statements

statement indices, but also actual principals that must exist in the base-linked role in order for the dependency to exist. In the example demonstrated by Figure 10, *B.r* will include *exp1* if and only if statements 0 and 3 are included in the policy state, as well as *C.r* contains the principal *A*.

Finally, Type IV statements that introduce circular dependencies when one or both of the intersected roles is a parent in the RDG. Again unrolling is found effective when coupled with the realization that *A.r* ← *A.r* ∩ *B.r* does not contribute anything unique to *A.r*. In fact, this is a base case such that if the circular dependency exists in that form then it can be safely removed. Figure 11 illustrates an example of circular dependency involving Type IV statements. Here the only thing contributing to *A.r* through *B.r* is *exp2* and not *A.r & C.r*.

For the sake of completion, note that Type I statements cannot contribute to circular dependency. Also, statements such as *A.r* ← *A.r* can safely be removed since it doesn't contribute anything new into *A.r*.

4.6. Chain Reduction

Certain optimizations can be incorporated to further reduce the state space by recognizing logically equivalent states with respect to a particular role. Consider the example using Type II statements where we want to determine the membership of A.r in Figure 12. In this case, there are a total of 4 statements and $2^4 = 16$ states possible. If statement 3 is removed, then not only is the membership of D.r is empty, but also the membership of *A.r, B.r*, and *C.r*. Thus under the condition that statement 3 does not exist, we do not need to check the eight states representing combinations of statements 0, 1 and 2. This is handled by conditional

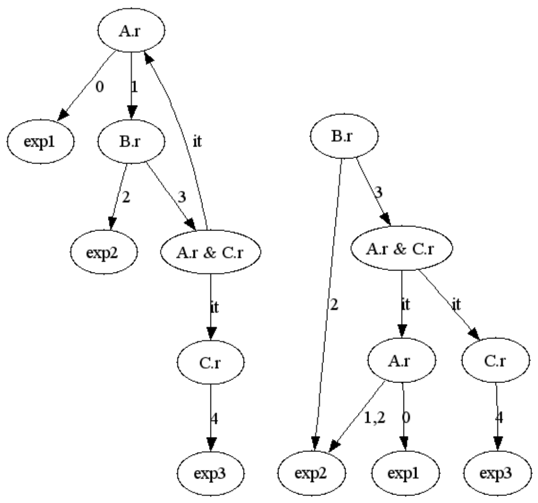

Figure 11. Circular Dependency with Type IV Statements

statements, one example of which is Figure 13. The effective result is that we now test only a single state (the empty policy) if statement 3 does not exist. The same idea can be used if a role is defined by multiple policy statements.

Index	Statement
0	$A.r \leftarrow B.r$
1	$B.r \leftarrow C.r$
2	$C.r \leftarrow D.r$
3	$D.r \leftarrow E$

Figure 12. Example of Chain Reduction

```
if(next(statement[3]))
   next(statement[2]) = {0,1};
else
   next(statement[2]) = 0;
```

Figure 13. Example of SMV Chain Reduction

Type III and Type IV statements may also be candidates for this reduction. For example, given the Type III statement $A.r \leftarrow B.r.s$, if the base-linked role $B.r$ is empty, than the linked role $B.r.s$ contributes nothing to $A.r$. Type IV statements are often easy to reduce because if either of the intersected roles is empty, then nothing is contributed to the defined role and thus we can force the other intersecting role to also be empty. A series of these conditions may occur leading to a chain reduction. A chain reduction may imply many logically equivalent states are able to be checked for a property with only a single test. This may yield a smaller state space.

4.7. Disconnected Graphs

Disconnected graphs are non-connected RDGs. It is possible that there are multiple sub-graphs in a system that, while not connected by any statements, are queried for containment. Our current translation approach works correctly because we do not depend on if the queried roles are connected or not. However, analyzing the RDG may provide insight to some optimization. For example, removing subgraphs that do not contain the roles specified in the query will further reduce the state space.

5. A Trust Management Case Study

Consider the access control policy of a fictitious company Widget Inc. Widget has a marketing strategy and an operations plan that it must protect from competitors, while at the same accessible to those employees with a need to know. Some properties of interest are:

1. Is the marketing strategy and operations plan only available to employees? *HR.employee* $\sqsupseteq$ *HQ.marketing, HR.employee* $\sqsupseteq$ *HQ.ops*
2. Does everyone who has access to the operations plan also have access to the marketing plan? *HQ.marketing* $\sqsupseteq$ *HQ.ops*

In this case, the significant roles are *HR.marketingDelg, HR.employee, HR.managers, HQ.specialPanel*, and *HR.researchDev* from the initial policy and *HQ.marketing* from the second query. This leads to a maximum of 64 new principals added to the model, 77 unique roles and a total of 4765 policy statements, 13 of which are permanent due to shrink restrictions. Note that not all of the roles are growable. Although 64 principals is the upper bound on number of new principals added, it is intuitive that there is a much smaller upper bound, which is the topic of future work. The number of principals needed directly affects how many Type I policy statements we need. Thus a smaller number of principals yields a smaller state space. While the current state space of 2^{4765} is quite large, SMV is able to check both properties. The translation from RT took about 9.9 s, and the first two properties were verified using SMV in approximately 400 ms. The third was found to be false in about 480 ms with a counterexample where the statement *HR.manufacturing* $\leftarrow$ *P9* is included and all other non-permanent statements are removed. The value of *P9* is a generic principal name and has no effect on the outcome. This leads to a state where *HQ.ops* contains *P9*, but *HQ.marketing* is empty. This example was performed on a Pentium 4 2.8 GHz with Windows XP. Cadence SMV was the target model checker.

6. Related & Future Work

Security requirements of business systems express the goals for protecting the confidentiality, integrity and availability of assets. There has been substantial work on developing models and policy languages for addressing these

Initial Policy
HQ.marketing $\leftarrow$ HR.managers
HQ.marketing $\leftarrow$ HQ.staff
HQ.marketing $\leftarrow$ HR.sales
HQ.marketing $\leftarrow$ HQ.marketingDelg $\cap$ HR.employee
HQ.ops $\leftarrow$ HR.managers
HQ.ops $\leftarrow$ HR.manufacturing
HQ.marketingDelg $\leftarrow$ HR.managers.access
HR.employee $\leftarrow$ HR.managers
HR.employee $\leftarrow$ HR.sales
HR.employee $\leftarrow$ HR.manufacturing
HR.employee $\leftarrow$ HR.researchDev
HQ.staff $\leftarrow$ HR.managers
HQ.staff $\leftarrow$ HQ.specialPanel $\cap$ HR.researchDev
HR.manager $\leftarrow$ Alice
HR.researchDev $\leftarrow$ Bob
Growth & Shrink Restricted
HQ.marketing
HQ.ops
HR.employee
HQ.marketingDelg
HQ.staff

Figure 14. Consider the queries: *HR.employee* $\sqsupseteq$ *HQ.marketing, HR.employee* $\sqsupseteq$ *HQ.ops, HQ.marketing* $\sqsupseteq$ *HQ.ops*

security concerns [13, 9]. To enforce the correctness (e.g. completeness and lack of conflicts) of policy specifications, policy language formalization and analysis have been performed using different techniques such as formal languages, automata theory, logic programming [9], and theorem proving [5]. However, these reasoning approaches require more expertise and efforts, and sometimes have less tool support, which are barriers for practitioners to adopt these formal techniques in developing secure software systems.

To alleviate this problem, researchers have been working towards developing automated tools to examine security properties using lightweight formal analysis techniques [3, 4, 6, 10, 14], such as model checking. Zhang et. al. [16] developed a model checking approach to examine the access rights of a group of principals. The access control is modeled in the RW language, which is a propositional logic-based policy language to express reading and writing access [6]. However, role delegation expressed as Type II or Type III RT statements cannot be expressed and checked using their approach. May et. al. [10] formalized the rules of Health Insurance Portability and Accountability Act into an extended access control matrix, which can be analyzed by model checker SPIN. However delegation in access control matrices does not scale well. Fisler et. al. [3] introduced Margrave as a tool to analyze the impact of changes in XACML policy. Their focus is on role-based access control as opposed to TM and thus they do not address delegation. Sistla et. al. [15] provided a framework for reasoning about dynamic policies in trust management systems. Of significant value is their proof of a lower bound time complexity for role containment queries in RT. Additionally, they suggest a structure for use in model checking to verify security properties, however they do not address how an unbounded number of additional roles/principals may affect the role containment analysis. We address this issue by constructing the MRPS.

In the future, we plan to optimize the preprocessing using RDG to reduce the state space and reduce the number of statements/principals necessary to verify a property. In addition, it is desirable to find the tight bound of extra principals in the MRPS. Finally we intend to show that this approach is feasible on extended variants of RT, to possibly include negated policy statements.

7. Conclusions

Security analysis of trust management policies is an important step towards provably secure software systems. Whereas the trust management approach provides the scalability and flexibility to handle real world access control requirements, the dynamic and indirect nature of delegation does not always provide an intuitive sense as to what the limitations on resources are actually in place.

This paper proposes a fully automated approach to performing security analysis in trust management. We demonstrate the feasibility of translating trust management policies into the input language of a model checker to analyze role containment. By translation, we support reuse of existing policy language and analysis tools to take advantage of policy language expressiveness and analysis tools' optimization. We also show that this approach can also be used in other security policy analysis such as separation of duty, safety, and availability.

Acknowledgments

This research was sponsored by the NSF through ITR award number CCR-0325951 (through a sub-award from Brigham Young University), and CyberTrust award number CCF-0524010. We thank Ninghui Li of Purdue University for his suggestions as well as the anonymous reviewers for their helpful comments.

References

[1] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 164, Washington, DC, USA, 1996. IEEE Computer Society.

[2] E. 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 ans Systems (TOPLAS)*, 8(2):244–263, 1986.

[3] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tshchantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th IEEE International Conference on Software Engineering (ICSE '05)*, pages 196–205, 2005.

[4] 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.

[5] P. Giorgini, F. Massacci, J. Mylopoulos, and N. Zannone. Modeling security requirements through ownership, permission and delegation. In *Proceedings of the 13th International Conference on Requirements Engineering (RE'05)*, pages 167–176. IEEE Computer Society, 2005.

[6] 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.

[7] D. Jackson and J. Wing. Lightweight formal methods. contribution to an invitation to formal methods. *IEEE Computer*, 29:16–30, 1996.

[8] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130. IEEE Computer Society Press, May 2002.

[9] N. Li, J. C. Mitchell, and W. H. Winsborough. Beyond proof-of-compliance: Security analysis in trust management. *Journal of the ACM (JACM)*, 52(3):474–514, 2005.

[10] 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.

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

[12] 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.

[13] R. S. Sandhu, E. J. Coyne, H. L. Feinstern, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[14] 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.

[15] A. P. Sistla and M. Zhou. Analysis of dynamic policies. In *Proceedings of Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis (FCS-ARSPA06)*, pages 233–262, 2006.

[16] 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 *Lecture Notes in Computer Science*, pages 446–460. Springer-Verlag, 2005.