

FRAC: Implementing Role-Based Access Control for Network File Systems*

Aniruddha Bohra,[†] Stephen Smaldone, and Liviu Iftode

Department of Computer Science, Rutgers University, Piscataway NJ

{bohra,smaldone,iftode}@cs.rutgers.edu

Abstract

We present FRAC, a Framework for Role-based Access Control in network file systems. FRAC is a reference monitor that controls the message flow between file system clients and servers. FRAC supports role hierarchies, user sessions, and static and dynamic separation of duty constraints. It also allows administrators to define dynamic policies based on access history and the environment, e.g., time of day.

FRAC introduces a virtual control namespace (VCN) that provides an interface to query and update the state of the access control framework over the standard file system protocol. This namespace eliminates the need for executing specialized user agents either at the client or at the server. Therefore, FRAC does not require any modification to either the file system client or the file server. We have implemented FRAC for the widely deployed NFS protocol using FileWall, a file system proxy previously developed by us. Our experimental evaluation shows that FRAC imposes minimal overheads for the common file system operations.

1. Introduction

File management has traditionally been split between file owners and file system administrators. Owners manage file organization, naming, and protection, while administrators manage file system configuration, backup and recovery, hardware and performance, disk space, etc. As file systems grow in size, complexity, and criticality, managing them has become increasingly complex both for file owners and file system administrators. Significant research and software development effort has been devoted to simplify management. For file owners, tools like desktop search [6], semantic file systems [19, 17], stackable file systems [24], etc., have been developed. For administrators, tools for improved backup and recovery [1, 25], interposed request routing [3], union file systems [23], and federated file systems [22] have simplified file system management.

While research effort has been focused on management and organization, file access control has been largely ignored. Policies for file access are defined through file at-

tributes, e.g., permissions and access control lists, and their control is at the discretion of the file owner. From an administrator's perspective, file access control policies are very limited. Administrators have no simple way to express file access policies, such as: administrator access to a file is allowed only between 9 AM and 5 PM, a directory is accessible by all members of a department except visitors, etc.

In this paper, we present FRAC, a realization of the role-based access control model (RBAC) for network file systems. FRAC interposes on all file system messages between clients and servers, and enforces RBAC through message transformation. At the highest level, FRAC is a reference monitor that controls the flow of file system messages based on the RBAC policy.

In recent years, role-based access control (RBAC) has emerged as a model for enforcing dynamic access control policies across a wide range of enterprise resources [10, 5]. There are three main benefits of using RBAC. First, RBAC models have been shown to be "policy neutral", that is by using role hierarchies and constraints a wide range of security policies can be expressed. Second, security administration is simplified by using roles to organize access privileges. For example, if a user moves to a new function in an organization, the user's role can simply be reassigned. In contrast, without RBAC, permissions on individual files would have to be updated. Third, by using constraints on the activation of user assigned roles, the principle of least privilege [18] can be enforced. In fact, today, RBAC models have matured to the point where they are prescribed as a generalized approach to access control.

While RBAC is attractive, there are several reasons for the reluctance of file system administrators to adopt RBAC. First, network file systems are performance critical and user applications expect their performance to be similar to local file systems. Second, due to the reliance on standardized file-system interfaces, users and applications expect and tolerate no modifications to existing behavior. Finally, deployability of any new mechanism requires that it does not modify either the clients or the servers. The servers may be proprietary and closed source precluding any modifications to them, and clients may refuse or be unable to execute any

*This work is partially supported by National Science Foundation Grant No. CCR 0133366.

[†]Also with NEC Labs America

agents to support the modified access control protocols.

This paper presents the design and implementation of FRAC, a Framework for Role-based Access Control for network file systems. Previous efforts to implement RBAC either modified the client-visible attributes of individual files [7], required modifications to or execution of specialized agents at the server, clients, or both [8]. In contrast, FRAC requires no modifications to standardized file system protocols and maintains the existing file system interfaces for user interaction.

FRAC introduces a *virtual control namespace* (VCN) that provides a snapshot of the state of the access control system, and defines interfaces for users to update permissions and administrators to update access control policies over the standard file system protocol. The contents of file system objects in the VCN are generated dynamically by handlers defined by FRAC. By introducing an active component in the path of every file system message, FRAC enables fine-grained dynamic control over file system accesses. Since all file system operations, including permission and session updates, are performed over file system messages, FRAC evaluates the access control policy for each operation.

We implement FRAC over FileWall [21], a network middlebox for file systems, which provides support for specifying network file system policies through interposition, and evaluate it using an RBAC policy with support for role-hierarchies, delegation, and dynamic separation of duty constraints. We show through our evaluation that FRAC imposes low overheads while enforcing the role-based access control policy.

This paper makes the following contributions. First, we show how to implement and deploy RBAC for network file systems using a file system proxy, without modification to existing client and server software (including operating system software). Second, we demonstrate the use of virtual namespaces to provide agent-free deployment for access control. Finally, through our evaluation, we demonstrate that the RBAC mechanisms implemented by FRAC are efficient and do not significantly degrade client observed performance.

The rest of this paper is organized as follows. Section 2 presents a brief overview of the RBAC model and outlines a sample RBAC policy. Section 3 presents the FRAC design. We describe the FRAC implementation in Section 4 and evaluate it in Section 5. Section 6 discusses limitations and possible extensions to FRAC and Section 7 presents the related work. Finally, Section 8 concludes the paper.

2. Background

Role-Based Access Control: According to the recently proposed NIST RBAC standard [10], the basic concept of RBAC is that users and permissions are assigned to roles,

and users acquire permissions by being members of roles. The same user can be a part of multiple roles and a role may have multiple users. Therefore, there is a many-to-many relationship between users and roles. Hierarchical RBAC adds requirements for supporting role hierarchies, whereby senior roles acquire the permissions of the juniors, and junior roles acquire the user membership of seniors.

The RBAC user assignment maps the users to roles in a *user session*, subject to the static and dynamic separation of duty constraints. The roles available to a user are *specified* by an administrator in an RBAC policy, while roles are *activated* by users in a session. Permission assignment defines the privileges associated with roles for each operation on an object in the system. Permissions are defined as an access control matrix for all operation-object pairs where RBAC is applied.

Separation of duty constraints are used to enforce conflict of interest policies. Conflict of interest may arise as a result of a user gaining authorization for permissions associated with conflicting roles. The separation of duty constraints can be static or dynamic. Static separation of duty (SSD) constraints are specified in the RBAC policy defining restrictions on roles across all sessions. Dynamic separation of duty (DSD) are specified as a constraint on the user-role mapping for each session. Intuitively, the SSD constraints govern the *available* roles, while the DSD constraints govern the *active* roles in a session.

2.1. Access Control Example

In this section, we define an RBAC security policy (\mathcal{F}) applied to a network file system. We use this policy to illustrate how FRAC applies RBAC concepts and use it as a running example throughout this paper. This example illustrates four basic principles of RBAC. First, the principle of least privilege is enforced. Specifically, a user accesses files at the lowest privilege level she is assigned to that is required for accessing an object. Second, dynamic escalation of roles is defined across user sessions. Third, delegation of roles is illustrated through a dynamic time-based policy, and fourth, per-file access control policies, defined by the user, are enforced.

Principal Definition: We assume a system with four users, *USERS*, who are assigned to three roles, *ROLES*, which have a partial order relationship defined by *HIER*. Permissions (*PERMS*) are derived from the NFSv4 ACL model and apply to each file system object (files, directories, links, etc) in the context of NFS operations defined for the object. Formally, the principal sets are defined as:

$$USERS \leftarrow \{alice, bob, charles, root\}$$
$$ROLES \leftarrow \{user, developer, threat, admin\}$$
$$HIER \leftarrow \{user \leq developer \leq admin, threat\}$$
$$PERMS \leftarrow \{ALLOW, DENY, LOG, ALARM\}$$

Role	Users
user	alice, bob*, charles*, root*
developer	bob, charles, root*
admin	root
threat	NULL

Table 1. User assignment for \mathcal{F} .

	GETATTR	READ	WRITE	REMOVE
ALLOW	<i>user</i>	<i>user</i>	<i>developer</i>	<i>OWNER</i>
LOG	<i>admin</i>	<i>admin</i>	<i>admin</i>	<i>admin</i>
ALARM	<i>threat</i>	<i>threat</i>	<i>threat</i>	<i>threat</i>

Table 2. Subset of the FRAC AC matrix

User Assignment: User assignment defines the mapping between users and active roles. In a hierarchical RBAC system, the users are assigned to all roles below the initial assignment through inheritance. The default role assignments for \mathcal{F} are shown in Table 1. The users marked by a * in the table show the inherited role memberships.

While the user assignment provides a list of all available roles, the active roles are determined for each session subject to dynamic separation of duty constraints. \mathcal{F} includes a dynamic separation of duty constraint, which states that a user cannot acquire the *admin* role in a session with any other role (*user*, *developer*, or *threat*) active.

Permission Assignment: Permission assignment maps the roles to the set of permissions for operations on an object. The permission assignment is statically defined by the policy, and may be dynamically updated by the owner of the object.

Table 2 shows a subset of the permission assignment for a file. Here, the *user* role is allowed GETATTR and READ operations, while WRITE operation is permitted only for the *developer* role. The special tag *OWNER* is derived from the file attributes and only the owner (or the admin) is allowed to remove the file. If the operation is allowed, FRAC additionally evaluates the LOG permission and logs the operation if required. Also, if the operation is denied, the ALARM permission is evaluated to possibly generate an alarm for the system administrator.

The members of the role *user* can only access the files owned by them. Members of the role *developer* derive all the user properties and additionally can access all files owned by users in the developer role. Administrators belong to the *admin* role and have unrestricted access to the file system. However, all administrator actions that modify the file system are logged. Finally, a user can be made a part of the *threat* role, based on a dynamic rule. As an illustration, all users with role *user* who try to escalate their privileges to the role *admin* are threats and all updates made by members of this role are logged.

Session Management: A *session* determines the set of active roles assigned to a user. All role assignments, as well

as, dynamic separation of duty constraints are evaluated within a session. While the user may acquire multiple roles, the set of active roles within a session is fixed. If the user wishes to modify this set of roles, the old session must be terminated and a new session created with the new roles.

In our example policy, on session initiation, all users are part of the *user* role. If a user is a member of the *developer* role, she additionally retains the developer privileges. Modification of the active roles in a session can be *explicit* or *implicit*. For explicit modification of active roles, the user must initiate modifications to the role set through FRAC. In \mathcal{F} , the administrator delegates his role to members of the developer group during the non-work hours. That is, the members of the developer group can acquire administrator privileges between 5PM and 9AM. This delegation is often desired but is seldom implemented without explicit RBAC support. The escalation of *developer* to *admin* is performed explicitly, and a new session is created.

\mathcal{F} also supports an implicit session update using the environment. When the access control system determines a user is a threat, based on previously specified access patterns or attempted accesses to sensitive files, FRAC terminates the active user session and initiates a new session with the *threat* role. This update does not involve the user and prohibits access to the file system including any role updates. Administrator intervention is required to re-enable the user access.

3. FRAC Design

In this section, we describe the design of the FRAC access control framework. Our goal is to provide RBAC functionality in network file systems, for existing file server deployments with a diverse client population. By insisting on no server modifications, FRAC is readily deployable and provides RBAC functionality even for closed-source or proprietary file servers.

We further restrict our system to not require any *client* modifications. This precludes designs where a separate user agent executes on the client system to interact with the access control mechanisms. However, we believe that expecting a client to run any software other than the default is difficult. This requires administrators to install and maintain the software, and train the users in its use. On the other hand, standard file system interfaces are well understood and satisfy the principle of least astonishment (POLA), where the system does not surprise the user with modified or non-traditional interfaces.

3.1. Overview

FRAC implements an RBAC framework for network file systems by interposing on the client-server network path and transforming file system messages. At the highest level, it is a reference monitor that controls the flow of requests from clients to servers. FRAC assumes that the underlying

system provide minimal support for message capture and injection, for forwarding or denying requests, and attribute extraction and transformation to implement access control policies.

Figure 1 shows the flow of requests and responses through FRAC. The client generates file system requests for both real (solid boxes) and virtual (patterned boxes) objects. The requests for virtual objects are handled by a special VCN handler, which generates the file system responses. Requests for real objects are forwarded to the file server, which generates the responses sent to the client. All requests are subject to permission evaluation and deny responses are generated for all requests that are denied.

Policy Specification and Update: Access control policies must be defined before any accesses to the system are performed by clients. Therefore, to bootstrap the system, FRAC requires administrators to provide an initial policy description during startup. This initial description contains the principal definitions and access control constraints. Several policy specification languages have previously been proposed for RBAC. We use XACML [2], a standardized RBAC specification language that provides not only the core RBAC specification but also allows us to define temporal and dynamic access control policies.

During execution, FRAC provides interfaces for administrators to update access control policies, and for users to manage their sessions and permissions for the files they own. These interfaces are realized using a virtual namespace, called the Virtual Control Namespace (VCN), contained within the existing file system namespace (see Section 3.2).

State Maintenance: FRAC maintains persistent state, which must be available through the lifetime of the system and is essential for correct enforcement of RBAC. We identify two persistent state components for FRAC: (i) Policy definition, which includes user and role identifiers, role hierarchies, user assignment, and any separation of duty constraints. (ii) Permission assignment, which includes per-file access control matrix and owner information.

FRAC also maintains soft state, which is throw-away, and can be reconstructed from the persistent state and by observing the file system requests and responses. This state has two main components: (i) Session state, which consists of active user sessions and their currently active roles. FRAC assumes each user has at most one session on each machine. This creates a unique identifier for a user session, which is then used to identify the active roles associated with the session, and (ii) a Virtual File Handle mapping, which represents an association between FRAC generated client-visible file identifiers, distinct from server-defined file identifiers.

Permission Evaluation: FRAC maintains the state required for making access control decisions and generates

Algorithm 1 FRAC algorithm for implementing RBAC.

```

1:  $sid \leftarrow SESSIONS[MSG.IP][MSG.UID]$ ;
2:  $roles_{sid} \leftarrow ROLES[sid]$ ;
3:  $vfh \leftarrow MSG.FH$ ;
4:  $vfhentry \leftarrow VFHMAP[vfh]$ ;
5:  $fh \leftarrow vfhentry.fh$ ;
6: if  $fh = NULL$  then
7:    $vcnhandler(MSG, vfhentry.shadow)$ ;
8:   return
9: end if
10:  $shadow \leftarrow SHADOWFILES[vfhentry.shadow]$ ;
11:  $ownerid \leftarrow shadow.ownerid$ ;
12:  $perms \leftarrow shadow.perms$ ;
13:  $op \leftarrow MSG.OP$ ;
14:  $minrole \leftarrow perms[op]$ ;
15: for all  $r \in roles_{sid}$  do
16:    $found \leftarrow DFS(HIER, r, minrole)$ 
17:   if  $found = true$  then
18:      $MSG' \leftarrow MSG$ 
19:      $MSG'.UID \leftarrow ownerid$ ;
20:      $MSG'.FH \leftarrow fh$ ;
21:      $forward(MSG')$ ;
22:     return
23:   end if
24: end for
25:  $deny(MSG)$ ;
26: return

```

a boolean *ALLOW* or *DENY* verdict for each file system request. The requests that are allowed are forwarded to the file server. For requests that are denied, FRAC generates an appropriate deny response *without* contacting the server.

Algorithm 1 shows the algorithm used by FRAC for permission evaluation. The algorithm uses the incoming request message as input and uses attributes contained in the message to identify the set of active roles (Lines 1-2). These roles are maintained in order of increasing privilege level. It then extracts the virtual file handle (*vfh*) from the request message and uses this VFH to obtain the structure containing the real file handle (*fh*). If this mapping structure does not contain a FH, then the VFH refers to a VCN object, which is handled by the *vcnhandler* (Line 6-9).

For the case of real file system objects, the algorithm obtains the corresponding access control information (shadow file) (Line 10). The least privileged role required to perform *op* is identified from the permission map. Finally, a depth first search (DFS) is performed on the role hierarchy defined by the policy starting at the current role (Line 16). If the target role (defined by the permission) is found during the DFS, the message is allowed. Otherwise, if all active roles are exhausted, the request is denied.

While forwarding the message, FRAC replaces the UID and VFH in the request with the *ownerid* and real *fh* respectively and the message is forwarded to the server (Line 21). This ensures that the operation is performed as the owner of the file and the base file system permissions are still enforced. All other messages are denied (Line 25).

FRAC uses four tables to maintain the state required for permission evaluation. The *SESSIONS* table maintains a mapping of the user identifier and the IP address to a local session identifier. This identifier indexes into the *ROLES*

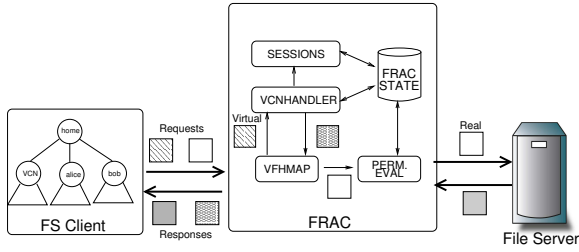


Figure 1. Overview of FRAC. The client namespace include the real file system and the VCN. FRAC handles requests for virtual objects (patterned boxes) and the file server handles requests for the real objects (solid boxes).

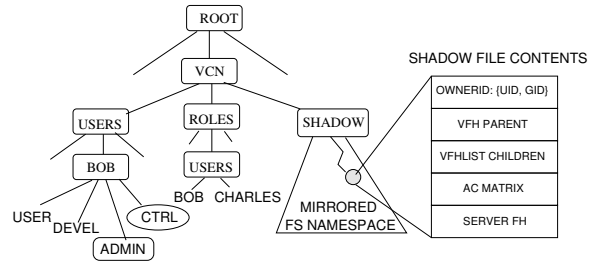


Figure 2. Subset of the VCN. User *bob* has the *user* and *devel* roles active and the *admin* role inactive. The *ctrl* file allows session updates. The contents of a shadow file are shown at the bottom.

table, which maintains the set of active roles for each active session. The *VFHMAP* stores a mapping from the VFH to the 2-tuple $\langle fh, shadowfh \rangle$. Finally, the *SHADOWFILES* map maintains the permission assignment and the file owner information for each file indexed by the file identifier (file handle).

3.2. Virtual Control Namespace

FRAC defines a virtual namespace, called the Virtual Control Namespace (VCN), for session management and dynamic permission assignment. The VCN is initialized when a file system is mounted over NFS. All operations on objects in the VCN are sent using the standard file system protocol and are interpreted as commands at FRAC. Through the VCN, clients can interact with FRAC over the well known file system interface using a familiar set of tools. Therefore, the VCN eliminates the need to run separate software at the clients or servers. It is important to note that user authentication is performed externally and is not the focus of FRAC. However, authentication mechanisms can be easily incorporated using the VCN.

The functionality of the VCN is similar to the well known */proc* file system in Linux. FRAC handlers are invoked on receiving a file system request for virtual objects. For read-only requests, for example, READ, REaddir, GETATTR, etc., the handlers query the FRAC state and generate the file contents dynamically. The query operation presents a snapshot of the FRAC state at the clients. The WRITE operations update the FRAC state and are used to modify active roles, permission assignment, and other session state defined by the system-wide policy. No metadata update operations, e.g. SETATTR, are supported on virtual files to restrict the update interface.

Figure 2 shows a subset of the VCN for the example in Section 2.1. The VCN has two main components: Session namespace, and Shadow namespace. The directories *users* and *roles* make up the session namespace and provide an interface to control the user sessions and user assignments, respectively. In the session namespace, a control file, *ctrl*, provides the update interface for each direc-

tory in the VCN. Users write the identifiers, of the new roles they want activated, to this file. If successful, the new active roles are instantiated and any modifications to the available roles are reflected in the contents of the directory. If unsuccessful, a permission denied message is returned to the client and the session remains unmodified.

The *shadow* namespace mirrors the file system. The names for objects in the *shadow* directory are derived directly from the files they represent in the primary file system. The shadow files provide an interface to query and update the permission assignments for each file. Effective permissions are determined using the owner permissions, specified as file attributes (*rwX*) of real files, and using the policy defined role-based permissions.

The initial permission assignment identifies the owner of each file and assigns her least privileged role the rights defined by the file permission bits. For example, if a file owned by *alice* has permissions *rw-*, FRAC identifies the least privileged role for *alice* (*user*), and provides that role with read and write privileges to the file. To update permissions for a file, users write the updated permission assignment to the corresponding file in the shadow namespace. If successful, the modifications are reflected in the contents of the shadow file. Owners can also update the permissions on the files through the *chmod* interface. For such updates, the effective permissions are re-evaluated and the contents of the corresponding shadow files are also updated.

Figure 2 shows the contents of a shadow file. For a shadow directory, the file handle information and shadow file identifiers of each of its children are also stored. This enables the VCN handlers for directory listing to construct a virtual namespace hierarchy that mirrors the real file system namespace.

The shadow file hierarchy is built lazily, using the file system requests to populate itself. FRAC generates shadow files by extracting records from the responses to directory listing and file lookup client requests (REaddir, LOOKUP). The root directory is special since it is identified during the mount protocol and the VCN root directory

appears as an immediate child of the root directory. FRAC adds an additional record for the VCN for any root directory listing request. The state required to generate the shadow files includes owner information, server specified file identifiers, and the access control matrix. While the owner information and the file handles can be regenerated, the access control matrix must be available across FRAC restarts as it represents the permission assignments that may have been updated by the owners.

All objects in the VCN are virtual and there is no file (or directory) at the server that stores the content of these objects. Virtual File Handles (VFH) are file identifiers generated by FRAC to transparently implement the VCN. Each virtual object in the VCN has a unique VFH. To prevent collisions between VFHs and file server specified FHs, FRAC generates a VFH even for real file system objects. FRAC maintains the VFH to FH mapping in its local state as a file handle mapping table (VFHMap). For real objects, FRAC replaces the VFH with the corresponding FH when forwarding a request message, and the FH with the VFH for a response message. For virtual objects, the appropriate FRAC handler is invoked. VFHs are also used to implement the write-once semantics for virtual objects by generating a new VFH every time the virtual object is updated.

4. Implementation

We built our system using FileWall [21], a network middlebox for file systems, which allows administrators to specify file system policies in the network. FileWall interposes between file system clients and servers, and *mediates* their interaction. FileWall provides the basic functionality of message capture, transformation, and injection, required by FRAC. FileWall functionality is implemented through policies, which are specified using file system message transformation. FileWall policies are executed using the attributes contained in file system messages, in the context of state, called *access context* (AC), maintained by the policy or present in the execution environment. FileWall is implemented in the Click modular router [11] framework as an external user-level package. The access context is implemented using an open-source database, Berkeley DB [20].

FRAC is implemented as a FileWall policy that uses attribute transformation to (i) virtualize file handles through the $VFH \leftrightarrow FH$ mapping, (ii) modify requests to execute as the owner of the file, (iii) incorporate the VCN in the file system hierarchy, and (iv) generate deny responses for requests that are disallowed by the policy. FRAC stores the state it maintains to implement RBAC persistently access context in FileWall.

5. Evaluation

In this section, we present an evaluation of our system. Our goal in this evaluation is to measure the impact of in-

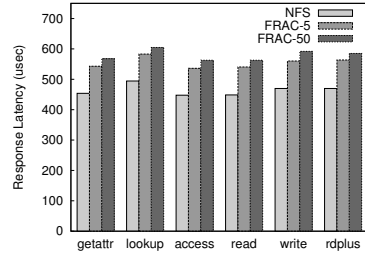


Figure 3. Policy enforcement over FRAC.

corporating FRAC on existing file system performance. All experiments assume that clients are operating under the policy defined in Section 2.1.

Setup: In our experimental setup, all systems are Dell Poweredge 2600 SMP systems with two 2.4GHz Intel Xeon II CPUs, and 2GB of RAM. All systems run a Linux-2.6.16 kernel and are connected using a Gigabit Ethernet switch. The average round-trip time between any two hosts is $300\mu s$. FRAC is configured to interpose on all NFS requests and responses.

In all experiments, the roles are arranged as a linear chain, with the least privileged role at the tail and the highest privileged role at the head. We assume a session with a role at the head of the chain, while the minimum role authorized for all operations is at the tail. This represents the worst case scenario for FRAC, as the permission evaluation must traverse all roles before generating an allow or deny verdict.

Microbenchmark: To study the behavior of the file system, with and without FRAC, we developed our own microbenchmark. This benchmark is an RPC client and issues NFS requests *without* relying on the client file system interface. Using this benchmark eliminates the noise due to the client buffer cache and other file system optimizations, and allows fine-grained measurements to be collected. The benchmark measures the CPU cycles between a request and the corresponding response.

First, we study the effect, on the client observed latency, of introducing FRAC on the network file system message path. To isolate the overheads, we study two different cases: (i) FRAC-5, which is the FRAC system configured with 5 different roles and (ii) FRAC-50, which is the FRAC system configured with 50 roles. As a baseline, we present the operation latency of the default NFS protocol.

Figure 3 shows the client observed latency for different NFS operations. For clarity, we show only the most common operations, as reported by various file system workload studies [4]. In the figure, each group of bars has 3 members, base NFS, FRAC-5, and FRAC-50. The height of each bar shows the average response latency for 1000 instances of the call.

We observe that FRAC imposes modest additional overhead when compared to the base NFS case. In the case of

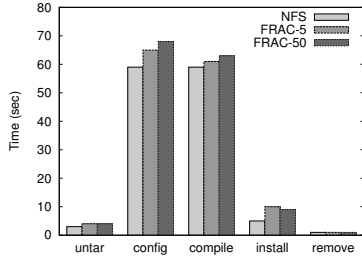


Figure 4. OpenSSH compilation.

5 roles, the overhead is less than $94\mu s$, while it is less than $123\mu s$ for the 50 role case.

Application Benchmark: In the following, we compare the performance of FRAC with the RBAC policy against the default NFS for a multi-stage software build similar to a modified Andrew Benchmark [9]. We measure the time taken to *untar*, *configure*, *compile*, *install*, and *remove* an OpenSSH 4.5 distribution.

Figure 4 shows the average time over ten runs with a cold cache for each phase of the OpenSSH compilation benchmark from left to right. The bars in each group are NFS, FRAC-5, and FRAC-50. We observe that FRAC imposes a modest overhead of around 11% (FRAC-5) and 14% (FRAC-50) for the sum of all phases of the benchmark. The most expensive data intensive phases have small ($< 10\%$ FRAC-5 and $< 15\%$ FRAC-50) overheads. However, the metadata intensive *untar* and *install* phases show significant degradation (33% and 100% respectively). This overhead is due to the creation of files, which requires generating new VFHs and shadow files for each file.

6. Discussion and Limitations

Stateful Policies: Stateful policies are useful, for example to identify suspicious user action based on pre-specified “threat-profiles” and to deny all subsequent file system accesses for the user. The threat profiles can be generated using access logs through offline analysis or specified statically as known malicious behavior. To implement such policies, FRAC maintains a per-session access history and matches the current access patterns against the threat profiles. On a match, the role associated with the session is updated to *threat* and no subsequent accesses are permitted. Re-enabling user access requires administrator action to update the FRAC state directly.

Concurrent Updates: We prevent concurrent, possibly inconsistent, updates to the objects in the VCN by forcing all objects to be write-once. That is any update to the permissions or ownership of a file must result in all cached copies of the virtual files to be purged. A new VFH is generated on each update and all other updates, possibly issued concurrently, are denied as the VFH is invalid or stale. However, our mechanism does not guarantee strict ordering or prioritization, and similar to UNIX write semantics, the winner of

the race to update is successful, while all others must update their cached copy and try the update again.

Fault Tolerance and Availability: FRAC is a centralized reference monitor. This creates the possibility of unavailability of the files even though both the clients and servers are alive and connected. However, FRAC stores its persistent state in a database and builds up its state using file system message contents. Therefore, we can easily design a primary-backup scheme where all messages are received at both systems. In this scheme, the database can be placed either at the file server or at an external node. While this might result in performance degradation, we believe the infrequent updates to file permissions and the user sessions make this acceptable.

FRAC does not allow direct access to users and allows access only through the standard file system interface. The limited interface to FRAC protects it against malicious users making the system unavailable. However, by generating a large number of carefully crafted requests, malicious users can cause a denial of service attack at FRAC. Such attacks can be mitigated by a rate control mechanism that discards messages from clients generating requests greater than a threshold rate. Attacks that prevent access to resources by inundating the network, must be handled external to FRAC.

7. Related Work

Role-Based Access Control (RBAC): The role-based access control model has evolved from the use of groups in UNIX and other operating systems, privilege groupings in database management systems, and separation of duty concepts. The modern concept of RBAC embodies the above in a single access control model ([10, 5] and references therein). Gustaffson et. al. [7] demonstrate the use of NFS to implement RBAC for distributed systems. They use modified NFS servers and clients to implement user-role mappings. However, their system does not support user session modification, role hierarchies, or separation of duty constraints.

Law Governed Interaction (LGI) [13] proposes a distributed policy definition and enforcement framework for heterogeneous distributed systems. He et. al. use the LGI framework to implement RBAC for network filesystems [8]. However, their system requires the server to execute an access control agent to hook into the security framework. It also requires specialized user agents at each client to communicate with the access control system. Their server modifications and client agents may limit the deployability of their system.

Namespaces and Semantic File Systems: The use of client namespaces in FRAC is motivated by seminal work on namespaces [12, 16, 15], each of which recognizes the importance of names as a unifying feature in all distributed systems. Plan 9 [16] uses a hierarchical file system to rep-

resent every resource in the system. A user or a process constructs a private namespace view by connecting these resources. FRAC uses the control namespace as a query and control mechanism for role based access control policies on network file systems. Therefore, unlike the above systems, file names in FRAC represent more than a mapping to a physical resource (files, network sockets, etc.). Along with the access context and policies, names are used as an active component of the RBAC framework.

Semantic File Systems [19], Jade [17], and Prospero [14] use programmable namespaces to organize and extend physical namespaces. FRAC shares the idea of programmability using the control namespace. However, there are several important differences. First, unlike the above where the filters execute on the client, FileWall policies are applied at an interposing proxy and do not require modifications to the server. Second, the policies are dynamic and are modified during execution. Finally, while FRAC can be used to organize data sources, its primary goal is protection by controlling access to portions of the namespace.

8. Conclusions

We presented an implementation of role-based access control for network file systems using FRAC, a reference monitor that controls the flow of file system messages between clients and servers. We demonstrated the use of our system through an access control policy that supports role hierarchies, user sessions, and static and dynamic separation of duty constraints.

We developed mechanisms to implement a virtual control namespace, which allows users and administrators to query and update user sessions, per-file policies, and the global access control policy over the standard file system interface. Our design based on VCN requires no modification to either clients or servers and does not require specialized user agents for clients to participate in the access control framework. Finally, our experiments with synthetic and real application workloads show that FRAC does not impose significant overheads while implementing RBAC policies.

References

- [1] The Advanced Maryland Automatic Network Disk Archiver. <http://www.amanda.org/>.
- [2] A. Anderson. XACML Profile for Role Based Access Control (RBAC). *OASIS Access Control TC Committee Draft*, 1:13, 2004.
- [3] D. C. Anderson, J. S. Chase, and A. Vahdat. Interposed request routing for scalable network storage. *ACM Trans. Comput. Syst.*, 20(1):25–48, 2002.
- [4] D. Ellard, J. Ledlie, P. Malkani, and M. I. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proc. FAST*, San Francisco, CA, 2003.
- [5] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. on Information and System Security (TISSEC)*, 4(3):224–274, 2001.
- [6] The Google Desktop. <http://desktop.google.com/>.
- [7] M. Gustafsson, B. Deligny, and N. Shahmehri. Using nfs to implement role-based access control. In *Proc. WET-ICE '97*, Washington, DC, USA, 1997.
- [8] Z. He, T. Phan, and T. D. Nguyen. Enforcing enterprise-wide policies over standard client-server interactions. In *Proc. of SRDS*, oct 2005.
- [9] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.
- [10] International Committee for Information Technology. Role-based access control. ANSI/INCITS 359-2004, Feb. 2004.
- [11] E. Kohler, R. T. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), August 2000.
- [12] B. W. Lampson. Designing a global name service. In *Proc. of PODC*, 1986.
- [13] N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Trans. Softw. Eng. Methodol.*, 9(3), 2000.
- [14] B. C. Neuman. The prospero file system: A global file system based on the virtual system model. *Computing Systems*, 5(4):407–432, 1992.
- [15] J. W. O’Toole and D. K. Gifford. Names should mean what, not where. In *Proc. of ACM SIGOPS-EW*, 1992.
- [16] R. Pike, D. Presotto, K. Thompson, H. Trickey, and P. Winterbottom. The use of name spaces in Plan 9. *Operating Systems Review*, 27(2):72–76, 1993.
- [17] H. C. Rao and L. L. Peterson. Accessing files in an internet: The jade file system. *IEEE Trans on Software Eng.*, 19(6), 1993.
- [18] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *IEEE, Proceedings*, 63:1278–1308, 1975.
- [19] M. Sheldon, D. Gifford, P. Jouvelot, and J. O. Jr. Semantic file systems. In *Proc. of SOSOP*, Pacific Grove, CA, 1991.
- [20] Sleepycat Software Inc. Berkeley DB. <http://dev.sleepycat.com/>.
- [21] S. Smaldone, A. Bohra, and L. Iftode. Firewall: Administrator controlled access to network file systems. Technical Report DCS TR 605, Rutgers University, Dept. of CS, Piscataway, NJ, Oct. 2006.
- [22] R. Tewari, J. M. Haswell, M. P. Naik, and S. M. Parkes. Glamour: A wide-area file system middleware using nfsv4. Technical Report RJ10368(A0507-011), IBM Research Division, Almaden Research Center, San Jose, CA, July 2005.
- [23] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and unix semantics in namespace unification. *ACM TOS*, 2(1), March 2006. To appear.
- [24] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proc. of USENIX*, San Diego, CA, June 2000. USENIX Association.
- [25] N. Zhu, D. Ellard, and T.-C. Chieuh. TBBT: Scalable and Accurate Trace Replay for File Server Evaluation. In *Proc. of FAST*, San Francisco, CA, December 2005.