

Role Assignment Service

Logi Ragnarsson

Thesis submitted to the University of London
for the degree of Master of Science September 2002

The work was supervised by Professor Steve Schneider
and funded by deCODE genetics.



Department of Computer Science
Royal Holloway, University of London

Acknowledgements

I would like to thank my mother *Anna Ingólfssdóttir* as well as *Luca Aceto* and *Margrét Sóley Jónsdóttir* for both general advice and constructive criticism of the thesis.

Others whose comments or discussion helped form the ideas set forth here include *Arnaldur Gylfason*, *Bertel I. Arnfinnsson*, *Björgvin Áskelsson*, *Neil Evans*, *Hákon Guðbjartsson*, *Malcolm Hole*, *Skeggi Þormar*, *Stefán Freyr Stefánsson*, *Sverrir Karlsson*, *Helen Treharne*, *Laufey E. Jóhannesdóttir*

Others who have rendered me support and motivation include my family, my girlfriend *Raffaella Grasso* and various friends and colleagues too numerous to name.

Finally I would like to thank deCODE genetics for allowing me to do this work.

Contents

1 Overview

1.1 Role-Based Access Control

The aim of this work is the development of a service for storing authorisation information and conveying this information to other parts of large software systems, especially in situations such as with Application Service Providers, where the user population is diverse and centralised management may be inappropriate. The methodology used involves the development of a special purpose logical language which will be used both to describe the system and argue about its security properties.

The service we offer in this study will go by the name of RAS or Role Assignment Service. The basic model we will base the RAS on is the Role-Based Access Control (RBAC) model introduced in [?], but with several extensions to make it more applicable to large and complex systems where full accountability must be maintained.

Role-based access control systems are, in short, authorisation systems where a set of roles is defined in addition to the usual set of principals (corresponding mostly to users) and resources. Rather than assigning permissions to principals directly, permissions are assigned to roles and these roles are assigned to the principals. This makes the management of authorisation simpler, by allowing particular roles to evolve without having to modify principals' permissions directly. It also simplifies each principal's permission configuration, making mistakes less likely.

In the standard classification system set forth by Sandhu et al. [?] our system will fall into the RBAC₁ category, which implies a hierarchy of roles. Rather than each role standing alone without relation to other roles, a hierarchy of roles is defined. A role implies all the roles below it in the hierarchy, so a principal assigned a role high in the hierarchy has the permissions assigned to that role as well as those assigned to any roles below it in the hierarchy.

Sandhu et al. also define the RBAC₂ model, which does not have the role hierarchy of RBAC₁, but instead allows for the definition of policies which the state of the system must follow. A common policy would for example be separation of concerns where it would not be legal for any principal to simultaneously hold a given pair of roles.

We have chosen not to implement the RBAC₂ functionality in the RAS. This is done both so that implementation of the system can be started sooner, but also because there are potential problems in the interaction between the distributed management model of the RAS and even simple policy specifications. As an example, imagine that a policy is in effect that no principal may hold both the *A* and *B* roles. Principals P_a and P_b have the authority to administer one of these roles each, but not the other. P_a , who can delegate role *A* to principal Q can effectively block P_b from delegating the role *B* to Q . Thus, delegating a role can suddenly lead to a reduction in the authority of Q and this reduction in Q 's authority is in an area which P_a , who caused the reduction, has not been specifically granted any privileges.

For the sake of completeness, we mention that Sandhu et al. also define RBAC₃ which is a combination of the RBAC₁ and RBAC₂ models having both a role hierarchy and policy definitions.

1.2 Extensions to RBAC₁

The envisioned uses of the RAS include being shared among various systems, which may not be otherwise related, controlling access to resources in many different places, both physically and logically. It will also be used with systems which store historical data such as logging servers. This calls for various extensions to the RBAC₁ model which are briefly described below and treated formally in later sections. The RBAC₁ model thus extended will be referred to as the RAS-model.

1.2.1 Domain Restrictions

The need to apply the RAS server simultaneously to data stored in different locations and controlled by different groups within an organisation, or even unrelated organisations using a common computing facility, calls for domain restriction on authorisation. There are various models available for domain restriction, but

the one chosen here is to apply an extended version of the x.500 name hierarchy to all resources.¹ A role is held within a particular domain w.r.t this extended x.500 hierarchy as described later.

1.2.2 Availability of Historical State

Because the RAS server will be used in conjunction with historical data, it is often required to view the authorisations of a particular principal at a particular instant in the past. Reasons for this may include verifying that an operation which was entered into an execution log was properly authorised and to see on whose authority the authorisation was established. The RAS will therefore never discard information about past authorisation state and queries can be made about past authorisations as easily as about the current state.

1.2.3 Temporal Restrictions

To support people working on projects temporarily, and in fact to allow people to lose authorisation now that no authorisation state is ever discarded, authorisation must be restricted to a particular period of time. This, along with the domain restrictions, clearly defines where and when a particular role may be used by a principal who has been assigned that role.

1.2.4 Management Model

When an authorisation system is applied to a large and disparate system with a growing amount of resources to control and complex relationships between the people responsible for these resources, centralised management of authorisation becomes cumbersome and may in fact not be sufficiently secure.

If authorisation management is centralised, the people responsible for managing authorisations may not be sufficiently familiar with the resources they are managing to do so effectively. Also, in the case of highly sensitive data or otherwise valuable resources in a large organisation, there may not be any one group of people trusted to manage all the resources of the organisation. Finally, in the case of Application Service Providers, it may be of value to allow customers to control access to their own data by giving each customer administrative privileges over a particular domain which they can then give their own personnel access to as required.

The approach adopted by the RAS system is to decentralise assignment of roles to principals, as well as the definition of roles and associated permissions. However, all authorisation information is stored centrally and is not accepted by other parties of the system unless retrieved from the central RAS server.

The authority to assign a role is assigned to principals in much the same way as the authority to act in that role. The organisation can then choose to keep it central, by assigning to a few central administrators the authority to assign all roles to principals, or it can have it distributed to various departments, divisions or customers by assigning to department administrators the authority to assign roles to principals, restricted to the domain that they manage.

1.2.5 Signed Authorisation Records

The management model described above leaves us with the problem of ensuring that only correct authorisation data is used in deciding whether a given principal has a particular authorisation. This is done by having every record used when deciding a principal's authorisations signed by the principal who inserted it into the database. These records are of various types and will be described later. Another advantage in this approach is that the database used to store authorisation records is no longer as critical in the security of the system.

1.2.6 Certified Execution

We will, however, go even further in reducing the part of the system which can cause the most serious security problems, known as the Trusted Computing Base. By applying certified execution methods [?],

¹The x.500 name hierarchy is taken from the ISO/ITU x.500 standard for directory servers, to which LDAP is a simplified interface. The name hierarchy is also used for x.509 certificates

whereby a proof of the correctness of an authorisation is built by the RAS server, and may be verified by any other systems making use of the RAS, we reduce the trusted computing base to only the verification component. This verification component is much smaller and simpler than the central RAS server and much more easily verified, making a sufficient verification of the RAS system much easier, whether it is done by fully formal methods, using the Common Criteria methodology or any other means.

Also, the verification of the proof can be done in a higher-security environment than the one the RAS server is running in, which allows us to share the RAS between different systems with differing security requirements. A low-security system may require many users to be able to manage role assignments, while a high-security system may require that very strict controls be in place for authorisation control. They can both share the RAS server, with the high-security system verifying every authorisation proof generated by the RAS server. The high-security system would therefore ensure that authorisation was ultimately established on the authority of a trusted principal and signed with a key which would not be available to the administrators of the low-security system.

1.2.7 Public Key Infrastructure

All of these signed records and certified execution make heavy use of public keys, which may be issued by various organisations. The RAS has no facilities for issuing keys, but uses its domain and temporal restriction mechanism for assigning different levels of trust to the root-certificates of various organisations Certification Authorities, referred to as CA's. Typically we will trust an organisation's CA to issue keys to its own members (as reflected in the principal's x.500 distinguished name), but not to identify other principals.

This information about key-ownership of various principals and the associated mechanisms for judging the validity and trustworthiness of particular public key certificates are also made available through the RAS server.

1.3 RAS Operation

Figure ?? shows the topology of a typical client-server system using the Role Assignment Service (RAS) and the Secure Logging Service (SLS) in conjunction with a number of servers (S_i) and clients (C). The arrows point from the originator of the connection towards the provider of the service which the connection is to carry.

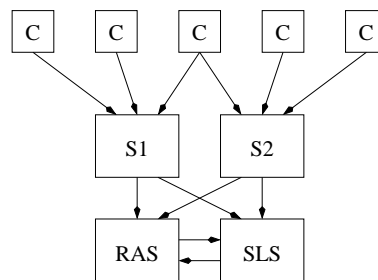


Figure 1: Connection diagram of typical RAS installation.

We will describe the typical use of each of these connections below.

Client → Server Very little can be said about the client-server connection, as it depends entirely on the function of the system. However, this will generally implement the main function of the overall system, which the RAS and SLS are support.

Server → RAS This is the main function of the RAS server. The RAS accepts queries from another server about the access privileges or public keys of a particular client. Depending on parameters controlled by the server, the RAS will reply with a

- “Yes” or “no”
- Signed token certifying that the RAS server believes that the client has the key or the access privileges in question, or a “no”
- A proof of the client having the access privileges in question, or a “no”.

The server will then decide depending on the answer returned by the RAS server whether to allow certain operations on behalf of the client.

Server → **SLS** Servers may log any operations performed by or on behalf of the clients with the Secure Logging Server. This is not a requirement for employing the RAS, but as the RAS makes use of the SLS it is reasonable for the various servers to use it for any logging they require.

RAS → **SLS** The RAS server logs any updates made to its state with the SLS. It may also be set to log access control queries and their results.

SLS → **RAS** The SLS uses the RAS to control who may view which logs entries depending on the server field in the log entry. This is done in a manner similar to other Server-RAS connections.

1.4 On Re-Using Logics

The first question that arises when we set about formally describing an access control system, is whether to re-use a previously developed formalism such as the well-known BAN-logic [?], or whether to develop a new special-purpose logic. After reviewing several formalisms in the literature, some of which are discussed below, it was decided to follow the latter approach.

Amongst the reasons to adopt a previously developed formalism, and possibly adapt it to our needs, are:

- Expediency. It would certainly be more expedient if one could find a suitable formalism, prune away redundant parts and extend it as needed.
- Legitimacy. Using a preexisting formalism would lend a certain legitimacy to any results obtained as the formalism would have been studied more thoroughly and more theoretical results about it would be available.
- Compound Systems. A more general purpose formalism would be useful for analysing a compound system made up of the RAS server along with, for example, authentication protocols, billing systems or secure logging systems.

The reasons for developing an entirely new formalism, however, include:

- Each of the formalisms considered is intended for a slightly different problem domain. They would need augmented notation and added logical postulates to make them suitable for our purposes. Extending them in this way may well invalidate results about their suitability for their original purposes, which would cancel many of the benefits listed above.
- Each of the encountered formalisms would carry a rather large load of excess baggage in the form of notation for irrelevant ideas and logical postulates for reasoning about these irrelevancies. This would distract its user from the actual problem at hand.
- We will require the idea of validity periods for all our statements, which the encountered formalisms do not support. They tend to be based on knowledge logics where we only reason about our knowledge about the current state based on events that have happened, allowing the current state to change as more events occur. Adding validity periods to a formalism would mean modifying it to such a large extent that it would become unfamiliar and probably overly cumbersome.

The chosen approach is to develop a small and focused logic, suitable for exactly the purpose we have in mind. It will be easy to verify the sensibility of it in an informal way and any structural proofs we may require will be much simplified.

1.5 Comparison with Other Formalisms

We will now compare a selection of formalisms from the literature with the RAS logic as it will be developed in section ???. We will highlight similar ideas and explain where the formalisms diverge from the purpose of the RAS logic and become unsuitable.

1.5.1 BAN-Logic

BAN-Logic [?] is the oldest of these formalisms and the best known. It is a many-sorted knowledge logic for reasoning about the belief of parties to an authentication protocol. By formalising the steps of a protocol conclusions are reached about the belief of these parties at the end of the protocol, based on the messages they have seen throughout its execution. They include belief about the freshness of keys (i.e. they were generated during this run of the protocol), that a key is shared only with a proper principal and other beliefs relevant to authentication.

There are significant differences between this setting and the one where we propose to make use of the RAS-logic. First, in the RAS we are not concerned with authentication but authorisation. BAN-logic, as given in [?] does not have the notation to express authorisation or the logical postulates to argue about it, but does have a lot of features which we would not find useful, such as belief about freshness which is not relevant in our system where authorisation information is valid long after it is generated. So we have a basic mismatch in the aims of the BAN-logic and the RAS-logic.

The BAN-logic is based on temporal logic, which is required for examining authentication protocols which proceed in many steps and the principals' belief evolves throughout the run of the protocol. In the RAS case, however, we are looking at proofs of authorisation which are assembled from many pieces of information which have been generated previously and which are assembled by an RAS server on behalf of an RAS client in one step of the protocol.

The BAN-logic is therefore not a good starting point for the RAS-logic. However, it is well known and the notation of the RAS-logic is based on the BAN-logic where this is appropriate. The notation for signatures “ $\{ \sigma \}_K$ ” is taken from the BAN-logic and the *may-delegate* notation “ $P \mapsto_t [r \triangleright d]$ ” is based on the *jurisdiction* notation in BAN-logic.

1.5.2 Kailar's Accountability Logic

Where the BAN-logic concerns itself with the belief of participants in an authentication protocol, Kailar's Accountability Logic [?] focuses on which statements the parties are able to prove to third parties, either on their own or in cooperation, after (or during) a run of the protocol.

This is in many ways a closer match with the aims of the RAS-logic, in that we want to assemble a proof which can be shown to one of the parties of the protocol and convince him of a principal's authorisation in a single step, much as if we were convincing a third party after the run of a protocol. In fact, we can view “the protocol” as being the interaction between the RAS server and the principals delegating authority. The party who is being sent a proof of a principals' authorisation is then in fact not a party to “the protocol”, but a third party in the sense of Kailar's logic, who is being shown a proof of the result of a protocol run involving the RAS server and any number of delegating principals.

However, Kailar's logic is actually a framework within which accountability properties can be discussed and not a directly applicable logic in itself. In each application of the logic to a problem, a domain-specific logic must be constructed to work within the framework and the RAS-logic can be used as such a domain-specific logic.

1.5.3 Abadi et al. Access Control Calculus

Abadi et al have suggested a calculus for access control in distributed systems [?]. This very closely matches the problem domain of the RAS, but without considering temporal aspects nor giving details

about domain restrictions of authorisations. However, many interesting high-level aspects of the problem are studied.

The basis of the calculus is the notion of composite principals. Principals can make statements in conjunction or on each other's behalf. In the first case the composite principal $A \wedge B$ may have certain privileges that neither A nor B have, acting on their own. Also, B acting on behalf of A , written $B \mid A$, may have reduced privileges compared to A acting directly or compared to a more trusted principal C acting on behalf of A . In this case, B and C are often channels (network connections, computer systems, application programs, etc.) through which A requests access to a resource and depending on how well we trust the channel, we restrict the access permitted to A . Much of the rest of the calculus is a treatment of how to assign privileges to such composite principals, but since there is already a sharp divergence at this point, the comparison becomes of little value.

The RAS adopts a very simple principal model. Adding various composite principals remains a possibility and may be done in a future version. A simplistic implementation of principals acting on behalf of other principals can be added by deciding that $B \mid A \mapsto_t [r \triangleright d] \Leftrightarrow A \mapsto_t [r \triangleright d] \wedge B \mapsto_t [r \triangleright d]$ (borrowing the notation to be introduced in section ??). B would then be the principal corresponding to a particular computer system or the author of (or someone willing to vouch for) a particular piece of software.

1.6 Overview of Subsequent Sections

The approach taken to the development of the RAS centres around a special-purpose logic for reasoning about access control, tailored to match the RAS model exactly. It is kept as small as possible to make structural induction easier and to make the logic decidable, both of which will be exploited in later sections. The logic will be introduced in section ?. It is seen as giving a semantics to the various kinds of delegation records, public key certificates, etc. that may be issued by principals in using the RAS. A large part of the workings of the RAS can be seen directly from this logic and we will also use it to reason about other aspects of the system.

In section ?? we give an algorithm for proving statements in the RAS logic and prove its correctness. This of course makes heavy use of the notation and results from section ??

In section ?? we briefly discuss the algorithm for verifying proofs generated by the prover.

In section ?? we give a specification of the RAS server using Z-notation, with the predicate logic normally used in the Z-schemas augmented by the RAS-logic to allow us to make much higher level statements succinctly. We also prove several important security properties of the RAS system.

In section ?? we discuss some aspects of the implementation of the RAS server and the libraries for accessing it.

Finally, in section ?? we discuss how the project progressed, what we have learned from it and how we might proceed.

2 A Logic For The RAS Model

In what follows we will develop a logic formalising how roles may be delegated, keys issued and the relationship between roles specified in the RAS model. We will give a language for statements about role membership, key ownership and role ordering and rules for deriving new statements from old ones. The intent is that the validity of the derivation rules be as self evident as possible, so that all parties making use of a system described using this logic will be confident that they correspond to real relationships between principals within an organisation.

We will develop a many-sorted predicate logic without negation, quantifiers, disjunction or the means to construct them. This results in a very restricted formalism which is not useful for general reasoning, but on the other hand is easier to treat formally and to implement efficient tools for.

2.1 Scope

The logic to be given will act as a formal framework for describing and reasoning about a particular extended $RBAC_1$ model, which we will call the RAS model. It does not capture all the details of an implementation of this model, but gives us a language with which to describe both the model and an implementation. An implementation may, for example, restrict the delegations, key certificates and role orderings it will accept beyond what is given below. There are also critical issues regarding freshness of information, particularly in relation to revocations, which are not treated by the logic. They can only be discussed in relation to a particular implementation, such as the RAS server.

2.2 Sorts

We will now introduce the sorts of the logic. Many of these sets will be ordered and in each case we will use the symbol ' \leq ' for the ordering. We will refer to the set of elements larger than a given set a as the upper-set of a , written a^\uparrow and the set of elements smaller than a is referred to as the lower-set of a and written a^\downarrow . Given a universe N , a partial ordering \leq and an element $a \in N$ we define:

$$\begin{aligned} a^\downarrow &:= \{ b \in N \mid b \leq a \} \\ a^\uparrow &:= \{ b \in N \mid a \leq b \} \end{aligned}$$

2.2.1 Periods

Time is considered in intervals of discrete instants, represented by the integers, much as it would be measured by the system clock of an actual computer system. This is appropriate here, since time is only used to compare time-stamps and the current time, all of which are generated from the system clock. We shall not and need not to argue about an arbitrarily dense sequence of events which might require a more advanced model of time. We will use the symbol t to range over time and apply the subset ordering on periods, but write $t_1 \leq t_2$ for $t_1 \subseteq t_2$.

The single instant t' and the period $\{t'\}$ containing only a single instant are not distinguished. We also define the notation $t_1 \dots t_2$ to mean $\{t \mid \min(t_1) \leq t \leq \min(t_2)\}$, which coincides with the meaning of this notation when t_1 and t_2 are single instants. Finally, $t_1 \uparrow t_2$, read t_1 revoked at t_2 , denotes $\{t \in t_1 \mid t \leq \min(t_2)\}$, which will be used for writing the validity period of revoked statements. We define α to be the instant before any other instant and ω to be the instant after all others. The period $\alpha \dots \omega$ thus encompasses all time.

Since the ' \leq ' relation is simply a renaming of the ' \subseteq ' relation, we inherit all the properties of that relation, which implies that the periods, along with the ' \leq ' ordering, form a lattice. We don't have a full ordering on the periods, since if we have four instants, $t_1 < t_2 < t_3 < t_4$ (without equality), then we will have that $t_1 \dots t_3 \leq t_1 \dots t_4$ and $t_2 \dots t_4 \leq t_1 \dots t_4$, but $t_1 \dots t_3$ and $t_2 \dots t_4$ are not comparable. We don't have a tree structure, since $t_2 \dots t_3 \leq t_1 \dots t_3$ and $t_2 \dots t_3 \leq t_2 \dots t_4$.

What we do have, is that any set of periods has a least upper bound and a greatest lower bound, which is the definition of a lattice. For two periods $t_a \dots t_b$ and $t_c \dots t_d$, the least upper bound is given by $\min(t_a, t_c) \dots \max(t_b, t_d)$ and the greatest lower bound is $\max(t_a, t_c) \dots \min(t_b, t_d)$, which may easily

be the empty period. For any finite set of periods we proceed by induction, while for infinite sets a more sophisticated approach is required, but as we have no interest in infinite sets of periods, we will not explore them here. We also have a well defined smallest period, given by the empty set and if we have a smallest (α) and largest (ω) element in the set of instants², then we have a largest period given by $\alpha \dots \omega$.

Visualising the graph induced by the ' \leq ' relation on the periods is not easy, since the set of periods is infinite and in fact each period will have an infinite set of 'parents'. However, if we restrict ourselves to only the set of periods which are explicitly mentioned in a set of statements, we will be able to generate a single-rooted directed acyclic graph from the ' \leq ' relation, possibly by adjoining the $\alpha \dots \omega$ period to serve as the root.

$t \in \{a \dots b \mid a, b \in N\}$	Periods are intervals of integers.
$t_1 \leq t_2 \Leftrightarrow t_1 \subseteq t_2$	Use the subset relation for ordering.
$t_1 \dots t_2 = \min(t_1) \dots \min(t_2)$	The interval between two intervals.
$t_1 \dagger t_2 = \{t \in t_1 \mid t \leq \min(t_2)\}$	The period t_1 revoked at t_2 .
$(t_1 \cap t_2) \dagger t_3 = t_1 \cap (t_2 \dagger t_3)$	Associativity of ' \cap ' and ' \dagger '

Table 1: Period Summary

2.2.2 Public keys

We have a set of public keys with no structure, which is ranged over by the symbol K . Presumably each public key will correspond to a private key held by the owner of the key (this is in fact the meaning of a public key belonging to a principal), but we shall not need to refer to private keys.

However, the existence of a statement whose signature can be verified with a particular public key implies that this statement was signed by the principal holding the corresponding private key. Further, since the principal signed the statement he must agree with it and be willing to accept the consequences of making it. This concept is the basis of many of the derivation rules which will be given in section ??.

2.2.3 Domains

The purpose of domains is to restrict the scope of any kind of authorisation to a particular physical or logical area. They are specified in terms of criteria on the items they contain and are ranged over by the symbol d . The set of criteria is formally given by a partial function from *component names* to *component values*. We shall not need to refer to these sets specifically, but they will typically be subsets of the finite character strings.

We will consider a domain d_1 to be a sub-domain of d_2 exactly when the criteria for an object to belong to d_1 is stronger than those for it to belong to d_2 . This leads us to write $d_1 \leq d_2$ whenever $d_2 \subseteq d_1$ ³. We finally define 'world' to be the universal domain, given by the partial function \emptyset which sets no criteria and is therefore all-inclusive.

The domains with the ' \leq ' ordering also form a lattice. Since ' \leq ' is simply a re-naming of the ' \supseteq ' ordering, we again inherit the lattice structure from there. To show that the domains form a lattice we need to show that any set A of domains has a smallest upper bound and a greatest lower bound. These are given by $\bigcap_{a \in A} a$ and $\bigcup_{a \in A} a$ respectively, where domains are considered to be sets of maplets.

The domain given by the empty set clearly contains all other domains by definition and it is also natural that the domain with no criteria should contain the most objects. However, there can be no smallest domain. Given any domain other than the largest one, we can modify the value that a particular component name maps to, which gives us an incomparable domain. We could do this to any candidate for the smallest domain, so none would be acceptable. Also, in actual implementations, each domain would be given by a finite set of maplets, while the set of component names is potentially infinite. We could therefore create a sub-domain of any proposed smallest domain simply by adjoining another maplet.

²This is required for infinite sets of periods to have upper and lower bounds.

³This construction corresponds to the x.500 distinguished names for domains, but with an extended ordering. x.500 also specifies various types of string comparisons for component values which we will model by converting all values to a canonical form.

As for the lattice structure, we can give an intuitive meaning to the smallest upper bound and greatest lower bound as well. The smallest upper bound of two domains, d_1 and d_2 is the domain whose set of maplets is the intersection of d_1 and d_2 . It therefore has the largest set of criteria such that they hold for the objects in both d_1 and d_2 and is the smallest domain which contains both d_1 and d_2 . Similarly the greatest lower bound has the smallest set of criteria such that they hold either for objects in d_1 or in d_2 and is the largest domain contained in both d_1 and d_2 . In fact it is the intersection of the two. Note that the smallest upper bound of d_1 and d_2 is not the union of the two domains.

$d \in \text{COMP_NAME} \setminus \{DN\}$ $\rightarrow \text{COMP_VALUE}$	Domains are partial functions from names to values.
world = \emptyset	the universal domain is given by the empty set of criteria.
$d_1 \leq d_2 \Leftrightarrow d_2 \subseteq d_1$	Ordering of domains.
$d_1 \wedge d_2 = d_1 \cup d_2$	Greatest lower bound.
$d_1 \vee d_2 = d_1 \cap d_2$	Least upper bound.

Table 2: Domain Summary

2.2.4 Principals

Principals are given names similar to the domains and are ranged over by the symbols P , Q and R . The names given to principals are of the same form as those for domains with the difference that the domain of the partial function contains the special value CN (for Common Name) which is not in the domain of any domain name. This allows us to distinguish between names for principals and domains.

We can now talk about principals belonging to domains. If all the criteria set by the domain are fulfilled by a principal, then the principal belongs to the domain and we write $P \in d$ when $d \subseteq P$.

See table ?? for a summary.

$P \in \text{COMP_NAME}$ $\rightarrow \text{COMP_VALUE}$	Principals are partial functions from names to values.
$CN \in \text{dom } P$	Principal names have a CN component.
$P \in d \Leftrightarrow d \subseteq P$	A principal belongs to a domain if it fulfils all criteria.

Table 3: Principal Summary

2.2.5 Roles

Roles denote labels for capabilities granted principals or required for operations and are ranged over by the symbol r . They have no intrinsic structure, but will typically be represented by a subset of the finite character strings. Usually in the *RBAC* model a distinction is made between roles and permissions. Principals are assigned to roles and the roles imply certain permissions. We will not make this distinction in the logic, but only in relation to the RAS implementation given in section ??.

A partial ordering on the set of roles and permissions will be introduced by the derivation rules in section ?? and depends on statements signed by principals in the system, which in turn are time dependent. When $r \leq r'$ at an instant t , according to the statements kept by the RAS server, we will write $r \leq_t r'$ and say that r is *dominated* by r' in the period t .

For these reasons the ordering of the roles only has meaning in relation to a particular RAS server. This also allows the server to control what role structures are considered acceptable. It would, for example, reject any statement which would cause a loop in the role-structure, i.e. a sequence of orderings by which we could deduce that $r \leq_t r'$ and $r' \leq_t r$. The RAS server may also decide to enforce other policies such as forcing a strict tree-structure or to distinguish between roles and permissions and force a particular structure based on this distinction.

$r \in \text{STRING}$	roles are strings.
$r \leq_t r'$	Role order is time-dependent and decided by statements held by server.

Table 4: Role Summary

2.3 Statements

The definition of any logic must begin with the definition of the language of statements which can be made in the logic. We will define a set Σ of statements, which is the union of Σ_0 and Σ_s which are defined separately below.

2.3.1 Simple Statements – Σ_0

We now define the language of statements which can be asserted or proven in our logic. They are concerned either with the assignment of principals to roles, keys to principals, signatures on statements or revocation of statements.

We first have a language of primitive statements which we will call Σ_0 and which describes access privileges that principals have, keys they hold and relations between roles and permissions. Queries made to the RAS system will be formulated in this language.

Capabilities The most important statements concern themselves with the capabilities of particular principals. The most basic function of the RAS server is to answer queries regarding the capabilities of particular a principal at a particular moment. This is done by sending a statement $P \mapsto_t [r \triangleright d]$ to the RAS server for verification, to query whether the principal P holds the role (or has the permission) r in domain d in period t .

What operations are open to the principal P if this query is successful depends entirely on the server making the query and is outside the scope of the logic. However, we will give derivation rules which show how statements of this type and other statements in Σ_0 can be derived from previously accepted statements, which constitutes the formal part of the semantics of these statements.

Essentially, there is a formal mechanism for deriving statements in the RAS logic, but the effect of deriving a statement conferring particular capabilities to a principal depends on the calling application and is decided by the application developer and the principal who defined the role or permission in question.

The second form of capability statement is $P \mapsto_t [r \triangleright d]$. It signifies that the principal P is allowed to delegate the role r in the domain d in the period t . The meaning of this statement is given formally by the derivation rules in sections ?? – ??, but informally means that the principal P can dictate that $Q \mapsto_{t'} [r' \triangleright d']$ for any principal Q , any period t' which is contained in t , any role r' which is dominated by r (see section ??) and any domain d' contained in d .

As an example, an accounting system could have defined a pair of roles, read-accounts and update-accounts⁴. If a principal P wishes to examine the accounts of a company d , the accounting system would send a capability query $P \mapsto_{now} [\text{read-accounts} \triangleright d]$ and would presumably allow P read access to the accounts if the RAS server verifies the statement. For accountants Q within the company it would be true that $Q \mapsto_{now} [\text{update-accounts} \triangleright d]$ and for the head accountant R we would have $R \mapsto_t [\text{update-accounts} \triangleright d]$ allowing him to delegate accounting capabilities to other principals.

Role Ordering Role ordering is written $r' \leq_t r$ and was mentioned previously in section ?? and the derivation rules for deriving role ordering are give in section ?. The intent is that a principal holding a role or permission r will implicitly hold any role or permission which is dominated by r , as per the derivation rules given in ??.

To continue the previous example, suppose that read-accounts \leq_{now} update-accounts, then we would expect that any accountant Q , who we know have $Q \mapsto_{now} [\text{update-accounts} \triangleright d]$, would also have $Q \mapsto_{now}$

⁴In a real-world system, these would probably be permissions embedded in a more complete hierarchy of roles and permissions, but this will suffice for our example.

[read-accounts \triangleright d] and therefore be able to read the accounts as well as update them. Further, the head accountant would be able to delegate the read-accounts role as well as update-accounts.

Key Ownership The way that authority is delegated in the RAS system, as well as defining new roles and permissions, is via signed statements. In order to be able to accept such a statement, we need to know that the key used to sign the statement actually belongs to the principal who supposedly issued the statement. We write $P \text{ Pub}_t K$ for principal P owning the public key K whose validity period is t .

The RAS will need to be able to verify many key ownership statements in the process of verifying capability statements and the semantics for this are made precise in section ???. However, other systems may also want to verify that particular keys belong to particular principals so they can be used in authentication protocols, to encrypt information for particular principals, to verify miscellaneous signatures or any other uses. The semantics here are outside the scope of the RAS logic, which simply establishes that a key belongs to a principal in a particular period.

$P \mapsto_t [r \triangleright d]$	P may perform operations requiring role r in domain d and period t .
$P \mapsto_t [r \triangleright d]$	P may delegate the role r in domain d and period t .
$r_1 \leq_t r_2$	The role r_1 is dominated by the role r_2 during the period t .
$P \text{ Pub}_t K$	K is a valid public key for P in the period t .
$P \text{ PPub}_t [K \triangleright d]$	P is a trusted CA with key K in the domain d in the period t .

Table 5: Σ_0 Summary

2.3.2 Signed Statements – Σ_s

Now that we have a language Σ_0 for describing the state that the access control system takes we build on it to define a language Σ_s which allows principals to issue statements to change this state.

Digital Signatures This is done by issuing *signed statements* which are initially formed by applying a digital signature with a key K to a statement $\sigma \in \Sigma_0$ to form a statement $\{\sigma\}_K$. The exact effect of such a statement is given by the derivation rules given in ??, but the intent is that the principal who owns the key K intends the statement σ to be true. Whether this is sufficient for a third party to take it to be true depends on the third party's view of this principal's authority, which in turn may depend on further signed statements.

We give names to the various types of signed statements, as can be seen in table ??. Each signed statement also has a revoked form with a revocation instant and additional signature.

Revocations The other form of signed statements is the revoked statement $\{\{\sigma\}_{K_i} \dagger t\}_{K_r}$, where a statement is first signed by the owner of K_i and subsequently revoked at instant t^5 and re-signed by the owner of K_r . The intent here is that at the instant t the owner of K_r would no longer wish σ to be true. He therefore embeds the signed statement of σ within a revocation statement and signs it. As discussed in ??, it is up to the RAS server to decide whether to accept this revoked statement as a replacement for the original.

Key Handling An important aspect of the RAS logic is the way keys are handled. Although we are making use of a public-key crypto-system, where keys come in pairs, a private key which can be used to sign data and a public key which can be used to verify signatures on data, we have no need explicitly to refer to the private keys. Rather, we simply judge statements based on the public key which can be used to verify the signature the statement carries, along with statements about this public key. It is of course implicit in being able to verify a signature with a public key that it was generated with the corresponding private key, but we have no need to encumber the formalism with this.

⁵More precisely, t may be an arbitrary period and the revocation is effective at the beginning instant of t .

$\{ \sigma \}_K$	Statement σ signed with key K .
$\{ \{ \sigma \}_{K_t} \dagger t \}_{K_r}$	Revocation of σ at instant t , signed with key K_r .
$\{ P \mapsto_t [r \triangleright d] \}_K$	Delegation of right to act in a role.
$\{ P \mapsto_t [r \triangleright d] \}_K$	Delegation of right to delegate a role.
$\{ r \leq_t R \}_K$	Label activation
$\{ P \text{ Pub}_t K_p \}_K$	Key certificate
$\{ P \text{ Pub}_t [K_p \triangleright K] \}_d$	CA certificate

Table 6: Σ_s Summary

Another thing that the reader might notice is that there is no notion of valid signatures and invalid signatures. In fact, there is no notion of a signature as such, but only of signed statements. I.e. we always write the data to be signed (which in our usage will be a statement from Σ_0) along with the key which can be used to verify a signature which the statement implicitly carries. If no signature is present, or if it can't be verified with the key specified, then it simply isn't a signed statement in the sense of the RAS logic and the various derivation rules don't apply.

2.3.3 Expressiveness of Σ

We have now defined the language Σ_0 in which we will specify what access privileges are required for various actions, as well as the language Σ_s in which principals can attempt to grant further privileges, issue keys or manage the role hierarchy. In section ?? we give the relationship between these two languages and the interpretation of statements in Σ_s in the form of derivation rules, but first we will consider the question of the expressiveness of the language. Is Σ sufficient for expressing the access privileges we need?

We will later give formal semantics to the statements in Σ_s in the form of rules by which we can derive statements from Σ_0 . It should be clear from these rules that the signed statements are sufficient to assign to principals any privileges that can be specified in Σ_0 . A more interesting question is how suitable Σ_0 is to specifying access control policies.

When a query is made whether a principal P has sufficient privileges to perform a particular action, the query will be of the form $P \mapsto_t [r \triangleright d]$.

We've noted before that access privileges may change with time and this is the purpose of the t parameter. However, there is no facility in this notation to give intermittent access to a resource. I.e. it is quite common to have the authority to enter a building during working hours, but not during the night or not at week-ends. This could only be managed in the RAS logic by repeatedly issuing and revoking delegations.

The second parameter is the role r . For expressiveness, it would be sufficient to have a non-structured set of roles as in the $RBAC_0$ model [?]. The structure introduced by $RBAC_1$ and which we manage through label activations is a mere convenience.

Domains The third and by far most interesting parameter, is the domain d , where most of the expressiveness of the logic is concentrated. The domain structure creates an infinite dimensional space with a dimension for each domain component. A domain is an assignment of a value to a finite set of components, fixing those components but leaving others free.

For example a company may have been issued the x.500 name "C=BE, O=Flex" and making this part of a domain name fixes the C and O dimensions, leaving all others free, creating an infinite-dimensional sub-space representing the company in question. This domain can then be further divided into departments (the OU component, according to the x.500 standard), divisions, etc.

Another view of a domain name is a list of constraints. These can be entirely arbitrary and include names projects, types of data or any other constraints. For example, the intermittent access that couldn't be specified using the temporal parameter in the beginning of this section, could be specified by having a constraint "Hours=Working" as part of a principal's access privileges and queries for access during working hours would include the "Hours=Working" constraint, while off hours it would include "Hours=Evening", "Hours=Week-end" and others, to any level of detail we may wish. Most users would be granted access

with “Hours=Working”, but occasionally, a user might be granted the authority twice, with two different “Hours” values and high-level people might be granted access with no constraints on “Hours” at all. However, these constraints will all be application specific and must be specified and correctly handled by the application which is querying the RAS server, creating a potential source of mistakes.

Most often, domains will be composed of components in both the x.500 style and the arbitrary constraints style. For example, a principal might be granted the administrator role (implying various other roles) in a domain “C=BE, O=Flex, OU=IT, Project=Web Services, Hours=Working”, to allow him to work with the IT department’s web services.

Given a coherent design of an organisation’s domain structure, people can be given access to certain aspects of resources which would normally be available to disjoint groups of people. For example, to be able to look at the logs for various systems and projects belonging to various departments, a principal might be granted the auditor role in the domain “C=BE, O=Flex, DataType=Logs”.

Supposing both the administrator role and the auditor role imply the role needed to examine logs, both the principals in the previous examples would be able to examine the logs in the domain “C=BE, O=Flex, OU=IT, Project=Web Services, DataType=Logs, Hours=Working”.

Pattern Matching One problem remains in the domain system. There is no way to specify that a domain component must have one of several values, must hold a numerical value greater than a certain constant, matches a regular expressions or returns a particular value when a function is applied to it.

These would allow much finer control in the delegation of authority, but are complications deferred to a later version of the RAS. They would also require the introduction of binary operators to allow the construction of the intersection of two domains, since if one domain specifies a regular expression for a component and another specifies that the value must be all in lower case, then both criteria must be met for an item to belong to the intersection of the domains and we must have conjugation.

2.4 Derivation Rules

Here we will give the rules by which statements may be derived from other statements. The derivation rules are given using the notation $\frac{a_1, \dots, a_n}{d}$ (NAME), with any number of antecedents $a_i \in \Sigma$, a single descendant $d \in \Sigma_0$ and a name. The intent is that if all the antecedents are derivable, for example by being taken as axioms, then the descendant is derivable.

The aim of these derivation rules is to assign meaning to the signed statements. Each rule will have a descendant in $d \in \Sigma_0$ and at most one of the antecedents will be from Σ_s . This assigns a meaning to the antecedents from Σ_s in terms of statements from Σ_0 .

2.4.1 Role Delegation

The following two derivation rules say how principals are assigned to roles and given the authority to assign principals to roles.

$$\frac{P \models_{t_1} [r \triangleright d] ; P \text{ Pub}_{\min t_2} K ; \{ Q \models_{t_2} [r \triangleright d] \}_K}{Q \models_{t_1 \cap t_2} [r \triangleright d]} \text{ (DEL-DEL)}$$

$$\frac{P \models_{t_1} [r \triangleright d] ; P \text{ Pub}_{\min t_2} K ; \{ Q \mapsto_{t_2} [r \triangleright d] \}_K}{Q \mapsto_{t_1 \cap t_2} [r \triangleright d]} \text{ (DEL-MAY)}$$

The two rules differ only in that the former describes how the right to delegate roles is delegated, while the latter describes how the right to act in roles is delegated.

The antecedents imply that:

1. The principal P has the authority to delegate the role/domain pair $[r \triangleright d]$ during a period t_1 .
2. The principal P holds the key K at the instant $\min t_2$, which is the instant when the delegation is to take effect.
3. A statement $Q \models_{t_2} [r \triangleright d]$ or $Q \mapsto_{t_2} [r \triangleright d]$ about role assignment has been signed with the key K .

We then derive that the statement which was signed with K is valid, as long as P 's authority to delegate the role/domain pair holds.

2.4.2 Role Weakening

The previous two rules are in fact too narrow and rigid to be useful by themselves. They only allow a principal to delegate the exact same role he holds in the exact same domain for the entire validity period of his own delegation of the role.

We therefore add the following two weakening rules, which along with a similar rule for weakening public keys allows for the appropriate flexibility in restricting the scope of a delegation.

$$\frac{P \Vdash_{t_1} [r_1 \triangleright d_1] ; r_2 \leq_{t_2} r_1 ; d_2 \leq d_1 ; t_2 \leq t_1}{P \Vdash_{t_2} [r_2 \triangleright d_2]} \text{ (WEAK-DEL)}$$

$$\frac{P \mapsto_{t_1} [r_1 \triangleright d_1] ; r_2 \leq_{t_2} r_1 ; d_2 \leq d_1 ; t_2 \leq t_1}{P \mapsto_{t_2} [r_2 \triangleright d_2]} \text{ (WEAK-MAY)}$$

Again, these two rules differ only in whether they describe the right to delegate a role or the right to act in a role. In either case a principal P is considered to hold a given role in a particular domain for a specific period if he has been shown to hold a dominating role in a larger domain in an enclosing period.

The ordering for domains was given in section ?? and the one for periods in section ?. The ordering for roles, however, depends on the signed label activations that we have as described in section ??

2.4.3 Role Revocation

Role delegations may need to be revoked for various reasons. In such cases the delegation of the role would be replaced by a truncated or revoked version, the rules for which are given below:

$$\frac{P \Vdash_{t_1} [r \triangleright d] ; P \text{Pub}_{\min t_2} K_p \quad ; \quad \{ \{ R \Vdash_{t_2} [r \triangleright d] \}_{K_p} \uparrow t_3 \}_{K_q}}{R \Vdash_{t_1 \cap t_2 \uparrow t_3} [r \triangleright d]} \text{ (DEL-DEL}\uparrow\text{)}$$

$$\frac{P \Vdash_{t_1} [r \triangleright d] ; P \text{Pub}_{\min t_2} K_p \quad ; \quad \{ \{ R \mapsto_{t_2} [r \triangleright d] \}_{K_p} \uparrow t_3 \}_{K_q}}{R \mapsto_{t_1 \cap t_2 \uparrow t_3} [r \triangleright d]} \text{ (DEL-MAY}\uparrow\text{)}$$

Yet again the two rules differ only in whether they describe the right to delegate a role or the right to act in a role. In either case, if we had only the knowledge about the principal P and the statement he signed, embedded in the overall statement, then we could apply one of the DEL- rules given previously. Here, however, the embedded statement was revoked at instant t_2 ⁶. The result is the same as the application of the previous DEL- rule, except that its effect will be cancelled for any instant after (the beginning instant of) t_2 .

For the truncated delegation to be valid, we require both that the full delegation would have been valid, i.e. we could have applied the previous DEL- rule successfully and that the revoking principal Q has the right to delegate the role/domain pair at the instant t_2 and that he properly signed the complete message with a valid key.

2.4.4 Key Issue

We now turn to the rules for issuing keys. This has many similarities with the x.509 system, which is described in section ?. We will refer to signed statements about key ownership or about the right to issue keys using a particular key as key certificates and the principals authorised to issue them as Certification Authorities.

⁶technically, of course, t_2 may be a period and we would only consider its starting instant.

$$\frac{P \mathbb{P}ub_{t_1} [K_p \triangleright d] ; Q \in d ; \{ Q \mathbb{P}ub_{t_2} [K_q \triangleright d] \}_{K_p}}{Q \mathbb{P}ub_{t_1 \cap t_2} [K_q \triangleright d]} \text{ (CA-CA)}$$

$$\frac{P \mathbb{P}ub_{t_1} [K_p \triangleright d] ; Q \in d ; \{ Q \mathbb{P}ub_{t_2} K_q \}_{K_p}}{Q \mathbb{P}ub_{t_1 \cap t_2} K_q} \text{ (CA-KEY)}$$

These two rules are similar to the two non-revoked DEL- rules, but with some notable differences. They are concerned with issuing certificates for CA's and other principals respectively, mirroring the two types of delegations for roles.

The antecedents imply that:

1. The principal P has the authority to issue keys in the domain d using the key K during a period t_1 .
2. The principal Q to whom the key is being issued belongs to the domain d in which P may issue keys.
3. If a CA certificate is being issued (first case), then the domain d noted in the certificate must be identical to the domain in which the issuer may issue keys.
4. A statement $Q \mathbb{P}ub_{t_2} [K_q \triangleright d]$ or $Q \mathbb{P}ub_{t_2} K_q$ about key ownership has been signed with the key K .

We then derive that the statement which was signed with K is valid, as long as P 's authority to issue keys with K holds.

2.4.5 Key Weakening

As with role delegations the above key certification rules are too rigid to be useful and again we supply weakening rules to make them more applicable.

$$\frac{P \mathbb{P}ub_{t_1} [K_1 \triangleright d_1] ; t_2 \leq t_1 ; d_2 \leq d_1}{P \mathbb{P}ub_{t_2} [K \triangleright d_2]} \text{ (WEAK-CA)}$$

$$\frac{P \mathbb{P}ub_{t_1} K ; t_2 \leq t_1}{P \mathbb{P}ub_{t_2} K} \text{ (WEAK-KEY)}$$

Again, we have a version of the rules for CA's and another for normal key holders. We can in either case shorten the period in which we consider a key certificate to hold and in the case of CA's we can shrink the domain in which keys may be issued. The ordering for domains was given in section ?? and the one for periods in section ??.

2.4.6 Key Revocation

As with role delegations, key certificates may need to be revoked for a variety of reasons. This can for example be because a principal leaves an organisation and we no longer wish him to be able to identify himself with a name which specifies that he belongs to the organisation. Revoking the key certificate will make him unable to successfully participate in an authentication protocol using the revoked key.

Another possible reason is that the key may have been compromised, in which case we no longer trust that messages signed with it were signed by the principal who was previously believed to control it.

$$\frac{P \mathbb{P}ub_{t_1} [K_p \triangleright d] ; R \in d ; \{ \{ R \mathbb{P}ub_{t_2} [K_r \triangleright d] \}_{K_p} \dagger t_3 \}_{K_q}}{R \mathbb{P}ub_{t_1 \cap t_2 \dagger t_3} [K_r \triangleright d]} \text{ (CA-CA}\dagger\text{)}$$

$$\frac{P \mathbb{P}ub_{t_1} [K_p \triangleright d] ; R \in d ; \{ \{ R \mathbb{P}ub_{t_2} K_r \}_{K_p} \dagger t_3 \}_{K_q}}{R \mathbb{P}ub_{t_1 \cap t_2 \dagger t_3} K_r} \text{ (CA-KEY}\dagger\text{)}$$

The mechanism is very similar to that for role delegations. A version is given for certificates for CA's and another for normal certificates.

For the truncated key to be valid, we require both that the key would have been considered valid prior to revocation, i.e. we could have applied the previous CA- rule successfully to the statement embedded in the revocation and the statements about P , that the revoking principal Q has the right to issue keys in the domain at the instant t_2 and that he properly signed the complete message with a valid key.

2.4.7 Role Ordering

The final set of rules concerns the ordering of roles. This ordering is time dependent and changes when statements about it are made by principals trusted to do so.

$$\frac{r_1 \leq_t r_2 ; r_2 \leq_t r_3}{r_1 \leq_t r_3} \text{ (TRANSITIVITY)} \quad \frac{}{r \leq_t r} \text{ (REFLEXIVITY)}$$

Which establishes ' \leq_t ' as a partial ordering on roles for every t

$$\frac{P \mapsto_{\min t} [\text{rm} \triangleright \text{world}] ; P \text{Pub}_{\min t} K ; \{ r_2 \leq_t r_1 \}_K}{r_2 \leq_{t..w} r_1} \text{ (ORDER)}$$

We can obtain that a role r is dominated by another role or set of roles r' if a statement to this effect is signed by a principal who is active in the role-manager role (abbreviated rm in the logic) in the global domain at the instant the statement is made. In the absence of a revocation (see below) we will heed this statement for all eternity.

And yet again we can shorten the period for which a statement holds.

$$\frac{r_2 \leq_{t_1} r_1 ; t_2 \leq t_1}{r_2 \leq_{t_2} r_1} \text{ (WEAK-ORDER)}$$

$$\frac{P \mapsto_{\min t_1} [\text{rm} \triangleright \text{world}] ; P \text{Pub}_{\min t_1} K_p ; \{ \{ r_1 \leq_{t_1} r_2 \}_{K_p} \dagger t_2 \}_{K_q}}{Q \mapsto_{\min t_2} [\text{rm} \triangleright \text{world}] ; Q \text{Pub}_{\min t_2} K_q ; \{ \{ r_1 \leq_{t_1} r_2 \}_{K_p} \dagger t_2 \}_{K_q}}}{r_1 \leq_{t_1..t_2} r_2} \text{ (ORDER}\dagger\text{)}$$

Statements about the ordering of roles may be revoked. Since the ordering on roles does not bestow any powers upon principals in the absence of role delegations and key certificates, they are not revoked in response to a change in status of principals. Rather, they may change as the structure of the organisation making use of the system changes.

For the truncated ordering statement to be valid, we require both that it would have been considered valid prior to revocation, i.e. we could have applied the previous ORDER rule successfully to the statement embedded in the revocation and the statements about P , that the revoking principal Q has the right to manage roles at the instant t_2 and that he properly signed the complete message with a valid key.

$$\frac{r_1 \leq_t R ; r_2 \in R}{r_1 \leq_t r_2} \text{ (ORDER-SET)}$$

If a role is dominated by a set of roles, then it is dominated by each role in the set.

2.4.8 Avoiding Negative Action

It may seem like the revoked rules, i.e. the ones whose names end with \dagger , imply *negative action*, that adding them to the logic might render statements unprovable which would otherwise be provable. This is clearly not true as the previous derivation of the statement, presumably using the DEL-DEL or DEL-MAY rules, would still be possible. The point of these rules is to interpret revoked statements when the original statement has been removed from the set of trusted statements.

In fact the reason for having adding revoked statements to the logic, as they have been previously described, was to avoid having to resort to negative action to revoke otherwise seemingly valid statements. The problem is apparent in x.509, for example, where key certificates are issued by a CA who also keeps a Certificate Revocation List (CRL). Every time a certificate is used, it must be looked up in the most recent and relevant CRL to verify that it has not been revoked. We would have to model this by first giving rules similar to the CA-CA and CA-KEY rules which assign meaning to certificates, but then add negative action

rules saying that if the certificate were found in a CRL, then the validity period of the certificate should be truncated at the revocation instant. This would cause innumerable technical problems including making the derivability relation non-monotone and making it difficult to assemble derivation trees from sub-trees.

2.4.9 Avoiding Complexity

There are several derivation rules which are deliberately excluded from the logic for simplicity or to make an automated prover for the logic practical.

For example, there is no rule of the form

$$\frac{Q \mapsto_{t_1} [r \triangleright d] ; Q \mapsto_{t_2} [r \triangleright d] ; t_1 \cap t_2 \neq \emptyset}{Q \mapsto_{t_1 \cup t_2} [r \triangleright d]} \text{ (NOT USED)}$$

even though it would make intuitive sense. This means that even though a principal may hold a role for every instant t in a period t_3 , we may need several independent proof-trees to show this, for different portions of t_3 .

The reason that this rule would increase the complexity of the logic is that in general, the rules for delegation allow a principal to issue a delegation which is less powerful than the authority he himself holds. This rule, however, would allow delegation by two principals to be combined to give a third principal more authority than either of the two, although in this case it is only for a longer period. Rules of this kind would make it much more difficult for a proving algorithm to decide which delegations and certificates might be of interest when proving a particular statement and might easily lead to asymptotic running times for the prover to rise dramatically.

Another element conspicuously missing from the logic is any kind of negation. Proving the negative of a statement would require the introduction of universal quantifiers. I.e. if $\neg \sigma$ can be derived, then it may be because $A, S \not\rightarrow \sigma$, which follows from $\forall a \in (A \cup S)^* \bullet (a, \sigma) \notin e$, using the notation to be introduced in section ???. This could be verified by a mechanical prover by examining the set of statements it has available to construct a proof from, much as when it proves a positive statement. However, to construct a proof which could be verified independently it would need to include the entire set of statements. The proof would then take as long to verify as to construct and would become quite large and costly to transmit over a network. Finally, in order to trust the conclusion of the proof, the verifier must assume that the prover did in fact include all the statements at its disposal, which is as big an assumption as simply assuming the prover will honestly answer “unable to prove” when it encounters a (non-negative) statement it can’t construct a proof for.

2.4.10 Temporal Considerations

Most of the derivation rules in section ??? has one or more periods as subscripts on the various statements. These subscripts are easy to overlook on a first reading of the rules, but are really quite intricate and very important for the logic not to exhibit behaviour contradicting the likely intent of issuing delegations, etc.

Delegations The most important case is the rules for interpreting delegations, i.e. the DEL-* and DEL-* \dagger rules in section ???. In the DEL-DEL rule, for example, there are two periods referenced, t_1 is the period during which the issuer P of the delegation holds the authority to issue such delegations and t_2 is the period during which he intends the subject Q to hold the delegated authority. The effective period of the delegation is then taken to be the intersection of these two periods. It might be tempting, for the sake of simplicity, to write the rule with t_1 and t_2 referring to the same period and rely on weakening rules to adapt the actual period of P ’s authority. The rule would then look like this:

$$\frac{P \Rightarrow_{t_1} [r \triangleright d] ; P \text{ Pub}_{\min t} K ; \{ Q \Rightarrow_{t_2} [r \triangleright d] \}_K}{Q \Rightarrow_{t_1 \cap t_2} [r \triangleright d]} \text{ (NOT USED)}$$

However, there are problems with this approach, the most serious of which becomes apparent if P ’s authority to delegate is revoked. Suppose that a delegation $Q \Rightarrow_{t_2} [r \triangleright d]$ is present and the criteria to be able to apply the modified DEL-DEL rule above are satisfied. Then, it will be true that $Q \Rightarrow_{\min t} [r \triangleright d]$ by

WEAK-DEL. However, now suppose that P 's authority to delegate is revoked at some instant during t . It will now no longer be true that $Q \models_{\min t} [r \triangleright d]$, since $P \models_t [r \triangleright d]$ has been revoked. With the DEL-DEL rule which we are using, however, the result of the DEL-DEL rule will still be valid, but with a truncated validity period.

Another detail of the DEL-DEL rule and other rules for delegation is that the key K used to sign a delegation need only be valid at the instant $\min t_2$ when the delegation is to take effect. This choice may seem arbitrary, but we will later see that the RAS server only accepts statements that will take effect immediately, so this implies that the key was valid when the statement was accepted by the server. Even if it were later compromised, signatures known to have been made earlier would still be acceptable. The alternative would be to require K to be valid during t_1 and rely on weakening to select a t_1 during which both the authority to delegate and the key used to sign the delegation is valid. This, however, would mean that once a key becomes compromised or expires, anything that was signed with the key before it being compromised would immediately become invalid. This would be a great disruption to the users of the RAS system and would encourage the issuing of keys with very long validity periods and not revoking keys that are in doubt. Requiring K to be valid during t_2 would cause the same problem as requiring P 's authority to be valid during t_2 .

The revocation rules, DEL- $\ast\ddagger$ are now a simple extension of the other DEL- \ast rules.

Key Certificates The key issuing rules, CA- \ast and CA- $\ast\ddagger$ are similar to the delegation rules, except that a single statement $P \text{ Pub}_t [K \triangleright d]$ implies both the validity of the key K and P 's authority to use it to issue key certificates. This means that we are effectively forced to choose the equivalent of the option to let delegations expire when the keys that they were issued with expires. For key certificates we have that if a CA's certificate becomes compromised, then all key certificates issued with that certificate immediately become invalid. This matches the standard rules for x.509 certificates.

Label Activations Finally we have label activations. The rules for label activation have somewhat different temporal characteristics to the previous ones. In the case of delegations and key certificates, a principal P makes a claim about a principal Q and we believe the statement about Q based on P 's authoritativeness. When P loses his authority, we no longer trust statements that he has made about Q to hold. This is not so with label activations, where P 's authority is only relevant at the instant when the a label is activated or removed, i.e. when a change is made to the label hierarchy. If P activates a label, this label stays active until he or another principal with the required authority revokes the activation. This is clear in the ORDER and ORDER \ddagger rules, where we only require the issuer of an activation to hold a key and the appropriate authority at the instant $\min t$, where t is the period noted in the label activation. The label activation results in a label ordering statement which is valid in the period $t \dots \omega$, i.e. until the end of time.

Revoked label activations are handled in a similar manner and are valid in the period $t_1 \dots t_2$, if they are issued at t_1 and revoked at t_2 .

2.5 Assumed Trust

We have now seen how statements signed by trusted principals can lead to new trusted statements. However, there are no axioms in the previous section. An axiom in this setting corresponds to implicit trust in particular principals or keys.

Below we give an example list of axioms which would be sufficient to set up any valid role hierarchy, any valid configuration of trusted certification authorities and, most importantly, any valid set of role assignments.

$$R \models_{\alpha \dots \omega} [\text{root} \triangleright \text{world}] ; R \text{ Pub}_{\alpha \dots \omega} K_r$$

These axioms give to a principal R the ability to delegate the root role in the universal domain for all time. A public key is also assigned to R . Assuming the root role dominates all other roles⁷, then the rule

⁷This will be enforced by the RAS implementation, but is not implicit in the RAS logic.

WEAK-DEL can be applied to the first statement yielding any combination of role, domain and period. This, along with the second axiom allows R to delegate any role by the DEL-DEL or DEL-MAY rules.

$$CA \text{ Pub}_{\alpha \dots \omega} [K_{ca} \triangleright \text{world}]$$

This axiom gives the principal CA the ability to issue key certificates within any domain and with any validity period. More importantly, though, it also gives CA the ability to appoint other CA 's to issue certificates within particular domains.

$$\text{rm} \leq \text{root}$$

Finally, we assume that the rm role is subordinate to the root role at all times. This is needed so that the R principal of the first two axioms is able to delegate the authority to build the role hierarchy to some principal.

2.6 Derivability

We now have a set of rules for deriving statements from other statements and an example set of axioms to initialise the system. Given a set of statements, either asserting role or key assignments or claiming that a sub-statement was signed with a particular key, we can attempt to apply one or more derivation rules to derive further statements.

Each derivation rule can be viewed as a partial function $d : \Sigma^n \rightarrow \Sigma_0$ from one or more antecedent statements to a single result statement. The union of these functions is a relation relating a vector of statements to a single statement: $e : \Sigma^* \rightarrow \Sigma_0$. We say that a statement s is *directly derivable* from a set of statements A if there exists an $a \in A^*$ such that $(a, s) \in e$. Thus direct derivability means that a single application of a derivation rule will yield the result.

However, we will usually need multiple applications of derivation rules to get the result we want. We will write $A \rightarrow B$ for two sets of statements $A, B \subseteq \Sigma$ when B contains only statements from A and at most one additional statement which is directly derivable from statements in A . We therefore have $\rightarrow = \{(A, B) \mid B \subseteq A \vee (\exists s \in \Sigma_0 \bullet (A, s) \in e \wedge B \subseteq A \cup \{s\})\}$. The derivability operator can now be defined as $\vdash \Rightarrow \rightarrow^*$.

$A \rightarrow B$	all statements in B are directly derivable from A .
$A \vdash B$	all statements in B are derivable from A .

Table 7: Notation for derivability.

Often we will write either A or B as a list of statements or a list of sets of statements. This should be interpreted as the set containing all the statements in the list or the union of all the sets in the list. We note that the empty set is now derivable from any set of statements by virtue of being a subset of any set and that only the empty set is derivable from the empty set. In this notation axioms are not introduced by application of an axiom introduction rule, but by including them in the set from which statements are being derived.

By the monotonicity of \rightarrow and thus \vdash , with respect to the subset relation, we also have obvious properties such as that if $A \vdash B$ and $A \subset A'$ then $A' \vdash B$ and further if $B' \subset B$ then $A \vdash B'$.

We can immediately write the derivation rules in this notation, so for example the rule DEL-DEL becomes

$$P \Rightarrow_t [r \triangleright d] ; P \text{ Pub}_t K ; \{ Q \Rightarrow_t [r \triangleright d] \}_K \rightarrow Q \Rightarrow_t [r \triangleright d]$$

We can also combine the DEL-DEL and WEAK-DEL rules for:

$$P \text{ Pub}_t K ; \{ Q \Rightarrow_t [r \triangleright d] \}_K \vdash Q \Rightarrow_{t'} [r' \triangleright d']$$

$$P \Rightarrow_t [r \triangleright d] ; t' \leq t ; r' \leq r ; d' \leq d$$

We can now finally state how we define a statement $\sigma \in \Sigma_0$ to be valid. The RAS server will have a set of axioms $A \subseteq \Sigma_0$, which it shares with all systems making use of its services. It will also hold a set

$S \subseteq \Sigma_s$ of current statements, which are statements it has been sent by any number of principals and which it has decided to accept and which have not been superseded by revocations.

We will take a set B of statements to be valid when ever $A, S \vdash B$. In other words, if $A, S \vdash P \mapsto_t [r \triangleright d]$, then the principal P may in fact be active in the role r in the domain d in the period t .

Example

By applying several derivation rules we can obtain

$$\left\{ \begin{array}{l} R \Rightarrow_{\alpha \dots \omega} [\text{root} \triangleright \text{world}] ; R \text{Pub}_{\alpha \dots \omega} K_r ; \\ \text{rm} \leq_{\alpha \dots \omega} \text{root} ; \{ \text{user} \leq_{5 \dots \omega} \text{root} \}_{K_r} ; \\ \{ R \mapsto_{\alpha \dots \omega} [\text{rm} \triangleright \text{world}] \}_{K_r} ; \\ \{ P \mapsto_{10 \dots 20} [\text{user} \triangleright \text{CMP}] \}_{K_r} \end{array} \right\} \vdash P \mapsto_{10 \dots 15} [\text{user} \triangleright \text{DPT}]$$

where

$$\begin{aligned} \text{CMP} &= \{ C \mapsto \text{'BE'}, O \mapsto \text{'Flex'} \} \\ \text{DPT} &= \{ C \mapsto \text{'BE'}, O \mapsto \text{'Flex'}, \text{OU} \mapsto \text{'Accounting'} \} \end{aligned}$$

To convince ourselves that this is so we must construct a proof-tree with $P \mapsto_{15} [\text{user} \triangleright \text{DPT}]$ as the result (i.e. as the result of the derivation at the root of the tree) and only axioms and statements from the left side of the ' \vdash ' as leaves. We construct such a proof-tree below.

We start by using the WEAK-MAY rule to match the desired result with the role appropriate delegation we have for the principal P .

$$\frac{P \mapsto_{10 \dots 15} [\text{user} \triangleright \text{CMP}] ; \text{user} \leq_{10 \dots 15} \text{user} ; \text{DPT} \leq \text{CMP} ; 10 \dots 15 \leq 10 \dots 20}{P \mapsto_{10 \dots 15} [\text{user} \triangleright \text{DPT}]}$$

The last two antecedents are clear from looking at the actual periods and domains and the definition of their orderings. The second antecedent is a result of ' \leq_{15} ' being a partial ordering and therefore reflexive. However, we still need to provide a sub-proof-tree for $P \mapsto_{15} [\text{user} \triangleright \text{CMP}]$.

We apply the WEAK-KEY and DEL-MAY rules to obtain

$$\frac{R \Rightarrow_{10 \dots 15} [\text{user} \triangleright \text{CMP}] ; \frac{R \text{Pub}_{\alpha \dots \omega} K_r ; 10 < \alpha \dots \omega}{R \text{Pub}_{10} K_r} ; \{ P \mapsto_{10 \dots 20} [\text{user} \triangleright \text{CMP}] \}_{K_r}}{P \mapsto_{10 \dots 15} [\text{user} \triangleright \text{CMP}]}$$

The only incomplete branch of the proof-tree is now the one now ending in $R \Rightarrow_{10 \dots 15} [\text{user} \triangleright \text{world}]$. We replace this statement in the proof-tree with a sub-proof-tree using only the WEAK-DEL rule.

$$\frac{R \Rightarrow_{\alpha \dots \omega} [\text{root} \triangleright \text{world}] ; \frac{\text{user} \leq_{5 \dots \omega} \text{root} ; 10 \dots 15 < 5 \dots \omega}{\text{user} \leq_{10 \dots 15} \text{root}} ; \text{CMP} \leq \text{world} ; 10 \dots 15 \leq \alpha \dots \omega}{R \Rightarrow_{10 \dots 15} [\text{user} \triangleright \text{world}]}$$

This leaves us with only the unproven statement $\text{user} \leq_{5 \dots \omega} \text{root}$, which we can replace with the following sub-proof-tree which uses the ORDER, WEAK-MAY and WEAK-KEY rules.

$$\frac{\frac{R \mapsto_{\alpha \dots \omega} [\text{root} \triangleright \text{world}] ; \text{rm} \leq_5 \text{root} ; \text{world} \leq \text{world} ; 5 < \alpha \dots \omega}{R \mapsto_5 [\text{rm} \triangleright \text{world}]} ; \frac{R \text{Pub}_{\alpha \dots \omega} K_r ; 5 < \alpha \dots \omega}{R \text{Pub}_5 K_r} ; \{ \text{user} \leq_{5 \dots \omega} \text{root} \}_{K_r}}{\text{user} \leq_{5 \dots \omega} \text{root}}$$

which finally closes the last branch of the proof-tree.

Although this proof may seem cumbersome to go through manually the way it is constructed lends itself easily to automation, which will be explored later. Also note that this proof is in fact rather small. All delegations are made by the root principal directly and there is no need to use any of the key issuing rules or any revocation rules. The nature of the proof would not change if these rules were needed, but the proof would become somewhat larger.

2.7 Formal Proofs

The RAS system works by constructing proofs of capabilities which may be examined by other parties in the system. For this purpose we need a more precise formulation of a proof than the one given previously. The approach is naturally similar to the semi-formal proofs in the last section, but rather than graphically representing proofs, we will encode them as functions mapping hypothetical statements to sets of premise statements which are sufficient proof of the hypothesis. If the premises are axioms or current statements, the proof is complete. Otherwise each non-axiomatic premise which is not a current statement must in turn be proven recursively by having a similar maplet to another set of premises.

Definition 1 Given two finite sets $A \subset \Sigma_0$ and $S \subset \Sigma_s$, let \vdash be the smallest relation relating functions $p : \Sigma_0 \rightarrow \mathbb{P}\Sigma$ to statements σ , which is closed under the two rules given below. We will read $p \vdash S$ as $\forall \sigma \in S \mid p \vdash \sigma$ when S is a set or list of statements and write $A, S \vdash_p \sigma$ when \vdash relates p to σ .

$$\frac{\sigma \in A}{A, S \vdash_p \sigma} \text{ (AXIOM)}$$

$$\frac{\sigma \in \text{dom } p; A, S \vdash_p p(\sigma); p(\sigma) \rightarrow \sigma}{A, S \vdash_p \sigma} \text{ (DERIVE)}$$

A formal proof is a function $p : \Sigma \rightarrow \mathbb{P}\Sigma$ such that for any $\sigma \in \text{dom } p$ we have $A, S \vdash_p \sigma$. When $A, S \vdash_p \sigma$ we say that p is a proof of σ or that p proves σ , adding “from A and S ” when needed to avoid confusion.

It is clear from the above definition that the existence of a formal proof p of σ implies that σ is derivable from $A \cup S$, since each use of the DERIVE rule in showing that p is a proof corresponds to one application of a derivation rule in deriving σ from $A \cup S$.

2.7.1 Combining proofs

Proofs often need to be combined from proofs of premises of a particular derivation, along with the derivation itself, to produce a larger proof. This allows us to write divide and conquer algorithms to prove statements, such as the one seen in section ???. We will give a theorem for the validity of such combinations.

Proofs are combined using the function overriding operator “ $p \oplus q$ ”, which is defined below. In combining proofs using overriding, any maplets in the left-hand operand which have the same origin as maplets in the right-hand operand will be lost and replaced by those from the right-hand operand. This is needed to maintain the functional property of the relation and we must show that the result is still a valid proof.

$$p \oplus q(\sigma) = \begin{cases} q(\sigma), & \text{if } \sigma \in \text{dom } q \\ p(\sigma), & \text{if } \sigma \in \text{dom } p \setminus \text{dom } q \\ \text{undefined}, & \text{otherwise} \end{cases}$$

Lemma 2.1 (Combined Proofs) Given two proofs p and q , we have that $p \oplus q$ is a proof and in particular:

$$\frac{A, S \vdash_p a; A, S \vdash_p b}{A, S \vdash_{p \oplus q} a, b}$$

Proof: $\text{dom } p \oplus q = \text{dom } p \cup \text{dom } q$ and we must show that any statement $\sigma \in \text{dom } p \oplus q$ is proved by $p \oplus q$. This is clear when σ is in the domain of q , so we will assume that $\sigma \in \text{dom } p \setminus \text{dom } q$ and we proceed by a very small structural induction over the derivation that $A, S \vdash_p \sigma$.

If $A, S \vdash_p \sigma$ by the AXIOM rule, this is independent of the proof and we are done.

If $A, S \vdash_p \sigma$ by the DERIVE rule then we will show that the DERIVE rule applied to $p \oplus q$ gives $A, S \vdash_{p \oplus q} \sigma$. Clearly, $\sigma \in \text{dom } p \oplus q$ and by the induction hypothesis we have $A, S \vdash_{p \oplus q} p(\sigma)$, so $A, S \vdash_{p \oplus q} p \oplus q(\sigma)$. We also have that $p \oplus q(\sigma) = p(\sigma)$ and since $A, S \vdash_p \sigma$ by the DERIVE rule, then $p(\sigma) \rightarrow \sigma$, whereby $p \oplus q(\sigma) \rightarrow \sigma$.

Lemma 2.2 (Extended Proofs) *Given a proof p , statement a and a set of statements B we have:*

$$\frac{A, S \vdash_p B; B \rightarrow a}{A, S \vdash_{\{a \rightarrow B\} \oplus p} a}$$

Proof: *If $p \vdash a$, the extended proof is equal to p and we have the result. Otherwise we can apply the DERIVE rule to the extended proof.*

Theorem 2.3 (Recursive Proofs) *Given a statement a , along with a sequence of statements $(b_i)_{i=1}^n$ and a sequence $(p_i)_{i=1}^n$ of proofs we, have:*

$$\frac{p_i \vdash b_i; b_1 \dots b_n \vdash a}{\{a \mapsto \{b_1 \dots b_n\}\} \oplus p_1 \dots p_n \vdash a}$$

Proof: *This follows from repeated application of lemmas ?? and ??.*

We can now prove a hypothesis by first proving all premises needed to apply a derivation rule and then combining the proofs with a single additional maplet using theorem ??. This will be applied in section ??

2.8 Soundness and Completeness

Before deciding whether we consider the logic to be sound, we need to ask ourselves what we mean by soundness in this setting. Since we have no negations in the logic we are not likely to encounter contradictions of the form $a \wedge \neg a$. We may get various judgements about the periods in which a principal holds a particular role or key, but these specifically say nothing about what may be true outside their period. Neither is there any claim that they give the largest or smallest period, domain or role which would give a particular permission. In fact, the question of which such judgement would give the ‘largest’ answer is not well defined.

With this in mind, even two judgements about a principal having a particular permission, where one is based on a delegation truncated by revocation and therefore giving a shorter period would not be an inconsistency. Ensuring that revoked delegations and certificates are not used in their un-revoked form is, as stated earlier, not within the scope of the logic and any treatment of it is deferred until the specification of the authorisation server.

Normal predicate logic and many specially constructed ones are expected to capture the properties of a pre-existing model and questions of soundness and completeness are asked and answered in relation to this model. Here, however, there is no pre-existing model which we can compare the RAS logic against. In effect we are constructing the model to be identical to the logic. Had we not chosen to exclude derivation rules whenever possible for performance and simplicity reasons, we might have been able to compare the more complete logic with a model which would look very much like our current logic, to verify that the additional derivation rules were consistent. Here, however, we have nothing to do.

2.9 Interoperability with PKI

In the real world, the two most widely deployed public key infrastructures (PKI) are x.509 [?] and the PGP web of trust [?, ?, ?]. The PGP web of trust tends to be used in ad-hoc systems where installing a rigid infrastructure is not feasible, while x.509 tends to be used in more centralised environments. The RAS logic could be made to interact with either system, but the kinds of organisations likely to employ an RBAC system are more likely to employ x.509 than PGP, so we will concentrate on that.

x.509 is a standard for public key certificates. It is used in conjunction with the x.500 directory service as well as the simplified LDAP [?] version. It has also been adopted for the SSL/TLS transport level security protocols [?], the S/MIME secure e-mail standard [?] as well as a variety of other standards and special purpose applications. It is therefore important that any access control system be able to interoperate with PKI systems based on x.509.

2.9.1 Basic Case

In its simplest form an x.509 certificate is a statement issued by a certification authority (P) and signed with its key (K_p), claiming that a particular public key (K_Q) belongs to a particular principal (Q) and that it is valid for a particular period t . This corresponds neatly with a statement in the RAS logic of the form $\{ Q \text{ Pub}_t K_q \}_{K_p}$ and is in fact the motivation for the format of the key assignment statements.

The other form of x.509 certificate is a CA certificate. It is also issued by a CA (P) and signed with its key (K_p) delegates to another CA (Q) the authority to issue certificates using a particular key (K_q) in a particular period (t). A number of restrictions may be put on Q 's ability to issue certificates, but the only one which translates into the RAS logic is the permitted subtree form of Name Constraints. This becomes the domain (d) in the statement $\{ Q \text{ Pub}_t [K_q \triangleright d] \}_{K_p}$.

2.9.2 Restriction Mismatches

Not all x.509 certificates can be translated directly into statements in the RAS logic. Certificates may contain restrictions on the way a particular key is used, the length of certification chains springing from a particular certificate, subtle interactions between string encoding mechanisms and a lot of complexity we do not want to introduce into the logic.

Similarly we may want to restrict the power granted by a certificate, particularly CA certificates, when they are accepted into a particular RAS server's set of current statements. For example, we will in most cases want to introduce a validity domain other than world so that the statement can become part of a proof tree and also so that we can accept the CA certificate of a potentially hostile organisation with the caveat that they can only issue certificates for their own employees.

2.9.3 x.509 Certificates as Signatures

The approach adopted is to approximate the meaning of the certificate as shown above and then shrink either the period (t) or (where applicable) domain (d). The certificate is stored as a signature on the statement which is then counter-signed by the RAS server and entered into the database of current statements.

The algorithm for building proof-trees will use the interpreted statement as the basis for its search for possible continuations of a proof-tree, allowing it to make full use of the underlying database. However, when verifying the signature on a statement, it is considered in relation to other statements in the proof tree, which is sufficient to build an x.509 certification path and allows us to apply the full x.509 rules to verify the validity of the path. Failure to verify the path is treated as a failed signature verification and the proof algorithm will discard this proof-branch and search for the next likely one.

While this approach is not perfect, it allows us to maintain the simplicity of the logic, ensure that invalid certificates are not accepted and, since most logical interpretations will correspond fairly closely to the certificate, does not appreciably slow down the proof generator.

3 Prover

The core of the RAS system is the proof generator embedded in the RAS server. The server accepts simple hypotheses encoded in the RAS logic and attempts to prove their correctness using the prover described below. We will give pseudo-code for a non-caching version of the algorithm employed by the prover. The caching mechanisms employed and their effects will be discussed and their effect on the running time.

3.1 Composite Derivation Rules

The derivation rules as presented in section ?? are sufficient to capture the semantics of statements made in the RAS logic. In fact they are the definition of these semantics. However, as presented, the rules very often require us to apply both a DEL-* or CA-* rule and a WEAK-* rule in succession. While this is fine for reasoning about the properties of the logic and for convincing ourselves that the rules are reasonable, it does make implementing and reasoning about provers and verifiers for the logic more cumbersome by introducing an unnecessary intermediate step between the application of e.g. a DEL-* or CA-* rule and the WEAK-* rule which almost certainly follows it. We will therefore extend the logic to include several composite derivation rules which can be seen to be equivalent to combinations of rules already present in the logic. When constructing formal proofs in the following sections, we will then allow for the use of these composite rules along-side the basic ones.

Table ?? lists the combination rules and from which basic rules they are constructed. We will show a few examples of the partial proof-trees in the original logic which correspond to an application of a combination rule. The reader should by now be familiar with how this extends to the remaining cases.

A combination of DEL-MAY and WEAK-MAY gives the WEAK-DEL-MAY rule:

$$\frac{\frac{P \mapsto_{t_1} [r_1 \triangleright d] ; P \text{ Pub}_{t_1} K ; \{ Q \mapsto_{t_2} [r_1 \triangleright d] \}_K}{Q \mapsto_{t_1 \cap t_2} [r_1 \triangleright d]} ; r_2 \leq_{t_3} r_1 ; d_2 \leq d_1 ; t_3 \leq t_1 \cap t_2}{Q \mapsto_{t_3} [r_2 \triangleright d_2]} \quad (\text{WEAK-DEL-MAY})$$

A combination of DEL-DEL and WEAK-DEL would yield a similar WEAK-DEL-DEL rule. A version of the WEAK-DEL-MAY rule for dealing with revoked delegations is the combination of DEL-MAY \dagger and WEAK-MAY:

$$\frac{\frac{P \mapsto_{t_1} [r \triangleright d] ; P \text{ Pub}_{t_1} K_p ; \{ \{ R \mapsto_{t_2} [r \triangleright d] \}_{K_p \dagger t_3} \}_{K_q}}{Q \mapsto_{t_3} [r \triangleright d] ; Q \text{ Pub}_{t_3} K_q}{R \mapsto_{t_1 \cap t_2 \dagger t_3} [r \triangleright d]} ; r_2 \leq_{t_2} r_1 ; d_2 \leq d_1}{R \mapsto_{t_4} [r_2 \triangleright d_2]} ; t_4 \leq t_1 \cap t_2 \dagger t_3 \quad (\text{WEAK-DEL-MAY})$$

We then combine the key-issuing rules in a similar way and finally make a role-ordering rule more suitable for building chains of role-ordering statements and another version (not shown here) for revoked role ordering statements:

$$\frac{\frac{P \mapsto_{t_1} [\text{rm} \triangleright \text{world}] ; P \text{ Pub}_{t_1} K ; \{ r_2 \leq_{t_1} r_1 \}_K}{r_2 \leq_{t_1 \dots \omega} r_1} ; t_3 \leq_{t_1} t_2 ; r_2 \leq_{t_2} r_3 ; t_3 \leq_{t_2} t_1}{r_1 \leq_{t_3} r_3} \quad (\text{CHAIN-ORDER})$$

3.2 Algorithm

We now give the algorithm for proving arbitrary statements in the RAS logic. The algorithm makes references to the sets A and S of axioms and current statements that were introduced in section ??, which are here assumed to finite. It also introduces the set B which is carried through all recursive calls and contains the statements which we are attempting to prove in all calls to any component of the algorithm or, differently stated, any statement which will be the result of a derivation closer to the root of the proof-tree if it is completed. This is used in a simple loop-detection algorithm to avoid infinite recursions as noted in section ??.

Combination Rule	Constituent Rules
WEAK-DEL-MAY	WEAK-MAY, DEL-MAY
WEAK-DEL-MAY†	WEAK-MAY, DEL-MAY†
WEAK-DEL-DEL	WEAK-DEL, DEL-DEL
WEAK-DEL-DEL†	WEAK-DEL, DEL-DEL†
WEAK-CA-KEY	WEAK-KEY, CA-KEY
WEAK-CA-KEY†	WEAK-KEY, CA-KEY†
WEAK-CA-CA	WEAK-CA, CA-CA
WEAK-CA-CA†	WEAK-CA, CA-CA†
CHAIN-ORDER	WEAK-ORDER, ORDER, TRANSITIVITY
CHAIN-ORDER†	WEAK-ORDER, ORDER†, TRANSITIVITY

Table 8: Combination Derivation Rules

The algorithm consists of two mutually recursive algorithms for proving that a principal has a particular capability (alg. ??) and that roles have a certain relative order (alg. ??), which call on a third recursive algorithm for proving that a principal owns a particular public key (alg. ??).

We will start by considering a somewhat simplified version of the algorithm, which does not perform any caching of intermediate results and which does not take revoked statements into account and then introduce these extensions in sections ?? and ??.

Notation We will give the algorithm using fairly straight forward pseudo-code, but the binding rules used are worth expanding upon. The use of these binding rules could be avoided by using existential quantifiers, but this would severely reduce the readability of the code.

Each algorithm starts with a heading of the form “**Algorithm N** AlgorithmName(parameter-list)”, which gives the number by which the algorithm is referenced, the name by which it is called and enumerates its parameters. The following two lines will be of the form “**Require:** predicate” and “**Ensure:** predicate”. giving respectively pre- and post-conditions for the algorithm. These predicates may also serve to bind a number of names to the parameters to a RAS statement which is passed as a parameter to the algorithm. For example, in Algorithm ??, the names P , t , K_p and d are all bound to the parameters of the statement s passed to the algorithm.

Also, a construct of the form **for all** $x \in \{ \text{parametrised object} \mid \text{predicate on parameters} \}$ will bind the parameters inside the set comprehension to the x which is taken to be in the set. This is used in Algorithm ?? to bind in line 9 the names t' , P , d' and K_q which are used in line 10.

We also assume that a signed statement of the form $\{s\}_K$ will contain hints to allow us to look up the principal expected to own the key K , even though this is not explicit in the notation. This allows us for example to conjure up the principal Q in line 15 of Algorithm ?? without any apparent reason. See the `certificate`, `delegation` and `label_act` tables in appendix ?? for a concrete implementation.

ProveKeyOwnership Algorithm ?? is straight-forward. It accepts a statement about key ownership as input and returns either the empty set or a proof of the statement.

It recursively attempts to build a chain of public-key certificates in accordance with the WEAK-CA-KEY and WEAK-CA-CA rules until it can terminate the chain with an axiom which implies the validity of the last certificate in the chain. The chain (along with side-branches) then becomes a proof of the validity of the postulated key ownership. There are two flavours of this algorithm, the one given here for CA certificates and a slightly simpler version for normal key-ownership⁸.

If statements 6-7 in the algorithm are reached, then the hypothesis is the direct result of an axiom and the trivial proof $\{s \mapsto \{axiom\}\}$ is returned. Otherwise, each certificate with P as the subject and a large enough validity period is examined, as long as it hasn't been considered before in the chain that led to this recursive call. We let ko be the hypothesis that the issuer of the certificate is authorised to issue it

⁸The version for normal key ownership has the “Requires” statement and statements 5,9 and 10 changed to refer to normal key ownership and to not refer to domains at all.

and if this hypothesis can be proven, we reach statement 13-15 where we combine the previous proof for the issuer being a sufficiently trusted Certification Authority with the certificate yielding a proof for the hypothesis. The correctness of this follows from the recursive proofs theorem (thm. ?? in combination with the WEAK-CA-CA rule.

If the hypothesis *ko* can't be proven, then the next certificate is examined until the available certificates are exhausted in which case we reach statement 18 and return an empty proof.

Algorithm 1 ProveKeyOwnership(s, B)

Require: $s = P \text{Pub}_t [K_p \triangleright d] \wedge B \subseteq \Sigma$

Ensure: $(A, S \vdash s \Rightarrow A, S \vdash_p s) \wedge (\neg A, S \vdash a \Rightarrow p = \emptyset)$

```

1: if  $s \in B$  then
2:    $p \leftarrow \emptyset$ 
3:   exit
4: end if
5: for all  $axiom \in \{ P \text{Pub}_{t'} [K_p \triangleright d'] \in A \mid (t, d) \leq (t', d') \}$  do
6:    $p \leftarrow \{s \mapsto \{axiom\}\}$ 
7:   exit
8: end for
9: for all  $cert \in \{ \{ P \text{Pub}_{t'} [K_p \triangleright d'] \}_{K_q} \in S \mid (t, d) \leq (t', d') \}$  do
10:   $ko \leftarrow Q \text{Pub}_{t'} [K_q \triangleright P \vee d']$ 
11:   $koProof \leftarrow \text{ProveKeyOwnership}(ko, B \cup \{s\})$ 
12:  if  $koProof \neq \emptyset$  then
13:     $p \leftarrow \{s \mapsto \{cert, ko\}\}$ 
14:     $p \leftarrow p \oplus koProof$ 
15:    exit
16:  end if
17: end for
18:  $p \leftarrow \emptyset$ 

```

ProveCapability Algorithm ?? attempts to prove a given hypothesis about a principal having a particular permission or holding a particular role in a given domain and during a given period⁹. This is done by first examining known axioms and if one is found which directly implies the hypothesis, then a minimal proof is returned. Otherwise each delegation which implies the hypothesis is examined in an arbitrary order and an attempt is made to prove the premises needed for each of these delegations to be effective. If none of the delegations yield a proof, then the empty proof is returned signifying failure.

We first iterate over the axioms which have the correct subject principal and a large enough validity period and domain. We will reach statement 6 whenever we find one. There we will attempt to prove that the role in the axiom implies the one in the hypothesis which is done by a mutually recursive call to Algorithm ?. If we reach lines 9-11, then this will have succeeded for some axiom and we return the proof obtained by adding a mapping from the hypothesis to the axiom and the root of the ordering proof to the ordering proof returned by the recursive calls. The correctness of this follows from the recursive proofs theorem (thm. ??) in combination with the WEAK-MAY rule.

If none of the axioms lead to a proof, we will examine the delegations with the appropriate subject principal and a large enough validity period and domain. Each of these will require 3 premises in order to be valid, given by *del*, *ko* and *ord*. We attempt to prove each of these by recursive calls to algorithms ?, ? and ?. If all premises hold, the three proofs, along with a mapping from the hypothesis to the delegation and the 3 premises are combined into a proof of the hypothesis in statements 22-24. The correctness of this follows from the recursive proofs theorem (thm. ??) in combination with the WEAK-DEL-MAY rule.

⁹This is the version for holding a role. The version for being able to delegate a role is similar with only the "Requires" statement and statements 1 and 11 changed to use the double arrow.

Algorithm 2 ProveCapability(s, B)

Require: $s \in \{P \mapsto_t [r \triangleright d], P \mapsto_t [r \triangleright d]\} \wedge B \subseteq \Sigma$
Ensure: $(A, S \vdash s \Rightarrow A, S \vdash_p s) \wedge (\neg A, S \vdash a \Rightarrow p = \emptyset)$

- 1: **if** $s \in B$ **then**
- 2: $p \leftarrow \emptyset$
- 3: **exit**
- 4: **end if**
- 5: **for all** $axiom \in \{P \mapsto_{t'} [r' \triangleright d'] \in A \mid (t, d) \leq_t (t', d')\}$ **do**
- 6: $ord \leftarrow r \leq_t r'$
- 7: $ordProof \leftarrow \text{ProveRoleOrder}(ord, B \cup \{s\})$
- 8: **if** $ordProof \neq \emptyset$ **then**
- 9: $p \leftarrow \{s \mapsto \{axiom, ord\}\}$
- 10: $p \leftarrow p \oplus ordProof$
- 11: **exit**
- 12: **end if**
- 13: **end for**
- 14: **for all** $del \in \{\{P \mapsto_{t'} [r' \triangleright d']\}_K \in S \mid (t, d) \leq_t (t', d')\}$ **do**
- 15: $cap \leftarrow Q \mapsto_{t'} [r' \triangleright d']$
- 16: $capProof \leftarrow \text{ProveCapability}(cap, B \cup \{s\})$
- 17: $ko \leftarrow Q \text{Pub}_{\min t'} K$
- 18: $koProof \leftarrow \text{ProveKeyOwnership}(ko, B \cup \{s\})$
- 19: $ord \leftarrow r \leq_t r'$
- 20: $ordProof \leftarrow \text{ProveRoleOrder}(ord, B \cup \{s\})$
- 21: **if** $capProof, koProof, ordProof \neq \emptyset$ **then**
- 22: $p \leftarrow \{s \mapsto \{del, cap, ko, ord\}\}$
- 23: $p \leftarrow p \oplus capProof \oplus koProof \oplus ordProof$
- 24: **exit**
- 25: **end if**
- 26: **end for**
- 27: $p \leftarrow \emptyset$

ProveRoleOrder The final part is algorithm ??, which attempts to prove a hypothesis about the ordering of roles. This is done by building a chain of signed role-ordering statements of the form $r_1 \leq_t r_2, r_2 \leq_t r_3, \dots, r_{n-1} \leq_t r_n$ when attempting to prove that $r_1 \leq_t r_n$. However, the generated proof must also show that the issuer of each of these role-ordering statements had the authority to issue it and that the signature was made with a valid key.

We don't expect axioms about role-ordering, so none are examined. Instead we iterate over the current statements about roles which dominate r within the given period, assuming they haven't been used yet on the proof-branch we are examining. We must always prove the premises ko and rm and if this is successful and if the signed ordering statement was for exactly the roles we have in mind, we will reach statements 12-14 where we combine these proofs with a mapping from the hypothesis to these two premises along with $r' = r'$ (which serves as an axiom to allow termination) using the recursive proofs theorem (thm. ??) and the CHAIN-ORDER rule.

If the signed role ordering statement we are attempting to prove does not match the hypothesis, then we will need to apply the TRANSITIVITY rule for role ordering and must first prove the $ordSub$ premise. If this succeeds we reach statements 19-21 where we again use the recursive proofs theorem (thm. ??) and the CHAIN-ORDER rule to build a proof of the hypothesis.

If none of the role-ordering rules lead to a proof, then we will reach statement 25 where we return the empty proof.

Revocations The pseudo-code given above completely ignores revoked statements for readability. However, the additions needed to handle revoked delegations, key certificates and role activations are straight-

Algorithm 3 ProveRoleOrder(s, B)**Require:** $s = r \leq_t r' \wedge B \subseteq \Sigma$ **Ensure:** $(A, S \vdash s \Rightarrow A, S \vdash_p s) \wedge (\neg A, S \vdash a \Rightarrow p = \emptyset)$

```

1: if  $s \in B$  then
2:    $p \leftarrow \emptyset$ 
3:   exit
4: end if
5: for all  $ord \in \{ \{ r \leq_{t'} r'' \}_K \in S \mid t \leq t' \}$  do
6:    $ko \leftarrow P \text{Pub}_{\min t'} K$ 
7:    $koProof \leftarrow \text{ProveKeyOwnership}(ko, B \cup \{s\})$ 
8:    $rm \leftarrow P \mapsto_{\min t'} [rm \triangleright \text{world}]$ 
9:    $rmProof \leftarrow \text{ProveCapability}(rm, B \cup \{s\})$ 
10:  if  $koProof, rmProof \neq \emptyset$  then
11:    if  $r'' = r'$  then
12:       $p \leftarrow \{s \mapsto \{ko, rm, r' = r'\}\}$ 
13:       $p \leftarrow p \oplus koProof \oplus rmProof$ 
14:      exit
15:    end if
16:     $ordSub \leftarrow r'' \leq_t r'$ 
17:     $ordSubProof \leftarrow \text{ProveRoleOrder}(ordSub, B \cup \{s\})$ 
18:    if  $ordSubProof \neq \emptyset$  then
19:       $p \leftarrow \{s \mapsto \{ko, rm, ordSub\}\}$ 
20:       $p \leftarrow p \oplus koProof \oplus rmProof \oplus ordSubProof$ 
21:      exit
22:    end if
23:  end if
24: end for
25:  $p \leftarrow \emptyset$ 

```

forward. A loop for examining revoked statements is added after the one for non-revoked statements and sufficient sub-proofs are generated to apply the revoked version of the derivation rule applied in the previous loop. As an example a second version of ProveKeyOwnership (alg. ??) is given.

3.3 Correctness, Completeness and Termination

The three properties we are most interested in when dealing with algorithms are correctness, completeness and termination, followed by various flavour of time-complexity.

In our setting, correctness is a security property, since returning an incorrect proof may lead to “proving” an incorrect hypothesis, which may (in case the verifier were disabled) allow principals access they should not have. Completeness and termination, however, are “merely” liveness properties. The possible effects of these properties lacking are that principals are denied access they should have, which is not considered as harmful as unlawful access. Also, confidence in the liveness of the system may be developed through use and any errors should be detectable, while security problems resulting from an incorrect prover may be hard to detect. Correctness is therefore of much higher importance in this setting than is completeness or termination.

3.3.1 Correctness

Correctness means that any result returned by the algorithm will be correct, i.e. a proof constructed by this algorithm is a proof of the original hypothesis in the sense of definition ???. This is shown by induction following the recursive calls of the algorithms and as the inductive step has already been performed in the description of each algorithm, the proof is complete.

Algorithm 4 ProveKeyOwnership(s, B) (with revocations)**Require:** $s = P \text{Pub}_t [K_p \triangleright d] \wedge B \subseteq \Sigma$ **Ensure:** $(A, S \vdash s \Rightarrow A, S \vdash_p s) \wedge (\neg A, S \vdash a \Rightarrow p = \emptyset)$

```

1: if  $s \in B$  then
2:    $p \leftarrow \emptyset$ 
3:   exit
4: end if
5: for all  $axiom \in \{ P \text{Pub}_{t'} [K_p \triangleright d'] \in A \mid (t, d) \leq (t', d') \}$  do
6:    $p \leftarrow \{s \mapsto \{axiom\}\}$ 
7:   exit
8: end for
9: for all  $cert \in \{ \{ P \text{Pub}_{t'} [K_p \triangleright d'] \}_{K_q} \in S \mid (t, d) \leq (t', d') \}$  do
10:   $ko \leftarrow Q \text{Pub}_{t'} [K_q \triangleright P \vee d']$ 
11:   $koProof \leftarrow \text{ProveKeyOwnership}(ko, B \cup \{s\})$ 
12:  if  $koProof \neq \emptyset$  then
13:     $p \leftarrow \{s \mapsto \{cert, ko\}\}$ 
14:     $p \leftarrow p \oplus koProof$ 
15:    exit
16:  end if
17: end for
18: for all  $cert \in \{ \{ \{ P \text{Pub}_{t'} [K_p \triangleright d'] \}_{K_q \dagger t''} \}_{K_r} \in S \mid (t, d) \leq (t' \dagger t'', d') \}$  do
19:   $ko \leftarrow Q \text{Pub}_{t'} [K_q \triangleright P \vee d']$ 
20:   $koProof \leftarrow \text{ProveKeyOwnership}(ko, B \cup \{s\})$ 
21:   $koRev \leftarrow R \text{Pub}_{t''} [K_r \triangleright P \vee d']$ 
22:   $koRevProof \leftarrow \text{ProveKeyOwnership}(koRev, B \cup \{s\})$ 
23:  if  $koProof, koRevProof \neq \emptyset$  then
24:     $p \leftarrow \{s \mapsto \{cert, ko, koRev\}\}$ 
25:     $p \leftarrow p \oplus koProof \oplus koRevProof$ 
26:    exit
27:  end if
28: end for
29:  $p \leftarrow \emptyset$ 

```

3.3.2 Completeness

p-Normalisation Completeness of the prover requires that it be able to produce a proof as per section ?? for any statement which is derivable in the logic. We note that any derivable statement is, by definition, derivable by a finite number of applications of the derivation rules of section ??, augmented in our case by the composite rules of section ?. We will define a normal form for such derivations and proofs, but before we can do this, we must note a few facts:

- There are really two kinds of derivation rules in the original RAS-logic. There are derivation rules such as DEL-DEL, CA-KEY and ORDER \dagger where signed statements are introduced, i.e. delegations, key certificates or label activations.
- Two applications of weakening rules in succession must be two applications of the same rule, since the conclusion of a weakening rule is not compatible with any of the postulates of another weakening rule. They can therefore be replaced by a single application of the rule.
- Any use of a signed statement, such as a delegation, key certificate or role ordering, whether or not it is followed by a weakening, and whether or not it has been revoked, may be replaced by a single use of one of the composite derivation rules.
- The only places a weakening rule may be applied is following another weakening rule, following a rule introducing the appropriate signed statement or following an axiom.

Since the composite statements all include the appropriate weakening rule, we can now convert any derivation of a statement in the RAS-logic into a derivation which consists solely of applications of the composite derivation rules whenever signed statements are introduced and an axiom followed by a weakening whenever an axiom is introduced.

We will apply one final transformation to the derivations before continuing. Consider a branch of the derivation tree from root to leaf, hypothesis to axiom. Since any derivation will be finite, then if the same statement is ever twice the result of a derivation rule, then the portion of the branch between the two occurrences of the same result as well as any side-branches on this portion of the branch we are considering can be removed and we will still have a derivation with the same ultimate result.

We can now finally restrict ourselves to statements with derivations of the form described above, using only the composite derivation rules, axioms followed by weakening, and with no repeated results along any branches of the derivation tree. Derivations and formal proofs in this form will be said to be in *p-normal form*, since this is the form of the proofs generated by the prover.

Finally, it is clear that a sub-derivation of a derivation in p-normal form is itself in p-normal form.

p-Completeness Given a statement s along with a derivation of s in p-normal form, we must now show that the prover is capable of generating a proof of s .

We will proceed by induction on the structure of the proof tree. If the derivation of s is an introduction of an axiom followed by a weakening, it is easy to verify by inspection that a proof having the same form will be produced by the appropriate component of the prover.

Next, suppose the derivation of s is the application of a composite derivation rule the postulates of which are derived in any number of sub-derivations. We again prove by inspection that the appropriate prover component will return a proof combining the appropriate composite derivation rule and the proofs generated for the sub-derivations. Since the sub-derivations are themselves in p-normal form the prover is by the inductive hypothesis able to generate proofs for them and the induction step is complete.

3.3.3 Termination

Termination follows easily from the assumption that the set S is finite. Since each recursive call made by the prover algorithm adds an element to the B set, it is sufficient to show that there is a finite number of possible elements which may be added to B .

An element in B is a statement which may be the result of a derivation rule. There are 15 types of such statements, 5 types in Σ_0 given in section ?? and 10 in Σ_s formed by signing or revoking statements from Σ_0 as shown in section ?. Each of these has a finite number of parameters of the sorts introduced in ?. The values that are given to these parameters will always be derived from values found in the statements in A and S and if we have a finite number of values which may be used, we will have a finite number of candidates for inclusion in B whereby B is finite and we will have no infinite recursions.

This, along with the fact that all the loops in the algorithm loop over a subset of the statements in S or axioms in A , both of which are finite, implies termination.

We now only need to show that the values which may be used as parameters to statements for inclusion in B are finite as a consequence of A and S being finite.

Periods The result of any operation on periods defined in the RAS-logic returns an interval between two of the end-points of the operands. Thus the number of end-points of periods which might be used is the finite set of end-points of statements in A and S and the set of periods which can be formed from these is also finite.

Public keys Only keys specifically mentioned in a statement in A or S will ever be used.

Domains Only domains which are formed from maplets found in domains or principals mentioned in statements in A or S will be used. Since each map is finite and the set of maps is finite, the set of maplets mentioned in A or S is finite. Any maps which may be used are subsets of this set of maplets and so there is a finite number of these.

Principals Only principals specifically mentioned in a statement in A or S will ever be used.

Roles Only roles specifically mentioned in a statement in A or S will ever be used.

3.4 Implications for RAS-Logic

Previously, following theorem ?? in section ??, we noted that given a formal proof of a statement it were clearly derivable. By giving the prover algorithm we have shown that for any statement which is derivable, a formal proof exists and that this proof can be computed. In other words we have shown that formal provability is equivalent to derivability and that the logic is decidable.

3.5 Caching Mechanisms

The algorithm described above is functionally equivalent to the one actually implemented in the RAS system. However, going through the entire proving process to answer every access control query would likely be too much load even for small installations. This is to some extent alleviated by the caching strategies described in the following sections.

3.5.1 Caching Proof-Trees

Fortunately, access-control queries tend to be clustered in a few ways. First, principals who are currently on-line with a service which uses the RAS server for access control are more likely to be the subject of queries. Secondly, the principals who hold very high-level roles and have issued delegations which appear in many proof-trees are likely to be the subject of many queries and finally there will be more queries about the current state of the system than about an arbitrary past state.

This leads to a fairly obvious caching strategy where proof-trees which have been generated in response to previous queries are cached. However, caching only the complete proof-trees would only give us a small part of the potential improvement in speed, since many proof-trees will have common sub-trees. For example the proof of validity of the public key certificate of a local certification authority may be a sub-tree in most queries, but very rarely will it be requested directly.

We therefore modify the prover to have two additional global variables, the *yes-cache* and the *no-cache*. At the beginning of each call to one of its mutually recursive portions, the prover checks if a statement is present in the yes-cache which implies the hypothesis, and if so, the previous proof is returned. If this fails and a proof is generated normally, it is added to the yes-cache before returning. Because of the recursive structure of the algorithm, the complete proof-tree will be added to the cache in the top-level call and each sub-tree will be added in a recursive call to one of the component algorithms, ready for inclusion in other proof-trees later. This is technically a case of dynamic programming, but as we don't expect many repeated sub-proofs in generated proofs, the main benefit appears only when repeated calls are made to the prover.

It is not only sub-proof-trees that are common between various runs of the prover. Often, the prover will attempt to incorporate a public key certificate or a delegation in a proof tree, for example when a likely certificate is found on line 9 in algorithm ??, only to find that it is not valid or that some prerequisite can't be proven. This may be repeated on several calls to the prover if there is a certificate or delegation in the set of current statements which often looks promising to the prover, but which never leads to a valid proof. To avoid this happening, after looking for a proof in the yes-cache, the prover looks for a statement in the no-cache which is implied by the hypothesis. If the hypothesis could be proven, then the statement in the no-cache could have been proven as a consequence of the hypothesis, so no proof of the hypothesis will be possible. We therefore immediately return the empty proof which allows us to avoid a potentially large search through the prover's execution space.¹⁰

This way, any sub-proof-trees which have been generated before (and not flushed from the cache) will be re-used and sub-proofs which have been shown to be hopeless are not repeatedly attempted. It is hoped that this will greatly speed up the execution of the program on average, but examining this requires both a statistical model of the usage of the RAS server, which we do not have, and sophisticated analysis which we will not attempt.

¹⁰This argument shows the need to flush the no-cache when ever new information is gained by the RAS server, but this is deemed to happen very infrequently compared to queries and not be a significant performance problem. In fact, all caches are flushed at any change to the state of the RAS server (other than the clock ticking), to be certain to avoid stale caches confusing the prover.

3.5.2 Caching Role-Graphs

We have now ensured that we don't need to attempt sub-proofs more often than is needed. We then turn to measures which speed up the proving process when it must be performed.

In the execution of ProveCapability (alg. ??), at lines 1 and 9 we loop over axioms and delegations respectively. In the latter case, this is eventually translated into an SQL query for a database back-end. This query will return a number of delegations and we will then attempt to prove for each delegation returned that the role in the delegation dominates the one in the hypothesis. It would be much more efficient if knowledge of the role hierarchy could be used when constructing the SQL query, which would reduce the load on the database and reduce the number of role ordering proofs we need to attempt.

Unfortunately, it is not possible to construct a definitive role hierarchy from role ordering statements before using it to search for delegations in the database, since the ORDER rule of the RAS-logic, which is used to accept a role ordering statement, requires a proof that the issuer of the statement has a particular capability which in turn requires us to search for delegations in the database, causing an infinite recursion.

The approach adopted is to construct the hierarchy from *potential* role ordering statements. Because the only way that a pair of roles may have a particular order is if there exists one or more current role orderings statements to this effect, this hierarchy may lead to too many delegations being considered, but not to too few delegations being considered.

Finally, we extend the ProveRoleOrder algorithm (alg. ??) so that when a role-order hypothesis fails to be proven, we remove the corresponding edges from the potential role-ordering graph, which will eventually lead to all cached role-graphs being perfectly correct.

4 Verifier

In this section we give an algorithm to verify proofs generated by the prover from section ??, which must be as defined in section ?. The verification algorithm is independent of the prover and both the RAS logic and the definition of formal proofs are designed to make verification as simple as possible. This allows developers of high-security systems who wish to use the RAS to verify the correctness of the verifier for themselves and then have their software verify each proof returned by the RAS server. This results in very high confidence in positive results of access control queries without needing to verify the entire system.

4.1 Algorithm

The algorithm is given in notation similar to that used for the prover. It also refers to the finite sets $A \subseteq \Sigma_0$ of axioms and $S \subseteq \Sigma_s$ of current statement from the environment.

Algorithm 5 VerifyProof(s, p, B)

Require: $s \in \Sigma_0 \wedge p : PROOF \wedge B \subseteq \Sigma_0$

Ensure: $A, S \vdash_p s \Leftrightarrow OK = TRUE$

```

1: if  $s \in B$  then
2:    $OK = FALSE$ 
3: exit
4: end if
5: if  $s \in A$  then
6:    $OK = TRUE$ 
7: exit
8: end if
9: if  $s \notin \text{dom } p \vee \neg (p(s) \rightarrow s)$  then
10:   $OK = FALSE$ 
11: exit
12: end if
13: for all  $s' \in p(s)$  do
14:  if  $\neg \text{VerifyProof}(s', p, B \cup \{s\}) = TRUE$  then
15:     $OK = FALSE$ 
16:    exit
17:  end if
18: end for
19:  $OK = TRUE$ 

```

4.2 Correctness and Termination

As with the prover, the main result we are interested in is that the verifier never returns *TRUE* for a proof which is not correct, which is our main security property. Termination is again a mere liveness property.

4.2.1 Termination

Termination is easy to show as $B \subseteq \text{dom } p$, $\text{dom } p$ is finite and in each recursive call a statement $s \notin B$ is added to B . We therefore have no infinite recursions. We also have that $p(s)$ is finite for each s and so the only loop in the algorithm is finite.

4.2.2 Correctness

We show correctness by induction over the execution of the algorithm. We will in fact show that the verifier returns with $OK = TRUE$ if and only if the proof p is a correct proof of s .

The set B is, as with the prover, used for loop detection. It is the set of statements from Σ_0 which are the result of a sub-proof being verified at some outer level of recursion. Therefore, if the algorithm exits at line 3 it is because p contains a loop and is not correct.

If the algorithm exits at line 7, however, it is because the statement s we are verifying the proof for is an axiom and therefore $A, S \vdash_p s$ by the AXIOM rule.

If the algorithm does not exit at line 7, the only way that the proof can be valid is by the DERIVE rule. The DERIVE rule requires that $s \in \text{dom } p$, $A, S \vdash_p p(s)$ and $p(s) \rightarrow s$. If the algorithm terminates on line 11, then either the first or the third of these conditions is not met and the proof is not valid. If it enters the loop at lines 13-18, however, then these two conditions are met and the proof is valid if and only if for each statement $s' \in p(s)$ we have that $A, S \vdash_p s'$. If the algorithm exits at line 16, then this condition is not met for some $s' \in p(s)$, while if the loop terminates normally and the algorithm exits after line 19, all the conditions of the DERIVE rule are met and the proof is correct.

5 RAS Server

5.1 Function

The RAS model depends on there being a central server holding the set of current statements. In addition to simply being a repository for statements, the server is responsible for disposing of statements which have been revoked, replacing them with the corresponding revocation statements. It may also refuse to accept statements which violate explicit policies beyond what is implicit in the RAS logic.

We will give a formal specification of the RAS server, followed by proofs of several properties we wish to hold true of the server.

5.1.1 Enforced Policies

The policies enforced by the server as specified here concern themselves with the role structure and maintaining the immutability of past state:

- The role hierarchy is forced to be weakly acyclic and single-rooted, with the exception of the role-manager role which is disjoint from the normal hierarchy.
- A distinction between roles and permissions is introduced, and roles will never be dominated by permissions.
- Permissions will never be delegated to principals, only normal roles.
- No delegation, label activation or key certificate is accepted if its validity period extends into the past.
- No revocation is accepted if the instant of revocation is in the past or if it would in any other way revoke privileges held in the past.

5.2 Extensions to the Z Notation

The RAS server will need to manipulate statements made in the RAS logic and base its operations on these statements or what can be derived from them. Rather than re-defining the logic along with its sorts using Z-notation, we will be using a higher order logic within the Z schemas and use the RAS logic as the lower order logic. The RAS logic therefore becomes part of the apparatus of the Z-method rather than being part of the specification.

In many places we will write predicates such as $A, S \vdash \sigma \Rightarrow p$ for some predicate p . This should be read as “if σ is derivable in the RAS logic, then p holds”. In other words, $A, S \vdash \sigma$ is a predicate in the RAS logic, the value of which is used in the higher order logic in the usual way. The prover algorithm given in section ?? allows for an implementation of operations such as these and is used frequently in implementing the RAS server, although not explicitly mentioned in the Z schemas.

5.2.1 Additional Types

The sorts of the logic will correspond to the types *INSTANT*, *PERIOD*, *LABEL*, *KEY*, *NAME*, *DOMAIN* and *PRINCIPAL*. We have $INSTANT = \alpha \dots \omega$ and $PERIOD = \{ a \dots b \mid a, b \in INSTANT \}$, while *LABEL* and *KEY* have no structure.

The following Z-code describes the *DOMAIN* and *PRINCIPAL* types, which are considered sub-types of the *NAME* type.

```
[COMP_NAME]
[COMP_VALUE]
CN : COMP_NAME
```

$$\begin{array}{l}
DNAME \hat{=} COMP_NAME \rightarrow COMP_VALUE \\
DOMAIN \hat{=} \mathbb{P} DNAME \\
PRINCIPAL \hat{=} \mathbb{P} DNAME \\
\hline
DOMAIN = \{ dn : DNAME \mid CN \notin \text{dom } dn \} \\
PRINCIPAL = \{ dn : DNAME \mid CN \in \text{dom } dn \}
\end{array}$$

We will also use the symbol Σ for the set of possible statements in the RAS logic and it is the disjoint union of Σ_0 and Σ_s as in section ???. Finally, the type *PROOF* contains all formal proofs as described in section ??? and is a sub-type of $\Sigma_0 \rightarrow \mathbb{P} \Sigma$.

We also define a bottom element “ \perp ” which may be adjoined to any of these sets and which is considered less than or equal to any element where an order is defined. This makes it equal to the α instant in the *INSTANT* type, the empty period in the *PERIOD* type and unlike any element in the *NAME*, *DOMAIN* and *PRINCIPAL* types where no smallest element exists. It is also unlike any element in the *LABEL* and *KEY* types where no order is defined.

5.2.2 Complex Types and Pattern Matching

One problem that faces software developers who wish to use state-based formalisms such as Z or B to model and develop applications is that these formalisms are based on static data types, while developers have become used to more dynamic typing systems such as object orientation. Where it is natural for a programmer in an object oriented language for example to create a class representing a delegation and instantiate it as needed (XXX refer to UML), this is generally not supported by state-based formalisms.

The standard way to proceed is to create complicated networks of functions and relations, so that a class with multiple fields would be represented by a set of identifying values along with a mapping from this set for each attribute of the class. An instance of the class is then an identifying value with corresponding maplets for each attribute.

The alternative employed here is to augment the logic with complex structures before developing the program specification. Rather than having a set of delegation ID’s and 7 mappings to represent delegations (including revocation data), we can now employ sets of $\{ P \mapsto_t [r \triangleright d] \}_K$ and $\{ \{ P \mapsto_{t_i} [r \triangleright d] \}_{K_i} \uparrow t_r \}_{K_r}$ statements, which tends to be much more readable.

We are also going to allow pattern matching on the types introduced by the RAS logic. When writing a statement in the RAS logic it implicitly carries predicates on the type of its parameters. So, for example, the set of all delegations issued to a principal P may be written $\{ P' \mapsto_t [r \triangleright d] K \in S \mid P' = P \}$, but in fact implies $\{ P' \mapsto_t [r \triangleright d] K \in S \mid P' = P \wedge P' : PRINCIPAL \wedge t : PERIOD \wedge r : LABEL \wedge d : DOMAIN \wedge K : KEY \}$. We will also allow the naming of statements taken as parameters to Z schemas, so a parameter given as $s = \{ P \mapsto_t [r \triangleright d] \}_K \in S$ is equivalent to the parameter $s \in S$ with an existential quantifier in the predicate portion of the schema which supplies the parameters to the statement and which encloses the body of the schema as it is written. As an example, these two schemas are equivalent:

$$\begin{array}{l}
\textit{Example} \\
\hline
\Delta Time \\
s = \{ P \mapsto_t [r \triangleright d] \}_K \in S \\
\hline
t \leq \textit{past}
\end{array}$$

$$\begin{array}{l}
\textit{Example} \\
\hline
\Delta Time \\
s \in S \\
\hline
\exists P : PRINCIPAL, t : PERIOD, r : LABEL, d : DOMAIN, K : KEY \bullet \\
s = \{ P \mapsto_t [r \triangleright d] \}_K \\
t \leq \textit{past}
\end{array}$$

5.3 Server Specification

We now finally give a specification of the RAS server using the notation given above and in the previous sections.

5.3.1 State

We begin by introducing the schemas holding the actual state of the RAS server.

\underline{Time} $now : INSTANT$ $past : PERIOD$ $future : PERIOD$
$past = \{ t : INSTANT \mid t \leq now \wedge t \neq now \}$ $future = \{ t : INSTANT \mid now \leq t \wedge now \neq t \}$

$\underline{\Delta Time}$ $Time$ $Time'$
$now \leq now'$

Time is split into three periods: the past, the current instant and the future. The current instant may increase, but will never decrease. It is however possible that the current instant, as measured by the timer in an actual computer system, will not advance during an operation of the server.

\underline{RAS} $Time$ $A : \mathbb{P} \Sigma_0$ $S : \mathbb{P} \Sigma_s$ $perm : \mathbb{P} LABEL$
$perm \subset root^\downarrow$ $root \notin perm$

$\underline{\Delta RAS}$ $\Delta Time$ RAS RAS'
$A' = A$

On the most abstract level the RAS server contains a set A of axioms giving assumed trust and a set S of statements it has seen and accepted into its database. The axioms are a subset of Σ_0 and all confer authority, key ownership or role ordering directly, while the current statements S are a subset of Σ_s and are all statements signed by principals and may be used to construct proofs in the RAS logic.

The server also keeps track of which of the labels active at the current instance are permissions, so that assigning them to principals can be avoided.

5.3.2 Initialisation

$\begin{array}{l} \textit{InitRAS} \\ \textit{RAS}' \\ \hline \exists P_r, P_{ca}, P_{ras} : \textit{PRINCIPAL}; K_r, K_{ca} : \textit{KEY} \bullet \\ A' = \{ \\ \quad P_r \Vdash_{\alpha \dots \omega} [\textit{root} \triangleright \textit{world}], \\ \quad P_{ca} \mathbb{P}\textit{ub}_{\alpha \dots \omega} [K_{ca} \triangleright \textit{world}], \\ \} \\ S' = \emptyset \\ \textit{perm}' = \emptyset \end{array}$
--

We initialise the RAS with a small set of axioms and an empty set of statements. The axioms give P_r the authority to delegate any role in any domain and period and gives P_{ca} the authority to issue any key certificates¹¹.

5.3.3 Label Queries

$\begin{array}{l} \textit{GetLabelActivation} \\ \hline \exists \textit{RAS} \\ t? : \textit{INSTANT} \\ \textit{name}? : \textit{LABEL} \\ \textit{activation}! : \{ \{ \textit{name}? \leq_t R \}_K \in \Sigma_s \mid \top \} \cup \{ \perp \} \\ \hline \textit{candidates} = \{ \{ \textit{name}? \leq_{t'} R \}_K \in S \mid t' \in t? \} \\ \textit{candidates} \neq \emptyset \Rightarrow \textit{activation}! \in \textit{candidates} \\ \textit{candidates} = \emptyset \Rightarrow \textit{activation}! = \perp \end{array}$
--

The *GetLabelActivation* schema accepts an instant and a label and returns the delegation for the label active at the given instant if one exists, but \perp otherwise.

We will show that only one activation for a given label can be active at a given instant, so the above schema is in fact deterministic.

$\begin{array}{l} \textit{ProveLabelOrder} \\ \hline \exists \textit{RAS} \\ s? : \{ r_1? \leq t? r_2 \in \Sigma_s \mid \top \} \\ p! : \textit{PROOF} \\ \hline A, S \vdash s? \Rightarrow A, S \vdash_{p!} s? \\ A, S \not\vdash s? \Rightarrow p! = \emptyset \end{array}$

The *ProveLabelOrder* schema accepts a simple statements that a label r_2 dominates a label r_1 . It returns a proof of this statement if it is derivable, but the empty proof otherwise.

¹¹This corresponds to installing a root CA certificate in traditional PKI systems

5.3.4 Key Certificate Queries

$\begin{array}{l} \textit{ListKeyCertificates} \\ \hline \exists RAS \\ \textit{subjects?} : NAME \\ \textit{domain?} : DOMAIN \cup \{\perp\} \\ \textit{instant?} : INSTANT \\ \textit{certificates!} : \mathbb{P}(\{\{P \text{Pub}_t K_s\}_{K_i} \in \Sigma_s \mid \top\} \cup \{\{P \mathbb{P}\text{ub}_t [K_s \triangleright d]\}_{K_i} \in \Sigma_s \mid \top\}) \\ \hline \textit{domain?} = \perp \Rightarrow \textit{certificates!} = \{ \\ \quad \{P \text{Pub}_t K_s\}_{K_i} \in S \mid \\ \quad P \leq \textit{subjects?} \wedge \textit{instant?} \leq t \\ \quad \} \\ \textit{domain?} \neq \perp \Rightarrow \textit{certificates!} = \{ \\ \quad \{P \mathbb{P}\text{ub}_t [K_s \triangleright d]\}_{K_i} \in S \mid \\ \quad P \leq \textit{subjects?} \wedge \textit{instant?} \leq t \wedge \textit{domain?} \leq d \\ \quad \} \end{array}$

The *ListKeyCertificates* schema accepts a list of criteria on subjects, a list of criteria on domains and an instant. It returns a list of certificates active at the given instant, where the subject fulfils the given criteria (which may be none by passing in the empty set). If $\textit{domain?} = \perp$, then normal key certificates are included, otherwise CA certificates which are valid within the given domain are included.

$\begin{array}{l} \textit{ProveKeyOwnership} \\ \hline \exists RAS \\ \textit{s?} : \{P? \text{Pub}_{t?} K? \mid \top\} \cup \{P? \mathbb{P}\text{ub}_{t?} [K? \triangleright d?] \mid \top\} \\ \textit{p!} : PROOF \\ \hline A, S \vdash \textit{s?} \Rightarrow A, S \vdash_{p!} \textit{s?} \\ A, S \not\vdash \textit{s?} \Rightarrow \textit{p!} = \emptyset \end{array}$

The *ProveKeyOwnership* schema accepts a simple statements that a principal P holds a key K during period t and possibly that K may be used to issue key certificates within a domain d . It returns a proof of this statement if it is derivable, but the empty proof otherwise.

5.3.5 Delegation Queries

<i>ListDelegations</i>
$\exists RAS$ $subjects? : NAME$ $label? : LABEL$ $useUpSet?, useDownSet? : BOOL$ $domain? : DOMAIN \cup \{\perp\}$ $instant? : INSTANT \cup \{\perp\}$ $toAct?, toDel? : BOOL$ $delegations! : \mathbb{P} \Sigma_s$
$labels = \{$ $ l : LABEL \mid$ $ (useUpSet = TRUE \wedge A, S \vdash label? \leq_{instant?} l) \vee$ $ (useDownSet = TRUE \wedge A, S \vdash l \leq_{instant?} label?)$ $\} \cup \{label?\}$ $toAct = TRUE \Rightarrow actdel = \{$ $ \{P \mapsto_t [r \triangleright d]\}_K \mid$ $ P \leq subjects? \wedge instant? \leq t \wedge r \in labels \wedge domain? \leq d$ $\}$ $toAct = FALSE \Rightarrow actdel = \emptyset$ $toDel = TRUE \Rightarrow deldel = \{$ $ \{P \mapsto_t [r \triangleright d]\}_K \mid$ $ P \leq subjects? \wedge instant? \leq t \wedge r \in labels \wedge domain? \leq d$ $\}$ $toDel = FALSE \Rightarrow deldel = \emptyset$ $delegations! = actdel \cup deldel$

The *ListDelegations* schema accepts a number of parameters. It returns a set of delegations, subject to the following criteria:

1. The subject of the delegation fulfils all the criteria set by the *subjects?* parameter. The *subjects?* parameter acts as a domain to which the subject of the delegation must belong, except that it may contain the special *CN* component.
2. A role is acceptable if one of the following is true: The role is equal to *label?*, *useDownSet?* = *TRUE* and *label?* dominates the role, or *useUpSet?* = *TRUE* and the role dominates *label?*,
3. If *domain?* $\neq \perp$, then only those delegations which are active within the given domain are included.
4. Certificates to act in a role are only included if *toAct?* = *TRUE* and administrative certificates are only included if *toDel?* = *TRUE*.

<i>ProveCapability</i>
$\exists RAS$ $s? : \{P? \mapsto_{t?} [r? \triangleright d?] \in \Sigma_s \mid \top\} \cup \{P? \mapsto_{t?} [r? \triangleright d?] \in \Sigma_s \mid \top\}$ $p! : PROOF$
$A, S \vdash s? \Rightarrow A, S \vdash_{p!} s?$ $A, S \not\vdash s? \Rightarrow p! = \emptyset$

The *ProveCapability* schema accepts a simple statements that a principal *P* may act in or administer a role *r* in domain *d* during period *t*. It returns a proof of this statement if it is derivable, but the empty proof otherwise.

5.3.6 Label Updates

<p><i>ActivateLabel</i></p> <p>ΔRAS</p> <p>$s? = \{ r \leq_t R \}_{K} : \Sigma_s$</p> <p>$isPermission? : BOOL$</p> <hr/> <p>$\exists P : PRINCIPAL \bullet$</p> <p>$A, S \vdash P \text{ Pub}_t K$</p> <p>$A, S \vdash P \mapsto_t [rm \triangleright world]$</p> <p>$t = now$</p> <p>$R \neq \emptyset$</p> <p>$\forall r' \in R \bullet A, S \vdash r \leq_{now} root$</p> <p>$\neg \exists \{ r' \leq_{t_i} R' \}_{K_i} \in S \bullet r' = r \wedge t \in t_i \dots \omega$</p> <p>$\neg \exists \{ \{ r' \leq_{t_i} R' \}_{K_i} \dagger t_r \}_{K_r} \in S \bullet r' = r \wedge t \in t_i \dots t_r$</p> <p>$r \neq rm$</p> <p>$isPermission = FALSE \Rightarrow R \cap perm = \emptyset$</p> <p>$S' = S \cup \{s?\}$</p> <p>$isPermission? = TRUE \Rightarrow perm' = perm \cup \{r\}$</p> <p>$isPermission? = FALSE \Rightarrow perm' = perm$</p>

The *ActivateLabel* schema accepts a label activation stating that a role r is smaller than a set of roles R , as well as a flag $isPermission$ deciding whether r should be considered a permission rather than a role.

The operation will block unless there exists a principal P who owns the key used to sign the label activation and who has sufficient privileges to edit the role hierarchy. It is further ensured that the new role takes effect at the current instant, that it has at least one parent given in R and that they are all active roles, that the new role r is not currently active and that if r is a role, then none of the labels in R are permissions.

When all of the above is true, the activation is added to the set of current statements, and depending on the $isPermission$ flag, r may be added to the $perm$ set of permissions.

<p><i>RemoveLabel</i></p> <p>ΔRAS</p> <p>$s? = \{ \{ r \leq_{t_1} R \}_{K_p} \dagger t_2 \}_{K_q} : \Sigma_s$</p> <hr/> <p>$\{ r \leq_{t_1} R \}_{K_p} \in S$</p> <p>$\exists Q : PRINCIPAL \bullet$</p> <p>$A, S \vdash Q \text{ Pub}_{t_2} K_q$</p> <p>$A, S \vdash Q \mapsto_{t_2} [rm \triangleright world]$</p> <p>$r \notin \{root, rm\}$</p> <p>$t_2 = \{now\}$</p> <p>$\neg \exists r' : LABEL \bullet A, S \vdash r' \leq_{now} r$</p> <p>$\neg \exists \{ P \mapsto_t [r' \triangleright d] \}_{K} : S \mid r' = r \wedge now \leq t$</p> <p>$\neg \exists \{ P \mapsto_t [r' \triangleright d] \}_{K} : S \mid r' = r \wedge now \leq t$</p> <p>$\neg \exists \{ \{ P \mapsto_{t_1} [r' \triangleright d] \}_{K_1} \dagger t_2 \}_{K_2} : S \mid r' = r \wedge now \leq t_1 \dagger t_2$</p> <p>$\neg \exists \{ \{ P \mapsto_{t_1} [r' \triangleright d] \}_{K_1} \dagger t_2 \}_{K_2} : S \mid r' = r \wedge now \leq t_1 \dagger t_2$</p> <p>$S' = S \cup \{s?\} \setminus \{ \{ r \leq_{t_1} R \}_{K_p} \}$</p> <p>$perm' = perm \setminus \{r\}$</p>

The *RemoveLabel* schema accepts a revoked label activation, which must be a revocation of an activation already in the S set.

The operation will block unless there exists a principal Q who owns the key used to sign the revoked role activation and who has sufficient privileges to edit the role hierarchy. It is further ensured that the role is being revoked at the current instant, that the role has no children and that it is not one of the special roles root or rm. Finally it is ensured that there is no active delegation of this role in S . It does not matter whether

the delegation would infer any actual privileges at the moment, since it might be made to grant privileges, for example by inserting an appropriate key certificate.

When all of the above is true, the revoked activation is added to the set of current statements and the original activation is removed.

5.3.7 Key Certificate Updates

We now turn to the transition schemas involved in issuing and revoking keys.

<i>IssueNormalCertificate</i>
ΔRAS $s? = \{ Q \text{Pub}_t K_q \}_{K_p} : \Sigma_s$
$\exists P : \text{PRINCIPAL} \bullet$ $A, S \vdash P \text{Pub}_t [K_p \triangleright d_p]$ $Q \in d_p$ $now = \min(t)$ $S' = S \cup \{s?\}$ $perm' = perm$

<i>IssueCACertificate</i>
ΔRAS $s? = \{ Q \text{Pub}_t [K_q \triangleright d_q] \}_{K_p} : \Sigma_s$
$\exists P : \text{PRINCIPAL} \bullet$ $A, S \vdash P \text{Pub}_t [K \triangleright d_p]$ $Q \in d_p$ $d_q \leq d_p$ $now = \min(t)$ $S' = S \cup \{s?\}$ $perm' = perm$

$$\text{IssueCertificate} = \text{IssueNormalCertificate} \vee \text{IssueCACertificate}$$

The *IssueCertificate* schema is split into a schema for issuing normal certificates and another almost identical one for issuing CA certificates. The combined schema accepts a certificate issuing a key K_q to a principal Q in a period t . In the case of CA certificates it is restricted to a domain d .

The operation will block unless there exists a principal P who owns the key used to sign the certificate and who has sufficient privileges to issue the certificate. It is further ensured that the revocation takes effect at the current instant and that if s is a CA certificate, that it's domain falls within the domain in which the issuer may issue certificates with K_p .

When all of the above is true, the certificate is added to the set of current statements.

<i>RevokeNormalCertificate</i>
ΔRAS $s? = \{ \{ Q \text{Pub}_{t_1} r \}_{K_q} \dagger t_2 \}_{K_r} : \Sigma_s$
$\{ Q \text{Pub}_{t_1} r \}_{K_q} \in S$ $\exists R : \text{PRINCIPAL} \bullet$ $A, S \vdash R \text{Pub}_{t_2} [K_r \triangleright d']$ $Q \in d'$ $t_2 = \{now\}$ $S' = S \cup \{s?\} \setminus \{ \{ Q \text{Pub}_{t_1} r \}_{K_q} \}$ $perm' = perm$

$\text{RevokeCACertificate}$ <hr/> ΔRAS $s? = \{ \{ Q \text{Pub}_{t_1} [r \triangleright d] \}_{K_q} \uparrow t_2 \}_{K_r} : \Sigma_s$ <hr/> $\{ Q \text{Pub}_{t_1} [r \triangleright K_q] \} \in S$ $\exists R : \text{PRINCIPAL} \bullet$ $A, S \vdash R \text{Pub}_{t_2} [K_r \triangleright d']$ $Q \in d'$ $d \leq d'$ $t_2 = \{now\}$ $S' = S \cup \{s?\} \setminus \{ \{ Q \text{Pub}_{t_1} [r \triangleright K_q] \} \}$ $perm' = perm$

$$\text{RevokeCertificate} = \text{RevokeNormalCertificate} \vee \text{RevokeCACertificate}$$

The *RevokeCertificate* schema is split into a schema for revoking normal certificates and another almost identical one for revoking CA certificates. The combined schema accepts a revocation of a certificate in S which issued a key K_q to a principal Q in a period t_1 . In the case of CA certificates it will have been restricted to a domain d . The revocation becomes active at time t_2 and is signed with a key K_r .

The operation will block unless there exists a principal R who owns the key used to sign the revocation and who has sufficient privileges to issue (and thereby revoke) the certificate. It is further ensured that the revocation takes effect at the current instant.

When all of the above is true, the revoked certificate is added to the set of current statements and the original certificate is removed.

5.3.8 Delegation Updates

We next give the transition schemas involved in delegating and revoking roles to/from principals.

$\text{IssueNormalDelegation}$ <hr/> ΔRAS $s? = \{ Q \mapsto_t [r \triangleright d] \}_K : \Sigma_s$ <hr/> $\exists P : \text{PRINCIPAL} \bullet$ $A, S \vdash P \text{Pub}_t K$ $A, S \vdash P \mapsto_t [r \triangleright d]$ $now = \min(t)$ $r \in \text{root}_{now}^\downarrow \cup \{rm\} \setminus perm$ $S' = S \cup \{s?\}$ $perm' = perm$

The *IssueAdminDelegation* schema is exactly as the above, except that $s?$ has the form $\{ Q \mapsto_t [r \triangleright d] \}_K$

$$\text{IssueDelegation} = \text{IssueNormalDelegation} \vee \text{IssueAdminDelegation}$$

The *IssueDelegation* schema is split into a schema for issuing normal delegations and another almost identical one for issuing administrative delegations. The combined schema accepts a delegation conferring to the principal Q the right to act in or to administer the role r in the domain d during the period t .

The operation will block unless there exists a principal P who owns the key used to sign the delegation and who has sufficient privileges to issue the delegation. It is further ensured that the delegation takes effect at the current instant and that r is a currently active role and not a permission.

When all of the above is true, the delegation is added to the set of current statements.

$\text{RevokeNormalDelegation}$ <hr/> ΔRAS $s? = \{ \{ Q \mapsto_{t_1} [r \triangleright d] \}_{K_p} \dagger t_2 \}_{K_r} : \Sigma_s$ <hr/> $\{ Q \mapsto_{t_1} [r \triangleright d] \}_{K_p} \in S$ $\exists R : \text{PRINCIPAL} \bullet$ $A, S \vdash R \text{Pub}_{t_2} K$ $A, S \vdash R \mapsto_{t_2} [r \triangleright d]$ $t_2 = \{ \text{now} \}$ $S' = S \cup \{s?\} \setminus \{ \{ Q \mapsto_{t_1} [r \triangleright d] \}_{K_p} \}$ $\text{perm}' = \text{perm}$
--

The *RevokeAdminDelegation* schema is exactly as the above with two exceptions: The statement $s?$ has the form $\{ \{ Q \mapsto_{t_1} [r \triangleright d] \}_{K_p} \dagger t_2 \}_{K_r}$, and the delegation to remove from S has the form $\{ Q \mapsto_{t_1} [r \triangleright d] \}_{K_p}$

$$\text{RevokeDelegation} = \text{RevokeNormalDelegation} \vee \text{RevokeAdminDelegation}$$

The *RevokeDelegation* schema is split into a schema for revoking normal delegations and another almost identical one for revoking administrative delegations. The combined schema accepts a revocation of a delegation in S which conferred to the principal Q the right to act in or to administer the role r in the domain d during the period t .

The operation will block unless there exists a principal R who owns the key used to sign the revocation and who has sufficient privileges to issue (and thereby revoke) the delegation. It is further ensured that the revocation takes effect at the current instant.

When all of the above is true, the revoked delegation is added to the set of current statements and the original delegation is removed.

5.4 Security Properties

We have now defined all the different parts of the RAS system and can finally set about proving that the whole has the kinds of properties we require of an access control system. By then examining the proofs we can see which assumptions about the system are required to maintain certain properties, by which we see which parties we must trust in order for the entire system to have these properties.

We will start by noting a few lemmas which are more or less self-evident and which will be needed in the more interesting proofs that follow. The first lemma tells us that any answers to access control questions that the RAS server makes will depend on what is provable from its sets of axioms and current statements. This allows us to concentrate in the proofs of other theorems on what effect operations have on the provability of various statements.

Lemma 5.1 (Server Responses Depend on Provability) *Any operation of the server which purports to decide whether a particular statement is true will only answer positively (i.e. by returning a non-empty proof) when the statement is provable from $A \cup S$. Further, any non-empty proof returned is a valid proof of the statement in question.*

More formally, in each case, when asked about the validity of statement $s? \in \Sigma_0$, we have $p! \neq \emptyset \Leftrightarrow A, S \vdash s?$ and $p! = \emptyset \vee A, S \vdash_{p?} s?$, where $p!$ is the returned proof.

Proof:

It is clear from their definitions that ProveLabelOrder, ProveCapability, and ProveKeyOwnership have this property. We also see that these are the only operations purporting to decide the validity of a statement.

When we simply verify that the proof is non-empty, this property requires trust in the RAS server to answer truthfully whether statements are provable. On the other hand, if the proof is verified by the recipient, no trust in the RAS server is required.

We now prove that nobody has the authority to act in a role or holds a permission unless there is a delegation of the authority by a principal with the appropriate authority. This allows us to concentrate our

attention on under which circumstances principals have the authority to delegate authority, since controlling this is sufficient for controlling the authority to act in roles.

Lemma 5.2 (Authority To Act Comes From Delegation) *Given a principal P such that $A, S \vdash P \mapsto_t [r \triangleright d]$, one of the following is true for some t', d' and r' such that $t \leq t', d \leq d'$ and $r \leq_t r'$:*

1. *There exists an axiom $P \mapsto_{t'} [r' \triangleright d'] \in A$.*
2. *There exists a signed statement $\sigma \in S$, which is either a delegation $\{P \mapsto_{t'} [r' \triangleright d']\}_K$ or a revoked delegation $\{\{P \mapsto_{t'} [r' \triangleright d']\}_K \dagger t_r\}_K$, where there exists a principal Q with $Q \text{ Pub}_{t'} K$ and $A, S \vdash Q \mapsto_{t'} [r' \triangleright d']$.*

Proof:

Given a proof of $P \mapsto_t [r \triangleright d]$, we can assume, without loss of generality, that it is in p -normal form. It can now either be a trivial proof where an axiom is introduced and then weakened, or it can be a proof-tree rooted in a composite derivation rule.

If the proof is an axiom followed by a weakening, we have case 1. If the proof is a tree, rooted in a composite derivation rule, we look at the list of rules and see that the only ones whose result is of the appropriate form are WEAK-DEL-MAY and WEAK-DEL-MAY \dagger . Either of these rules implies the existence in $A \cup S$ of σ along with the criteria on the parameters to σ and on Q and K that are given in case 2. Finally, since $\sigma \in \Sigma_s$, and $A \subseteq \Sigma_0$, we know that $\sigma \in S$.

The above lemma is entirely a property of the RAS-logic and requires no trust in any part of an implementation. However, we may still need to trust e.g. the RAS server to actually act in accordance with the RAS logic unless we choose to verify all proofs. Also note that the RAS server is not initialised to hold any axioms of the form given in case 1, so only case 2 is relevant in a properly working RAS setup.

Theorem 5.3 (Authority to Delegate Comes from Outside any Group) *If, for a principal P in group G , it holds that $A, S \vdash P \mapsto_t [r \triangleright d]$, then there must exist a principal Q in G such that one of the following is true for some t', d' and r' with $t \leq t', d \leq d'$ and $r \leq_t r'$:*

1. *There exists an axiom $Q \mapsto_{t'} [r' \triangleright d'] \in A$, conferring the authority to Q .*
2. *There exists a delegation $\{Q \mapsto_{t'} [r' \triangleright d']\}_K \in S$, issued by a principal $R \notin G$, where $R \text{ Pub}_t K$ and $Q \mapsto_{t'} [r' \triangleright d']$*

Proof:

Let us assume that $A, S \vdash P \mapsto_t [r \triangleright d]$ and consider the derivation of this in p -normal form. It will be rooted either in an axiom along with a weakening or in an application of the WEAK-DEL-DEL or WEAK-DEL-DEL \dagger rule. Both of these require a sub-derivation in p -normal form of the capabilities of the issuer of the delegation, which has the same form as the original derivation.

This highlights a branch of the derivation tree where a sequence of statements $Q_i \mapsto_{t_i} [r_i \triangleright d_i]$ for $i = 0 \dots n$ are being derived, with $Q_0 \mapsto_{t_0} [r_0 \triangleright d_0] = P \mapsto_t [r \triangleright d]$ and $Q_n \mapsto_{t_n} [r_n \triangleright d_n]$ an axiom or the weakening of an axiom.

If $Q_n \in G$ we have case 1. Otherwise we choose i to be the smallest index such that $Q_i \in G$ and $Q_{i+1} \notin G$. At this point along the branch a delegation as in case 2 is being used.

This theorem is also purely a property of the RAS logic and requires no particular assumptions about the trustworthiness of the RAS server.

We now turn to more complex properties of the RAS system, the first of which is that no operation of the RAS server will allow its state to change so that the answers to queries directed at periods in the past will change. In other words, if a query to the RAS server returns that a principal has (or does not have) a certain capability, then if the server is queried at any later time whether the principal had the capability at that instant in the past, it will return an equivalent result. The actual proof returned may be different (in the case of a positive answer), but the server will never change its decision on whether the capability was valid or not.

Theorem 5.4 (Past State is Preserved) Any statement in Σ_0 with a validity period t with $t \leq \text{past}$ (i.e. t is a sub-period of past), which is derivable from the A, S in state $(A, S, \text{perm}, \text{now})$ of the RAS server, will be derivable from the A', S' in state $(A', S', \text{perm}', \text{now}')$ achieved by any sequence of operations on the RAS server from state $(A, S, \text{perm}, \text{now})$

Similarly, any similar statement which is not derivable from A, S will not be derivable from A', S' either.

Proof:

Only an outline of the proof is given. It is sufficient to show that each operation of the RAS server preserves past state, as the theorem then follows by induction. It is also clear that query operations preserve past state, so only update operations need be considered.

It is clear that adding statements to S will not invalidate the derivation of any statement, so we have shown the first claim of the proof for all operations other than revocation operations.

By comparing derivation rules interpreting un-revoked signed statements with the rules interpreting revoked statements, we see that the revocation operations will replace a statement with one which can be used only to derive statements which were previously derivable, but not all such statements. Thus we have shown the second claim of the proof for all operations other than those adding an un-revoked signed statement to S .

To show that revocation operations don't render a previously derivable statement underivable we need a rather lengthy proof showing that for each of the operations, given a derivation of a statement with a period in the past, we can replace the application of the rule interpreting the signed statement being revoked, with an application of the rule interpreting the revoked statement replacing it. It will have the antecedents from the interpretation of the un-revoked statement concerning the issuer of the statement, as well as the derivation about the revoker which allowed the server to perform the replacement.

To show that non-revocation update operations don't render a previously underivable statement with a period in the past derivable, we note that each of these operations will block if the statement being inserted has a period overlapping past and proceed with a structural induction over the derivation rules showing that any derivation which includes a statement with validity period t will have a validity period contained in t .

The above theorem has not been proved. We don't expect the proof to be technically difficult, but large and involving a separate argument for most of the operations of the RAS server.

The theorem relies heavily on the RAS server operating correctly. In particular it relies on the server not discarding any information about delegations, key certificates and label activations, not accepting statements or revocations that take effect in the past or rolling back revocations.

Lemma 5.5 (Activation Time-stamps) The following properties hold of the RAS server:

1. All label activations that are accepted into S , have an activation times-tamp equal to now when they are accepted. Likewise, when a label activation is revoked, its revocation times-tamp is equal to now.
2. All label activations in S have activation and revocation time-stamps $\leq \text{now}$.
3. The only operations which can affect the role hierarchy are `ActivateLabel` and `RemoveLabel`.

Proof:

We note that the only operations (other than initialisation), which update the role ordering statements in S , are `ActivateLabel` and `RemoveLabel` in section ???. `ActivateLabel` has the property that $S' \setminus S = \{ \{ r \leq_{\text{now}} R \}_{K} \}$, and `ActivateLabel` has the property that $S' \setminus S = \{ \{ \{ r \leq_{t_i} R \}_{K_i} \dagger \text{now} \}_{K_r} \}$, where $\{ r \leq_{t_i} R \}_{K_i} \in S$. This tells us that non-revoked role-ordering statements will all have an activation times-tamp $t = \text{now}$ when they are inserted, and since ΔRAS includes ΔTime , which ensures that now is non-decreasing, we will have for all non-revoked role-ordering statements an activation times-tamp $t \leq \text{now}$.

Further, since a revoked label activation $\{ \{ r \leq_{t_i} R \}_{K_i} \dagger t_r \}_{K_r}$ will only be accepted into S provided $\{ r \leq_{t_i} R \}_{K_i} \in S$, and we have shown that $t_i \leq \text{now}$, we will always have the activation times-tamp of the period of a revoked role ordering statement less or equal to the current instant. The same logic as for non-revoked role-ordering statements now shows that we also have $t_r \leq \text{now}$, establishing properties 1 and 2.

Looking at the *ORDER* and *ORDER*[†] derivation rules, we see that their only antecedents are a label activation, possibly in a revoked state, along with statements about the key ownerships and capabilities of principals at the time the statement was issued and possibly when it was revoked. We can therefore apply theorem ?? to show that no changes in capabilities or key ownerships of principals will change the derivability of role ordering statements, so this must be done by changing the set of label activations. But we had previously noted that only the *ActivateLabel* and *RemoveLabel* operations do this, establishing property 3.

Theorem 5.6 (Role Structure is Correct) *For any instant t , the partial order “ \leq_t ”, relative to a set $A \subseteq \Sigma_0$ of axioms and a set $S \subseteq \Sigma_s$ of current statements¹², held by the RAS server has the following properties:*

1. *rm is not comparable to any other role or permission.*
2. *A role, which is comparable to any other role, will be dominated by root.*
3. *There exist no two distinct labels r_1 and r_2 such that $r_1 \leq_t r_2$ and $r_2 \leq_t r_1$*
4. *Given a role $r \notin \text{perm}$, a permission $p \in \text{perm}$, and a period t , we will not have $r \leq_t p$.*
5. *In S there is, for each label r , at most one label activation $\{r \leq_{t_i} R\}_{K_i}$ with $t \leq t_i \dots \omega$ or revoked label activation $\{\{r \leq_{t_i} R\}_{K_i} \dagger t_r\}_{K_r}$ with $t \leq t_i \dots t_r$.*

Proof:

We proceed by induction over the state transitions allowed by the RAS server. The base case is the state after initialisation, as per section ??, where we have no statements in A or S which allow us to use the *ORDER* or *ORDER*[†] to derive information about \leq_t (c.f. section ??). This leaves *REFLEXIVITY*, *TRANSITIVITY* and *WEAK-ORDER*, the transitive closure of which yields the identity relation for any t and this relation clearly has the properties listed above.

We now assume that a state $(A, S, \text{perm}, \text{now})$ of the RAS server has the properties listed and show that a state $(A', S', \text{perm}', \text{now}')$ after any operation given in the specification still has these properties.

It is sufficient by property 3 in lemma ?? to show that *ActivateLabel* and *RemoveLabel* maintain the properties listed in the theorem. We also have by property 1 in lemma ?? that role ordering statements are only accepted when they take effect at the current instant and only allowed to be revoked when the revocation takes effect at the current instant, which in turn implies that the label hierarchy is constant in instants after now. This, along with theorem ??, reduces the problem to showing that if the 5 properties hold for \leq_{now} at the beginning one of the two relevant operations, then it holds for $\leq_{\text{now}'}$ at the end of the operation. We will now do this for each of the 5 properties.

1. *It is easy to show that *RemoveLabel* will not make any two roles or permissions comparable which were not comparable previously, so we will look only at *ActivateLabel*.*

*The predicate in the body of *ActivateLabel* implies $r \neq rm$, so the role activation inserted into S to form S' will not make another role dominate rm .*

The predicate also implies $\forall r' \in R \bullet A, S \vdash r \leq_{\text{now}} \text{root}$, which will not be true by the inductive hypothesis if $rm \in R$, so there will never be a role-ordering statement in S' making rm dominate another role.

It is now easy to verify that the only ordering statements that can be derived are that $rm \leq_t rm$ for all t .

2. *When a label activation $\{r \leq_{\text{now}} R\}_K$ is added in an *ActivateLabel* operation, it is ensured that $\forall r' \in R \bullet A, S \vdash r \leq_{\text{now}} \text{root}$ and that $R \neq \emptyset$. Since the only effect on which labels are comparable are the results of making r dominated by all the labels in R , which by *TRANSITIVITY* and *ORDER-SET*, makes r dominated by all labels which dominate any label in R , it is sufficient to show that r will be dominated by root after the operation.*

¹²The ordering being relative to A and S means that $r_1 \leq_t r_2$ if and only if $A, S \vdash r_1 \leq_t r_2$ by the RAS logic.

We therefore select an $r' \in R$, and have that $A, S \vdash r' \leq_{\text{now}} \text{root}$ and that $A', S' \vdash r \leq_{\text{now}} r'$, which implies $A, S \vdash r \leq_{\text{now}} \text{root}$ which in turn implies $A', S' \vdash r \leq_{\text{now}'} \text{root}$ by the monotonicity of the \vdash operator and the uniformity of the label hierarchy in the future.

When a label activation $\{r \leq_t R\}_{K_i}$ is exchanged for a revoked version $\{\{r \leq_t R\}_{K_i} \dagger \text{now}\}_{K_r}$ in a `RemoveLabel` operation, it is ensured that there is no label r' such that $A, S \vdash r' \leq_{\text{now}} r$. This implies that the only labels that are comparable with r are the ones that dominate it and that no label's comparability with `root` depends on the label activation about to be revoked. This in turn implies that revoking the label activation makes r only comparable itself and has no effect on other labels comparability with `root`, maintaining property 2.

3. Let us examine derivation trees for label order statements. They will in general have the form of a chain of `TRANSITIVITY` applications, showing that since $r_1 \leq_t r_2$, $r_2 \leq_t r_3$, \dots , $r_{n-1} \leq_t r_n$, then we have $r_1 \leq_t r_n$. Each of these $r_i \leq_t r_{i+1}$ statements needs its own proof, which will either be an application of `REFLEXIVITY` if $r_i = r_{i+1}$ (which in any case is superfluous and we can discount) or an application of `ORDER-SET` with either an application of `ORDER` or `ORDER` \dagger to extract form a label activation.

Now, assume that the addition of $\{r \leq_t R\}_K$ to S causes there to exist distinct labels r_a and r_b with both $r_a \leq_{\text{now}} r_b$ and $r_a \leq_{\text{now}} r_b$. In other words, we would have $A', S' \vdash r_a \leq_{\text{now}} r_b$ and $A', S' \vdash r_b \leq_{\text{now}} r_a$, which implies a derivation tree for each result.

If neither derivation tree included the new label activation, then they would both have existed before the operation, contradicting the inductive hypothesis.

If the new label activation is included in a derivation, and we will assume without loss of generality that it is the derivation of $r_a \leq r_b$, then the derivation would have $r \leq_t r'$, with $r' \in R$, somewhere in the chain of `TRANSITIVITY` applications. But since `ActivateLabel` wouldn't allow the addition of the new label activation unless r had been previously incomparable to any other labels (by there being no label activation for it, implying it is not dominated by anything, along with property 2), r must be at the beginning of the chain, implying $r = r_a$. But then we could only have $r_b \leq r_a$ if the derivation of this also included the new label activation, since r_a was previously incomparable to any other label. But the same argument as before now shows that $r_b = r$ and so we have a contradiction and no such r_a and r_b can be found.

`RemoveLabel` trivially maintains property 3.

4. If we are unable to derive for any permission r_p and role r_r , using only `ORDER`, `ORDER` \dagger and `ORDER-SET`, that $r_r \leq_{\text{now}} r_p$, then an easy induction shows that this will also hold when `TRANSITIVITY` and `REFLEXIVITY` are used.

When `ActivateLabel` adds a label activation $\{r \leq_t R\}_K$ it ensures that $r \notin \text{perm}' \Rightarrow \text{isPermission?} = \text{FALSE} \Rightarrow R \cap \text{perm} = \emptyset$, meaning that if r is not a permission, none of the labels in R are permissions and no label activations will be accepted which would allow us to derive that a role is dominated by a permission.

`RemoveLabel` trivially maintains property 4.

5. A label activation $\{r \leq_t R\}_K$ is only accepted in a `ActivateLabel` operation if no other activation for the same label is active.

`RemoveLabel` trivially maintains property 3.

Corollary 5.7 (Future Label Hierarchy is Constant) It was shown in the proof of theorem ?? that the ordering of labels is constant in the period $\text{now} \dots \omega$, although this hierarchy may be exchanged for another constant hierarchy by either the `ActivateLabel` or `RevokeLabel` operations.

Corollary 5.8 (The Label Hierarchy is a Weakly Acyclic Directed Graph) It follows directly from theorem ?? that the time dependent graph constructed by taking `LABELS` to be the set of nodes and connecting a label r_1 to a label r_2 at instant t if it can be derived without using `TRANSITIVITY` that $r_2 \leq_t r_1$ is a weakly acyclic directed graph rooted in `root` along with an infinite number of unconnected labels including `rm`.

The above theorem and assorted corollaries again all rely heavily on the RAS server operating correctly, accepting only certain label activations and also, by the use of theorem ??, managing the past state of the system correctly as noted after that theorem.

6 Implementation

The implementation of the RAS server and the libraries needed to communicate with it and interpret its responses have been implemented alongside the development of the formalisms and the specification described in the previous sections. Here we will look at how the formalisms were and are being implemented.

6.1 Extreme Programming and Formal Specification

A fairly recent idea in development methodology is *extreme programming* (XP)[?] and similar agile development methods. Where traditional development methods, such as the Unified Process (UP) [?], call for a full specification of the application before any implementation starts, the agile methods advocate creating a minimal running application as soon as possible, to make code testable as soon as it is written and to allow the form of the application to be influenced by how the actual users of the application respond to it at each step in its development. Various forms of modelling are applied along the way as needed, but the real specification in fact lies in the test procedures which are run after each build of the program.

Many see this as the antithesis of formal development methods, but this is only partially true in the view of the author. It is true that with agile methods, a full specification of an application does not generally exist. However, in cases where safety or security is important, a specification along with arguments about the properties of the specification can be valuable in determining whether the application fulfils the appropriate safety or security criteria. A way to reconcile these apparently opposing views is to develop the specification of the system alongside the actual application. This specification would be at a fairly high level of abstraction and only concern itself with those aspects of the system which need to fulfil any safety and security properties.

In the particular case of XP, the methodology calls for the creation of an initial implementation which will be far from having all required features. Development then proceeds in short iterations, where a feature is added and tested to form a new, slightly more complete version of the application. What we suggest, and the approach taken with the implementation of the RAS is that the initial implementation of the application comes with an initial, fairly trivial specification. Each iteration in the XP process then consists of selecting a feature to implement, updating the specification, writing the code, and update the proofs of any properties that have been established as needed. In many cases the feature being added in an iteration will not affect the specification, as it will only concern itself with safety and security related issues. Only enough detail is included to allow proofs that safety and security criteria are met.

This is in fact how most specifications are written. An initial, simplified specification is written and more detail is injected until it captures the full scope of the system under development. The only difference is that now these two iterative processes are run concurrently are synchronised at the points where proofs need to be updated.

6.2 Implementing the Specification

Here we will describe how the various parts of the previous sections of this paper have been and are being implemented as Java code and give UML diagrams as appropriate. Only a broad outline of the implementation is given, since a full explanation of all implementation details would be too large and time consuming for this medium.

The sequence of implementation approximately follows the sequence of definition and specification in the paper as you would expect from the discussion of XP methodology. However, certain details in the implementation have been deferred for practical reasons which will be discussed later.

6.2.1 Sorts

The various sorts of the logic are implemented as Java classes and instances used liberally in the classes for statements which are discussed below. The implementation is straight forward from the discussion in section ?? and will not be shown explicitly.

We will use the Java type `long` for instants and the class `Period` for periods. The standard library interface `PublicKey` is used for public keys and we will use the Java type `String` for the names of roles

and permissions. There is also a class `DName` which has sub-classes `RasDomain` and `RasPrincipal` for domain and principal names respectively. It implements a mapping from `String` to `String` and various useful methods are implemented to test membership in a domain, generate the smallest domain containing several names etc.

6.2.2 Statements

Statements are also implemented as Java classes. There are classes denoting statements from Σ_0 , which either denote a single type of statements or a pair of such types. Other classes denote the signed or revoked statements in Σ_s corresponding to the statements denoted by one of the previous classes. This creates 3 pairs of classes: `KeyOwnership/KeyCertificate`, `Capability/Delegation` and `LabelOrder/LabelActivation`.

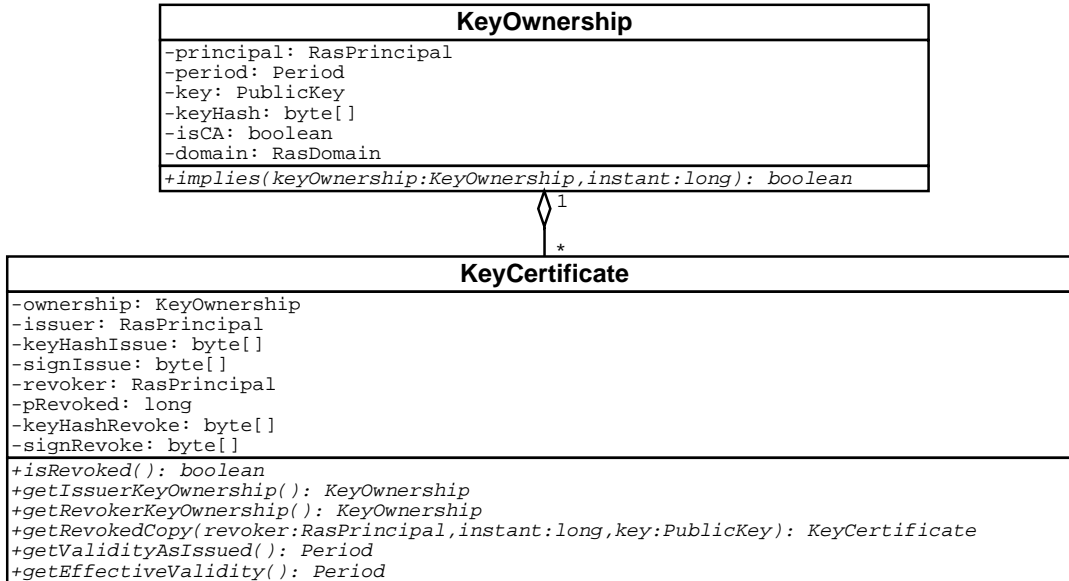


Figure 2: KeyOwnership and KeyCertificate classes

The `KeyOwnership` class, shown in figure ??, denotes the statement “principal $\text{Pub}_{\text{period}}$ key” if `isCA==false`, or “principal $\text{Pub}_{\text{period}}$ [key \triangleright domain]” if it is true. It has various accessor methods to read its member variables, but these are not shown here. It also has a method `implies`, which can be used to decide whether this key ownership statement implies another one by weakening, i.e. by shrinking the validity period.

The `KeyCertificate` class, also shown in figure ??, contains a statement ownership of the `KeyOwnership` type. It denotes “{ ownership } K_i ” if `revoker==null` or “{ { ownership } K_i \uparrow pRevoked } K_r ” otherwise. In either case, K_i and K_r parameters are the public keys whose hash values are equal to `keyHashIssue` and `keyHashRevoke` respectively. This allows it to denote at either normal key certificates or CA certificates in either original or revoked form.

The `Capability` and `Delegation` classes shown in figure ?? follow much the same pattern as `KeyOwnership` and `KeyCertificate`, with `Capability` denoting either “principal \mapsto_{period} [label \triangleright domain]” if `toDelegate==false` or “principal \mapsto_{period} [label \triangleright domain]” if it is true. `Delegation` has a field `capability` of the `Capability` type and denotes either “{ capability } K_i ” if `revoker==null`, or “{ { capability } K_i \uparrow pRevoked }” if it is true. The K_i and K_r parameters are as before.

The `LabelOrder` and `LabelActivation` classes shown in in figure ?? are slightly different, since a label activation does not correspond to a single label order, but to many label orderings with the same smaller label and various greater labels. The `LabelActivation` class denotes “{ name \leq_{period} parents } K_i ” whenever we have, `revoker==null`, but “{ { name \leq_{period} parents } K_i \uparrow pRevoked } K_r ” otherwise, with K_i and K_r as before.

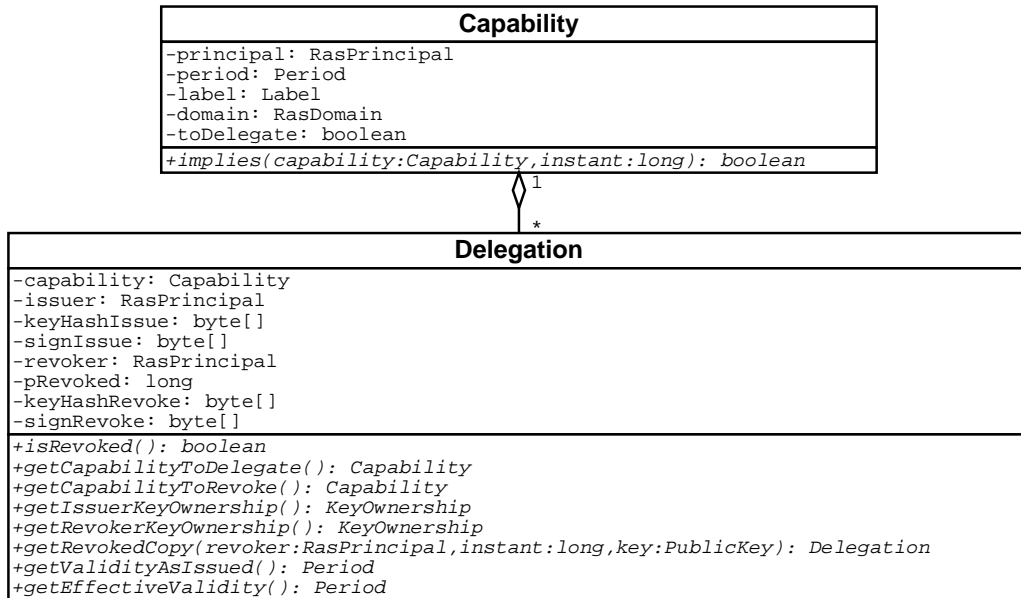


Figure 3: Capability and Delegation classes

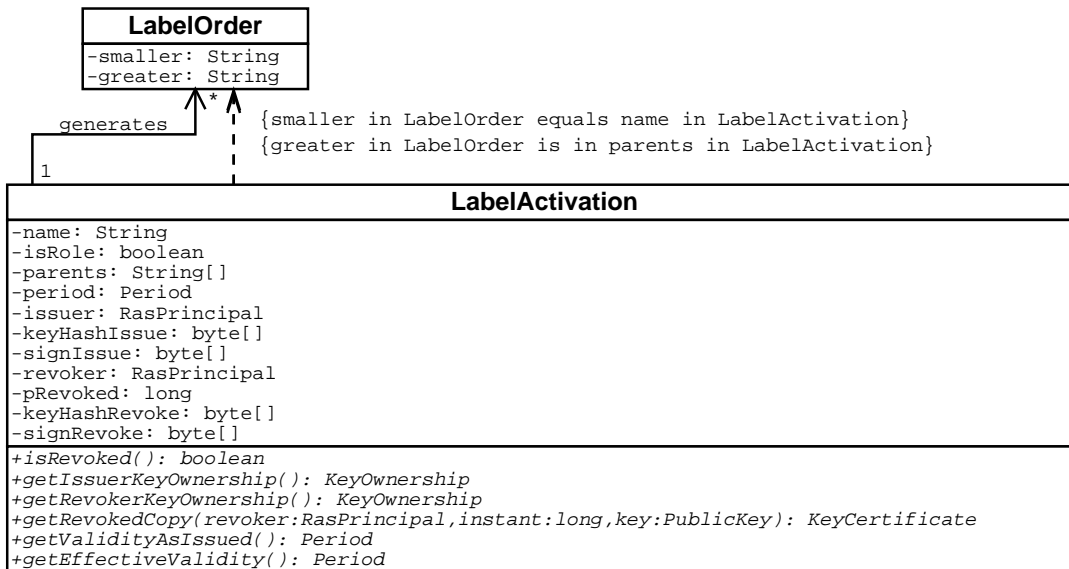


Figure 4: LabelOrder and LabelActivation classes

This LabelActivation can then be used to produce an instance of KeyOwnership denoting the statement $\text{smaller} \leq_{\text{period}} \text{greater}$ for every element greater in the parents array.

6.2.3 Proofs and Derivation Rules

Now that we have classes for the statements in our logical language, it is time to turn to the methods by which we will derive these statements from other statements. The function of deriving new statements is performed by the prover, discussed in section ???. However, the proof generated consists of the application of a number of derivation rules, for which we exhibit implementations here.

Figure ?? shows the classes implementing proofs of capability statements, along with closely related classes. It does not show the equivalent hierarchies for key ownership and label order proofs or the details of the Capability and Delegation classes shown in in figure ??.

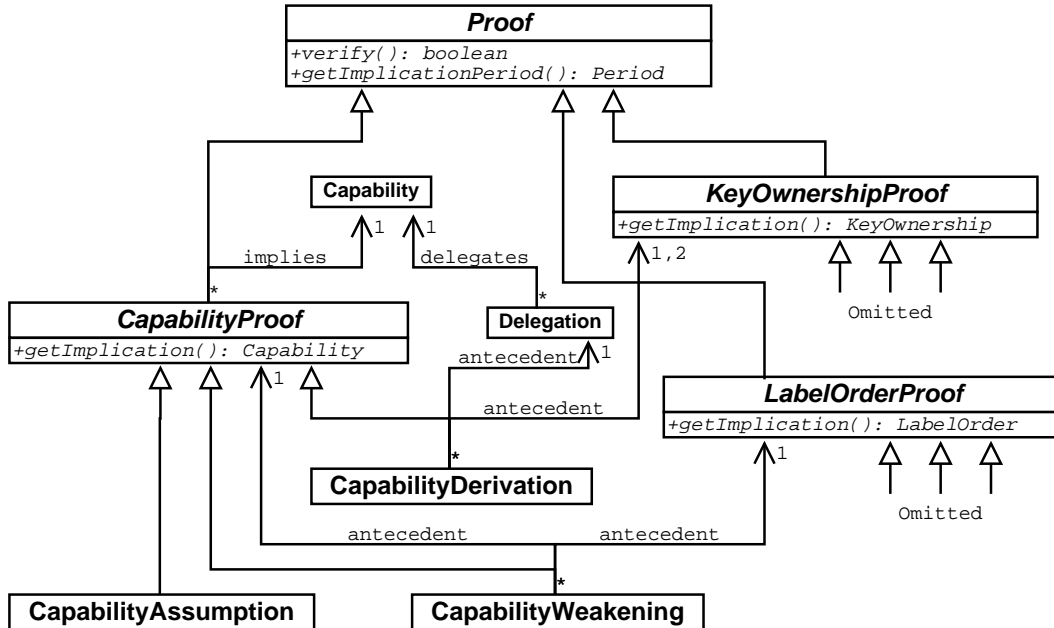


Figure 5: Capability derivation rules and related classes

The root of the hierarchy is the abstract Proof class, which has three direct sub-classes named CapabilityProof, KeyOwnershipProof and LabelOrderProof, all of them abstract. They are used for proving the three types of unsigned statements described in section ??, as seen by the different definitions of the getImplication methods. This diagram only shows the part of the full class diagram related to the CapabilityProof class, but omits the sub-classes of its siblings, which would form a similar structure with KeyOwnership and LabelOrder taking the place of Capability and KeyCertificate and LabelActivation taking the place of Delegation.

A proof is represented as a hierarchy of sub-classes of the Proof class, where the actual class of each instance decides the appropriate types of the antecedents, in accordance with the appropriate derivation rules of the RAS logic. The Proof class also has a verify which is implemented in each concrete sub-class of Proof. It checks that the criteria for the application of the corresponding derivation rules are met before recursively calling the verify method of each antecedent proof, thus verifying the entire proof.

The CapabilityAssumption class corresponds to the introduction of an axiom and so doesn't correspond directly to any derivation rule. It also has no antecedents and its verify method will always return true.

The Capability Weakening class corresponds to one of the WEAK-MAY or WEAK-DEL rules, depending on whether the Capability it purports to imply (via the CapabilityProof) is the capability to act in a role or to delegate it. Looking at those rules we see that they have antecedents which are

directly verifiable about periods and domains as well as an antecedent that $r_2 \leq_{t_2} r_1$, which calls for a sub-proof of the `LabelOrderProof` type. This is shown in the diagram by an *antecedent* with the `LabelOrderProof` class.

The `CapabilityDerivation` class similarly corresponds to one of the DEL-DEL, DEL-MAY, DEL-DEL \dagger , or DEL-MAY \dagger rules depending on the delegation statement it contains. This delegation statement is shown as an antecedent in the diagram, along with one or two `KeyOwnershipProof` and `CapabilityProof` instances as you would expect from the derivation rules being implemented.

6.2.4 Prover

Now that we have the means to represent both statements and proofs, we can look at the class responsible for producing the proofs. This is the `Prover` class shown in in figure ??.

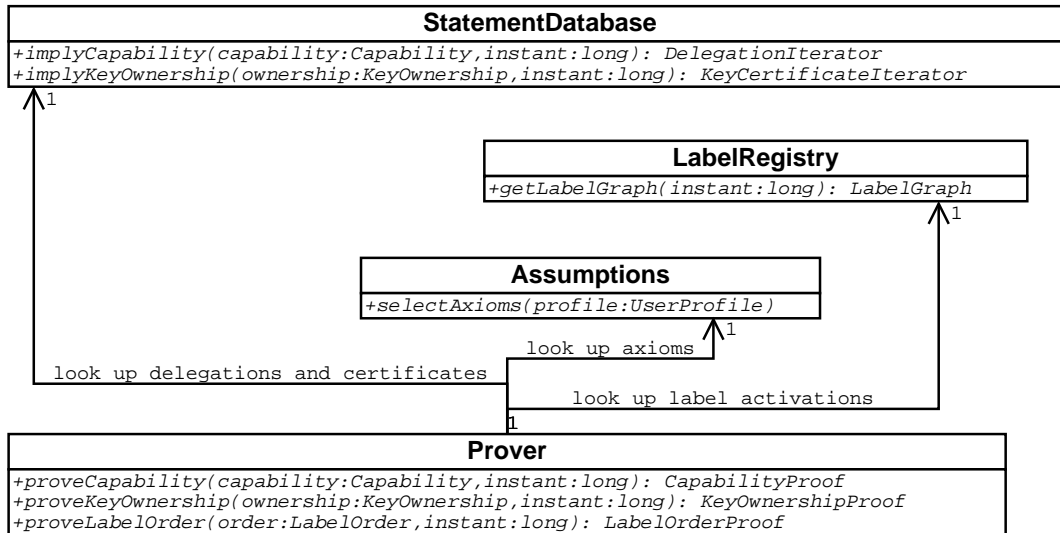


Figure 6: Prover and statement repositories

The `Prover` class implements the algorithm given in section ???. The set A of axioms is represented by the `Axioms` class, which holds a static set of statements assumed to be true. Other statements are looked up in the `StatementDatabase` and `LabelRegistry` classes as needed. These two classes are kept separate to allow the caching of label hierarchies as discussed in section ??.

6.2.5 Server

The server is fully specified in section ??? and repeating this in UML notation would not add to our understanding. We will only note that of course the statements which the various operations of the server may accept as input or give as output will be represented either by the appropriate statement or proof classes that we described above.

6.3 Current State

At the moment the RAS is in a usable state, as per the XP methodology, but certain features are not implemented. Some of these features are required for the full security benefits to be realised, but are deferred for practical reasons.

RAS Server is running with a working prover and database back-end. This is the most important part of the system to have implemented, as it shows that the algorithms provided earlier are workable. Their scalability is still not certain, but looks good so far.

Signed Statements are not given proper cryptographic signatures, but mere tokens to help in testing. The reason for this is to maintain a running and testable application. The system can be tested without cryptographic signatures, but testing just the signatures would require writing large amounts of code for various statements and statement encodings first. Also, it slows down development to have to keep track of cryptographic keys during testing and small changes to the system may render all signatures invalid, which calls for re-building the state of the system from scratch, involving keys from many sources.

It was therefore deemed better to implement signature signing as one of the last features of the system, when most of the code has stabilised.

Client-Side Interface for Java clients has been defined and partially implemented. Applications can already now be written to take advantage of the RAS server, although not all the security goals will be realised until the server is further developed. Also, the query interface may be expanded as the need arises, in XP style, but the current minimal interface is very unlikely to change.

Verifier is not implemented. This was in fact the last section of the report to be written and it could be said that at the time of this writing, the project is half-way through an iteration of improvement, where the verifier feature is being added.

Command-Line Administration Interface has been implemented fully, with the exception that it does not currently apply cryptographic signatures to statements sent to the RAS server.

Graphical Administration Interface has not been implemented, designed or thought about much.

6.4 Statement Encoding

So far we have spoken about principals “having a statement”, for example in a set of axioms, in a set of current statements or in a formalisation of a proof. We have both given an abstract notation for these statements and concrete implementations in Java. However, since the statements may need to be communicated between different machines and since signatures need to be generated on statements, some thought must be given to how these statements should be encoded in concrete terms.

6.4.1 Internal Manipulation

Most of the operations done with statements will be internally in the RAS server while it is attempting to build a proof-tree for a particular hypothesis. We will not in any way attempt to specify or restrict the internal representation of the statements in the RAS server or any other application. Different representations will be suitable for different purposes and the Java implementation shown previously is only an example.

For example, statements are likely to be stored in a relational database, scattered through several tables and allowing for fast searches when building proof-trees. Once a candidate statement is found in the database, however, it will be converted to a coherent object in memory and may form part of proof trees or lookup tables or be used in any number of ways.

6.4.2 Communication

In the normal use of the RAS server, statements will need to be sent back and forth amongst applications. Mostly, servers who use the RAS server for access control will send statements to the RAS server and ask for a proof tree in response and user interface applications may send statements to the RAS server to update authorisation information.

In a more heterogeneous environment, it would be important to specify how to encode statements and proof-trees for transfer between applications. However, in the currently envisioned environment, all applications will be written in the Java language and all will have access to common library code. Therefore, the classes representing statements internally to the RAS server can be sent via Serialisation and/or Remote Method Invocation between the individual servers.

If and when applications written in other languages will need to access the RAS server, other encoding formats will be developed. The might be based on XML, which has become popular recently for data transmission where interoperability is important, ASN.1/DER which is used in the x.509 [?] certificate standard and the PKCS cryptography standards or S-expressions which are used in the SPKI standard [?] and is more easily human-readable than ASN.1/DER. Alternatively, the signature encoding could be used for communication encoding.

6.4.3 Signatures

The final form that statements take is when they are to be signed. Unlike the internal representation and the communication representation, it is very important to have a unique signing representation of each statement, so that a statement in any of the other representations can be converted to a unique form suitable for verifying the signature on a statement or for generating the signature in the first place. Any difference in the representation used to form the signature and the one used to verify it will result in the signature being rejected. This is true even if the difference is a single bit or a space inserted or removed and furthermore, since there is no notion of “almost correct” signatures, slightly differently represented statement will be rejected in the same way as statements with forged signatures.

We will give the signature encoded form of each of the statements in the RAS logic. The encoding function is constructed to be non-ambiguous, i.e. any two different statements will produce different signature encodings, which in turn means that a signature on one statement could not be used to verify another statement¹³.

We recursively define a function *senq*, from the sets of statements, periods, keys, domains, principals and roles to the set of finite character strings by the following rules. On the right side of the equations, $\{X\}$ should be replaced by *senq*(X) enclosed in actual curly brackets and $[X]$ should be replaced by *senq*(X) without any added brackets.

$$\begin{array}{ll}
 \textit{senq}(t) = [t_1] \text{ } \text{to} \text{ } [t_2] & \text{with } t_1 \text{ and } t_2 \text{ instants and } t = t_1 \dots t_2. \\
 \textit{senq}(K) = H([e]:[n]) & \text{with } K \text{ an RSA key with exponent } e \text{ and modulus } n \\
 \textit{senq}(d) = [c_1]=[v_1], [c_2]=[v_2], \dots, [c_N]=[v_N] & \text{with } d=\{c_1 \mapsto d_2, c_2 \mapsto d_2, \dots, c_N \mapsto d_N\} \\
 \textit{senq}(P) = [c_1]=[v_1], [c_2]=[v_2], \dots, [c_N]=[v_N] & \text{with } d=\{c_1 \mapsto d_2, c_2 \mapsto d_2, \dots, c_N \mapsto d_N\} \\
 \textit{senq}(r) = \text{The name of the role} & \\
 \textit{senq}(r_1, r_2, \dots, r_N) = [r_1], [r_2], \dots, [r_N] &
 \end{array}$$

We can now signature encode the sorts of the logic, but must note how the primitive string and integer values they are composed of are encoded.

- Instants are represented as 64-bit non-negative integers counting milliseconds since midnight (before) January 1st 1970. They are encoded using normal unsigned integer notation and we see that $\alpha = -2^{-63}$ and $\omega = 2^{63} - 1$.
- The components of public keys are arbitrary size positive integers and are encoded as hexadecimal strings with at most one leading zero, and always of even length.
- The H function maps its input to a hex representation of the SHA-1 hash of the input. The input string is UTF-8 encoded before the hash operation is performed and the resulting hexadecimal string is 40 characters, including any leading zeroes.
- The names of roles and the components of domains and names must not contain curly brackets for this encoding to be unambiguous.
- We re-order the components of distinguished names, whether for domains or principals, by first writing out those components of the name which appear in the list below, in the order given, followed by any other components in alphabetic order.

Component list: [C, ST, L, STREET, O, OU, CN, UID, DC]

¹³barring collisions in the hash function applied before signing.

We can now turn to the signature encoding of statements:

$$\begin{aligned}
senq(P \text{ Pub}, K) &= \text{pub}\{P\}\{t\}\{K\} \\
senq(P \text{ Pub}_t [K \triangleright d]) &= \text{ca}\{P\}\{t\}\{K\}\{d\} \\
senq(P \mapsto_t [r \triangleright d]) &= \text{may}\{P\}\{t\}\{r\}\{d\} \\
senq(P \mapsto_t [r \triangleright d]) &= \text{del}\{P\}\{t\}\{r\}\{d\} \\
senq(r \leq_t \{r_1, \dots, r_N\}) &= \text{ord}\{r\}\{t\}\{r_1, \dots, r_N\} \\
senq(\sigma \dagger t) &= \text{rev}\{\sigma\}\{t\} \\
senq(\{\sigma\}_K) &= \text{sign}\{\sigma\}\{K\}
\end{aligned}$$

We can now signature-encode any statement in the RAS logic. Note that none of the values on the right hand side of the equations in the last group contain any whitespace except where it is returned by the recursive calls to *senq* denoted by curly-brackets.

6.4.4 Example Encodings

Let us signature encode the set of axioms of the RAS server and a couple of other statements, choosing suitable names for principals. We truncate the hashes of keys to conserve space and since they are not legible anyway.

$$\begin{aligned}
senq(\{CN \mapsto \text{root}, O \mapsto \text{Flex}, C \mapsto \text{BE}\} \mapsto_{\alpha\dots\omega} [\text{root} \triangleright \text{world}]) = \\
\text{del}\{C=\text{BE}, O=\text{Flex}, CN=\text{root}\}\{-9223372036854775808 \text{ to } 9223372036854775807\}\{\text{root}\}\{\}
\end{aligned}$$

$$\begin{aligned}
senq(\{CN \mapsto \text{root}, O \mapsto \text{Flex}, C \mapsto \text{BE}\} \text{ Pub}_{\alpha\dots\omega} (3, 437)) = \\
\text{pub}\{C=\text{BE}, O=\text{Flex}, CN=\text{root}\}\{-9223372036854775808 \text{ to } 9223372036854775807\}\{\text{a4}..5\text{d}\}
\end{aligned}$$

$$\begin{aligned}
senq(\{CN \mapsto \text{CA}, O \mapsto \text{Flex}, C \mapsto \text{BE}\} \text{ Pub}_{\alpha\dots\omega} [(3, 437) \triangleright \text{world}]) = \\
\text{ca}\{C=\text{BE}, O=\text{Flex}, CN=\text{Root}\}\{-9223372036854775808 \text{ to } 9223372036854775807\}\{\text{a4}..5\text{d}\}\{\}
\end{aligned}$$

$$senq(\{\text{rm} \leq \text{root}\}_{(3,437)}) = \text{sign}\{\text{ord}\{\text{role-manager}\}\{\text{root}\}\}\{\text{a4}..5\text{d}\}$$

7 Discussion of Project

We will now briefly discuss how the RAS project progressed, what approaches were discarded and how it might further evolve.

7.1 The Pure Z Approach

The initial attempt at formally specifying the workings of an RAS server was done in standard Z notation with only normal predicate logic used in the schemas. The development progressed in what in hindsight may be considered XP fashion, with an initial simple specification without any temporal aspects or signatures on delegations, etc. This specification was slowly expanded to include the missing features and re-factoring applied when appropriate, mostly in the form of breaking schemas into several sub-schemas as they became large and complicated. However, even with this approach, the schemas got overly large and the interactions between the various sub-schemas became hard to comprehend. This approach was abandoned when the first schema reached a full page of text and before revocations were introduced.

It was clear that too many mathematical formulas and too complex data structures were needed to encode what seemed like fairly simple ideas. It was also clear that the same ideas and patterns were being restated multiple times throughout the specification, most notably the relation that was to become the derivability relation. This relation was defined on a rather low level in many places in the specification. Defining it to be part of the main state of the system simplified this somewhat, but we now had a specification with infinite state. Also, the application of the derivability relation was cumbersome since we did not have elegant entities corresponding to what became the statements in Σ_0 .

7.2 Introducing the RAS Logic

The above observations led to the realisation that the task would be much simplified by having a language in which to encode access control queries and responses and in which to encode why a response should be accepted. The development of this language, which became Σ , was relatively straight forward. A statement type for each of the types of queries we were interested in and signed versions of these statements were defined to make up the initial version of the language.

The requirement for principals to be able to revoke signed statements they have issued often complicates any delegation based system. The usual approach, as taken in x.509, is to issue lists of revoked statements or otherwise have a repository of revocations which a statement is compared with before it can be used. This poses both practical and theoretical difficulties, with revocation lists possibly being unreachable or outdated and the need to introduce negative action into the logic. Luckily, in the RAS setting we can avoid this problem completely, since the signed statements which may be revoked will all reside on a central server. A new form of signed statements with embedded revocation information was introduced and the RAS server (which was yet to be specified at this point) was expected to purge a statement being revoked and replace it with this revoked form whenever revocation took place.

Of course, we also needed to formalise how the delegation statements and revocations should be interpreted. This took the form of the derivation rules and this is where most of the theoretical work on the project was done. A surprising number of subtleties came up, especially in the way periods were handled. Slight changes in the derivation rules can lead to a complete break-down in the system.

A real-world example of this is the current versions of the Internet Explorer browser by Microsoft Corporation[?], which is in fact only one manifestation of a bug in Windows system libraries up to and including Windows XP. The problem is that no distinction is made between CA certificates and normal key certificates when deciding whether a certificate can be used to issue a key certificate. This is equivalent to introducing the following rule into the RAS logic:

$$\frac{P \text{Pub}_{t_1} K_p ; \{ Q \text{Pub}_{t_2} K_q \}_{K_p}}{Q \text{Pub}_{t_1 \cap t_2} K_q} \text{ (KEY-KEY)}$$

which allows any principal who holds a valid public key to issue keys to other principals. In particular they could issue a key to a bank, but actually keep the private key themselves. The potential for abuse is obvious.

7.3 RAS Server

Once the logic had been developed, the rest of the system was surprisingly obvious. The prover simply has a sub-algorithm for each type of statement we may wish to prove. For each derivation rule which has that type of statement as its descendant (albeit from the list of composite derivation rules) it will loop over the signed statements that may be used along with this derivation rule to derive the statement to be proven and recursively attempt to prove the other antecedents. In other words, the prover algorithm follows directly from the derivation rules. The verifier is even more obvious.

The RAS server could now be specified. There was no need for the successive iterations of adding complexity to the specification, since most of the complexity had been embedded in the RAS logic. Some thought had to be given to the criteria on statements to accept to ensure that past state was conserved and that the role hierarchy would have the form we wanted, but apart from those points, the specification sprang naturally from the requirements of the server and the RAS logic.

The proofs of security the properties of the RAS server caused some trouble, since the proofs tend towards large structural inductions. The proof that past state is preserved was even omitted, since no concise proof has been developed by the author.

7.4 Possible Future Work

The most pressing task at the moment is to complete the implementation of the RAS server. This will test the workability of the scheme in real-world settings and bring to light any further requirements.

Although difficult to foresee, additional requirements might involve encoding other privileges than simple role membership. This would call for adding new primitive statements to Σ and propagating the addition through the various parts of the specification and implementation. Such additional statements might for example include being trusted to write application code to allow other principals to perform certain roles, which is distinct from being allowed to perform the role. This addition would correspond to the Java signed applet model. A similar addition would be certifying communication channels or storage media for use with particular roles. With both additions in place, a principal P using software written by Q on a system R would only be able to perform actions implied by role r if he holds the role, Q can write code for the role and R is a trusted medium for the execution of the role.

In a more theoretical vein it might be interesting to expand Σ drastically. Allowing existential and universal quantifiers, possibly the construction of arbitrary classes of statements and even allowing principals to add derivation rules while the system is running. One can imagine a logic where only a small language is defined and only a few derivation rules, all having to do with how to add further statement to the logic, how to add derivation rules and how to apply the derivation rules to the statements. Such a system would probably require a notion of jurisdiction over statement types or principals such that it is clear which principals can define derivation rules for which statements and which other principals are bound by the addition of these rules.

A Database Schema

```
-- Create RAS server tables for postgres database.
--
-- PostgreSQL types:
--   BIGINT is 8 byte = 64 bit signed integer.
--   SERIAL implicitly creates a sequence and inserts values
--     from there by default.
--   BIT VARYING is a variable-length bit-string but max 8192 bits.
--     I.e. RSA keys for signing must be < 8192 bits

BEGIN TRANSACTION;

-- Table holding basic information about principals.
-- This table will be linked to by activations, and delegations
CREATE TABLE principal (
    id          SERIAL,
    PRIMARY KEY(id)
);

-- Table holding domain-restrictions on delegations and ca certificates.
-- The comp=value pairs from the restrictions on a delegation taken together
-- form the distinguished name for the delegation's domain.
CREATE TABLE principal_name (
    principal   BIGINT NOT NULL REFERENCES principal(id),
    comp        VARCHAR(32) NOT NULL,
    value       VARCHAR(1024) NOT NULL,

    PRIMARY KEY(principal,comp)
);

-- Table holding key certificates
CREATE TABLE certificate (
    id          SERIAL NOT NULL,

    subject     BIGINT NOT NULL REFERENCES principal(id),
    key         CHAR VARYING NOT NULL,
    keyHash     CHAR VARYING NOT NULL,
    p_from      BIGINT NOT NULL,
    p_until     BIGINT NOT NULL,
    ca          BOOLEAN NOT NULL,

    issuer      BIGINT NOT NULL REFERENCES principal(id),
    key_i       CHAR VARYING NOT NULL, -- may reference certificate(hash)
    sign_i      CHAR VARYING NOT NULL,

    revoker     BIGINT REFERENCES principal(id),
    p_revoked   BIGINT,
    key_r       CHAR VARYING, -- may reference certificate(hash)
    sign_r      CHAR VARYING,

    PRIMARY KEY(id),
    UNIQUE(keyHash)
);
```

```
-- Table holding domain-restrictions on delegations and ca certificates.  
-- The comp=value pairs from the restrictions on a delegation taken together  
-- form the distinguished name for the delegation's domain.
```

```
CREATE TABLE cert_domain (  
    cert          BIGINT NOT NULL REFERENCES certificate(id),  
    comp          VARCHAR(32) NOT NULL,  
    value         VARCHAR(1024) NOT NULL,  
  
    UNIQUE(cert,comp)  
);
```

```
-- Table holding activations of labels for particular periods.  
-- Delegations will reference label activations to specify which  
-- role is being delegated.
```

```
CREATE TABLE label_act (  
    id            SERIAL NOT NULL,  
  
    name          VARCHAR(32) NOT NULL,  
    is_role       BOOLEAN NOT NULL,  
  
    issuer        BIGINT NOT NULL REFERENCES principal(id),  
    p_from        BIGINT NOT NULL,  
    key_i         CHAR VARYING NOT NULL, -- may reference certificate(hash)  
    sign_i        CHAR VARYING NOT NULL,  
  
    revoker       BIGINT REFERENCES principal(id),  
    p_revoked     BIGINT,  
    key_r         CHAR VARYING, -- may reference certificate(hash)  
    sign_r        CHAR VARYING,  
  
    PRIMARY KEY(id),  
    UNIQUE(name,p_from)  
);
```

```
-- Graph structure on labels
```

```
CREATE TABLE label_link (  
    parent        BIGINT NOT NULL REFERENCES label_act(id),  
    child         BIGINT NOT NULL REFERENCES label_act(id),  
  
    UNIQUE(parent,child)  
);
```

```
-- Table holding delegations of roles to principals
```

```
CREATE TABLE delegation (  
    id            SERIAL,  
  
    subject       BIGINT NOT NULL REFERENCES principal(id),  
    role          BIGINT NOT NULL REFERENCES label_act(id),  
    p_from        BIGINT NOT NULL,  
    p_until       BIGINT NOT NULL,  
    to_delegate   BOOLEAN NOT NULL,  
  
    issuer        BIGINT NOT NULL REFERENCES principal(id),  
    key_i         CHAR VARYING NOT NULL, -- may reference certificate(hash)  
    sign_i        CHAR VARYING NOT NULL,
```

```
    revoker      BIGINT REFERENCES principal(id),
    p_revoked    BIGINT,
    key_r        CHAR VARYING, -- may reference certificate(hash)
    sign_r       CHAR VARYING,

    PRIMARY KEY(id)
);

-- Table holding domain-restrictions on delegations.
-- The comp=value pairs from the restrictions on a delegation taken together
-- form the distinguished name for the delegation's domain.
CREATE TABLE del_domain (
    del          BIGINT NOT NULL REFERENCES delegation(id),
    comp         VARCHAR(32) NOT NULL,
    value        VARCHAR(1024) NOT NULL,

    UNIQUE(del,comp)
);

-----
-- Insert BOOTSTRAP DATA

-- root user
INSERT INTO principal (id)
    SELECT nextval('principal_id_seq');

-- Give root a distinguished name.
INSERT INTO principal_name(principal, comp, value)
    VALUES (1, 'C' , 'IS');
INSERT INTO principal_name(principal, comp, value)
    VALUES (1, 'O' , 'deCODE genetics');
INSERT INTO principal_name(principal, comp, value)
    VALUES (1, 'CN', 'Root User');

-- CA user
INSERT INTO principal (id) SELECT nextval('principal_id_seq');

-- Give CA a distinguished name.
INSERT INTO principal_name(principal, comp, value)
    VALUES (2, 'C' , 'IS');
INSERT INTO principal_name(principal, comp, value)
    VALUES (2, 'O' , 'deCODE genetics');
INSERT INTO principal_name(principal, comp, value)
    VALUES (2, 'CN', 'Certification Authority');

-- activation for root role
INSERT INTO label_act (name, is_role, issuer, p_from, key_i, sign_i)
    VALUES ('root', true, 1, 0, '10001', '20001'); -- id=1

-- activation for role-manager role
INSERT INTO label_act (name, is_role, issuer, p_from, key_i, sign_i)
    VALUES ('role-manager', true, 1, 0, '10001', '20001'); -- id=2

COMMIT;
```