Formal Analysis of Dynamic, Distributed File-System Access Controls

Avik Chaudhuri[1] and Martín Abadi[1,2]

[1] Computer Science Department, University of California, Santa Cruz
[2] Microsoft Research, Silicon Valley

Abstract. We model networked storage systems with distributed, cryptographically enforced file-access control in an applied pi calculus. The calculus contains cryptographic primitives and supports file-system constructs, including access revocation. We establish that the networked storage systems implement simpler, centralized storage specifications with local access-control checks. More specifically, we prove that the former systems preserve safety properties of the latter systems. Focusing on security, we then derive strong secrecy and integrity guarantees for the networked storage systems.

1 Introduction

Storage systems are typically governed by access-control policies, and the security of those systems depends on the sound enforcement of the necessary access-control checks. Unfortunately, both the policies and their enforcement can be surprisingly problematic, for several reasons. In particular, the policies may be allowed to change over time, often via interactions with the file-system environment; it is then crucial to prevent unauthorized access-control administration, and to guarantee that authorized access-control administration has correct, prompt effects. Another source of substantial difficulties is distribution. In networked, distributed storage systems, file access is often not directly guarded by access-control checks. Instead, file access is guarded by the inspection of capabilities; these capabilities certify that the relevant access-control checks have been done elsewhere in the past. Yet other difficulties result from the scale and complexity of systems, which present a challenge to consistent administration.

In this paper, we aim to simplify security analyses for storage systems. Specifically, we model network-attached storage (NAS) systems [7,15,11]. We prove that NAS systems are as safe (from the point of view of passing tests [14]) as corresponding centralized storage systems with local access-control enforcement. In other words, reasoning about the safety of the centralized storage systems can be applied for free to the significantly more complicated NAS systems. As important special cases, we derive the preservation of secrecy and integrity guarantees.

The systems that we study include distributed file-system management across a number of access-control servers and disks on the network; they also include dynamic administration of access control. At the same time, we avoid commitments to certain specific choices that particular implementations might make—on file-operation and policy-administration commands, algorithms for file allocation over multi-disk arrays, various scheduling algorithms—so that our results remain simple and apply broadly. We describe those systems and analyze their security properties in an applied pi calculus [3].

E. Najm et al. (Eds.): FORTE 2006, LNCS 4229, pp. 99–114, 2006.

This calculus includes cryptographic primitives and supports file-system constructs. It also enables us to incorporate a basic but sufficient model of time, as needed for dynamic administration.

Background and Related Work. Various cryptographic implementations of distributed access control have been proposed as part of the security designs of NAS protocols [6,8,7,15,11,17]. However, the security analyses of these implementations have been at best semi-formal. Some exceptions are the work of Mazières and Shasha on data integrity for untrusted remote storage [10], and Gobioff's security analysis of a NAS protocol using belief logics [7].

In a recent paper [5], we consider a restricted class of NAS systems, with fixed access-control policies and a single network-attached disk interface. We show that those systems are fully abstract with respect to centralized file systems. Full abstraction [12] is a powerful criterion for the security of implementations [1]: it prohibits any leakage of information. It is also fairly fragile, and can be broken by many reasonable implementations in practice. In particular, capability revocation and expiry (more broadly, dynamic administration, as we study it here) give rise to counterexamples to full abstraction that appear impossible to avoid in any reasonable implementation of NAS. We discuss these issues in detail in Section 5. In sum, the systems that we study in this paper are considerably more general and complex than those we consider in our previous work, so much so that we cannot directly extend our previous full-abstraction result. Fortunately, however, we can still obtain strong secrecy and integrity guarantees while retaining the simplicity of our specifications.

We employ a variation of may-tests to observe the behaviour of systems. Proofs based on may-testing for safety and security properties have also been studied elsewhere (*e.g.*, [14,4]). Our treatment of secrecy is also fairly standard (*e.g.*, [4]). On the other hand, our treatment of integrity properties is not. We formalize integrity properties via "warnings". Warnings signal violations that can be detected by monitoring system execution. In this way, our approach to integrity is related to enforceable mechanisms for security policies [16]. Warnings can also indicate the failure of correspondences between events, and hence may be used to verify correspondence assertions (*e.g.*, [9]). On the other hand, it does not seem convenient to use standard correspondence assertions directly in implementation proofs such as ours.

Outline of the Paper. In the next section we give an overview of the applied pi calculus that serves as our modeling language. In Section 3, we present a simple storage specification based on a centralized file system with local access-control checks. In Section 4, we show a NAS implementation that features distributed file-system management and cryptographic access-control enforcement. Then, in Section 5, we extract specifications from NAS systems, state our main theorem (safety preservation), and derive some important security consequences. We conclude in Section 6.

2 The Applied pi Calculus

We use a polyadic, synchronous, applied pi calculus [13,3] as the underlying language to describe and reason about processes. The syntax is standard. We use the notation $\widetilde{\varphi}$ to mean a sequence $\varphi_1, \ldots, \varphi_k$, where the length k of the sequence is given by $|\widetilde{\varphi}|$.

$M, N ::=$	terms
$m, n, \ldots$	name
$x, y, \ldots$	variable
$f(\widetilde{M})$	function application

The language of terms contains an infinite set of names and an infinite set of variables; further, terms can be built from smaller ones by applying function symbols. Names can be channel names, key names, and so on. Function symbols are drawn from a finite ranked set $\mathcal{F}$, called the signature. This signature is equipped with an equational theory. Informally, the theory provides a set of equations over terms, and we say that $\mathcal{F} \vdash M = N$ for terms M and N if and only if $M = N$ can be derived from those equations.

For our purposes, we assume symbols for shared-key encryption $\{\cdot\}_{\cdot}$ and message authentication $\mathbf{mac}(\cdot, \cdot)$, and list the only equations that involve these symbols below. The first equation allows decryption of an encrypted message with the correct key; the second allows extraction of a message from a message authentication code.

$$\mathbf{decrypt}(\{x\}_y, y) = x \qquad \mathbf{message}(\mathbf{mac}(x, y)) = x$$

We also assume some standard data structures, such as tuples, numerals, and queues, with corresponding functions, such as projection functions $\mathbf{proj}_\ell$. Several function symbols are introduced in Sections 3 and 4. Next we show the language of processes.

$P, Q ::=$	processes
$\overline{M}\langle\widetilde{N}\rangle.\, P$	output
$M(\widetilde{x}).\, P$	input
$P \mid Q$	composition
$(\nu n)\, P$	restriction
0	nil
$!P$	replication
$\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$	conditional

Processes have the following informal semantics.

- The nil process 0 does nothing.
- The composition process $P \mid Q$ behaves as the processes P and Q in parallel.
- The input process $M(\widetilde{x}).\, P$ can receive any sequence of terms $\widetilde{N}$ on M, where $|\widetilde{N}| = |\widetilde{x}|$, then execute $P\{\widetilde{N}/\widetilde{x}\}$. The variables $\widetilde{x}$ are bound in P in $M(\widetilde{x}).\, P$. The notation $\{\widetilde{M}/\widetilde{x}\}$ represents the capture-free substitution of terms $\widetilde{M}$ for variables $\widetilde{x}$. The input blocks if M is not a name at runtime.
- The synchronous output process $\overline{M}\langle\widetilde{N}\rangle.P$ can send the sequence of terms $\widetilde{N}$ on M, then execute P. The output blocks if M is not a name at runtime; otherwise, it waits for a synchronizing input on M.
- The replication process $!P$ behaves as an infinite number of copies of P running in parallel.
- The restriction process $(\nu n)\, P$ creates a new name n bound in P, then executes P. This construct is used to create fresh, unguessable secrets in the language.
- The conditional process $\mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q$ behaves as P if $\mathcal{F} \vdash M = N$, and as Q otherwise.

We elide $\mathcal{F} \vdash$ in the sequel. The notions of free variables and names (`fv` and `fn`) are as usual; so are various abbreviations (*e.g.*, Π and Σ for indexed parallel composition and internal choice, respectively). We call terms or processes closed if they do not contain any free variables. We use a commitment semantics for closed processes [13,4]. Informally, a commitment reflects the ability to do some action, which may be output ($\overline{n}$), input (n), or silent (τ). More concretely,

- $P \xrightarrow{\overline{n}} (\nu\widetilde{m})\,\langle\widetilde{M}\rangle.\,Q$ means that P can output on name n the terms $\widetilde{M}$ that contain fresh names $\widetilde{m}$, and continue as Q.
- $P \xrightarrow{n} (\widetilde{x}).Q$ means that P can input terms on n, bind them to $\widetilde{x}$ in Q, and continue as Q instantiated.
- $P \xrightarrow{\tau} Q$ means that P can silently transition to Q.

3 Specifying a Simple Centralized File System

In this section, we model a simple centralized file system. The model serves as a specification for the significantly more complex distributed file-system implementation of Section 4. We begin with a summary of the main features of the model.

- The file system serves a number of clients who can remotely send their requests over distinguished channels. The requests may be for file operations, or for administrative operations that modify file-operation permissions of other clients.
- Each request is subject to local access-control checks that decide whether the requested operation is permitted. A request that passes these checks is then processed in parallel with other pending requests.
- Any requested modification to existing file-operation permissions takes effect only after a deterministic, finite delay. The delay is used to specify accurate correctness conditions for the expiry-based, distributed access-control mechanism of Section 4.

We present a high-level view of this "ideal" file system, called IFS, by means of a grammar of *control states* (see below). IFS can be coded as a process (in the syntax of the previous section), preserving its exact observable semantics. An IFS control state consists of the following components:

- a pool of threads, where each thread reflects a particular stage in the processing of some pending request to the file system;
- an access-control policy, tagged with a schedule for pending policy updates;
- a storage state (or "disk"); and
- a clock, as required for scheduling modifications to the access-control policy.

IFS-Th $::=$	file-system thread
$\mathsf{Req}_k(op, n)$	file-operation request
$\mathsf{App}(op, n)$	approved file operation
$\mathsf{Ret}(n, r)$	return after file operation
$\mathsf{PReq}_k(adm, n)$	administration request
$\Delta ::=$	thread pool
$\varnothing$	empty

IFS-Th, Δ	thread in pool
IFS-Control ::=	file-system control state
Δ: $\mathcal{R}^{\mathcal{H}}$: ρ: Clk	threads: tagged access policy: disk state: clock

The threads are of four sorts, explained below: $\mathsf{Req}_k(op, n)$, $\mathsf{App}(op, n)$, $\mathsf{Ret}(n, r)$, and $\mathsf{PReq}_k(adm, n)$. The clock Clk is a monotonically increasing integer. The storage state ρ reflects the state maintained at the disk (typically file contents; details are left abstract in the model). The access-control policy $\mathcal{R}$ decides which subjects may execute operations on the storage state, and which administrators may make modifications to the policy itself. The schedule $\mathcal{H}$ contains a queue of pending modifications to the policy, with each modification associated with a clock that says when that modification is due.

Let $\mathcal{K}$ be a set of indices that cover both the subjects and the administrators of access control. We assume distinguished sets of channel names $\{\beta_k \mid k \in \mathcal{K}\}$ and $\{\alpha_k \mid k \in \mathcal{K}\}$ on which the file system receives requests for file operations and policy modifications, respectively. A file-operation request consists of a term *op* that describes the operation (typically, a command with arguments, some of which may be file names) and a channel n for the result. When such a request arrives on β_k, the file system spawns a new thread of the form $\mathsf{Req}_k(op, n)$. The access-control policy then decides whether k has permission to execute *op* on the storage state. If not, the thread dies; otherwise, the thread changes state to $\mathsf{App}(op, n)$. The request is then forwarded to the disk, which executes the operation and updates the storage state, obtaining a result r. The thread changes state to $\mathsf{Ret}(n, r)$. Later, r is returned on n, and the thread terminates successfully.

A policy-modification request consists of a term adm that describes the modification to the policy and a channel n for the acknowledgment. When such a request arrives on α_k, the file system spawns a thread of the form $\mathsf{PReq}_k(adm, n)$. Then, if the policy does not allow k to do adm, the thread dies; otherwise, the modification is queued to the schedule and an acknowledgment is returned on n, and the thread terminates successfully. At each clock tick, policy modifications that are due in the schedule take effect, and the policy and the schedule are updated accordingly.

Operationally, we assume functions **may**, **execute**, **schedule**, and **update** that satisfy the following equations. (We leave abstract the details of the equational theory.)

- $\mathbf{may}(k, op, \mathcal{R}) = \mathbf{yes}$ (*resp.* $\mathbf{may}(k, adm, \mathcal{R}) = \mathbf{yes}$) if the policy $\mathcal{R}$ allows k to execute file operation *op* (*resp.* make policy modification adm), and $= \mathbf{no}$ otherwise.
- $\mathbf{execute}(op, \rho) = \langle \rho', r \rangle$, where ρ' and r are the storage state and the result, respectively, obtained after executing file operation *op* on storage state ρ.
- $\mathbf{schedule}(adm, \mathcal{H}, \mathsf{Clk}) = \mathcal{H}'$, where $\mathcal{H}'$ is the schedule after queuing an entry of the form $adm@\mathsf{Clk}'$ (with $\mathsf{Clk}' \geq \mathsf{Clk}$) to schedule $\mathcal{H}$. The clock Clk', determined by adm, $\mathcal{H}$, and Clk, indicates the instant at which adm is due in the new schedule.
- $\mathbf{update}(\mathcal{R}^{\mathcal{H}}, \mathsf{Clk}) = \mathcal{R}'^{\mathcal{H}'}$, where $\mathcal{R}'$ is the policy after making modifications to policy $\mathcal{R}$ that are due at clock Clk in schedule $\mathcal{H}$, and $\mathcal{H}'$ is the schedule left.

Further, we assume a function **lifespan** such that $\mathbf{lifespan}(k, op, \mathcal{H}, \mathsf{Clk}) \geq 0$ for all k, *op*, $\mathcal{H}$, and Clk. Informally, if $\mathbf{lifespan}(k, op, \mathcal{H}, \mathsf{Clk}) = \lambda$ and the file

(Op Req)

$$\begin{array}{c}\Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk} \xrightarrow{\beta_k}\\ (x,y).\ \mathsf{Req}_k(x,y), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk}\end{array}$$

(Op Deny)

$$\frac{\mathbf{may}(k, op, \mathcal{R}) = \mathbf{no}}{\mathsf{Req}_k(op,n), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk} \xrightarrow{\tau} \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk}}$$

(Op Ok)

$$\frac{\mathbf{may}(k, op, \mathcal{R}) = \mathbf{yes}}{\begin{array}{c}\mathsf{Req}_k(op,n), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk} \xrightarrow{\tau}\\ \mathsf{App}(op,n), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk}\end{array}}$$

(Op Exec)

$$\frac{\mathbf{execute}(op, \rho) = \langle \rho', r\rangle}{\begin{array}{c}\mathsf{App}(op,n), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk} \xrightarrow{\tau}\\ \mathsf{Ret}(n,r), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho'\colon \mathsf{Clk}\end{array}}$$

(Op Res Ret)

$$\begin{array}{c}\mathsf{Ret}(n,r), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk} \xrightarrow{\overline{n}}\\ \langle r\rangle.\ \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk}\end{array}$$

(Adm Req)

$$\begin{array}{c}\Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk} \xrightarrow{\alpha_k}\\ (x,y).\ \mathsf{PReq}_k(x,y), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk}\end{array}$$

(Adm Deny)

$$\frac{\mathbf{may}(k, adm, \mathcal{R}) = \mathbf{no}}{\begin{array}{c}\mathsf{PReq}_k(adm,n), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk} \xrightarrow{\tau}\\ \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk}\end{array}}$$

(Adm Ok Ack)

$$\frac{\begin{array}{c}\mathbf{may}(k, adm, \mathcal{R}) = \mathbf{yes}\\ \mathbf{schedule}(adm, \mathcal{H}, \mathsf{Clk}) = \mathcal{H}'\end{array}}{\begin{array}{c}\mathsf{PReq}_k(adm,n), \Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk} \xrightarrow{\overline{n}}\\ \langle\rangle.\ \Delta\colon \mathcal{R}^{\mathcal{H}'}\colon \rho\colon \mathsf{Clk}\end{array}}$$

(Tick)

$$\frac{\mathbf{update}(\mathcal{R}^{\mathcal{H}}, \mathsf{Clk}) = \mathcal{R}'^{\mathcal{H}'}}{\Delta\colon \mathcal{R}^{\mathcal{H}}\colon \rho\colon \mathsf{Clk} \xrightarrow{\tau} \Delta\colon \mathcal{R}'^{\mathcal{H}'}\colon \rho\colon \mathsf{Clk}+1}$$

Fig. 1. Semantics of a file system with local access control

operation op is allowed to k at Clk, then op cannot be denied to k before $\mathsf{Clk} + \lambda$. Formally, we extend **schedule** to sequences by letting $\mathbf{schedule}(\varnothing, \mathcal{H}, \mathsf{Clk}) = \mathcal{H}$ and $\mathbf{schedule}(adm'\,\widetilde{adm}, \mathcal{H}, \mathsf{Clk}) = \mathbf{schedule}(\widetilde{adm}, \mathbf{schedule}(adm', \mathcal{H}, \mathsf{Clk}), \mathsf{Clk})$; we require that if $\mathbf{lifespan}(k, op, \mathcal{H}, \mathsf{Clk}) = \lambda$ then there do not exist (possibly empty) sequences of policy-modification commands $\widetilde{adm}_{\mathsf{Clk}}, \widetilde{adm}_{\mathsf{Clk}+1}, \ldots, \widetilde{adm}_{\mathsf{Clk}+\lambda}$ and policy $\mathcal{R}_{\mathsf{Clk}}$ such that the following hold at once:

- $\mathbf{may}(k, op, \mathcal{R}_{\mathsf{Clk}}) = \mathbf{yes}$
- $\mathcal{H}_{\mathsf{Clk}} = \mathcal{H}$
- $\widehat{\mathcal{H}}_{\mathsf{Clk}'} = \mathbf{schedule}(\widetilde{adm}_{\mathsf{Clk}'}, \mathcal{H}_{\mathsf{Clk}'}, \mathsf{Clk}')$ for each $\mathsf{Clk}' \in \mathsf{Clk} \ldots \mathsf{Clk} + \lambda$
- $\mathcal{R}^{\mathcal{H}_{\mathsf{Clk}'+1}}_{\mathsf{Clk}'+1} = \mathbf{update}(\mathcal{R}^{\widehat{\mathcal{H}}_{\mathsf{Clk}'}}_{\mathsf{Clk}'}, \mathsf{Clk}')$ for each $\mathsf{Clk}' \in \mathsf{Clk} \ldots \mathsf{Clk} + \lambda - 1$
- $\mathbf{may}(k, op, \mathcal{R}_{\mathsf{Clk}+\lambda}) = \mathbf{no}$

For instance, $\mathbf{lifespan}(k, op, \mathcal{H}, \mathsf{Clk})$ can return a constant delay λ_c for all k, op, $\mathcal{H}$, and Clk, and $\mathbf{schedule}(adm, \mathcal{H}, \mathsf{Clk})$ can return $[\mathcal{H};\ adm@\mathsf{Clk}+\lambda_c]$ for all adm. When $\lambda_c = 0$, any requested modification to the policy takes effect at the next clock tick.

The formal semantics of the file system is shown as a commitment relation in Figure 1. The relation describes how the file system spawns threads, how threads evolve, how access control is enforced and administered, how file operations are serviced, and how time goes by, in terms of standard pi-calculus actions.

We assume a set of clients $\{C_k \mid k \in \mathcal{K}\}$ that interact with the file system. We provide macros to request file operations and policy modifications; clients may use these macros, or explicitly send appropriate messages to the file system on the channels $\{\alpha_k, \beta_k \mid k \in \mathcal{K}\}$.

Definition 1 (Macros for IFS clients).

File operation on port k: *A file operation may be requested with the macro* $\mathsf{fileop}_k\ op/x;\ P$*, which expands to* $(\nu n)\ \overline{\beta_k}\langle op, n\rangle.\ n(x).\ P$*, where* $n \notin \mathtt{fn}(P)$.

Administration on port k: *A policy modification may be requested with the macro* $\mathsf{admin}_k\ adm;\ P$*, which expands to* $(\nu n)\ \overline{\alpha_k}\langle adm, n\rangle.\ n().\ P$*, where* $n \notin \mathtt{fn}(P)$.

We select a subset of clients whom we call *honest*; these clients may be arbitrary processes, as long as they use macros on their own ports for all interactions with the file system. Further, as a consequence of Definitions 2 and 3 (see below), no other client may send a request to the file system on the port of an honest client.

Definition 2. *A set of honest* IFS *clients indexed by* $\mathcal{I} \subseteq \mathcal{K}$ *is a set of closed processes* $\{C_i \mid i \in \mathcal{I}\}$*, so that each* C_i *in the set has the following properties:*

- *all macros in* C_i *are on port* i*,*
- *no name in* $\{\alpha_{i'}, \beta_{i'} \mid i' \in \mathcal{I}\}$ *appears free in* C_i *before expanding macros.*

Let $\mathcal{J} = \mathcal{K} \setminus \mathcal{I}$. We impose no restrictions on the "dishonest" clients C_j ($j \in \mathcal{J}$), except that they may not know the channels $\{\alpha_i, \beta_i \mid i \in \mathcal{I}\}$ initially. In fact, we assume that dishonest clients are part of an arbitrary environment, and as such, leave their code unspecified. The restriction on their initial knowledge is expressed by leaving them outside the initial scope of the channels $\{\alpha_i, \beta_i \mid i \in \mathcal{I}\}$.

Definition 3. *An ideal storage system denoted by* $\mathrm{IS}(\mathbb{C}_{\mathcal{I}}, \mathcal{R}, \rho, \mathsf{Clk})$ *is the closed process* $(\nu_{i \in \mathcal{I}}\ \alpha_i \beta_i)\ (\Pi_{i \in \mathcal{I}} C_i \mid \varnothing\colon \mathcal{R}^{\varnothing} : \rho\colon \mathsf{Clk})$*, where*

- $\mathbb{C}_{\mathcal{I}} = \{C_i \mid i \in \mathcal{I}\}$ *is a set of honest* IFS *clients indexed by* $\mathcal{I}$*,*
- $\varnothing\colon \mathcal{R}^{\varnothing} : \rho\colon \mathsf{Clk}$ *is an initial* IFS *control state, and* $\{\alpha_i, \beta_i \mid i \in \mathcal{I}\} \cap \mathtt{fn}(\mathcal{R}, \rho) = \varnothing$.

4 An Implementation of Network-Attached Storage

In this section, we model a distributed file system based on network-attached storage (NAS). A typical network-attached file system is distributed over a set of disks that are "attached" to the network, and a set of servers (called managers). The disks directly receive file-operation requests from clients, while the managers maintain file-system metadata and file-access permissions, and serve administrative requests. In simple traditional storage designs, access-control checks and metadata lookups are done for every request to the file system. In NAS, that per-request overhead is amortized, resulting in significant performance gains. Specifically, a client who wishes to request a file operation first contacts one of the managers; the manager does the relevant checks and lookups, and returns a cryptographically signed *capability* to the client. The capability is a certification of access rights for that particular operation, and needs to be obtained only once. The client can then request that operation any number of times at a disk, attaching to its requests the capability issued by the manager. The disk simply verifies the capability before servicing each of those requests. NAS implementations are further optimized by allocating different parts of the file system to different managers and disks. This kind of partitioning distributes load and increases concurrency.

Perhaps the most challenging aspect of NAS's access-control mechanism, and indeed of distributed access controls in general, is the sound enforcement of access revocation. In particular, whenever some permissions are revoked, all previous capabilities that certify those permissions must be invalidated. On the other hand, when issuing a capability, it is impossible to predict when a permission certified by that capability might be revoked in the future. It is possible, in theory, to simulate immediate revocation by communicating with the disks: the disks then maintain a record of revoked permissions and reject all capabilities that certify those permissions. However, this "solution" reduces the performance and distribution benefits of NAS.

A sound, practical solution exists if we allow a deterministic finite delay in revocation. Informally, a capability is marked with an unforgeable timestamp that declares its expiry, beyond which it is always rejected—and any revocation of the permissions certified by that capability takes effect only after the declared expiry. By letting the expiry depend on various parameters, this solution turns out to be quite flexible and effective.

Following the design above, we model a fairly standard network-attached file system, called NAFS. Much as in Section 3, we present the file system using a grammar of control states and a semantics of commitments. A NAFS control state consists of the following components:

- a pool of threads distributed between the managers and the disks;
- the local access-control policy and modification schedule at each manager;
- the local storage state at each disk; and
- a global clock shared between the managers and the disks.

NAFS-Th-Server$_a$ $::=$	thread at a^{th} manager
$\quad\mathsf{AReq}_{a.k}(op, c)$	capability request
$\quad\mathsf{PReq}_{a.k}(adm, n)$	administration request
NAFS-Th-Disk$_b$ $::=$	thread at b^{th} disk
$\quad\mathsf{Req}_b(\kappa, n)$	authorized file-operation request
$\quad\mathsf{App}_b(op, n)$	approved file operation
$\quad\mathsf{Ret}(n, r)$	return after file operation
$\ddot{\Delta}$ $::=$	distributed thread pool
$\quad\varnothing$	empty
$\quad$NAFS-Th-Server$_a, \ddot{\Delta}$	a^{th}-manager thread in pool
$\quad$NAFS-Th-Disk$_b, \ddot{\Delta}$	b^{th}-disk thread in pool
NAFS-Control $::=$	distributed file-system control state
$\quad\ddot{\Delta} : \widetilde{\mathcal{R}^{\mathcal{H}}} : \widetilde{\rho} : \mathsf{Clk}$	threads : tagged policies : disk states : clock

Let $\mathcal{A}$ (*resp.* $\mathcal{B}$) index the set of managers (*resp.* disks) used by the file system. For each $a \in \mathcal{A}$, we assume a distinguished set of names $\{\alpha_{a.k} \mid k \in \mathcal{K}\}$ on which the a^{th} manager receives requests for policy modifications. A request on $\alpha_{a.k}$ is internally forwarded to the manager a' allocated to serve that request, thereby spawning a thread of the form $\mathsf{PReq}_{a'.k}(adm, n)$. This thread is then processed in much the same way as $\mathsf{PReq}_k(adm, n)$ in Section 3. At each tick of the shared clock, due modifications to each of the local policies at the managers take effect.

Next, we elaborate on the authorization and execution of file operations. For each $a \in \mathcal{A}$ and $b \in \mathcal{B}$, we assume distinguished sets of names $\{\alpha^{\circ}_{a.k} \mid k \in \mathcal{K}\}$ and

$\{\beta_{b.k} \mid k \in \mathcal{K}\}$ on which the a^{th} manager and the b^{th} disk receive requests for authorization and execution of file operations, respectively. An authorization request consists of a term op that describes the file operation and a channel c to receive a capability for that operation. Such a request on $\alpha^{\circ}_{a.k}$ is internally forwarded to the manager a' allocated to serve that request, thereby spawning a thread of the form $\mathsf{AReq}_{a'.k}(op, c)$. If the access-control policy at a' does not allow k to do op, the thread dies; otherwise, a capability κ is returned on c, and the thread terminates successfully. The capability, a term of the form $\mathbf{mac}(\langle op, T, b\rangle, \mathrm{K}_b)$, is a message authentication code whose message contains op, an encrypted timestamp T, and the disk b responsible for executing op. The timestamp T, of the form $\{\langle m, \mathsf{Clk}\rangle\}_{\mathrm{K}_b}$, indicates the expiry Clk of κ, and additionally contains a unique nonce m. (The only purpose of the nonce is to make the timestamp unique.) A secret key K_b shared between the disk b and the manager is used to encrypt the timestamp and sign the capability. (In concrete implementations, different parts of the key may be used for encryption and signing.) The rationale behind the design of the capability is discussed in Section 5. Intuitively, the capability is unforgeable, and verifiable by the disk b; and the timestamp carried by the capability is unique, and unintelligible to any other than the disk b.

An execution request consists of a capability κ and a return channel n. On receiving such a request on $\beta_{b.k}$, the disk b spawns a thread of the form $\mathsf{Req}_b(\kappa, n)$. It then extracts the claimed operation op from κ (if possible), checks that κ is signed with the key K_b (thereby verifying the integrity of κ), and checks that the timestamp decrypts under K_b to a clock no earlier than the current clock (thereby verifying that κ has not expired). If these checks fail, the thread dies; otherwise, the thread changes state to $\mathsf{App}_b(op, n)$. This thread is then processed in much the same way as $\mathsf{App}(op, n)$ in Section 3.

Operationally, we assume a function **manager** (*resp.* **disk**) that allocates file operations and policy modifications to managers (*resp.* file operations to disks). We also assume functions $\mathbf{may}_a$, $\mathbf{execute}_b$, $\mathbf{schedule}_a$, and $\mathbf{update}_a$ for each $a \in \mathcal{A}$ and $b \in \mathcal{B}$, with the same specifications as their analogues in Section 3. Further, we assume a function $\mathbf{expiry}_a$ for each $a \in \mathcal{A}$ with the following property (*cf.* the function **lifespan**, Section 3): if $\mathbf{expiry}_a(k, op, \mathcal{H}, \mathsf{Clk}) = \mathsf{Clk}_e$, then $\mathsf{Clk}_e \geq \mathsf{Clk}$ and there do not exist sequences of policy-modification commands $\widetilde{adm}_{\mathsf{Clk}}, \widetilde{adm}_{\mathsf{Clk}+1}, \ldots, \widetilde{adm}_{\mathsf{Clk}_e}$ and policy $\mathcal{R}_{\mathsf{Clk}}$ such that the following hold at once:

- $\mathbf{manager}(\widetilde{adm}_{\mathsf{Clk}'}) = a$ for each $\mathsf{Clk}' \in \mathsf{Clk} \ldots \mathsf{Clk}_e$
- $\mathbf{may}_a(k, op, \mathcal{R}_{\mathsf{Clk}}) = \mathbf{yes}$
- $\mathcal{H}_{\mathsf{Clk}} = \mathcal{H}$
- $\widehat{\mathcal{H}}_{\mathsf{Clk}'} = \mathbf{schedule}_a(\widetilde{adm}_{\mathsf{Clk}'}, \mathcal{H}_{\mathsf{Clk}'}, \mathsf{Clk}')$ for each $\mathsf{Clk}' \in \mathsf{Clk} \ldots \mathsf{Clk}_e$
- $\mathcal{R}^{\mathcal{H}_{\mathsf{Clk}'+1}}_{\mathsf{Clk}'+1} = \mathbf{update}_a(\mathcal{R}^{\widehat{\mathcal{H}}_{\mathsf{Clk}'}}_{\mathsf{Clk}'}, \mathsf{Clk}')$ for each $\mathsf{Clk}' \in \mathsf{Clk} \ldots \mathsf{Clk}_e - 1$
- $\mathbf{may}_a(k, op, \mathcal{R}_{\mathsf{Clk}_e}) = \mathbf{no}$

In Section 5, we show how the functions $\mathbf{expiry}_a$ and **lifespan** are related: informally, the lifespan of a permission can be defined as the duration between the current clock and the expiry of any capability for that permission.

The formal semantics of NAFS is shown in Figure 2. Next we provide macros for requesting file-operation capabilities and policy modifications at a manager, and authorized file operations at appropriate disks.

At the a^{th} manager:

(Auth Req)

$$\ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\alpha^{\circ}_{a.k}} (op, c).\ \mathsf{AReq}_{a.k}(op, c), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}$$

(Auth Deny)

$$\frac{\mathbf{manager}(op) = a \qquad \mathbf{may}_a(k, op, \mathcal{R}_a) = \mathbf{no}}{\mathsf{AReq}_{a.k}(op, c), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\tau} \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}}$$

(Auth Ok Cap)

$$\frac{\begin{array}{c}\mathbf{manager}(op) = a \qquad \mathbf{may}_a(k, op, \mathcal{R}_a) = \mathbf{yes} \qquad \mathbf{disk}(op) = b \\ \{\langle m, \mathbf{expiry}_a(k, op, \mathcal{H}_a, \mathsf{Clk})\rangle\}_{\mathrm{K}_b} = T \text{ for fresh } m \qquad \mathbf{mac}(\langle op, T, b\rangle, \mathrm{K}_b) = \kappa\end{array}}{\mathsf{AReq}_{a.k}(op, c), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\overline{c}} (\nu m)\ \langle \kappa\rangle.\ \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}}$$

(Adm Req)

$$\ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\alpha_{a.k}} (adm, n).\ \mathsf{PReq}_{a.k}(adm, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}$$

(Adm Deny)

$$\frac{\mathbf{manager}(adm) = a \qquad \mathbf{may}_a(k, adm, \mathcal{R}_a) = \mathbf{no}}{\mathsf{PReq}_{a.k}(adm, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\tau} \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}}$$

(Adm Ok Ack)

$$\frac{\begin{array}{c}\mathbf{manager}(adm) = a \qquad \mathbf{may}_a(k, adm, \mathcal{R}_a) = \mathbf{yes} \\ \mathbf{schedule}_a(adm, \mathcal{H}_a, \mathsf{Clk}) = \mathcal{H}'_a \qquad \forall a' \neq a : \mathcal{H}'_{a'} = \mathcal{H}_{a'}\end{array}}{\mathsf{PReq}_{a.k}(adm, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\overline{n}} \langle\rangle.\ \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}'}}\colon \widetilde{\rho}\colon \mathsf{Clk}}$$

Across managers:

(Auth Fwd)

$$\frac{\mathbf{manager}(op) = a' \neq a}{\mathsf{AReq}_{a.k}(op, c), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\tau} \mathsf{AReq}_{a'.k}(op, c), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}}$$

(Adm Fwd)

$$\frac{\mathbf{manager}(adm) = a' \neq a}{\mathsf{PReq}_{a.k}(adm, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\tau} \mathsf{PReq}_{a'.k}(adm, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}}$$

(Tick)

$$\frac{\forall a : \mathbf{update}_a({\mathcal{R}_a}^{\mathcal{H}_a}, \mathsf{Clk}) = {\mathcal{R}'_a}^{\mathcal{H}'_a}}{\ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\tau} \ddot{\Delta}\colon \widetilde{\mathcal{R}'^{\mathcal{H}'}}\colon \widetilde{\rho}\colon \mathsf{Clk} + 1}$$

At the b^{th} disk:

(Exec Req)

$$\ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\beta_{b.k}} (\kappa, n).\ \mathsf{Req}_b(\kappa, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}$$

(Op Ok)

$$\frac{\begin{array}{c}\kappa = \mathbf{mac}(\langle op, T, b\rangle, \mathrm{K}_b) \\ \mathbf{decrypt}(T, \mathrm{K}_b) = \langle m, \mathsf{Clk}'\rangle \qquad \mathsf{Clk} \leq \mathsf{Clk}'\end{array}}{\mathsf{Req}_b(\kappa, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\tau} \mathsf{App}_b(op, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}}$$

(Exec Deny)

$$\frac{\nexists op, T, m, \mathsf{Clk}' \text{ s.t. } \mathbf{mac}(\langle op, T, b\rangle, \mathrm{K}_b) = \kappa, \mathbf{decrypt}(T, \mathrm{K}_b) = \langle m, \mathsf{Clk}'\rangle, \text{and } \mathsf{Clk} \leq \mathsf{Clk}'}{\mathsf{Req}_b(\kappa, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\tau} \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}}$$

(Op Exec)

$$\frac{\mathbf{execute}_b(op, \rho_b) = \langle \rho'_b, r\rangle \qquad \forall b' \neq b : \rho'_{b'} = \rho_{b'}}{\mathsf{App}_b(op, n), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\tau} \mathsf{Ret}(n, r), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho'}\colon \mathsf{Clk}}$$

(Op Res Ret)

$$\mathsf{Ret}(n, r), \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk} \xrightarrow{\overline{n}} \langle r\rangle.\ \ddot{\Delta}\colon \widetilde{\mathcal{R}^{\mathcal{H}}}\colon \widetilde{\rho}\colon \mathsf{Clk}$$

Fig. 2. Semantics of a network-attached file system with distributed access control

Definition 4 (Macros for NAFS clients).

Authorization on port k: *Authorization may be requested with* $\mathsf{auth}_k\ x\ \mathsf{for}\ op;\ P$*, which expands to* $(\nu c)\ \overline{\alpha^\circ_{a.k}}\langle op, c\rangle.\ c(x).\ P$*, for some* $a \in \mathcal{A}$*, and* $c \notin \mathtt{fn}(P)$*. The variable x gets bound to a* capability *at runtime.*

File operation using κ on port k: *An authorized file operation may be requested with* $\mathsf{fileopauth}_k\ \kappa/x;\ P$*, which expands to* $(\nu n)\ \overline{\beta_{b.k}}\langle \kappa, n\rangle.\ n(x).\ P$*, where* $n \notin \mathtt{fn}(P)$*,* $\mathbf{proj}_3(\mathbf{message}(\kappa)) = b$*, and* $b \in \mathcal{B}$*. (Recall that for a capability κ that authorizes op, the third component of* $\mathbf{message}(\kappa)$ *is the disk responsible for op.)*

Administration on port k: *Administration may be requested with* $\mathsf{admin}_k\ adm;\ P$*, which expands to* $(\nu n)\ \overline{\alpha_{a.k}}\langle adm, n\rangle.\ n().\ P$*, for some* $a \in \mathcal{A}$*, and* $n \notin \mathtt{fn}(P)$*.*

As in Section 3, we select a subset of clients whom we call honest; these can be any processes with certain static restrictions on their interactions with the file system. In particular, an honest client uses macros only on its own port for sending requests to the file system; each file-operation request is preceded by a capability request for that operation; a capability that is obtained for a file operation is used only in succeeding execution requests for that operation; and finally, as a consequence of Definitions 5 and 6, no other client may send a request to the file system on the port of an honest client.

Definition 5. *A set of honest* NAFS *clients indexed by* $\mathcal{I} \subseteq \mathcal{K}$ *is a set of closed processes* $\{\ddot{C}_i \mid i \in \mathcal{I}\}$*, so that each* $\ddot{C}_i$ *in the set has the following properties:*

- *all macros in* $\ddot{C}_i$ *are on port* i*,*
- *no name in* $\{\alpha^\circ_{a.i'}, \alpha_{a.i'}, \beta_{b.i'} \mid i' \in \mathcal{I}, a \in \mathcal{A}, b \in \mathcal{B}\}$ *appears free in* $\ddot{C}_i$ *before expanding macros,*
- *for each subprocess in* $\ddot{C}_i$ *that is of the form* $\mathsf{auth}_i\ \kappa\ \mathsf{for}\ op;\ P$*, the only uses of* κ *in P are in subprocesses of the form* $\mathsf{fileopauth}_i\ \kappa/x;\ Q$*,*
- *every subprocess Q in* $\ddot{C}_i$ *that is of the form* $\mathsf{fileopauth}_i\ \kappa/x;\ Q$ *is contained in some subprocess* $\mathsf{auth}_i\ \kappa\ \mathsf{for}\ op;\ P$*, such that no subprocess of P that strictly contains Q binds* κ*.*

Dishonest clients $\ddot{C}_j$ $(j \in \mathcal{J})$ are, as in Section 3, left unspecified. They form part of an arbitrary environment that does not have the names $\{\mathrm{K}_b, \alpha^\circ_{a.i}, \alpha_{a.i}, \beta_{b.i} \mid i \in \mathcal{I}, a \in \mathcal{A}, b \in \mathcal{B}\}$ initially.

Definition 6. *A NAS system denoted by* $\mathrm{NAS}(\ddot{\mathbb{C}}_{\mathcal{I}}, \widetilde{\mathcal{R}}, \widetilde{\rho}, \mathsf{Clk})$ *is the closed process* $(\nu_{i\in\mathcal{I},a\in\mathcal{A},b\in\mathcal{B}}\ \alpha^\circ_{a.i}\alpha_{a.i}\beta_{b.i})\ (\Pi_{i\in\mathcal{I}}\ddot{C}_i \mid (\nu_{b\in\mathcal{B}}\ \mathrm{K}_b)\ (\varnothing\colon \widetilde{\mathcal{R}^{\varnothing}} : \widetilde{\rho}\colon \mathsf{Clk}))$*, where*

- $\ddot{\mathbb{C}}_{\mathcal{I}} = \{\ddot{C}_i \mid i \in \mathcal{I}\}$ *is a set of honest* NAFS *clients indexed by* $\mathcal{I}$*,*
- $\varnothing : \widetilde{\mathcal{R}^{\varnothing}} : \widetilde{\rho} : \mathsf{Clk}$ *is an initial* NAFS *control state, and* $\{\mathrm{K}_b, \alpha^\circ_{a.i}, \alpha_{a.i}, \beta_{b.i} \mid i \in \mathcal{I}, a \in \mathcal{A}, b \in \mathcal{B}\} \cap \mathtt{fn}(\widetilde{\mathcal{R}}, \widetilde{\rho}) = \varnothing$*.*

5 Safety and Other Guarantees for Network-Attached Storage

We now establish that IFS is a sound and adequate abstraction for NAFS. Specifically, we show that network-attached storage systems safely implement their specifications as ideal storage systems; we then derive consequences important for security.

IFS *functions derived from* NAFS *functions*:

$$\frac{\mathbf{manager}(op) = a}{\mathbf{may}(k, op, \widetilde{\mathcal{R}}) = \mathbf{may}_a(k, op, \mathcal{R}_a)} \qquad \frac{\mathbf{manager}(adm) = a}{\mathbf{may}(k, adm, \widetilde{\mathcal{R}}) = \mathbf{may}_a(k, adm, \mathcal{R}_a)}$$

$$\frac{\mathbf{disk}(op) = b \qquad \forall b' \neq b : \rho'_{b'} = \rho_{b'} \qquad \langle \rho'_b, r\rangle = \mathbf{execute}_b(op, \rho_b)}{\mathbf{execute}(op, \widetilde{\rho}) = \langle \widetilde{\rho'}, r\rangle} \qquad \frac{\mathbf{manager}(adm) = a \qquad \forall a' \neq a : \mathcal{H}'_{a'} = \mathcal{H}_{a'} \qquad \mathcal{H}'_a = \mathbf{schedule}_a(adm, \mathcal{H}_a, \mathsf{Clk})}{\mathbf{schedule}(adm, \widetilde{\mathcal{H}}, \mathsf{Clk}) = \widetilde{\mathcal{H}'}}$$

$$\frac{\forall a : {\mathcal{R}'_a}^{\mathcal{H}'_a} = \mathbf{update}_a({\mathcal{R}_a}^{\mathcal{H}_a}, \mathsf{Clk})}{\mathbf{update}(\widetilde{\mathcal{R}}^{\widetilde{\mathcal{H}}}, \mathsf{Clk}) = \widetilde{\mathcal{R}'}^{\widetilde{\mathcal{H}'}}} \qquad \frac{\mathbf{manager}(op) = a}{\mathbf{lifespan}(k, op, \widetilde{\mathcal{H}}, \mathsf{Clk}) = \mathbf{expiry}_a(k, op, \mathcal{H}_a, \mathsf{Clk}) - \mathsf{Clk}}$$

Honest IFS*-client code derived from honest* NAFS*-client code*:

$$\lceil 0 \rceil = 0 \qquad \lceil (\nu n)\ P \rceil = (\nu n)\ \lceil P \rceil \qquad \lceil u(\widetilde{x}).\ P \rceil = u(\widetilde{x}).\ \lceil P \rceil \qquad \lceil \overline{u}\langle \widetilde{M} \rangle.\ P \rceil = \overline{u}\langle \widetilde{M} \rangle.\ \lceil P \rceil$$

$$\lceil P \,|\, Q \rceil = \lceil P \rceil \,|\, \lceil Q \rceil \qquad \lceil !P \rceil = !\lceil P \rceil \qquad \lceil \mathsf{if}\ M = N\ \mathsf{then}\ P\ \mathsf{else}\ Q \rceil = \mathsf{if}\ M = N\ \mathsf{then}\ \lceil P \rceil\ \mathsf{else}\ \lceil Q \rceil$$

$$\lceil \mathsf{admin}_i\ adm;\ P \rceil = \mathsf{admin}_i\ adm;\ \lceil P \rceil \qquad \lceil \mathsf{auth}_i\ \kappa\ \mathsf{for}\ op;\ P \rceil = \lceil P \rceil$$

$$\lceil \mathsf{fileopauth}_i\ \kappa / r;\ P \rceil = \mathsf{fileop}_i\ \mathbf{proj}_1(\mathbf{message}(\kappa))/r;\ \lceil P \rceil$$

Fig. 3. Abstraction of NAS systems

In our analyses, we assume that systems interact with arbitrary (potentially hostile) environments. We refer to such environments as *attackers*, and model them as arbitrary closed processes. We study the behaviour of systems via *quizzes*. Quizzes are similar to tests, more specifically to may-tests [14], which capture safety properties.

Definition 7. *A quiz is of the form $(E, c, \widetilde{n}, \widetilde{M})$, where E is an attacker, c is a name, $\widetilde{n}$ is a vector of names, and $\widetilde{M}$ is a vector of closed terms, such that $\widetilde{n} \subseteq \mathtt{fn}(\widetilde{M}) \setminus \mathtt{fn}(E, c)$.*

Informally, a quiz provides an attacker that interacts with the system under analysis, and a goal observation, described by a channel, a set of fresh names, and a message that contains the fresh names. The system passes the quiz if it is possible to observe the message on the channel, by letting the system evolve with the attacker. As the following definition suggests, quizzes make finer distinctions than conventional tests, since they can specify the observation of messages that contain names generated during execution.

Definition 8. *A closed process P passes the quiz $(E, c, \widetilde{n}, \widetilde{M})$ iff $E \mid P \xrightarrow{\tau}{}^{\star} \xrightarrow{\overline{c}} (\nu \widetilde{n})\ \langle \widetilde{M} \rangle.\ Q$ for some Q.*

Intuitively, we intend to show that a NAS system passes a quiz only if its specification passes a similar quiz. Given a NAS system, we "extract" its specification by translating it to an ideal storage system. (The choice of specification is justified by Theorem 2.)

Definition 9. *Let* $\mathrm{NAS}(\ddot{\mathbb{C}}_{\mathcal{I}}, \widetilde{\mathcal{R}}, \widetilde{\rho}, \mathsf{Clk})$ *be a network-attached storage system. Then its specification is the ideal storage system* $\Phi\mathrm{NAS}(\lceil \ddot{\mathbb{C}}_{\mathcal{I}} \rceil, \widetilde{\mathcal{R}}, \widetilde{\rho}, \mathsf{Clk})$*, with* $\lceil . \rceil$ *as defined in Figure 3, and with the* IFS *functions* **may***,* **execute***,* **schedule***,* **update***, and* **lifespan** *derived from their* NAFS *counterparts as shown in Figure 3.*

Next, we map quizzes designed for NAS systems to quizzes that are "at least as potent" on their specifications. Informally, the existence of this map implies that NAFS does not "introduce" any new attacks, *i.e.*, any attack that is possible on NAFS is also possible on IFS. We present the map by showing appropriate translations for attackers and terms.

Definition 10. *Let E be an attacker (designed for NAS systems). Then ΦE is the code*

$$\begin{aligned} E \mid (\nu_{b \in \mathcal{B}} \mathrm{K}_b) \;\; (\; & \mathit{\Pi}_{\alpha_{a.j} \in \mathtt{fn}(E)} !\alpha_{a.j}(adm, n).\, \overline{\alpha_j}\langle adm, n\rangle \\ \mid\; & \mathit{\Pi}_{\beta_{b.j} \in \mathtt{fn}(E)} !\beta_{b.j}(\kappa, n).\, \mathit{\Sigma}_{\beta_{b.j'} \in \mathtt{fn}(E)} \overline{\beta_{j'}}\langle \mathbf{proj}_1(\mathbf{message}(\kappa)), n\rangle \\ \mid\; & \mathit{\Pi}_{\alpha^\circ_{a.j} \in \mathtt{fn}(E)} !\alpha^\circ_{a.j}(op, c).\, \mathit{\Sigma}_{b \in \mathcal{B}} (\nu m)\, \overline{c}\langle \mathbf{mac}(\langle op, \{m\}_{\mathrm{K}_b}, b\rangle, \mathrm{K}_b)\rangle) \end{aligned}$$

Informally, E is composed with a "wrapper" that translates between the interfaces of NAFS and IFS. Administrative requests on $\alpha_{a.j}$ are forwarded on α_j. A file-operation request on $\beta_{b.j}$, with κ as authorization, is first translated by extracting the operation from κ, and then broadcast on all $\beta_{j'}$. Intuitively, κ may be a live, valid capability that was issued in response to an earlier authorization request made on some $\alpha^\circ_{a.j'}$, and a request must now be made on $\beta_{j'}$ to pass the same access-control checks. (This pleasant correspondence is partly due to the properties of **lifespan**.) Finally, authorization requests on $\alpha^\circ_{a.j}$ are "served" by returning fake capability-like terms. Intuitively, these terms are indistinguishable from NAFS capabilities under all possible computations by E. To that end, fake secret keys replace the secret NAFS keys $\{\mathrm{K}_b \mid b \in \mathcal{B}\}$; the disk b is non-deterministically "guessed" from the finite set $\mathcal{B}$; and an encrypted unique nonce replaces the NAFS timestamp. Notice that the value of the NAFS clock need not be guessed to fake the timestamp, since by design, each NAFS timestamp is unique and unintelligible to E.

We now formalize the translation of terms (generated by NAFS and its clients). As indicated above, the translation preserves indistinguishability by attackers, which we show by Proposition 1.

Definition 11. *Let m range over names not in $\{\mathrm{K}_b \mid b \in \mathcal{B}\}$, and $\mathcal{M}$ range over sequences of terms. We define the judgment $\mathcal{M} \vdash \diamond$ by the following rules:*

$$\varnothing \vdash \diamond \qquad \frac{\mathcal{M} \vdash \diamond}{\mathcal{M}, m \vdash \diamond} \qquad \frac{\mathcal{M} \vdash \diamond \qquad f \text{ is a function symbol} \qquad \widetilde{M} \subseteq \mathcal{M}}{\mathcal{M}, f(\widetilde{M}) \vdash \diamond}$$

$$\frac{\mathcal{M} \vdash \diamond \qquad \{\langle m, \_\rangle\}_{\mathrm{K}_b} \notin \mathcal{M} \qquad op \in \mathcal{M}}{\mathcal{M}, \mathbf{mac}(\langle op, \{\langle m, \mathsf{Clk}\rangle\}_{\mathrm{K}_b}, b\rangle, \mathrm{K}_b), \{\langle m, \mathsf{Clk}\rangle\}_{\mathrm{K}_b} \vdash \diamond}$$

We say that $\mathcal{M}$ is valid if $\mathcal{M} \vdash \diamond$, and define Φ on terms in a valid sequence:

$$\Phi m = m \qquad \Phi f(\widetilde{M}) = f(\widetilde{\Phi M}) \qquad \Phi\{\langle m, \mathsf{Clk}\rangle\}_{\mathrm{K}_b} = \{m\}_{\mathrm{K}_b}$$

$$\Phi\mathbf{mac}(\langle op, \{\langle m, \mathsf{Clk}\rangle\}_{\mathrm{K}_b}, b\rangle, \mathrm{K}_b) = \mathbf{mac}(\langle \Phi op, \{m\}_{\mathrm{K}_b}, b\rangle, \mathrm{K}_b)$$

Proposition 1. *Let M, M' belong to a valid sequence. Then $M = M'$ iff $\Phi M = \Phi M'$ (where $=$ is equational, and not merely structural, equality).*

Our main result, which we state next, says that whenever a NAS system passes a quiz, its specification passes a quiz that is meaningfully related to the former:

Theorem 1 (Implementation soundness). *Let* NAS *be a network-attached storage system. If* NAS *passes some quiz $(E, c, \widetilde{n}, \widetilde{M})$, then $\widetilde{M}$ belong to a valid sequence, and Φ*NAS *passes the quiz $(\Phi E, c, \widetilde{n}, \widetilde{\Phi M})$.*

The converse of this theorem does not hold, since $\varPhi E$ can always return a capability-like term, while NAFS does not if an access check fails. Consequently, full abstraction breaks. In [5], where the outcome of any access check is fixed, we achieve full abstraction by letting the file system return a fake capability whenever an access check fails. (The wrapper can then naïvely translate execution requests, much as in here.) However, it becomes impossible to translate attackers when dynamic administration is allowed (even if we let NAFS return fake capabilities for failed access checks). Intuitively, $\varPhi E$ cannot consistently guess the outcome of an access check when translating file-operation requests at runtime—and for any choice of $\varPhi E$ given E, this problem can be exploited to show a counterexample to full abstraction.

Full abstraction can also be broken by honest clients, with the use of expired capabilities. One can imagine more complex client macros that check for expiry before sending requests. (Such macros require the NAFS clock to be shared with the clients.) Still, the "late" check by NAFS (after receiving the request) cannot be replaced by any appropriate "early" check (before sending the request) without making additional assumptions on the scheduling of communication events over the network.

One might of course wonder if the specifications for NAS systems are "too weak" (thereby passing quizzes by design), so as to make Theorem 1 vacuous. The following standard completeness result ensures that this is not the case.

Theorem 2 (Specification completeness). *Let two systems be distinguishable if there exists a quiz passed by one but not the other. Then two ideal storage systems* IS_1 *and* IS_2 *are distinguishable only if there are distinguishable network-attached storage systems* NAS_1 *and* NAS_2 *such that* $\varPhi\mathrm{NAS}_1 = \mathrm{IS}_1$ *and* $\varPhi\mathrm{NAS}_2 = \mathrm{IS}_2$.

It follows that every quiz passed by an ideal storage system can be concretized to a quiz passed by some NAS system with that specification.

Several safety properties can be expressed as quiz failures.Next we show two"safety-preservation" theorems that follow as corollaries to Theorem 1. The first one concerns secrecy; the second, integrity. We model the initial knowledge of an attacker with a set of names, as in [2]; let S range over such sets.

Definition 12. *Let S be a set of names. An attacker E is a S-adversary if* $\mathtt{fn}(E) \subseteq S$.

We may then express the hypothesis that a system keeps a term secret by claiming that it fails any quiz whose goal is to observe that term on a channel that is initially known to the attacker.

Definition 13. *A closed process P keeps the closed term M secret from a set of names S if P does not pass any quiz $(E, s, \widetilde{n}, M)$ where E is an S-adversary and $s \in S$.*

We now derive preservation of secrecy by NAS implementations. For any S modeling the initial knowledge of a NAS attacker, let $\varPhi S$ be an upper bound on S, as follows:

$$\varPhi S = S \cup \{\alpha_j, \alpha^\circ_{j'}, \beta_{j''} \mid \alpha_{a.j}, \alpha^\circ_{a.j'}, \beta_{b.j''} \in S, a \in \mathcal{A}, b \in \mathcal{B}\}$$

Note that for any S-adversary E, $\varPhi E$ is a $\varPhi S$-adversary. Further, note that the inclusion of the name $\alpha_{a.j}$ (*resp.* $\alpha^\circ_{a.j'}$, $\beta_{b.j''}$) in S suggests that E knows how to impersonate the NAFS client $\ddot{C}_j$ for requesting policy modifications (*resp.* capabilities, file operations);

the corresponding inclusion of the name α_j (*resp.* $\alpha^\circ_{j'}$, $\beta_{j''}$) in ΦS allows the abstract attacker ΦE to impersonate the IFS client C_j. Thus, the following result says that a secret that may be learnt from a NAS system may be also be learnt from its specification with comparable initial knowledge; in other words, a NAS system protects a secret whenever its specification protects the secret.

Corollary 1 (Secrecy preservation). *Let* NAS *be a network-attached storage system, S a finite set of names, and M a closed term that belongs to a valid sequence. Then* NAS *keeps M secret from S if Φ*NAS *keeps ΦM secret from ΦS.*

Next we derive preservation of integrity by NAS implementations. In fact, we treat integrity as one of a larger class of safety properties whose violations may be detected by letting a system adequately monitor itself, and we derive preservation of all such properties in NAS. For this purpose, we hypothesize a set of monitoring channels that may be used to communicate warnings between various parts of the system, and to signal violations on detection; we protect such channels from attackers by construction. In particular, clients can use monitoring channels to communicate about begin- and end-events, and to warn whenever an end-event has no corresponding begin-event (thus indicating the failure of a correspondence assertion [9]).

Definition 14. *A name n is purely communicative in a closed process P if any occurrence of n in P is in the form $n(\widetilde{x}).\ Q$ or $\overline{n}\langle\widetilde{M}\rangle.\ Q$. Let S be a finite set of names. Then the set of names W monitors a closed process P under S if $W \cap S = \varnothing$ and each $w \in W$ is purely communicative in P.*

Any message on a monitoring channel may be viewed as a warning.

Definition 15. *Let W monitor P under S. Then S causes P to warn on W if for some S-adversary E and $w \in W$, P passes a quiz of the form $(E, w, \widetilde{n}, \widetilde{M})$.*

The following result says that whenever an attack causes a warning in a NAS system, an attack with comparable initial knowledge causes that warning in its specification. In other words, since a specification may contain monitoring for integrity violations, a NAS system protects integrity whenever its specification protects integrity.

Corollary 2 (Integrity preservation). *Let W monitor an abstracted network-attached storage system Φ*NAS *under ΦS. Then S does not cause* NAS *to warn on W if ΦS does not cause Φ*NAS *to warn on W.*

6 Conclusion

In this paper we study networked storage systems with distributed access control. In particular, we relate those systems to simpler centralized storage systems with local access control. Viewing the latter systems as specifications of the former ones, we establish the preservation of safety properties of the specifications in the implementations. We derive the preservation of standard secrecy and integrity properties as corollaries. We expect that such results will be helpful in reasoning about the correctness and the security of larger systems (which may, for example, include non-trivial clients that rely on file storage). In that context, our results imply that we can do proofs using the simpler centralized storage systems instead of the networked storage systems. In our current work, we are developing proof techniques that leverage this simplification.

Acknowledgments. We thank Cédric Fournet and Ricardo Corin for helpful comments. This work was partly supported by the National Science Foundation under Grants CCR-0204162, CCR-0208800, and CCF-0524078, and by Livermore National Laboratory, Los Alamos National Laboratory, and Sandia National Laboratory under Contract B554869.

References

1. M. Abadi. Protection in programming-language translations. In *ICALP'98: International Colloquium on Automata, Languages and Programming*, pages 868–883. Springer-Verlag, 1998.
2. M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. *Journal of the ACM*, 52(1):102–146, 2005.
3. M. Abadi and C. Fournet. Mobile values, new names, and secure communication. In *POPL'01: Principles of Programming Languages*, pages 104–115. ACM, 2001.
4. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, 1999.
5. A. Chaudhuri and M. Abadi. Formal security analysis of basic network-attached storage. In *FMSE'05: Formal Methods in Security Engineering*, pages 43–52. ACM, 2005.
6. G. A. Gibson, D. P. Nagle, K. Amiri, F. W. Chang, E. Feinberg, H. G. C. Lee, B. Ozceri, E. Riedel, and D. Rochberg. A case for network-attached secure disks. Technical Report CMU–CS-96-142, Carnegie Mellon University, 1996.
7. H. Gobioff. *Security for a High Performance Commodity Storage Subsystem.* PhD thesis, Carnegie Mellon University, 1999.
8. H. Gobioff, G. Gibson, and J. Tygar. Security for network-attached storage devices. Technical Report CMU-CS-97-185, Carnegie Mellon University, 1997.
9. A. D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoritical Computer Science*, 300(1-3):379–409, 2003.
10. D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *PODC'02: Principles of Distributed Computing*, pages 108–117. ACM, 2002.
11. E. L. Miller, D. D. E. Long, W. E. Freeman, and B. Reed. Strong security for network-attached storage. In *FAST'02: File and Storage Technologies*, pages 1–13. USENIX, 2002.
12. R. Milner. Fully abstract models of typed lambda-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.
13. R. Milner. The polyadic pi-calculus: a tutorial. In *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag, 1993.
14. R. D. Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, 1984.
15. B. C. Reed, E. G. Chron, R. C. Burns, and D. D. E. Long. Authenticating network-attached storage. *IEEE Micro*, 20(1):49–57, 2000.
16. F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
17. Y. Zhu and Y. Hu. SNARE: A strong security scheme for network-attached storage. In *SRDS'03: Symposium on Reliable Distributed Systems*, pages 250–259. IEEE, 2003.