Reconfiguring Role Based Access Control Policies Using Risk Semantics

Benjamin Aziz* Simon N. Foley† John Herbert†
Garret Swart‡

*Department of Computing, Imperial College, London SW7 2AZ, UK
baziz@doc.ic.ac.uk
†Department of Computer Science, University College Cork, Cork, Ireland
{s.foley,herbert}@cs.ucc.ie
‡IBM Almaden Research Center, 650 Harry Road, San Jose, CA, USA
gswart@us.ibm.com

Abstract

We present a refined model for Role Based Access Control policies and define a risk measure for the model, which expresses elements of the operational, combinatorial and conflict of interest risks present in a particular policy instance. The model includes risk-reducing mechanisms corresponding to practical mechanisms like firewalls, stack checking, redundancy, and event tracking that are frequently used to reduce risks in real systems. We also define policy transformation operators that produce new policies that allow the behaviours of the old policy while potentially reducing the risk measure. Sequences of these operators can be used to find policies that are less risky but still implement the initial policy. An example is give for Grid computing.

1 Introduction

In general, Role-Based Access Control (RBAC) policies [18] are rigid; once the policy is constructed out of the basic elements of users, roles and permissions, it is hard to reason about the security implications of any changes that may be applied to the

initial setup. For example, what is the risk associated with adding the user Alice into an accountant's role, given that she is already in the sales role? Or, is it safe to host the accounting system on the same server as the sales system, on the basis of the users who may have access to the server?

Existing research to address this category of problems have ranged from the analysis of security configurations for potentially dangerous conflicts of interest [5, 19] to frameworks that provide support for dynamic RBAC policies [12, 21]. Such approaches typically regard security modelling as a binary problem: a system and/or its configuration is either secure or not.

Setting up and maintaining a security configuration requires an ongoing process of security evaluation so that any changes made will, at least, make the system no less secure than it was prior to any changes. Each time a configuration is changed there is a chance for something unexpected. At best, any change, prompted for instance by business growth or changing business dynamics, leads to a more secure system. In practice, reconfiguration is a balance of security against other attributes such as price, availability, reliability and performance. So, one may pose questions like "does reconfiguring my system with the intention of improving performance increase or decrease the risk of security failure?"

Using formal methods to analyse and verify properties of secure systems requires considerable specification effort. The cost of in-depth specification and subsequent security analysis may be justified for small critical security mechanisms such as authentication protocols [4, 14, 16] and security kernels [17]. However, such in-depth analysis would not scale well to the configuration of a large and/or complex application system.

We are interested in developing shallow and pragmatic security analysis methods for systems. This is achieved through the analysis of how a system is *configured*, rather than an analysis of its underlying mechanisms and protocols. Instead of

concentrating on detailed semantics and complete formal verification of components, we are concerned more with the ability to trace, at a practical level of abstraction, how component security requirements relate to each other and any overall security requirements. We believe that a complete security verification of a system is not achievable in practice; we seek some degree of useful feedback from an analysis to indicate how reasonable a particular system configuration is.

In this paper, we propose a new approach to security configuration, which is based on defining transformations for RBAC policies. The transformations are based on a refined model of RBAC policies that incorporates the notion of *risk*. Another model of RBAC policies based on risk was defined in [13]. However, the approach of [13] is different; the notion of risk adopted does not provide for risk-enhancing mechanisms and the model does not define any formal risk-based semantics for the different elements of RBAC policies, and the related transformations. In [10], Millen adapts Meadow's lattice-based Chinese Wall mechanism [9] to determine optimal configurations that can survive component failures. Rather than taking a binary approach to the ability to survive failure (in the sense of [10]), our approach allows a measurement of the relative risks of failure associated with a configuration.

The paper is organised as follows. In Section 2 we introduce the basic components of our refined RBAC model. Section 3 defines an interpretation of risk and Section 4 uses this interpretation to provide a risk based semantics for the refined RBAC model. In Section 5 we define policy transformations in terms of implementation and equivalence relations. Section 6 illustrates how an example of Grid access control policy administration can be interpreted within our configuration framework.

2 A Refined RBAC Model

Role-based access control (RBAC) [18] is widely used to provide access control in database management systems, operating systems and middleware architectures. In

RBAC, access rights (permissions) are associated with roles, and users are members of a set of roles. When a user is a member of a role, they gain all the permissions of that role in the system. This allows an organisation to model its security infrastructure along the lines of its business use cases, assigning a role to a set of use cases and assigning users to the roles associated with the use cases they need to perform. Adding the indirection permissions through roles can lead to a coarser grain of control but can make the system easier to administer as it reduces the state space for administrators. For example, if the thousands of permissions can be grouped into tens of roles, then we've greatly reduced the difficulty in administering users.

Role Based Access Control (RBAC) is usually defined in terms of *Users*, *Roles* and *Permissions* [18].

- *Permissions*: represent actions, capabilities, applications or any other active behaviour that can be "performed" and, to which, we intend to control authorisation. We write the set of permissions as $perm \in PERM$.

- *Roles*: roles are logical groupings of permissions that reflect a particular task that can be assigned to some user. In our model, we assume that roles do intersect in their underlying permissions. We write the set of all roles as $role \in ROLE$.

- *Users*: users include humans or any other entity that can be assigned a role. We write the set of all users as $user \in USER$.

Based on these components, a standard RBAC policy $pol \in POLICY$ can be defined as a tuple, $(PERM, USER, ROLE, userRoles, rolePerms)$, such that:

$$\begin{aligned} userRoles : USER &\rightarrow \wp(ROLE) \\ rolePerms : ROLE &\rightarrow \wp(PERM), \end{aligned}$$

The former mapping assigns each user a set of roles and the latter assigns a set of permissions to each role. In the rest of this section, we refine the model of RBAC policies to include additional elements that model practical security measures.

2.1 Containers

The first such element is that of *containers*, which are defined as execution contexts that are shared by sets of computations. A container might represent a subnet, a server, a virtual machine inside a server, a process inside a virtual server, a container inside an application server, or a distributed transaction. Each container is implemented by a set of mechanisms, some inherent in the implementation of the container and some that are used to represent ad-hoc mechanisms that are used to 'harden' operations/permissions. They serve to protect the computations from each other in such a way that a failure in one computation does not compromise the safety of the container. We write the set of all containers as $cont, cont' \in CONTAINER$ and the set of mechanisms as $mech, mech' \in MECH$.

If a container is to be effective it must utilize permissions to perform actions. Thus we associate with each container the set of permissions it utilizes:

$$containerPerms : CONTAINER \rightarrow \wp(PERM)$$

and the set of mechanisms used to separate computations in the container:

$$containerMechs : CONTAINER \rightarrow \wp(MECH)$$

For example, the Stackguard compilation tool [3] compiles programs and incorporates code to defend against likely stack smashing attacks. Therefore, given a container c with $containerPerms(c) = \{Apache\}$, then having $containerMechs(c) = \{Stackguard\}$ would represent a more robust version of Apache. If, in addition,

the container running Apache were running on a secure system, such as security enhanced Linux, then $containerMechs(c) = \{selinux, Stackguard\}$ would represent an even more secure version of Apache. Other examples of protection mechanisms that can be included within a configuration include ad-hoc mechanisms such as TCP-wrappers [20] and more general mechanisms such as a Java Virtual Machine running a security manager.

In reality, certain mechanism-permission and mechanism-mechanism compositions do not make sense, for example a hardware network firewall doesn't compose with a stack frame checker. To avoid such bad compositions, we introduce the special commutative relation:

$$\psi_c \in (MECH \cup PERM) \times (MECH \cup PERM)$$

Any compatible mechanisms or permissions are in the relation. For a policy, *pol*, to be compatible with ψ_c it must be the case that

$$\begin{aligned} \forall c \quad &\in \quad pol.CONTAINER, \\ \forall m, m' \quad &\in \quad pol.containerMechs(c), \\ \forall p, p' \quad &\in \quad pol.containerPerms(c) : \\ (m, m') \in \psi_c \wedge (m, p) \in \psi_c \wedge (p, p') \in \psi_c. & \end{aligned}$$

For example, $(TCP\_wrapper, SMTP) \in \psi_c$, since a TCP wrapper can be composed with SMTP to protect email traffic, but $(JVMS, Apache) \notin \psi_c$, since it does not make sense to apply a Java Security Manager to the Apache server.

2.2 Roles

Like containers, roles are also composed from a set of permissions that are usually executed together, however in a use case rather than in an execution environment.

However, unlike a container, a role does not execute the activities underlying its permissions, but rather makes available for utilisation the permissions (and their resources) to any users that may be assigned to that role. More precisely, roles are the means through which users can avail of a subset of permissions associated with that role. For example, the role *Accountant* within Bank of America may have the specific permissions to check funds, prepare financial statements and make presentations.

Also just as containers have mechanisms to strengthen security inside a container, we extend roles with mechanisms to strengthen the security of that role. These mechanisms can include a diverse set of monitoring and checking tools, e.g., monitoring the activity of users in certain roles in real time using intrusion detection techniques such as [2] to predict if an attack is suspected. Another mechanism might restrict the activation of a role to users physically within a security perimeter. We choose the role mechanisms from the same set $MECH$, but we use a different typing mechanism (though having the same form as ψ_c), ψ_r. ψ_r is used along with a new relation:

$$roleMechs : ROLE \rightarrow \wp(MECH)$$

such that, $\forall r \in ROLE, \forall m, m' \in roleMechs(r), \forall p, p' \in rolePerms(r) : \ (m, p) \in \psi_r \wedge (p, p') \in \psi_r \wedge (m, m') \in \psi_r$. Intuitively, ψ_r maintains the correctness of the compositionality of permissions and mechanisms inside roles.

2.3 RBAC Policies

Based on the refined components of our RBAC model given in the previous sections, we define our extended RBAC policies as a tuple consisting of $USER$, $ROLE$, $PERM$, $userRoles$, and $rolePerms$, plus the new sets, $CONTAINER$ and $MECH$, and three new mappings, $containerMechs$, $containerPerms$ and $roleMechs$. The only restriction is that any permission that is accessible to a user must be contained

in at least one container and the compatibility rules given in ψ_r and ψ_c must hold.

A standard RBAC policy can be encoded into our refined model by using one container per permission, placing that permission in $containerPerms$ and finally, using no mechanisms for roles or containers.

3 Risk

We define in this section the concept of risk and we formalise its measure on our extended policy. Our concept of risk combines our intuition about the likelihood of a failure occuring and the consequence of that failure. These two concepts are intertwined: as the consequence of a failure in a policy increases, the likelihood will also increase, as the policy becomes a better target for hackers.

Intuitively, we deal with three main elements of risk:

- The first is *operational*; indicating the possibility of RBAC entities deviating from their specified operational behaviour. Such deviation may affect security properties of the entity, such as the failure in protecting data privacy by causing that data to be output over public communication channels. It may also affect quality properties, such as the failure to deliver information at a minimum quality of service (QoS) standard. Our notion of operational risk is linear; for example, consider a role that has two permissions, where the first could be misused to cause a privacy breach while the second could be misused to commit fraud; in this case the role has an operational risk level of the privacy and fraud vulnerabilities combined. Throughout the paper, we are interested only in security aspects of the operational risk.

- The second element of risk is *combinatorial*; it expresses the possibility of vulnerabilities that appear only as a result of the *combination* (co-existence) of a number of permissions together in the same container or role. Again, we are

interested in this paper in the security aspect of the combinatorial risk. The combinatorial risk level resulting from the co-existence of permissions within a particular role is perceived as the threat arising from the *collective usage* of those permissions by any of the users belonging to that role or the containers acting on the permissions. For example, a role with two permissions each utilising a server up to 75% of the server's capacity could introduce a denial of service vulnerability when utilised at the same time. This interpretation of the combinatorial risk is different within the context of a container, where the co-existence of multiple permissions within a container introduces threats resulting from the *collective management* of those permissions by the container. For example, a container running an insecure operating system underneath its permissions may introduce privacy vulnerabilities, compared to another container running on an operating system with multilevel security. In any case, combinatorial risk is non-linear. For example, it is not possible to infer the denial of service vulnerability of the role described above, by simply knowing that one of the two permissions has privacy vulnerability, the other fraud vulnerability, and then composing the two.

- The final risk element arises from *conflicts of interest*, which normally exist among competing roles and is controlled using separation of duties [7, 11, 18]. For example, one role may represent the auditing function and another the accounts payable. Having a single user in both roles may represent a conflict of interest as the same user could approve a payment to a bogus supplier and later, as part of an audit, confirm that the payment met corporate guidelines.

In order to formalise a measure of risk, we define *security threats* as a finite set, $Rset = \{l_1, l_2, \ldots, l_n\}$. For simplicity, we shall take these in English words, like $l_1 = ddos\_attack$, $l_2 = buffer\_overflow$, $l_3 = privacy$ etc. Then we define the lattice of risk levels, $L = (Rset, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$, where $\bot$ denotes the safest element in the

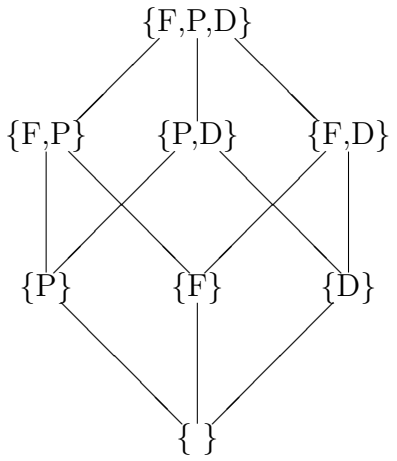

Figure 1: A lattice of the F, P and D risk levels.

lattice and $\top$ denotes the riskiest element. For example, assume that F is the risk level of fraud, P is the risk level of privacy breach and D is the risk level of denial of service, then Figure 1 illustrates one such lattice of risk levels ordered by subset inclusion.

In order to be able to denote the level of each of the operational, combinatorial and conflict of interest risks as described above in terms of the lattice of risk levels, we define the following functions:

$$\begin{aligned} \alpha &: (PERM \cup MECH) \rightarrow L \\ \beta &: \wp(PERM) \rightarrow L \\ \gamma &: \wp(ROLE) \rightarrow L \end{aligned}$$

The α function defines operational risk levels for permissions and mechanisms, whereas the β function defines the combinatorial risk level of different permissions co-existing together (represented as a set). Finally, the γ function represents a conflict of interest relation, where the risk of such a conflict is based on the roles within domains that a single user may be assigned.

4 Risk Measures

In the following sections, we define measures of risk for the refined RBAC model introduced in the previous section. This measure reflects the operational, combinatorial, and conflict of interest risk elements as discussed earlier. To make the description easier, we look first at the risk in containers, domains, and then users, and finally combine them to form a single risk measure for an RBAC policy.

4.1 Container Risk

We define the operational risk in a container inside a policy based on the permissions held by the container, the security mechanisms used in the container and the function α defined in the previous section:

$$\mathcal{R}_c^{op}(cont) = \bigsqcup_{perm \in containerPerms(cont)} \alpha(perm) \quad \sqcap \quad \bigsqcap_{mech \in containerMechs(cont)} \alpha(mech)$$

The risk in a container increases with the number of permissions being used in that container. This reflects the fact that both the risk that something will go wrong inside the container and the consequences of multiple failures will increase, which is reflected by the first least upper bound calculation. On the other hand, the risk level is lowered by using security mechanisms to implement the container. This decrease is reflected by taking the greatest lower bound of the resulting permissions with the risk levels of the mechanisms in use. If the policy is valid, the mechanisms (permissions) in a container must be compatible with each other with the corresponding permissions (mechanisms).

For example, an auditing mechanism to improve fraud detection (adm) could have a risk level of $\{P, D\}$, a cryptographic mechanism to improve privacy ($crypto$) might have a risk level of $\{F, D\}$ and a syn-cookies mechanism [1] to lessen DDOS vulnerabilities (syn) could have risk level $\{F, P\}$. We might use those mechanisms

to run a sensitive application for which $\alpha(app) = \{F, P, D\}$. By taking the greatest lower bound of the extra mechanisms of *adm*, *crypto* and *syn*, as well as *app*, the risk level of the policy is lowered. For example:

$$\begin{aligned}
\mathcal{R}_c^{op}(\{app\}) &= \{F, P, D\} \\
\mathcal{R}_c^{op}(\{adm\}, \{app\}) &= \{D, P\} \\
\mathcal{R}_c^{op}(\{syn, adm\}, \{app\}) &= \{P\} \\
\mathcal{R}_c^{op}(\{crypto, syn, adm\}, \{app\}) &= \{\}
\end{aligned}$$

The combinatorial risk in a container is simpler because the function β works directly on the set of permissions used inside the container:

$$\mathcal{R}_c^{cmb}(cont) = \beta(containerPerms(cont)) \sqcap \bigsqcap_{mech \in containerMechs(cont)} \alpha(mech)$$

For example, a container may have permission to delete a user and permission to refund money to a user. Having both those permissions together gives the container the opportunity to commit fraud and to hide the fact by deleting the evidence. This vulnerability must be encoded in the β function.

4.2 Roles

Roles also introduce an operational risk element using the same function α as above:

$$\mathcal{R}_r^{op}(role) = \bigsqcup_{perm \in rolePerms(role)} \alpha(perm) \sqcap \bigsqcap_{mech \in roleMechs(role)} \alpha(mech)$$

As with containers, having many permissions within a single role will increase the risk level of that role since the operational risk levels of individual permissions join together. However, adding mechanisms to a role lowers this risk level. For example,

given a role with permissions $pay\_supplier, refund\_customer$, users with this role could commit fraud by paying bogus suppliers or refunding money to their friends. One can imagine a mechanism, say $limit\_remits$, that limits a user to issuing a maximum of $1000 per day in funds to either suppliers or customers. By attaching that mechanism to the role, we can reduce the operational risk.

As one may expect, roles also have combinatorial risk defined using β:

$$\mathcal{R}_r^{cmb}(role) = \beta(rolePerms(role)) \quad \sqcap \prod_{mech \in roleMechs(cont)} \alpha(mech)$$

For example, a role may have permissions to execute two applications, each of which could consume up to 75% of the processor's computational resources. Executing both together will most likely cause a denial of service that is captured in β. However, that particular risk can be mitigated by using a special *sequential* mechanism which prevents the user utilizing the role from having more than one application active at a time.

4.3 Users

Unlike containers and roles, users have only one element of risk; i.e. the conflict of interest risk, which we define using the special function, γ, as follows:

$$\mathcal{R}_u^{conf}(user) = \gamma(userRoles(user)))$$

Note that while one could imagine mitigating this risk by some user mechanism (to augment the existing container and role mechanisms) in most systems this is not possible because it is difficult to track users as they switch roles. This is often due to the fact that these roles are actually implemented as separate users using disconnected login systems.

4.4 Policies

The risk semantics of a policy is defined by joining all of the risk components defined for its components:

$$\begin{aligned}\mathcal{R}_p(pol) \;=\; & \bigsqcup_{user \in pol.USER} \mathcal{R}_u^{conf}(user) \\ \sqcup\; & \bigsqcup_{role \in pol.ROLE} \mathcal{R}_r^{op}(role) \sqcup \mathcal{R}_r^{cmb}(role) \\ \sqcup\; & \bigsqcup_{cont \in pol.CONTAINER} \mathcal{R}_c^{op}(cont) \sqcup \mathcal{R}_c^{cmb}(cont)\end{aligned}$$

5 Policy Transformation

In this section, we define an *implementation* and an *equivalence* relation among policies as well as a set of policy transformations that produce policies that implement the input policy but have a different risk profile than the input policy. These transformations may be used in sequence to produce a policy with minimum risk.

5.1 The *Implements* Relation

The *implements* relation allows us to formalise the idea that a new policy preserves the permissions granted to a set of users in an older policy, while allowing for arbitrary new elements (users, containers, roles and permissions) to appear in the new policy. More formally, we define the *implements* relation as follows.

Definition 1 (The *implements* relation)

Given two policies $pol, pol' \in POLICY$*, we say that* pol' implements *pol (or pol is*

implemented by pol′), written as pol ▶ pol′, if the following holds:

$$\forall u \in pol.USER, p \in pol.PERM,$$
$$\exists r \in pol.ROLE :$$
$$r \in pol.userRoles(u) \wedge p \in pol.rolePerms(r)$$
$$\Rightarrow \exists r' \in pol'.ROLES : r' \in pol'.userRoles(u) \wedge p \in pol'.rolePerms(r')$$

Intuitively, every permission held by a user u in *pol* is also held by u in *pol′*. Note that this does not mean that the $USER$ or $PERM$ sets have to be identical in *pol* and *pol′*, as *pol* may contain users that have no permissions and permissions that no users have been granted that do not appear in *pol′*. Likewise, *pol′*, may contain new users and permissions as these do not inhibit its ability to implement *pol*.

5.2 The Equivalence Relation

Our equivalence relation captures scenarios where two policies are deemed to have the same risk semantics and they both implement each other. More formally, we define the equivalence of two policies as follows.

Definition 2 (The *equivalence* relation)
Two policies, pol and pol′, are said to be equivalent, written as pol ≐ pol′, if pol and pol′ differ only in the following:

- *The renaming of roles or containers*
- *The existence of roles with no permissions or roles with no users assigned*
- *The existence of users that have no permissions*
- *The existence of containers that have no permissions*
- *The existence of mechanisms that are unused by any container or role*

Moreover, one can prove the following properties.

Theorem 1 *Assuming* $pol \backsimeq pol'$*, then:*

1. $\mathcal{R}_p(pol) = \mathcal{R}_p(pol')$
2. $\forall pol'' \in POLICY : \quad pol'' \blacktriangleright pol \;\Leftrightarrow\; pol'' \blacktriangleright pol'$

Proof sketch. The proof of 1. is by induction over the definition of $\backsimeq$. Essentially, the definition of $\backsimeq$ involves either the renaming of permissions, the addition of roles/containers with no permissions or the addition of mechanisms that are unused by any container or role. According to our definition of risk measures in Section 4, none of these changes will induce any change in the risk value of a policy. As for part 2., we can show that this holds by showing that the definition of $\backsimeq$ preserves the definition of $\blacktriangleright$. This can be shown by induction over the definition of $\backsimeq$. □

5.3 Mechanism Replacement

The first technique we consider is quite straightforward; replacing the mechanisms used to reduce risk in containers and roles. The resulting policy trivially implements the input policy because mechanisms do not affect the ability of a user to make use of a granted permission. This is defined by a transformation $\mathcal{M}_c$ and $\mathcal{M}_r$:

$$\begin{aligned} \mathcal{M}_c &: POLICY \times CONTAINER \times \wp(MECH) \quad \rightarrow POLICY \\ \mathcal{M}_r &: POLICY \times ROLE \times \wp(MECH) \qquad\quad \rightarrow POLICY \end{aligned}$$

where the given set of mechanisms replace the mechanisms used in the given container or role. The container or role must exist within the policy and must be compatible using ψ_c or ψ_r with the existing permissions for the container or role.

Lemma 1 *Given a policy, pol, then the* $\mathcal{M}_c, \mathcal{M}_r$ *transformations preserve the* implements *relation, as follows:*

$$\forall pol, c \in pol.CONTAINER, r \in pol.ROLE, mset \in \wp(MECH) :$$
$$pol \blacktriangleright \mathcal{M}_c(pol, c, mset) \ \wedge \ pol \blacktriangleright \mathcal{M}_r(pol, r, mset)$$

Proof sketch. For the case of $\mathcal{M}_c$, given that the set of permissions in the new policy resulting from the transformation is the same as the old set, and by Definition 1 above for the *implements* relation, it is possible to show that $pol \blacktriangleright \mathcal{M}_c(pol, c, mset)$. Similar line of reasoning can be applied to $\mathcal{M}_r$. □

Depending on the mechanisms in place and the new mechanisms replacing the old, these transformations may reduce the overall risk of the policy. Note that since risk is not a total order, changing mechanisms can cause the resulting risk to be incomparable with the original risk. This can make doing local search for sequences of transformations difficult.

5.4 Reallocation of Permissions

The reallocation of permissions between containers or roles is necessary whenever we seek to change the combinatorial risk levels inside those containers or roles. Rearranging permissions can also increase the number of compatible mechanisms used to reduce risk.

We define the reallocation of permissions within containers as a function,

$$\mathcal{U}_c : POLICY \times CONTAINER \times PERM \times$$
$$(CONTAINER \cup \{\perp\}) \rightarrow POLICY$$

The resulting policy moves the given permission, which must be in the permissions set associated with the first container, to the second container. If the second container is $\perp$ then the permission is simply removed from the first container. If the second container is given, the permission must be compatible using ψ_c with the other

permissions and mechanisms used in that container. If the second container is not currently a member of $pol.CONTAINER$ then it is added.

For example, assume we want to remove one of two permissions, $perm_1$, $perm_2$, currently co-existing in a container, $cont$, to a new empty container, $cont_0$. The transformation $\mathcal{U}_c(pol, cont, perm_2, cont_0)$ will result in a policy holding two containers, $cont$ holding $perm_1$ and $cont_0$ holding $perm_2$. Since the allocation of permissions to containers does not affect the set of permissions a user is assigned, the resulting policy implements the input policy.

Lemma 2 *Given a policy, pol, then the* $\mathcal{U}_c$ *transformation preserves the* implements *relation, as follows:*
$\forall pol, c \in pol.CONTAINER, p \in containerPerms(c), c' \in CONTAINER \cup \{\perp\}$:
$pol \blacktriangleright \mathcal{U}_c(pol, c, p, c')$

Proof sketch. Given that $\mathcal{U}_c$ only moves permissions across the different containers, the set of permissions allocated to a particular role remains the same, and by Definition 1 above for the *implements* relation, it is possible to show that $pol \blacktriangleright \mathcal{U}_c(pol, c, pset, c')$. □

The next transformation defines the reallocation of permissions among roles:

$$\mathcal{U}_r : POLICY \times (ROLE \cup \{\perp\}) \times PERM \times ROLE \rightarrow POLICY$$

whose value is a new policy, pol' where the first role, r, no longer contains the given permission in $pol'.rolePerms(r)$ and where the target role, r', which may be a new role, contains that permission in $pol'.rolePerms(r')$. If the first role is $\perp$, then the given permission is simply added to the given role $pol'.rolePerms(r')$ and no changes are made to the $pol'.userRoles$. In all cases the new permission must be compatible with the existing permissions and policies for the target role using ψ_r.

This transformation is somewhat more difficult because the assignment can affect the set of permissions assigned to a user. The trick here is to give any user that was allowed access to the old role, access to the new role as well. So, we impose the following condition whenever $\mathcal{U}_r$ is used:

$$\forall u \in USER, pol \in POLICY, r \in pol.userRoles(u) :$$
$$r \in \mathcal{U}_r(pol, r, p, r').userRoles(u) \quad (1)$$

Now we can show that the implementation relation is preserved by $\mathcal{U}_r$.

Lemma 3 *Given a policy, pol, then the* $\mathcal{U}_r$ *transformation preserves the* implements *relation up to condition (1) above and as follows:*
$\forall pol, r \in (pol.ROLE) \cup \{\bot\}, p \in \wp(PERM), r' \in ROLE : \quad pol \blacktriangleright \mathcal{U}_r(pol, r, p, r')$

Proof sketch. According to condition (1) above, a user in the new policy still has access to the old set of permissions in the old policy, since roles are preserved across the $\mathcal{U}_r$ transformation. Therefore, considering Definition 1 for the *implements* function, it is possible to show that $pol \blacktriangleright \mathcal{U}_r(pol, r, pset, r')$. □

5.5 The Reconfiguration Problem

The reconfiguration problem is concerned with finding a policy with minimum risk that is an implementation of some initial high-risk policy using the transformations of the previous sections over finite sets of mechanisms and permissions.

Definition 3 (Minimum Risk Policy Reconfiguration)
Given an initial policy, pol_0*, a finite set of mechanisms,* $MECH_f$*, a finite set of permissions,* $PERM_f$*, a set of transformations,*
$\mathcal{T} \in \{\mathcal{M}_c(\_, cont, mechset), \mathcal{M}_r(\_, role, mechset),$
$\mathcal{U}_c(\_, cont, perm, cont'), \mathcal{U}_r(\_, role', perm, role)\}$

where,

$cont \in pol_0.CONTAINER$,

$cont' \in pol_0.CONTAINER \cup \{\perp\}$,

$role \in pol_0.ROLE$,

$role' \in pol_0.ROLE \cup \{\perp\}$,

$perm \in PERM_f$,

$mechset \in \wp(MECH_f)$

then the minimum risk policy reconfiguration *problem is concerned with finding a new policy,* pol_{min}*, such that:*

1. $pol_{min} \in \mathcal{T}^*(pol_0)$[1]
2. $\mathcal{R}_p(pol_{min}) = \bigsqcup_{pol \in \mathcal{T}^*(pol_0)} \mathcal{R}_p(pol)$

The following theorem shows that the minimum risk policy has indeed an improved risk level compared to the initial policy, that the former implements that latter and that it is unique up to the $\backsimeq$ equivalence.

Theorem 2 *Given an initial policy,* pol_0*, and a minimum risk policy,* pol_{min}*, that is a result of transforming* pol_0 *according to Definition 3, then the following properties hold true:*

1. $\mathcal{R}_p(pol_{min}) \sqsubseteq \mathcal{R}_p(pol_0)$
2. $pol_0 \blacktriangleright pol_{min}$
3. $\forall pol'_{min} \in \mathcal{T}^*(pol_0) : pol'_{min} \backsimeq pol_{min}$

Proof sketch. The proofs of 1. and 3. follow directly from the definition of a least upper bound in a partially ordered set. On the other hand, 2. can be proven using Lemmas 1,2 and 3. □

[1] $\mathcal{T}^*$ is the reflexive transitive closure of $\mathcal{T}$.

6 Example: A Risk-Based RBAC Model for Grid Computing

Grid computing has emerged in recent years as an authoritative model of large-scale resource sharing among *virtual organisations*; collections of users/institutes grouped together under a common policy/interest [6]. Like any other model of computation, security is of paramount importance and issues of authentication, authorisation and access control are mainly considered when designing/analysing the security of the Grid. One solution to such issues, which has been adopted is to construct RBAC models for grid computations [8, 15, 22]. This comes as no surprise since RBAC models have been shown to be more flexible and cheaper to maintain compared to the classical MAC and DAC models.

In this section, we demonstrate the applicability of our risk-based RBAC model to Grid computing. For this purpose, we define a *virtual organisation* based on the simplified architecture illustrated in Figure 2. According to this architecture, a virtual organisation consists of a set of users, each assigned to a specific set of roles. The organisation also owns a set of containers, within which permissions and mechanisms are managed. The management of the overall organisation is performed through an *organisation manager*, whose responsibilities include the following: First, it can accept requests from external users to join in and requests from internal users to leave. Second, it configures and maintains the RBAC policy of the organisation according to some acceptable level of risk. Finally, the organisation manager may query and update the local policy database with data about the current users, containers, roles and permissions of the organisation, as well as its current configuration.

Our main concern is with the issue of user mobility among different organisations and how this mobility affects the current policy configuration of each organisation. A user may either *join* a destination organisation and/or leave its home organisation.

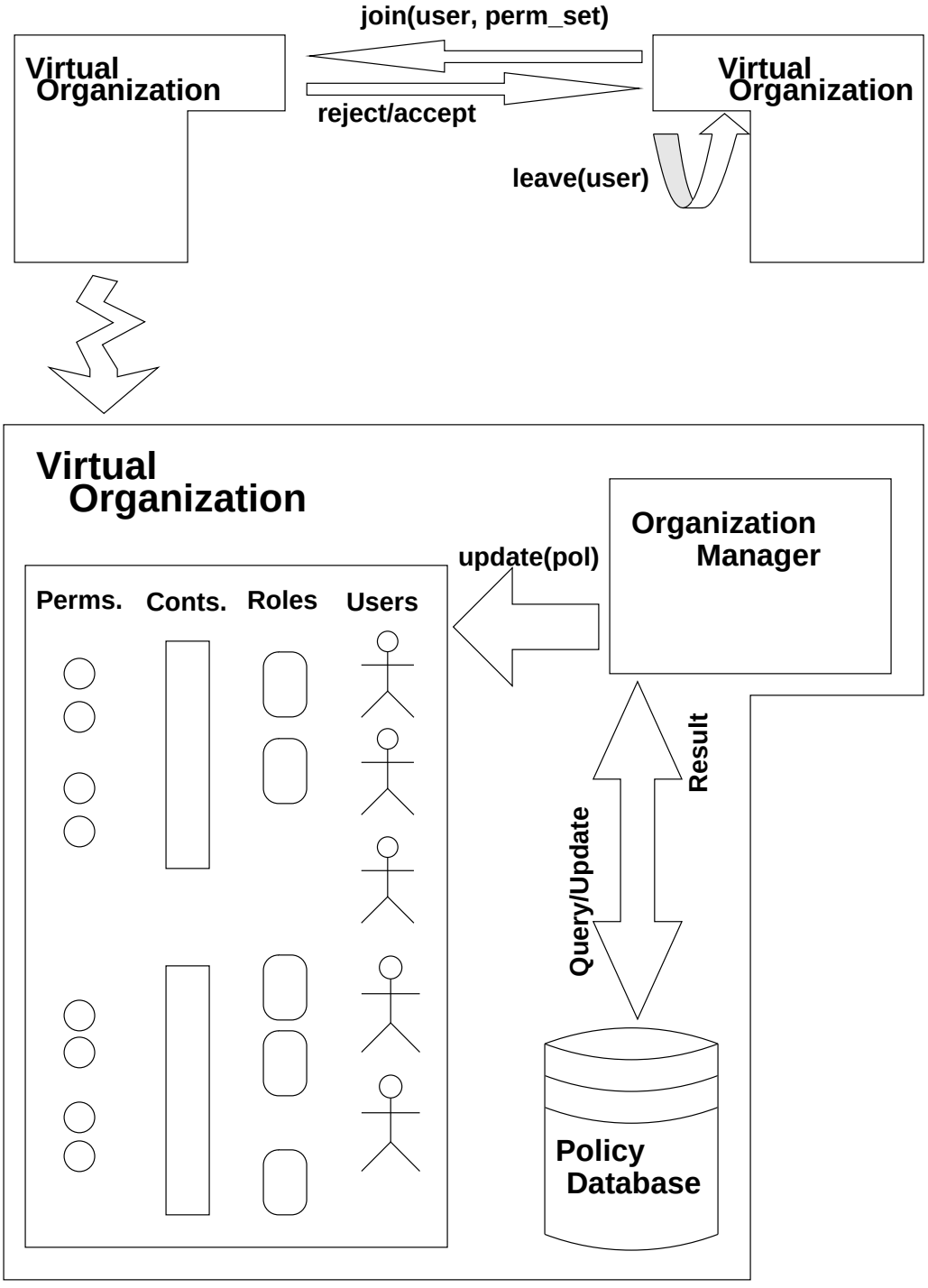

Figure 2: The architecture of the Grid.

The procedure of joining an organisation with the request to use a permission is initiated by the manager of the user's home organisation (*HomeManager*), which invokes the *join(user,perm)* operation on the manager of the destination organisation (*DestManager*). We assume that *HomeManager* in invoking the *join* operation also implicitly certifies the request. Once the *join* operation is invoked, *DestManager* then queries the current policy of its organisation, *pol*, and redesigns *pol* to a new policy, pol_{new}, reflecting the presence of the new user and preserving the definition of the *implements* relation:

$$
\begin{aligned}
&join(user, perm) \equiv \\
&\quad (pol_{new}.USER = pol.USER \cup \{user\}) \ \wedge \\
&\quad (\exists r \in pol_{new}.ROLES : r \in pol_{new}.userRoles(user) \ \wedge \ perm \in role.Perms(r)) \ \wedge \\
&\quad (pol \blacktriangleright pol_{new}) \qquad (2)
\end{aligned}
$$

Once (2) is satisfied, *DestManager* then seeks to find $pol_{min} \in \mathcal{T}^*(pol_{new})$; the solution to the minimum risk reconfiguration problem (Definition 3), given a set of transformations, $\mathcal{T}$, currently held in the database of the destination organisation. Finally, if such a solution exists, then *DestManager* informs *HomeManager* of the success of the join operation, otherwise, it rejects the operation. *HomeManager* then decides to invoke the *leave*(*user*) operation (which it may invoke at any other time) and update its current policy, *pol*. The definition of *leave*(*user*) is delicate; the operation simply results in computing a new policy, pol_{new}, such that *user* now has the following property:

$$
\begin{aligned}
&leave(user) \equiv \\
&\quad \forall r \in pol_{new}.ROLES : \\
&\quad r \in pol_{new}.userRoles(user) \Rightarrow pol_{new}.rolePerms(r) = \{\} \qquad (3)
\end{aligned}
$$

According to (3), *user* has no permissions in the new policy, pol_{new}, and according to Definition 2, we have that $pol = pol_{new}$. Furthermore, according to Theorem 1, pol_{new} has a risk level that is equal to the risk level of pol.

7 Conclusion

In this paper, we presented a refined model for RBAC policies consisting of permissions, containers, roles and users and defined a measure of risk the model that expresses elements of operational, combinatorial and conflict of interest risks. We then defined implementation and equivalence relations over policies. These relations are then used to define transformations that produce policies implementing the original policy. Based on these relations and the corresponding transformations, we defined the problem of finding the minimal risk policy configuration. Finally, to demonstrate the applicability of our theoretical framework, we presented an example of the minimal risk policy configuration problem for the case of Grid computing.

For the future, we would like to extend the idea of transformations and measures to handle other quantitative properties of security policies, such as complexity, probability, cost and performance, and then study the different tradeoffs among these properties within different security policies. We are also planning to extend the semantics to include other variations of RBAC models, mainly RBAC_1 and RBAC_2 with domain and role hierarchies.

8 Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments on this paper. This work was supported by the Boole Centre for Research in Informatics, University College Cork under the HEA-PRTLI scheme and by the Enterprise Ireland Basic Research Grant Scheme (SC/2003/007).

References

[1] D.J. Bernstein. Syn cookies. URL http://cr.yp.to/syncookies.html.

[2] E. Bertino et al. Intrusion detection in rbac-administered database. In *21st Annual Computer Security Applications Conference (ACSAC)*, 2005.

[3] C. Cowan et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the* 7^{th} *Usenix Security Symposium*, San Antonio, Texas, U.S.A., January 1998. Usenix Association.

[4] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[5] S.N. Foley. A non-functional approach to system integrity. *IEEE Journal on Selected Areas in Communications*, 21(1), Jan 2003.

[6] I.T. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid - enabling scalable virtual organizations. *The Computing Research Repository*, cs.AR/0103025, March 2001.

[7] V. Gligor et al. On the formal defintion of separation of duty policies and their composition. In *Symposium on Security and Privacy*. IEEE Press, 1998.

[8] M. Lorch, D.G. Kafura, and S. Shah. An xacml-based policy management and authorization service for globus resources. In *Proceedings of the* 4^{th} *International Workshop on Grid Computing*, pages 208–212, Phoenix, AZ, USA, November 2003. IEEE Computer Society.

[9] C. Meadows. Extending the brewer-nash model to a multilevel context. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 95–102, Oakland, California, USA, May 1990. IEEE Computer Society Press.

[10] J. Millen. Local reconfiguration policies. In *IEEE Symposium on Security and Privacy*, pages 48–56, 1999.

[11] M.J. Nash and K.R. Poland. Some conundrums concerning separation of duty. In *IEEE Symposium on Security and Privacy*, pages 201–207. IEEE, 1990.

[12] G. Neumann and M. Strembeck. An approach to engineer and enforce context constraints in an rbac environment. In *Proceedings of the* 8th *ACM Symposium on Access Control Models and Technologies*, pages 65–79, Como, Italy, June 2003. ACM Press.

[13] N. Nissanke and E. Khayat. Risk based security analysis of permissions in rbac. In L. Javier García-Villalba Eduardo Fernández-Medina, Julio César Hernández Castro, editor, *Proceedings of the* 2nd *International Workshop on Security In Information Systems*, Security In Information Systems, pages 332–341, Porto, Portugal, April 2004. INSTICC Press.

[14] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[15] W. Qiang, H. Jin, X. Shi, D. Zou, and H. Zhang. Rb-gaca: A rbac based grid access control architecture. In Minglu Li, Xian-He Sun, Qianni Deng, and Jun Ni, editors, *Proceedings of the* 2nd *International Workshop on Grid and Cooperative Computing*, volume 3032 of *Lecture Notes in Computer Science*, pages 487–494, Shanghai, China, December 2004. Springer Verlag.

[16] A.W. Roscoe. Intensional specifications of security protocols. In *Proceedings 9th IEEE Computer Security Foundations Workshop*, pages 28–38. IEEE Press, 1996.

[17] J. M. Rushby. The design and verification of secure systems. In *Proceedings 8th ACM Symposium on Operating System Principles*, December 1981. Available as ACM Operating Systems Review **15** 5.

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

[19] A. Schaad and D. Moffett. The incorportation of control principles into access control policies. In *Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, 2001.

[20] W. Venema. Tcp wrapper: Network monitoring, access control and booby traps. In *3rd UNIX Security Symposium*, 1992.

[21] W. Yamazaki, H. Hiraishi, and F. Mizoguchi. Designing an agent-based rbac system for dynamic security policy. In *Proceedings of the* $13^{13\text{th}}$ *IEEE International Workshops on Enabling Technologies, Infrastructure for Collaborative Enterprises*, pages 199–204, Modena, Italy, June 2004. IEEE Press.

[22] G. Zhang and M. Parashar. Dynamic context-aware access control for grid applications. In *Proceedings of the* 4^{th} *International Workshop on Grid Computing*, pages 101–108, Phoenix, AZ, USA, November 2003. IEEE Computer Society.