A Proof-Carrying File System with Revocable and Use-Once Certificates

Jamie Morgenstern, Deepak Garg, and Frank Pfenning

Carnegie Mellon University
{jamiemmt,dg,fp}@cs.cmu.edu

Abstract. We present the design and implementation of a file system which allows authorizations dependent on revocable and use-once policy certificates. Authorizations require explicit proof objects, combining ideas from previous authorization logics and Girard's linear logic. Use-once certificates and revocations lists are maintained in a database that is consulted during file access. Experimental results demonstrate that the overhead of using the database is not significant in practice.

1 Introduction

In the past decade, proof-carrying authorization (PCA) [4,6,7,15] has emerged as a promising, open-ended architecture for rigorous enforcement of authorization policies. In PCA, policy rules and other policy-relevant credentials are abstractly represented as formulas of a formal logic (as opposed to a possible low-level representation in system databases or access control lists), and published in signed certificates that are distributed to authorized principals. Access to a protected resource is allowed by a reference monitor if and only if the principal requesting access produces enough certificates to authorize the access and a *formal logical proof* which explains how the certificates combine to justify the access. Through this combination of public-key cryptography and logic, PCA rigorously enforces authorization policies at a high-level of abstraction. PCA-based authorization has been deployed and tested in a variety of systems, including a web server [6], physical devices like office doors in the Grey system [7], and a file system, PCFS [15].

A significant shortcoming of prior work on PCA is the lack of a satisfactory treatment of *use-once certificates*, i.e., certificates that can be used only once for authorization. For instance, if an individual buys a movie from a pay-per-view website, the certificate authorizing her to stream the movie should be usable only once. Incorporating use-once certificates in proof-carrying authorization is challenging because it not only requires the reference monitor to track consumption of such certificates (which adds extra work, and potentially slows down the reference monitor), but also requires a change to the logic itself to track uses of each use-once certificate in a proof. As its main contribution, the present paper fills this gap — we discuss the design, implementation, and evaluation of a PCA-based file system, LPCFS, that allows authorizations to depend on use-once certificates in addition to the usual persistent certificates.

LPCFS extends our prior PCA-based file system, PCFS [15], which does not allow use-once certificates. First, we extend the logic BL used for representing policies in PCFS with ideas from Girard's linear logic [16] that allows precise counting of use of resources in proofs. We call the resulting logic BL^L (the superscript L stands for linear). Second, we extend proof construction and proof verification tools of PCFS to deal with linearity. Third, we extend the implementation of PCFS with a database for storing and tracking use-once certificates. All such certificates are added to the database by their creators (this is in contrast to persistent certificates that are given directly to beneficiaries), and the reference monitor marks them consumed when it successfully authorizes access based on them. An important concern in the use of the database is atomicity — all certificates present in the justification of an access must be checked and marked consumed in a single atomic step. To ensure this property, we use exclusive transactions on the database. Because marking consumption requires an update to the database, authorizations with use-once certificates incur performance overhead in the reference monitor, but we show through experimental evaluation that, given the scarcity of use-once certificates in practice, this overhead is reasonable.

A second contribution of this paper is to extend PCFS with support for *revocation* of both use-once and persistent certificates by their issuers. To this end, we include a table of revoked certificates in the database; policy creators add revoked certificates to this table, and the reference monitor checks that each certificate in a submitted request is absent from this table. Since the reference monitor does not update this table when checking the revocation of certificates, it incurs very little overhead by allowing revocation, as we confirm in our experimental evaluation. An additional design consideration is that the check for certificate revocation must be made atomically with the check for use-once certificates described above.

The rest of this paper is organized as follows. We start by discussing related work and comparing LPCFS to it. In Section 2, we motivate use-once certificates, revocation, and the syntax of our logic BL^L through an example. Section 3 presents the proof theory of the logic BL^L briefly. Section 4 describes the design and implementation of our file system, LPCFS, that enforces policies written in BL^L. Section 5 presents experimental measurements of the overhead of tracking both use-once certificates and certificate revocations. Section 6 concludes the paper. The source code of LPCFS is available under a liberal license from the authors' webpages. A detailed description of the logic BL^L and proofs of its metatheorems are available in the second author's thesis [12, Chapter 9].

Related Work We briefly discuss some closely related work. The idea of using logic for authorization goes back to the work of Lampson et al. [17] and has been adopted in several subsequent proposals. For a general description of the area, we refer the reader to two surveys [2,3].

Proof-carrying authorization (PCA), or the use of formal proofs for authorization, was first proposed by Appel and Felten [4] and evolved in two implemented systems [6,7], before two of the present authors applied it to a file

system, PCFS [15], which the present paper extends with support for use-once and revocable certificates. To avoid the overhead of checking a proof and its certificates in the reference monitor at each access (as in PCA), PCFS offlines the work of proof and certificate verification to a trusted verifier that issues a signed capability in return, which is then used to authorize access at the file system's reference monitor. The same architecture is inherited by our file system LPCFS, except that we use the signed capability to also carry lists of both use-once and persistent certificates used in a proof to the reference monitor, where the lists are checked against the database (see Section 4 for details).

Our use of a *centralized* database for tracking use-once (and also revoked) certificates contrasts from a fully distributed implementation, as in the work of Bowers et al. [8]. In that work, Bowers et al. assume that each use-once certificate is tracked by a remote trusted party called a ratifier, and use a contract signing protocol between ratifiers to ensure that all use-once certificates in a proof are marked consumed atomically. However, this is slow in practice, and unnecessary for applications that are centralized, as is the case for our file system.

Linear logic was first proposed by Girard [16]. The use of linear logic for representing use-once certificates was first proposed by two of the authors [13] and independently by Cederquist et al. [9]. Our policy logic BL^L is an amalgamation of the logic BL used in PCFS and linearity from the work of the authors [13]. Barth and Mitchell [5] have used a fragment of linear logic to study monotonicity properties of algorithms for enforcement of digital rights (as in DRM applications).

Some systems, e.g., Nexus [20], support use-once credentials by tracking their use in the reference monitor, but do not distinguish them from persistent credentials in the policy logic. This approach has the disadvantage that proof construction and verification tools become oblivious to credential consumption and seemingly correct proofs of authorization may be rejected by the reference monitor because they utilize a use-once credential more than once.

2 Motivating Example

In this section, we motivate the need for use-once certificates through the example of a fictitious online movie rental service's authorization policy. We also give a brief overview of the syntax of our logic BL^L, describe a formalization of the example policy in the logic, and motivate the need for certificate revocation.

Example: A Movie Rental Service. Consider the following policy for WebFilm, a hypothetical online movie rental service. If principal `K` is a member of the service, then she has access to view movie listings. If `K` is a member, and `K` purchases a movie ticket, then `K` has the ticket which can be traded for the right to download a movie `M`. If `K` exchanges a ticket in order to watch Ferris Bueller's Day Off, then `K` no longer has the ticket but can read Ferris Bueller's Day Off from the server for the next 30 days. Different principals are responsible for different kinds of facts about the system: `MovieServer` controls access to

movies, `UserDB` keeps track of the user database, and `TicketHolder` keeps a list of tickets held by members and records of payment. `Alice` is a user of the rental service.

In order to formalize this policy within a logic, we need the notion of statements made by principals, consumable resources (such as money and tickets), and time-sensitive permissions (a principal can download a movie for 30 days after purchasing it with a ticket), all of which our logic BL^{L} supports.

A Brief Introduction to the Logic $\mathbf{BL^L}$. BL^{L} is a logic for distributed access control, different principals making statements about access rights, together with the notion of consumable facts or resources which are consumed in deriving other facts. As in its precursor BL [15], facts may be time-sensitive: the proposition $A @ [u_1, u_2]$ means that proposition A holds between time points u_1 and u_2.

Statements made by principals in the system together form an access policy, or a list of hypotheses from which inferences about access to resources can be made. Because statements can be either *persistent*, in that they may be used arbitrarily many times through the course of reasoning, or *linear*, in that they may be used at most once, it is necessary to have two different connectives to represent statements. K once A means that the principal K asserts the proposition A *as a consumable resource*, which can be used only once in a proof of authorization. This contrasts from K says A, which means that principal K asserts the persistent fact A. For example, Bank once ($\mathtt{HasMoney}$ K) is a proposition which represents the bank stating that K has money; this fact may be exchanged for some other fact (e.g, that K owns a Ferrari), but it cannot be used more than once.

Two other connectives, implication and conjunction, have linear counterparts with meanings different from conventional logic. Linear implication, written $A \multimap B$, describes an implication which consumes the (linear) fact A and produces the linear fact B. The proposition $A \otimes B$ means that both A and B are true. Finally $!A$ represents that the proposition A may be used arbitrarily many times (i.e., A is persistent).

Example Formalized. Next, we formalize the authorization policies of WebFilm in BL^{L}. We start by describing the predicates needed for the formalization. The atomic proposition ($\mathtt{may}$ K F R) represents the authorization of permission R on file F to principal K, e.g., ($\mathtt{may}$ Alice FBDO read) gives `Alice` permission to read FBDO (Ferris Bueller's Day Off). Another atom used in the formalization is `Member` K, representing the assertion that K is a member of the service; in our example, the user database will state this *persistent* fact. `HasTicket` K represents that K has a ticket, which will be stated by the `TicketHolder` as a *linear* (use-once) fact. `GetMovie` M, the assertion of desire for a movie M, will be asserted by users of the system with the wish to purchase the movie M. `Purchased` K M represents the record of K having bought the movie M, which will be asserted by the `TicketHolder` as a persistent fact. `HasMoneyForTicket` K, that K has the money to purchase a ticket, will be asserted by K's bank as a linear fact

that can be used to purchase movie tickets. Finally, BuyTicket is asserted as a linear fact by users wanting to purchase movie tickets.

Using these atoms, the policy of WebFilm described earlier can be represented in BL^L as the following propositions. As a convention, any variables in uppercase letters are implicitly assumed to be universally quantified inside the outermost assertion K says $\bullet$.

$$\gamma_1 = \mathsf{MovieServer\ says}\ \ ((\mathsf{UserDB\ says}\ (\mathtt{member}\ K)) \multimap\ !\,(\mathtt{may}\ K\ \mathit{movieList}\ \mathsf{read}))\ @\ [-\infty, \infty]$$

$$\begin{aligned}\gamma_2 = \mathsf{MovieServer\ says}\ \ (((&\mathsf{UserDB\ says}\ (\mathtt{member}\ K))\\ &\otimes (\mathsf{TicketHolder\ once}\ (\mathtt{HasTicket}\ K))\\ &\otimes (\mathsf{K\ once}\ (\mathtt{GetMovie}\ M))) \multimap\\ &!\,((\mathtt{may}\ K\ M\ \mathsf{read})\ @\ [T, T+30])\\ &\otimes (\mathsf{TicketHolder\ \ says}\ (\mathtt{Purchased}\ K\ M)))\ @\ [T, T]\end{aligned}$$

$$\begin{aligned}\gamma_3 = \mathsf{TicketHolder\ says}\ \ (((&\mathsf{UserDB\ says}\ (\mathtt{member}\ K))\\ &\otimes (\mathsf{Bank\ once}\ (\mathtt{HasMoneyForTicket}\ K))\\ &\otimes (\mathsf{K\ once}\ \mathtt{BuyTicket}))\\ &\multimap (\mathtt{HasTicket}\ K))\ @\ [-\infty, \infty]\end{aligned}$$

$$\gamma_4 = \mathsf{UserDB\ says}\ \ (\mathtt{member}\ \mathsf{Alice})\ @\ [-\infty, \infty]$$

The first rule above means that the MovieServer states that if the UserDB states that K is a member, then K can read the *movieList* any number of times. This rule (like all others above) is persistent because it contains the says connective at the top level. The permission granted by the rule is also persistent because it contains a ! connective in front of it. The suffix $@\ [-\infty, \infty]$ at the end of the rule means that the rule is valid in all time intervals.

As another example, the second rule above means that if at time T, principal UserDB states that K is a member, K holds a ticket (TicketHolder once (HasTicket K)), and K wants to buy the movie M (K once (GetMovie M)), then K may read movie M any number of times in the interval $[T, T+30]$ and we record the fact that K has purchased the movie M. Note that the ticket and K's desire to purchase the movie (K once (GetMovie M)) are consumed as part of the rule, thus preventing the rule from firing again, unless K produces another ticket and another certificate expressing the desire to purchase the movie.

In addition to these policy rules, we need several linear (use-once) propositions to draw meaningful conclusions. For instance, the following use-once credentials state respectively that at time T_0, Alice has enough money to buy a ticket, that she wants to buy a ticket, and that she wants to obtain the movie FBDO.

$$\delta_1 = \mathsf{Bank\ once}\ \ (\mathtt{HasMoneyForTicket}\ \mathsf{Alice})\ @\ [T_0, T_0]$$
$$\delta_2 = \mathsf{Alice\ once}\ \ (\mathtt{BuyTicket})\ @\ [T_0, T_0]$$
$$\delta_3 = \mathsf{Alice\ once}\ \ (\mathtt{GetMovie}\ \mathsf{FBDO})\ @\ [T_0, T_0]$$

Intuitively, we may expect that from the policy rules $\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$ and the use-once assumptions $\Delta = \{\delta_1, \delta_2, \delta_3\}$, we can construct a proof that Alice can read the file FBDO in the interval $[T_0, T_0 + 30]$. The proof would *consume*

the use-once assumptions Δ. We now explain informally how this deduction is done in BL^{L}.

First, by modus ponens on the rule γ_3 and the premises γ_4, δ_1, and δ_2, we obtain the linear fact $\delta_4 = \texttt{TicketHolder}\ \mathsf{once}\ (\texttt{HasTicket}\ \mathsf{Alice})\ @\ [T_0, T_0]$. Note that due to the use of the connective $\multimap$ in γ_3, this deduction consumes the linear assumptions δ_1 and δ_2, leaving only Γ, δ_3, and the new fact δ_4. Next, by modus ponens on the rule γ_2 with the premises γ_4, δ_4, and δ_3, we deduce that $(!(\texttt{may}\ \mathsf{Alice}\ \mathsf{FBDO}\ \mathsf{read})\ @\ [T_0, T_0 + 30]) \otimes (\mathsf{TicketHolder}\ \ \mathsf{says}\ (\texttt{Purchased}\ \mathsf{Alice}\ \mathsf{FBDO}))\ @\ [T_0, T_0]$. The first component of this tensor ($\otimes$) gives Alice the permission to read FBDO any number of times in the interval $[T_0, T_0 + 30]$, as expected. Also note that the second deduction step consumes both remaining linear facts δ_2 and δ_4.

Linear Proof-Carrying Authorization. How is deduction in BL^{L} related to policy enforcement in LPCFS? Consider a state of the system with persistent policy facts Γ and linear policy facts Δ. Suppose that a principal K constructs a proof M of authorization φ using a subset Δ' of the linear facts Δ and any subset of the persistent credentials Γ. When an access based on M is allowed, the reference monitor marks the subset Δ' consumed (in its central database), leaving only the linear facts $\Delta - \Delta'$ for use in future authorizations. All of Γ persists and can be used again.

Revocation. Revocation is a mechanism for canceling a previously issued certificate. For instance, assuming that WebFilm watermarks all its movies with identities of users who download them, the service may want to cancel Alice's membership if it discovers that Alice is illegally sharing movies downloaded from WebFilm. In our example and LPCFS, WebFilm can do this by telling the reference monitor that the certificate γ_4 that authorizes Alice's membership to the service has been revoked. The reference monitor stores this revocation in its database, thus rejecting any further authorizations that use γ_4.

It is important to note the distinction between use of a linear certificate, the revocation of a (persistent or linear) certificate, and time-based expiration of a certificate. A linear certificate is used when a proof based on it is successfully used to authorize access. Revocation takes place when a principal decides that a part P of her policy is flawed. She then adds the name of P to the revoked table in the reference monitor, so that no proof which relies on P will check successfully. A time-based expiration means that the certificate $A\ @\ [u_1, u_2]$ cannot be used to deduce an authorization valid in an interval other than $[u_1, u_2]$, unless the policy explicitly allows this. Unlike linearity and time-based expiration, both of which have explicit representation in the logic, revocation has no representation in the logic and is an artifact of the enforcement architecture only.

3 The Policy Logic BL^{L}

This section describes the syntax and, briefly, the proof theory of BL^{L}. To keep the presentation simple, we omit a description of some standard connectives of linear logic, including $\mathbf{1}$, $\oplus$ and &. We also do not describe BL^{L}'s treatment of stateful atoms and constraints, which are inherited from its predecessor BL [14]. Formulas (propositions) A, B have the following syntax. P denotes an atomic formula, which is a predicate applied to a list of terms, and σ denotes a type (sort) of terms.

$$\begin{array}{c}\textsf{Formulas}\ A, B ::= P \mid A \otimes B \mid A \multimap B \mid\ !\,A \mid \forall x{:}\sigma.A \mid \exists x{:}\sigma.A \mid \\ K\ \textsf{says}\ A \mid K\ \textsf{once}\ A \mid A\ @\ [u_1, u_2]\end{array}$$

The intuitive meanings of the connectives were explained and illustrated in Section 2. Deduction is formally defined over judgments, which are assertions with formulas as subjects [11,19], and which may be established through proofs. We need four judgments to describe the constructs of BL^{L}: (1) $A \circ [u_1, u_2]$: Formula A holds throughout the interval $[u_1, u_2]$, and this fact can be used any number of times (2) $A \star [u_1, u_2]$: Formula A holds throughout the interval $[u_1, u_2]$, and this fact must be used once, (3) $K\ \textsf{claims}\ A \circ [u_1, u_2]$: Principal K asserts throughout the interval $[u_1, u_2]$ that formula A holds, and this fact may be used any number of times, and (4) $K\ \textsf{claims}\ A \star [u_1, u_2]$: Principal K asserts throughout the interval $[u_1, u_2]$ that formula A holds, and this fact must be used exactly once. Although inference is performed over judgments, the latter can also be represented equivalently (internalized) in the syntax of formulas. $A \star [u_1, u_2]$ is internalized as $A\ @\ [u_1, u_2]$; $A \circ [u_1, u_2]$ is internalized as $!(A\ @\ [u_1, u_2])$; $K\ \textsf{claims}\ A \star [u_1, u_2]$ is internalized as $(K\ \textsf{once}\ A)\ @\ [u_1, u_2]$; $K\ \textsf{claims}\ A \circ [u_1, u_2]$ is internalized as $(K\ \textsf{says}\ A)\ @\ [u_1, u_2]$.

Deduction is formalized with inference rules, which establish hypothetical judgments or sequents: $\Sigma; \Gamma; \Delta \xrightarrow{\nu} A \star [u_1, u_2]$, where

- Σ is a list of variables occurring free in the rest of the sequent, together with their types (sorts)
- Γ is a list of persistent assumptions of the form $A \circ [u_1, u_2]$ and $K\ \textsf{claims}\ A \circ [u_1, u_2]$
- Δ is a list of use-once assumptions of the form $A \star [u_1, u_2]$ and $K\ \textsf{claims}\ A \star [u_1, u_2]$
- $\nu = K', u_1', u_2'$, a triple containing a principal K' and a time interval $[u_1', u_2']$, is called the *view* of the sequent

The meaning of the entire sequent is: "Parametrically in the variables in Σ, the judgment $A \star [u_1, u_2]$ can be derived using the persistent assumptions Γ any number of times, and each of the use-once assumptions Δ exactly once. Further, this derivation is relative to the assumption that all statements made by principal K about the interval $[u_1', u_2']$ are true." In the following we describe some of the inference rules of the logic's proof system.

Axiom. The logic BL^L has one axiom that allows us to conclude that an atom P holds during an interval from the linear assumption that P holds on a larger interval. Further, to properly account for the use of resources, Δ must not contain any other assumption.

$$\frac{\Sigma; \Gamma \models u_1' \leq u_1 \qquad \Sigma; \Gamma \models u_2 \leq u_2'}{\Sigma; \Gamma; P \star [u_1', u_2'] \xrightarrow{\nu} P \star [u_1, u_2]}\text{init}$$

Copy. The following rule allows copying of a persistent assumption into the linear context Δ, where it can be analyzed by rules presented later. The persistent assumption is retained in the premise to allow it to be used again.

$$\frac{\Sigma; \Gamma, A \circ [u_1, u_2]; \Delta, A \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2']}{\Sigma; \Gamma, A \circ [u_1, u_2]; \Delta \xrightarrow{\nu} B \star [u_1', u_2']}\text{copy}$$

Connective $\otimes$*.* The so-called linear multiplicative conjunction, $\otimes$, is defined by the following two inference rules:

$$\frac{\Sigma; \Gamma; \Delta_1 \xrightarrow{\nu} A_1 \star [u_1, u_2] \qquad \Sigma; \Gamma; \Delta_2 \xrightarrow{\nu} A_2 \star [u_1, u_2]}{\Sigma; \Gamma; \Delta_1, \Delta_2 \xrightarrow{\nu} A_1 \otimes A_2 \star [u_1, u_2]}\otimes\text{R}$$

$$\frac{\Sigma; \Gamma; \Delta, A_1 \star [u_1, u_2], A_2 \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2']}{\Sigma; \Gamma; \Delta, A_1 \otimes A_2 \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2']}\otimes\text{L}$$

The first rule says that to establish $A_1 \otimes A_2$ (in some interval), we must split the linear resources into Δ_1 and Δ_2, using the first set to prove A_1 and the other to prove A_2 (both in the same interval). Dually, the second rule means that the assumption $A_1 \otimes A_2$ is equivalent to having both A_1 and A_2. Note that the principal linear judgment $A_1 \otimes A_2 \star [u_1, u_2]$ is not included in the premise of the second rule, to prevent it from being used again.

Connective $\multimap$*.* Intuitively, the judgment $A_1 \multimap A_2 \star [u_1, u_2]$ means that there is a method to consume a proof of A_1 on any subset of $[u_1, u_2]$ and produce a proof of A_2 on the same subset. This is captured in the following rules of inference.

$$\frac{\Sigma, x_1{:}\mathsf{time}, x_2{:}\mathsf{time}; \Gamma, u_1 \leq x_1, x_2 \leq u_2; \Delta, A_1 \star [x_1, x_2] \xrightarrow{\nu} A_2 \star [x_1, x_2]}{\Sigma; \Gamma; \Delta \xrightarrow{\nu} A_1 \multimap A_2 \star [u_1, u_2]}\multimap\text{R}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma; \Delta_1 \xrightarrow{\nu} A_1 \star [u_1', u_2'] \\ \Sigma; \Gamma; \Delta_2, A_2 \star [u_1', u_2'] \xrightarrow{\nu} B \star [u_1'', u_2''] \qquad \Sigma; \Gamma \models u_1 \leq u_1' \qquad \Sigma; \Gamma \models u_2' \leq u_2\end{array}}{\Sigma; \Gamma; \Delta_1, \Delta_2, A_1 \multimap A_2 \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1'', u_2'']}\multimap\text{L}$$

Connective once*.* A proof of K once $A \star [u_1, u_2]$ is a proof of $A \star [u_1, u_2]$ in the view K, u_1, u_2 (rule onceR) using only assumptions of the forms K' claims $A' \circ [u_1', u_2']$ in Γ (notation $\Gamma|$) and K' claims $A' \star [u_1', u_2']$ in Δ (notation $\Delta|$). Note that to ensure that no linear resources are lost in moving from the conclusion of

the rule to the premise, the linear assumptions in the conclusion are exactly $\Delta|$. Dually, the assumption K once $A \star [u_1, u_2]$ can be used to deduce $A \star [u_1, u_2]$ if the view $\nu = K, u_b, u_e$ satisfies $[u_b, u_e] \subseteq [u_1, u_2]$ (rules onceL and lclaims).

$$\frac{\Sigma; \Gamma|; \Delta| \xrightarrow{K,u_1,u_2} A \star [u_1, u_2]}{\Sigma; \Gamma; \Delta| \xrightarrow{\nu} K \text{ once } A \star [u_1, u_2]}\text{onceR}$$

$$\frac{\Sigma; \Gamma; \Delta, K \text{ claims } A \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2']}{\Sigma; \Gamma; \Delta, K \text{ once } A \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2']}\text{onceL}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma; \Delta, A \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2'] \\ \nu = K, u_b, u_e \qquad \Sigma; \Gamma \models u_1 \leq u_b \qquad \Sigma; \Gamma \models u_e \leq u_2\end{array}}{\Sigma; \Gamma; \Delta, K \text{ claims } A \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2']}\text{lclaims}$$

Connective says. The connective says behaves similarly to once, except that in the rule saysR, we require the linear context to be empty. This is because K says A is a persistent fact, which may be used multiple times, so it cannot depend on any linear assumptions. Dually, in the rule (claims), we retain the principal formula in the premise to allow it to be used multiple times.

$$\frac{\Sigma; \Gamma|; \cdot \xrightarrow{K,u_1,u_2} A \star [u_1, u_2]}{\Sigma; \Gamma; \cdot \xrightarrow{\nu} K \text{ says } A \star [u_1, u_2]}\text{saysR}$$

$$\frac{\Sigma; \Gamma, K \text{ claims } A \circ [u_1, u_2]; \Delta \xrightarrow{\nu} B \star [u_1', u_2']}{\Sigma; \Gamma; \Delta, K \text{ says } A \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2']}\text{saysL}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma, K \text{ claims } A \circ [u_1, u_2]; \Delta, A \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2'] \\ \nu = K, u_b, u_e \qquad \Sigma; \Gamma \models u_1 \leq u_b \qquad \Sigma; \Gamma \models u_e \leq u_2\end{array}}{\Sigma; \Gamma, K \text{ claims } A \circ [u_1, u_2]; \Delta \xrightarrow{\nu} B \star [u_1', u_2']}\text{claims}$$

Metatheory. We have verified standard metatheoretic properties of the proof system of BL^{L}. For instance, we prove that the rules of cut and identity (which generalizes the init rule from atoms P to arbitrary formulas A) are both admissible in the logic.

Theorem 1 (Admissibility of cut). *$\Sigma; \Gamma; \Delta_1 \xrightarrow{\nu} A \star [u_1, u_2]$ and $\Sigma; \Gamma; \Delta_2, A \star [u_1, u_2] \xrightarrow{\nu} B \star [u_1', u_2']$ imply $\Sigma; \Gamma; \Delta_1, \Delta_2 \xrightarrow{\nu} B \star [u_1', u_2']$.*

Proof. By nested induction, first on the structure of the formula A and then on the heights of the two given derivations, as in prior work [15,18].

Theorem 2 (Identity). *$\Sigma; \Gamma; A \star [u_1, u_2] \xrightarrow{\nu} A \star [u_1, u_2]$ for every formula A.*

Proof. By induction on A.

4 The File System LPCFS

Like its predecessor PCFS, our file system LPCFS is implemented for the Linux operating system. Technically, both file systems are *virtual*, since they only add a layer of authorization checks to an existing file system, which is used for all disk I/O. The existing file system in all experiments reported in this paper is ext3. Both PCFS and LPCFS are implemented using the Fuse kernel module [1].

The general workflow in both file systems is the following. Users create policies, which are given to others in the form of certificates (in LPCFS, linear certificates are stored in a central database which can be read by all users). The certificates are used as assumptions to create proofs of authorization in a logic (BL for PCFS and BL^{L} for LPCFS). The proofs are verified by a trusted verifier (an independent program), and exchanged for signed capabilities called *procaps*, which are stored in an indexed store on the disk. During file system calls, the reference monitor looks up this store for appropriate procaps and checks them to authorize access and, in LPCFS, marks linear certificates as consumed. We explain each of these steps in more detail below but, briefly, policy enforcement in both PCFS and LPCFS follows the path:

$$\text{Policy} \rightarrow \text{Proof} \rightarrow \text{Procap} \rightarrow \text{File access}$$

Policy Creation. A policy is a set of logical formulas governing access rights to files. The policy consists of certificates, which contain formulas of BL^{L} signed with creators' (owners') private keys. Certificates may be persistent or linear (use-once).

A persistent certificate is stored in a file and given to others at the owner's discretion. Persistent certificates are created using the PCFS tool `pcfs-cert` that checks their syntax. There is no restriction on copying persistent certificates. New to LPCFS are linear certificates that are stored in a central SQLite database that is accessible to both users and the reference monitor. LPCFS provides a new tool `pcfs-parse-insert` to manage this database. The tool allows anyone to insert a well-formed, signed linear certificate into the database, and anyone to read certificates in the database, but only allows the reference monitor to mark a linear certificate consumed. To ensure the latter, the database file is accessible only to the superuser, and both the tool `pcfs-parse-insert` and the reference monitor run as superuser.

Revoked certificates are stored in a separate table in the same database that stores the linear certificates. This table can be manipulated using the LPCFS command-line tool `pcfs-view-remove` that allows listing of revoked certificates, and also allows the owner of a certificate to create an entry for revoking it.

Proof Construction. To authorize herself to access a file, a user must first construct a formal proof which shows that she has access. As discussed in Sections 2 and 3, this proof uses certificates as assumptions (the contexts Γ and Δ). Although users are free to construct proofs by any means they choose including heuristics and hard-coding, PCFS provides a tool called `pcfs-search`

that uses backchaining to construct proofs automatically. In LPCFS, we have modified this tool to make it linearity-aware, i.e., it correctly ensures that linear certificates are used exactly once in the proof. This raises new challenges; for instance, when applying the $\otimes$R rule (Section 3), we need to choose a split for the linear assumptions from an exponential number of choices. We avoid this problem by using an approach to backchaining proof construction due to Cervesato et al. [10], which keeps track of unused resources in a branch and avoids this exponential choice during proof search.

Proof Verification. Once the user has constructed a proof M, this proof, together with the certificates used to construct it, is given to a proof verifier, invoked using another command line program, `pcfs-verify`. The verifier is a simple piece of code and must be trusted. In PCFS, the verifier checks that the logical structure of the proof M is correct, and that the digital signatures of all certificates used in the proof are correct. LPCFS adds two new checks: (1) that none of the certificates used in the proof have been revoked by their authors, and (2) that all linear certificates exist within the database and are unused. If all these checks succeed, the verifier gives back the user a *procap*, which is a capability that mentions the user, file, and permission (read, write, etc.) authorized. The procap also contains conditions related to time and system state under which the proof is valid (we have not discussed system state in this paper, but LPCFS inherits it from PCFS). In LPCFS, lists of unique identifiers of all persistent certificates (P) and linear certificates (L) on which the proof depends are also added to the procap. Finally, the procap is signed using a shared symmetric key that is known only to the verifier and the file system reference monitor. The signature is necessary to prevent users from forging capabilities. After receiving a procap, the user calls another command line tool which puts the procap in an indexed store on the disk.

File System Call and Access. LPCFS, like PCFS, respects the standard POSIX interface for file systems. During a file system call (read, write, open, etc.) the PCFS/LPCFS file system looks up the indexed procap store to authorize the operation. The number of procaps needed varies from 1 to 3 depending on the operation; these are unchanged from the prior work on PCFS. If all relevant procaps are found, they are checked. In PCFS, this check covers the procap's time and system state conditions; in LPCFS, the procap's certificate lists (P and L) are also checked as follows:

- An *exclusive transaction* with the database containing the linear certificates and the revoked certificates is started
- The revoked certificates table is queried to ensure it contains no elements in the persistent list, P
- The linear certificates table is queried to ensure it contains all elements of the linear list, L, and that none of them have been marked as "used" previously
- All the elements of L are marked as "used" in the database

- The transaction is committed
- File access is granted

The order of these operations is imperative. All other conditions in a procap must be checked before checking the certificate lists in it, as we do not want to unnecessarily mark linear certificates as used when access may not be granted. Also for that reason, we must check both the linear and the revoked certificates before consuming the linear certificates. It is also necessary that the linear certificates be marked as used *prior to giving file access*; if not, we risk a system failure preventing us from marking the certificates used despite an access having been made. Of course, this allows the possibility that a system failure after the certificates are marked but prior to access incorrectly causes the certificates to be marked. However, we maintain access logs and time of use within the database, so certificates marked consumed this way can be unmarked by an administrator during system recovery.

Summary of the Implementation. Our implementation of the LPCFS front end tools (proof search, proof verification, and certificate management) comprises approximately 9,000 lines of SML code. The original PCFS implementation of these tools was nearly 7,000 lines of code; our modifications and additions are spread throughout that code. Because the front end tools are used less frequently than the reference monitor, their efficiency is also less of a concern. The bottleneck for performance is the LPCFS reference monitor, which comprises approximately 11,000 lines of C++ code (the PCFS reference monitor was 10,000 lines long). We evaluate performance of the reference monitor in Section 5.

5 Experimental Results

In this section, we present the results of several experiments that measure the efficiency of the reference monitor (back end) of LPCFS. We are specifically interested in the costs of checking and consuming certificates when performing basic operations such as stat-ing a file (to which we address our microbenchmarks), and during a typical build cycle (to which we address our macrobenchmarks). All experiments reported here were performed on a 2.8 GHz 8-core machine with 3.8 GB RAM with a 500GB 7200 RPM hard drive running Linux kernel version 2.6.35-27-generic. We used the GNU C++ compiler (g++) to compile the reference monitor.

5.1 Macrobenchmarks

We performed two typical build-cycle benchmarks: (1) Untar-ing, compiling, and deleting the source code of the fuse kernel module 5 times (Fuse $\times$ 5), and (2) Untar-ing, configuring, compiling, and deleting the Poco C++ Base Library (Poco/Base). In running LPCFS, we gave read, write, execute permissions on the parent directory in which the tests were being run, first dependent upon

no certificates, and then with each permission dependent upon one persistent certificate (which, of course, had not been revoked). No linear certificates were used in these benchmarks: we would not expect linear rights to be used in a build environment and so their effect is not a concern. The results of the benchmarks are shown below. All times are measured in seconds. Fuse/Null is a virtual file system with an architecture similar to that of LPCFS, except that it makes no access checks. This file system is our baseline for comparison.

Benchmark	LPCFS(0 certs)	LPCFS (1 cert/perm)	PCFS	Fuse/Null	Ext3
Poco/Base	614	638	614	611	538
Fuse $\times$ 5	97.55	98.64	96.41	91.18	85.48

In the absence of revocation checks (column 0 certs in the table), LPCFS overhead over Fuse/Null is 0.4% for Poco, and 7% for Fuse. These are similar to those of PCFS, which is to be expected, because in these cases the LPCFS and PCFS implementations behave similarly. The additional overhead of checking one certificate revocation per access (column 1 cert/perm in the table) is less than 1% for Fuse and less than 5% for Poco, which is not much. Interestingly, this overhead does not change with the size of the revoked certificate table, which is also supported by our microbenchmarks below.

5.2 Microbenchmarks

The purpose of microbenchmarking was to assess the cost of checking for existence of certificate revocations and linear certificates (and marking the linear certificates as used) in the database. In the first microbenchmark, we measured the amount of time taken to stat a file[1], when the permission to stat the file depended on $N = \{0, 1, 2, 10, 20, \ldots, 100\}$ linear certificates. Precisely, we created 10,000 files of size 1 byte each and procaps authorizing the execute permission to each of them (execute is the only permission needed to stat a file) with N linear certificates in each procap. The average time to stat a file for different values of N is shown in both tabular and graphical form in Figure 1. Note that stat-ing a file whose execute permission depends on N linear certificates requires N updates to the database (one update to mark each of the N certificates consumed).

With 0 certificates, the time taken by LPCFS per file (2.6ms) is similar to that taken by Fuse/Null (2.4ms) and PCFS (2.6ms). However, with even one linear certificate per procap, the time for access increases to 156ms. This is unsurprising because dependence on linear certificates implies that the database must be written to consume the linear certificates, which is an expensive operation. Note, however, that a linear certificate can be used only once after it is issued, so the *total* initial overhead due to linear certificates (156ms) across all system calls cannot exceed the number of such certificates issued. In practice, we may expect that not many linear certificates will be issued, so the overall cost should be manageable. As the chart in Figure 1 shows, after this initial overhead the time increases almost linearly with the number of linear certificates in each procap,

[1] The stat file system call reads a file's metadata, e.g., its length and owner.

Certs	0	1	2	10	20	30	40	50	60	70	80	90	100	PCFS	Fuse/Null
Time (ms)	2.6	156	155	157	190	205	228	231	235	243	250	261	279	2.6	2.4

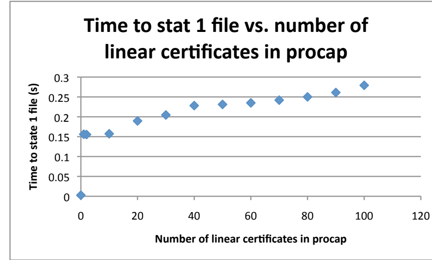

Fig. 1. Time to stat one file varying the number of linear certificates per procap

and is approximately 1ms per certificate. Practical authorizations are unlikely to use more than one or two linear certificates each, so the incremental overhead (1ms/certificate) is unlikely to add up to a significant number for any access.

Our second microbenchmark measures the cost of checking for certificate revocations. This experiment is similar to the previous one, except that instead of linear certificates, we use persistent certificates in procaps, for which only revocations are checked. Again we varied the number of certificates in each procap in the set $N = \{0, 1, 2, 10, 20, \ldots, 100\}$. However, in addition, we also varied the number of certificates in the revocation table in the set $\{1000, 2000, \ldots, 10000\}$ to observe the impact, if any, of changing the size of this table. Our observations are shown in Figure 2. First, as is evident from the data, there is no sudden increase in access time when moving from 0 to 1 revocation checks per procap, as was the case for linear certificates. This is because a revocation check does not require writing the database and is, therefore, relatively inexpensive. Second, there is a uniform growth in the overhead with increase in the number of revocation checks per procap. The slope of this growth is approximately 0.02ms per certificate. Finally, the effect of changing the number of revoked certificates in the database is negligible. This is because the reference monitor reads the revocation table in accordance with the table's index.

5.3 Summary of Experimental Results

Our experiments show that increasing the size of the database does not significantly affect the cost of checking certificates at the time of file access. Increasing the number of certificates (linear or persistent) upon which permissions rely has a roughly linear correlation with the time required to gain a permission to a file. Linear certificates incur a significant, but constant, overhead because marking them consumed requires writing the database. Our macrobenchmarks show that there is not a significant overhead in maintaining a revocation list within a database and checking certificates against this list for a typical build cycle.

Certs\DB Load	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
0	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6
1	3.2	3.1	3.1	3.1	3.1	3.1	3.1	3.1	3.1	3.1
2	3.3	3.3	3.2	3.2	3.2	3.2	3.2	3.3	3.2	3.2
10	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.4	3.5
20	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6	3.6
30	3.7	3.7	3.7	3.8	3.8	3.8	3.8	3.8	3.8	3.8
40	3.9	3.9	3.9	4.0	4.0	4.0	4.0	4.0	4.0	4.0
50	3.9	4 0	4.0	4.0	4.0	4.0	4.0	4.0	4.0	4.0
60	4.2	4.2	4.1	4.2	4.2	4.2	4.2	4.2	4.3	4.3
70	4.2	4.2	4.2	4.2	4.1	4.2	4.2	4.3	4.2	4.4
80	4.4	4.5	4.5	4.5	4.5	4.5	4.5	4.5	4.5	4.6
90	4.8	4.8	4.8	4.8	4.8	4.8	4.8	4.7	4.7	4.7
100	4.8	4.8	4.8	4.8	4.9	5.2	5.6	5.4	5.5	5.5

Fig. 2. Time to stat one file varying the number of required persistent certificates per procap (rows) and the size of the revocation list (columns). All times are reported in ms.

6 Conclusion

LPCFS extends previous work on the file system PCFS to support revocable and linear certificates within a proof-carrying authorization framework. Both extensions are implemented using a centralized database which maintains a list of revoked certificates and a table of linear certificates. Our experiments suggest that the cost of making additional checks to support these features is manageable.

An interesting future direction for this work is to consider linear and revocable certificates in a distributed setting: rather than requiring a centralized database, the certificates and revocation list could be kept at multiple nodes.

Further, we would like to study policy authoring and analysis. Owing to the complexity of the logic, policies may have unintended consequences if care is not taken in constructing them. It would be useful to develop tools for exploring possible consequences of a policy, or to aid in the proof of certain metatheorems about a particular policy. For example, for the policy in Section 2, it might be useful to prove that no statement made by `Alice` could affect the permissions available to `Bob`. Notions such as this are useful guidelines, both for individuals authoring the policy and for developers constructing policy verification tools.

Acknowledgments Jamie Morgenstern and Frank Pfenning were partially supported by NSF grant number 0716469. Jamie Morgenstern was also supported by a Graduate Research Fellowship from the National Science Foundation. Deepak Garg was supported by the U.S. ARO contract "Perpetually Available and Secure Information Systems" (DAAD19-02-1-0389) to Carnegie Mellon's CyLab and the AFOSR MURI "Collaborative Policies and Assured Information Sharing".

References

1. FUSE: Filesystem in Userspace, available from http://fuse.sourceforge.net/
2. Abadi, M.: Logic in access control. In: 18th Annual Symposium on Logic in Computer Science (LICS'03). pp. 228–233 (Jun 2003)
3. Abadi, M.: Logic in access control (tutorial notes). In: 9th International School on Foundations of Security Analysis and Design (FOSAD). pp. 145–165 (2009)
4. Appel, A.W., Felten, E.W.: Proof-carrying authentication. In: 6th ACM Conference on Computer and Communications Security (CCS). pp. 52–62 (1999)
5. Barth, A., Mitchell, J.C.: Managing digital rights using linear logic. In: 21st Annual IEEE Symposium on Logic in Computer Science (LICS). pp. 127–136 (2006)
6. Bauer, L.: Access Control for the Web via Proof-Carrying Authorization. Ph.D. thesis, Princeton University (2003)
7. Bauer, L., Garriss, S., McCune, J.M., Reiter, M.K., Rouse, J., Rutenbar, P.: Device-enabled authorization in the Grey system. In: 8th Information Security Conference (ISC). pp. 431–445 (2005)
8. Bowers, K.D., Bauer, L., Garg, D., Pfenning, F., Reiter, M.K.: Consumable credentials in logic-based access-control systems. In: Electronic Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS'07) (2007)
9. Cederquist, J.G., Corin, R., Dekker, M.A.C., Etalle, S., den Hartog, J.I., Lenzini, G.: Audit-based compliance control. International Journal of Information Security 6(2), 133–151 (2007)
10. Cervesato, I., Hodas, J.S., Pfenning, F.: Efficient resource management for linear logic proof search. Theoretical Computer Science 232, 133–163 (2000)
11. Chang, B.Y.E., Chaudhuri, K., Pfenning, F.: A judgmental analysis of linear logic. Tech. Rep. CMU-CS-03-131R, Carnegie Mellon University (2003)
12. Garg, D.: Proof Theory for Authorization Logic and Its Application to a Practical File System. Ph.D. thesis, Carnegie Mellon University (2009), available as Technical Report CMU-CS-09-168
13. Garg, D., Bauer, L., Bowers, K., Pfenning, F., Reiter, M.: A linear logic of affirmation and knowledge. In: 11th European Symposium on Research in Computer Security (ESORICS '06). pp. 297–312 (2006)
14. Garg, D., Pfenning, F.: Non-interference in constructive authorization logic. In: 19th Computer Security Foundations Workshop (CSFW). pp. 283–293 (2006)
15. Garg, D., Pfenning, F.: A proof-carrying file system. In: 31st IEEE Symposium on Security and Privacy (Oakland). pp. 349–364 (2010)
16. Girard, J.Y.: Linear logic. Theoretical Computer Science 50, 1–102 (1987)
17. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems 10(4), 265–310 (1992)
18. Pfenning, F.: Structural cut elimination I. Intuitionistic and classical logic. Information and Computation 157(1/2), 84–141 (2000)
19. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Mathematical Structures in Computer Science 11, 511–540 (2001)
20. Schneider, F.B., Walsh, K., Sirer, E.G.: Nexus Authorization Logic (NAL): Design rationale and applications. Tech. rep., Cornell University (2009), http://ecommons.library.cornell.edu/handle/1813/13679