A Proof-Carrying File System

Deepak Garg
CyLab
Carnegie Mellon University
Pittsburgh PA, USA
dg@cs.cmu.edu

Frank Pfenning
Computer Science Department
Carnegie Mellon University
Pittsburgh PA, USA
fp@cs.cmu.edu

***Abstract*—We present the design and implementation of PCFS, a file system that adapts proof-carrying authorization to provide direct, rigorous, and efficient enforcement of dynamic access policies. The keystones of PCFS are a new authorization logic BL that supports policies whose consequences may change with both time and system state, and a rigorous enforcement mechanism that combines proof verification with conditional capabilities. We prove that our enforcement using capabilities is correct, and evaluate our design through performance measurements and a case study.**

***Keywords*-Access control, logic, proof-carrying authorization, file system**

I. Introduction

Proof-carrying authorization (PCA) [7, 9, 10] has emerged as a promising, open-ended architecture for rigorous enforcement of access control policies. In PCA, policy rules and credentials are represented as logical formulas at a high level of abstraction and published in certificates that are signed digitally by policy creators; a trusted reference monitor allows an operation to be performed on a protected resource only if the principal requesting the operation produces enough certificates to authorize the access, as well as a *formal logical proof* which explains why the certificates entail access. Although PCA combines the rigor and flexibility of formal logic for expressing and interpreting access policies, and of digital signatures for maintaining their integrity, the reference monitor must check a logical proof and associated certificates at each access, which may together limit the throughput of resource access. To date, PCA has been practically deployed for access control on the web [9] and in physical devices [10]. In either case, latency of access checks is not a significant concern since it is overshadowed by the time taken for network communication and movement of physical parts, respectively.

In this paper we present the theory, design, and implementation of PCA in a local file system where latency of access checks is a significant concern. Given an effective disk cache, a local file system is expected to perform several thousand operations a second, which leaves only a few microseconds for each access check. In particular, a file system that verifies proofs and certificates at each access results in visible delays, a hypothesis we confirmed through an implementation that preceded the work presented in this paper. To attain high throughput during file access, our present file system PCFS (proof-carrying file system) off-lines proof and certificate checking to a trusted verifier program, which is invoked in advance of access and issues signed capabilities after verifying proofs and associated certificates. The capabilities, called procaps (proven capabilities), can then be used to authorize access to files and perform disk I/O. Verification of capabilities is approximately two orders of magnitude faster than verification of proofs and certificates and, as a result, PCFS attains high throughput during file access even in our prototype implementation.

At a high-level of abstraction, procaps are similar to entries of a cache that a reference monitor may maintain to record accesses it has authorized in the past. However, procaps are more general than cache entries since they scale easily to decentralized settings where the trusted verifier and reference monitor are running on different nodes of a network. Further, as opposed to a reference monitor maintained cache, procaps are closer in spirit to proof-carrying authorization where the principal seeking access is responsible for maintaining and providing evidence to authorize access. Another merit of using capabilities as opposed to caches is that both the design and implementation of the access control system factor into two parts that interact via capabilities only: (a) The *front end* that deals with policies, proofs, and verification of proofs and certificates, and (b) The *back end* that uses capabilities to authorize access and perform I/O. Indeed, the PCFS back end is independent of the logic used in the front end, and it can be used with any policy infrastructure that produces compatible capabilities. Similarly, the PCFS front end may be used for managing policies and proofs for controlling access to any class of resources, not necessarily a file system.

Although the design and implementation of an access control system that separates policy decision from policy enforcement via capabilities may seem technically straightforward (indeed such designs have been used in many prior systems, e.g., [6, 25, 33, 37], although none of these systems use logic for representing policies), a technically challenging and interesting aspect of our work that has not been addressed previously is a rigorous treatment of

dynamic authorizations, i.e. authorizations that are valid only at specific points of time or in particular system states, both of which may change between the generation of a capability and its use. To address this problem satisfactorily, we consider a new logic BL that can represent both time and state dependent policies, describe a proof verification procedure that extracts time and state conditions on which proofs rely (these conditions are then written to generated capabilities), require that the PCFS back end verify these conditions, and prove rigorously that this two-part checking of proofs where proof structure and certificates are verified off-line and conditions are checked during access ensures that a full proof authorizing access exists at the *time of access*. Thus file access in PCFS has the same formal guarantees as PCA, but with much smaller access check overheads.

As part of the infrastructure supporting the core PCFS architecture, we have built an automatic proof search tool for BL that is based on logic programming. As in PCA, this tool is not a part of the trusted computing base, but users may choose to use it for constructing proofs. To test the expressiveness of the framework, we have completed a realistic case study that represents in BL policies for controlling access to classified information in the U.S., and explains their enforcement through PCFS.

The current implementation of the PCFS back end maintains compatibility with existing programs by requiring that capabilities be put in an indexed store on disk in advance of program invocation. Capabilities are fetched from this store by the file system back end when needed, so the existing file system API remains unmodified. The PCFS architecture has been constructed with generality in mind, and we expect that it can be used in other centralized and decentralized settings (besides file systems) without significant changes to either its design or its implementation.

Contributions: Our work contributes to both the theory and practice of access control. At a theoretical level, the primary contribution of our work is the logic BL that can represent dynamic authorization policies, a detailed development of its proof theory (of which we present only a part in this paper), a verification procedure that generates capabilities conditional on time and system state from BL proofs, and a rigorous proof that enforcement of policies with off-line proof verification and capabilities has the same formal guarantees as PCA. On the practical side, we implement a prototype file system PCFS that includes a BL front end with an automatic proof search tool and a proof verification tool that generates procaps, as well as a back end that uses capabilities to authorize access. We show through measurements that capability based authorization checks are fast in practice even when capabilities are conditional on time and state, and verify expressiveness of the framework through a case study. The back end of PCFS also makes a minor contribution by adding two new permissions to the three permissions mandated by POSIX (read, write, and execute). This enables enforcement of both discretionary and mandatory access control policies, as becomes evident in our case study (it is known that mandatory access control policies are difficult to enforce with POSIX permissions).

Non-goals: Our focus in designing and building PCFS is on logical foundations and efficiency of access control. Issues of end-user friendliness, although important in practice, are beyond the scope of this paper. In particular, we do not consider methods for authoring policies, finding certificates relevant for proof construction (prior work on this subject may also apply here [12, 16]), and role-based access control (RBAC) or its administration. The implementation of PCFS is a prototype meant to evaluate access control checks; it does not focus on other aspects of security, e.g., encryption of data and communication protocols between the kernel and the file system. Further, we do not evaluate file-use patterns in common programs. In general, our architecture will perform better for programs whose working files can either be predicted prior to execution or are newly created during execution. In the former case, procaps needed for access can be generated in advance and for the latter case PCFS automatically creates procaps that provide a program access to newly created files for a fixed period of time, without requiring administrators to create new policy rules. Finally, we do not consider the issue of security of signing keys. Instead, we assume that administrators keep their signing keys secure at all times.

Organization: The rest of this paper is organized as follows. In Section II we introduce the architecture of PCFS and its various components. Section III covers the logic BL used to represent policies in PCFS and its proof theory. Section IV briefly describes a case study on access control for classified information in the U.S. Section V covers the front end of PCFS including our implementation of automatic proof search and proof verification. It also shows formally that procap-based enforcement of policies in PCFS is equivalent to an enforcement with proof-carrying authorization. Section VI discusses the back end of PCFS that uses procaps to authorize permissions. Section VII evaluates performance of the PCFS back end. Section VIII discusses some related work, and Section IX concludes the paper.

Proofs of theorems and several aspects of BL's metatheory have been omitted from this paper due to lack of space. These are present in the first author's thesis [20]. Details of the case study are in a separate technical report [23]. Source code of the entire implementation of PCFS is available from the first author's homepage.

II. Overview of PCFS

PCFS is implemented as a local file system for the Linux operating system. Technically, PCFS is a *virtual file system* that relies on the Fuse kernel module [2]. An underlying file

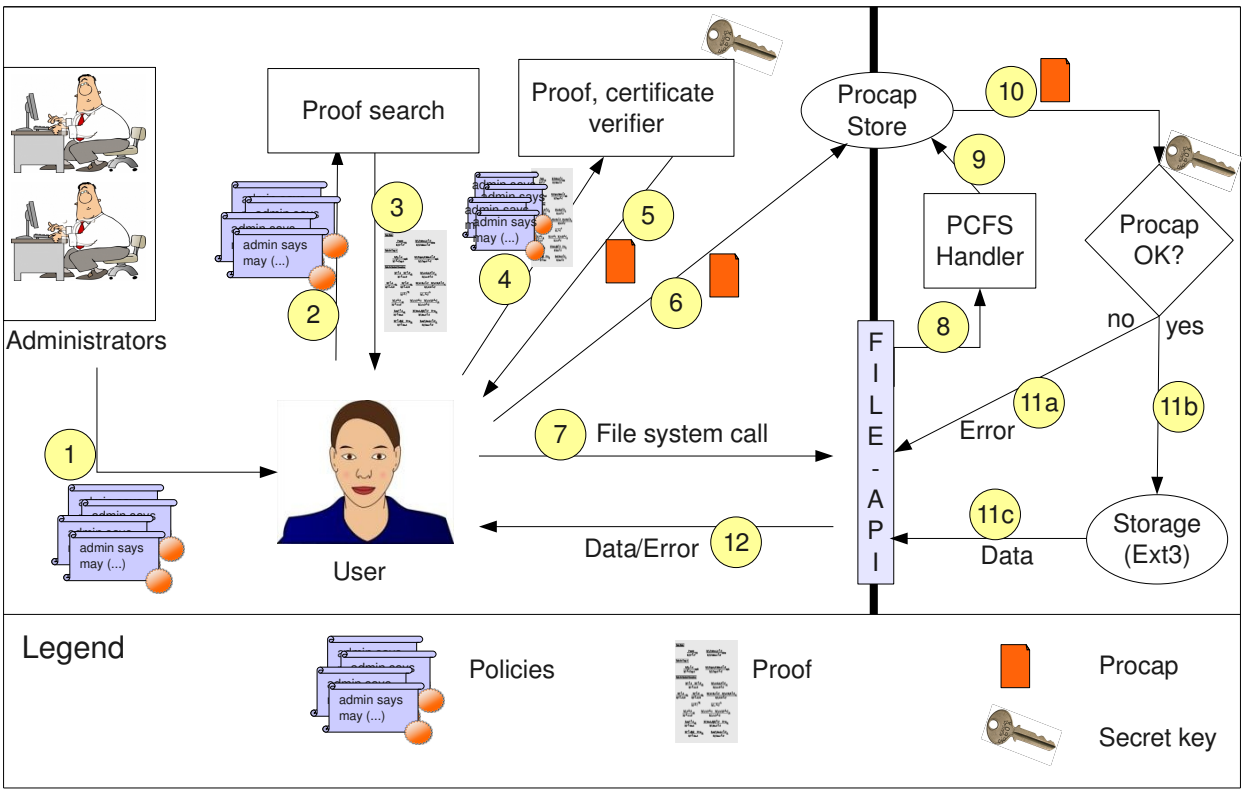

Figure 1. PCFS Workflow

system (ext3 in all experiments reported in this paper) is used for disk I/O after making access control checks. PCFS is mounted using the command:

```
$> sudo pcfs-main /path/to/src \
   /path/to/mountpoint
```

Here `/path/to/src` is an existing directory in an ext3 system, and `/path/to/mountpoint` is an empty directory. After the execution of this command, any file system call on a path inside `/path/to/mountpoint/` (e.g.,`/path/to/mountpoint/foo/bar`) results in a corresponding operation on the path inside `/path/to/src/` (e.g.,`/path/to/src/foo/bar`), but is subject to rigorous access checks. PCFS expects some configuration information in special files in `/path/to/src/`, which is described in Section VI.

Figure 1 shows the PCFS workflow. Numbers are used to label steps in order of occurrence. Steps 1–6 create and store procaps which show that a user is allowed certain permissions in the file system. These steps are performed in advance of file access, and happen infrequently (usually before a user accesses a file for the first time). Once procaps are stored, they can be used repeatedly to perform file operations (steps 7–12). The solid black vertical line in the figure separates steps that happen in user space, i.e. before and after a file system call (left side of the line) from those that happen during a file system call (right side of the line). Briefly, policy enforcement in PCFS follows the path:

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

Policy creation (Step 1): We define a *policy* as a set of rules and facts which determines access rights. An access right is a triple $\langle k, f, \eta \rangle$, which means that user k (Alice, Bob, etc.) has permission η (read, write, etc.) on file or directory f. We allow different rules and facts in a policy to be created by different individuals called *administrators* (this is necessary to faithfully represent separation of duty in many organizations). We require that each administrator write her portion of facts and rules as *logical formulas* in a text file and digitally sign the file with her private key. This signed file is called a *certificate.* In a concrete sense, therefore, a policy is a collection of certificates signed by different administrators. Abstractly, a policy is a collection of logical formulas which are contained in the certificates. We often denote this collection of logical formulas with the symbol Γ. PCFS provides its own logic for writing policies. This logic, BL, is described in Section III.

PCFS provides a command line tool, `pcfs-cert`, to help administrators check formulas for adherence to logical syntax, to digitally sign them, and to convert them to a custom certificate format. (We could have used a standard certificate format like X.509 [27], but found it easier to create our own format.) We do not assume a centralized store for certificates. Instead they are distributed to users to whom they grant permissions. Certificates may be issued and revoked over time (revocation is discussed in Section V), so the policy may change over time.

Proof generation (Steps 2–3): Once certificates have been created by administrators and given to users, the latter use them to show that they are allowed certain permissions in the file system. The basic tenet of PCFS (adapted from PCA) is that a user k is allowed permission η on resource f at time u, if and only if the user can provide a *formal logical proof* M which shows that the policies in effect (Γ) entail a fixed formula $\texttt{auth}(k, f, \eta, u)$, or in formal notation, $M :: \Gamma \vdash \texttt{auth}(k, f, \eta, u)$. The formula $\texttt{auth}(k, f, \eta, u)$ is defined in Section III.

To facilitate proof construction, PCFS provides an automatic theorem prover for BL through the command line tool `pcfs-search`, which we describe briefly in Section V-A. However, the proof search tool is *not* a trusted component of PCFS and users may create their own proof search tool or use heuristics to construct proofs.

Proof verification (Steps 4–5): 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` (Step 4 in Figure 1). The verifier is a simple piece of code and must be trusted. The verifier checks that the logical structure of the proof M is correct, and checks the digital signatures of all certificates used in the proof. If both checks succeed, the verifier gives back the user a procap, which is a capability that mentions the right $\langle k, f, \eta \rangle$ that the proof grants (Step 5). The procap also contains conditions related to time and system state on which the proof depends (see Section V-B for details) and 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.

Procap injection (Step 6): After receiving a procap, the user calls another command line tool which puts the procap in a central store marked "Procap Store" in Figure 1. This store is in a designated part of the PCFS file system, and is accessible to both users as well as the file system back end. The back end looks up this store to find relevant procaps when file system calls are made. We describe this store's organization in Section VI.

File system call (Step 7): A call to the PCFS file system is made through the usual POSIX file system API during the execution of a user program or through a shell command like `ls`, `cp`, `rm`, etc. The PCFS file access back end respects the standard POSIX interface, so user programs and shell commands don't need to change to work on PCFS. However, before a program is started or a shell command is executed, the user must ensure that enough procaps have been injected to allow the program to complete all its file operations. So steps 2–6 may have to be repeated many times by the user (this can be automated using simple scripts). Alternatively, the program must be augmented to possibly create, and certainly inject, procaps on the fly. For files that are created during the execution of a program, the back end automatically creates and stores some default procaps that allow temporary access to the user creating them. These are discussed in Section VI.

Procap look up and checking (Steps 8–10): When a system call is made on a PCFS file system, it is redirected by the Linux kernel to a process server which we have written (Step 8 in Figure 1). Depending on the specific operation requested, this server looks up one or more procaps in the procap store (Steps 9 and 10). The exact procaps needed for each operation vary, and are listed in Section VI. If all relevant procaps are found, they are checked.

Error (Steps 11a, 12): If any procap needed for performing the requested file operation is missing, or fails to check, an error code is returned to the user program.

File operation (Steps 11b, 11c, 12): If all relevant procaps needed to perform the requested file operation are found, and successfully check, then the underlying file system is used to perform the requested file operation (Step 11b). The result of the operation is returned to the user (Steps 11c and 12).

A. Implementation

Our implementation factors into two parts: (a) The **front end**, which comprises the command line tools for creating certificates, constructing proofs, checking proofs to create procaps, and injecting procaps into the central store (Steps 1–6 in Figure 1), and (b) the **back end** which handles the calls from the Fuse kernel module, looks up procaps in the store, checks them, and then makes calls on the underlying file system to perform disk operations (Steps 8–11c in Figure 1). The two parts interact via procaps which carry information from logical proofs into the file system's back end.

The front end, with the exception of the procap injection tool, has strong foundations in logic, and the technical challenge in its design has been the development theoretical principles for BL that can be practically implemented. Our implementation of the front end tools is written in Standard ML, and comprises nearly 7,000 lines of code. OpenSSL [3] is used for all cryptographic operations. Because the front end tools are used less frequently than the back end tools, their efficiency is also less of a concern.

The back end is the bottleneck for performance and needs to be extremely efficient. It is implemented in C++ using approximately 10,000 lines of code, most of which

is devoted to procap parsing and checking. We evaluate performance of the back end in Section VII.

III. BL: The Authorization Logic

PCFS provides a new logic for writing policies, which we call BL, and describe in this section.[1] BL is an extension of first-order intuitionistic logic with two modalities that have been studied in prior work [5, 18, 30]: k says s, which means that principal k states or believes formula s, and s @ $[u_1, u_2]$ which means that s holds from time u_1 to time u_2. The former is used to distinguish in the logic parts of the policy made by different individuals whereas the latter is needed to accurately represent time-dependent rules. The logical interpretation of k says s in BL is different from that in any existing work. This new interpretation is designed to facilitate faster proof search and to increase expressiveness (Sections V-A and III-C). In addition to these modalities, BL includes support for checking constraints, which are relations between terms decided using external decision procedures not formalized in the logic (e.g., the usual order $\leq$ on integers). BL also supports predicates that capture the state of the file system. Formulas in BL are denoted using the letters s and r. The syntax of BL is summarized below.

Sorts	σ	::=	principal | time | file | perm | ...
Terms	t	::=	$a \mid x \mid h(t_1, \ldots, t_n)$
I-Predicates	I		(Interpreted Predicates)
U-Predicates	P		(Uninterpreted Predicates)
I-Atoms	i	::=	$I(t_1, \ldots, t_n)$
U-Atoms	p, q	::=	$P(t_1, \ldots, t_n)$
Constraints	c	::=	$u_1 \leq u_2 \mid k_1 \succeq k_2 \mid \ldots$
Formulas	r, s	::=	$p \mid i \mid c \mid r \wedge s \mid r \vee s \mid r \supset s \mid$ $\top \mid \bot \mid \forall x{:}\sigma.s \mid \exists x{:}\sigma.s \mid$ k says $s \mid s$ @ $[u_1, u_2]$

As in first-order logic, subjects of predicates are called *terms*. They represent principals, files, time points, etc. Abstractly, terms can be either ground constants a, bound variables x, or applications of uninterpreted function symbols h to ground terms. Terms in BL are classified into sorts σ (sometimes called types). We stipulate at least four sorts: principal, whose elements are denoted by the letter k, time whose elements are denoted by the letter u, file whose elements are denoted by the letter f, and perm (for permission) whose elements are denoted by the letter η. Elements of time are called time points, and it is assumed that ground time points are integers (we could also have used rationals or real numbers instead). In the external syntax of the logic, we allow time points to be actual clock times written to second level accuracy as yyyy:mm:dd:hh:mm:ss, but internally time points are represented as integers that measure seconds elapsed from a fixed clock time.

[1] BL stands for "Binder Logic", as a tribute to the trust management framework Binder [17] from which the logic draws inspiration.

Predicates in BL are divided into three categories: (a) Uninterpreted predicates, denoted P, which are established using logical rules and a priori using signed certificates, (b) Constraints, which are interpreted through trusted decision procedures, and (c) Interpreted predicates, denoted I, which capture properties of the environment. By environment here we mean the state of the file system, including, but not limited to, meta data contained in the files such as extended attributes and file ownership information. We assume that the environment is volatile, i.e. it may change unpredictably. We denote an abstract logical representation of the environment by the letter E and write $E \models i$ to mean that the interpreted atomic formula i holds in the environment E.

In the implementation we require trusted decision procedures to decide the truth of both interpreted predicates and constraints. The difference between the two is that the truth of interpreted predicates may depend on state, while that of constraints may not. We stipulate at least two types of constraints: $u_1 \leq u_2$ that captures the usual total order on time points, and a pre-order $k_1 \succeq k_2$, which means that principal k_1 is stronger than principal k_2. If $k_1 \succeq k_2$, then BL's inference rules force $(k_1$ says $s) \supset (k_2$ says $s)$ for every formula s. In this sense, $k_1 \succeq k_2$ is similar to the "speaks for" relation of Abadi et al. [5, 30]. The difference is that the BL relation $k_1 \succeq k_2$ is a constraint, so it does not have to be defined entirely through logical formulas, and its verification may rely on external decision procedures. We assume that there is a strongest principal ℓ, i.e. $\ell \succeq k$ holds for every k. In particular $(\ell$ says $s) \supset (k$ says $s)$ for every k and s. For this reason ℓ is called the "local authority", a principal whom everyone believes. (We borrow this term from the language SecPAL [11].) The notation $\models c$ means that constraint c holds.

Time-invariance of state: An important design decision in BL is that interpreted predicates occurring in a proof are implicitly defined (and verified) relative to the environment E prevailing at the time that the procap derived from the proof is used for access. For the proof system of BL, this design choice has the consequence that the truth of interpreted atoms does not change with time, so i @ $[u_1, u_2]$ and i @ $[u_1', u_2']$ are equivalent for arbitrary intervals $[u_1, u_2]$ and $[u_1', u_2']$ (both mean that i holds in the environment prevalent at the time of access). Although counter-intuitive, because, in practice, the truth of interpreted predicates does change as system state changes, this design choice simplifies the proof verifier and the design of procaps (Section V-B). Further, limiting system state to one point in time does not reduce expressiveness: if some access control policy were to rely on a predicate over system state having been true in the past, this can still be represented in BL by requiring that there be explicit evidence – either an element of system state or a certificate – *still valid at the time of access* that witnesses this fact. Clearly, requiring such persistent

evidence is no harder than requiring the reference monitor to maintain a record of the entire history, and is in fact, a better design choice since it requires the policy to make explicit what evidence from the past is necessary to verify proofs. Our case study (Section IV) shows that this design choice does not preclude representation of realistic policies in BL.

A. Proof System

Next, we present a proof system for BL in the natural deduction style of Gentzen [24]. We have also designed a sequent calculus for BL but omit it here. Our proof theory is based on the judgmental method [34], where a syntactic category of judgments (distinct from formulas) is the subject of proofs and deductions. Using the judgmental method simplifies meta-theory of the logic. Our technical presentation closely follows prior work for a related logic by DeYoung and the authors [18]. As in that work, we introduce two judgments: $s \circ [u_1, u_2]$ meaning that formula s is provably true in the interval $[u_1, u_2]$, and k claims $s \circ [u_1, u_2]$ meaning that principal k states that s holds from u_1 to u_2. The symbol $\circ$ is read "during". $s \circ [u_1, u_2]$ is logically equivalent to the formula $s \ @ \ [u_1, u_2]$, whereas k claims $s \circ [u_1, u_2]$ is logically equivalent to the formula $(k \text{ says } s) \ @ \ [u_1, u_2]$.

Judgments	J	::=	$s \circ [u_1, u_2] \mid k \text{ claims } s \circ [u_1, u_2]$
Views	α	::=	k, u_b, u_e
Hypotheses	Γ	::=	$\pi_1 : J_1 \ldots \pi_n : J_n \qquad (n \geq 0)$
Sequent			$E; \Gamma \xrightarrow{\alpha} s \circ [u_1, u_2]$

Hypothetical judgments (which are established through proofs) have the form $E; \Gamma \xrightarrow{\alpha} s \circ [u_1, u_2]$. E is an abstract logical representation of the environment and Γ is the set of assumed judgments (hypotheses or policy). $\pi_1, \ldots, \pi_n$ are distinct names for assumptions in Γ. A novel feature of the proof system is the triple $\alpha = k, u_b, u_e$ on the entailment arrow, which we call the *view* of the sequent.[2] The view represents the principal and interval of time relative to which reasoning is being performed. It affects provability in the following manner: while reasoning in view k, u_b, u_e, an assumption of the form k' claims $s \circ [u_1', u_2']$ entails $s \circ [u_1', u_2']$ if $k' \succeq k$, $u_1' \leq u_b$, and $u_e \leq u_2'$. This entailment does not hold in general.

A proof is represented using a compact representation called a *proof term*, denoted M. We write $M :: E; \Gamma \xrightarrow{\alpha} s \circ [u_1, u_2]$ to mean that M is a proof term that represents a proof of the hypothetical judgment that follows it. For each deduction rule in our proof system, there is a unique constructor for proof terms. An entire proof can be reconstructed from its proof term and the hypotheses.

[2]We simplify the presentation of the proof system by not explicitly listing term parameters that are allowed to occur in hypothetical judgments. Further, we do not consider hypothetical constraints here. A full set of rules without these omissions is available in the first author's thesis [20, Chapter 4].

$$\frac{\models u_1' \leq u_1 \qquad \models u_2 \leq u_2'}{\pi :: E; \Gamma, \pi : s \circ [u_1', u_2'] \xrightarrow{\alpha} s \circ [u_1, u_2]}\text{hyp}$$

$$\frac{\begin{array}{ccccc} & & \alpha = k, u_b, u_e & \models u_1' \leq u_1 & \\ \models u_2 \leq u_2' & \models u_1' \leq u_b & \models u_e \leq u_2' & & \models k' \succeq k \end{array}}{\pi :: E; \Gamma, \pi : k' \text{ claims } s \circ [u_1', u_2'] \xrightarrow{\alpha} s \circ [u_1, u_2]}\text{claims}$$

$$\frac{M :: E; \Gamma| \xrightarrow{k, u_1, u_2} s \circ [u_1, u_2]}{(\texttt{pf\_saysI}\ M) :: E; \Gamma \xrightarrow{\alpha} (k \text{ says } s) \circ [u_1, u_2]}\text{saysI}$$

$$\frac{\begin{array}{c} M_1 :: E; \Gamma \xrightarrow{\alpha} s_1 \supset s_2 \circ [u_1, u_2] \\ M_2 :: E; \Gamma \xrightarrow{\alpha} s_1 \circ [u_1', u_2'] \\ \models u_1 \leq u_1' \leq u_1'' \qquad \models u_2'' \leq u_2' \leq u_2 \end{array}}{(\texttt{pf\_impE}\ M_1\ M_2\ u_1'\ u_2') :: E; \Gamma \xrightarrow{\alpha} s_2 \circ [u_1'', u_2'']}\supset\text{E}$$

$$\frac{M :: E; \Gamma \xrightarrow{\alpha} \forall x{:}\sigma.s \circ [u_1, u_2]}{(\texttt{pf\_forallE}\ M\ t) :: E; \Gamma \xrightarrow{\alpha} s[t/x] \circ [u_1, u_2]}\forall\text{E}$$

$$\frac{E \models i}{(\texttt{pf\_sinjI}) :: E; \Gamma \xrightarrow{\alpha} i \circ [u_1, u_2]}\text{interI}$$

$$\frac{\models c}{(\texttt{pf\_cinjI}) :: E; \Gamma \xrightarrow{\alpha} c \circ [u_1, u_2]}\text{consI}$$

Figure 2. BL: Natural Deduction (Selected rules)

Figure 2 shows selected and slightly simplified rules of the proof system. As usual, we have introduction and elimination rules for each connective (marked I and E respectively). For a syntactic entity R, $R[t/x]$ denotes the capture avoiding substitution of term t for variable x in R. The rule (hyp) states that the assumption $s \circ [u_1', u_2']$ entails $s \circ [u_1, u_2]$ if $u_1' \leq u_1$ and $u_2 \leq u_2'$, i.e. the interval $[u_1, u_2]$ is a subset of the interval $[u_1', u_2']$. This makes intuitive sense: if a formula s holds throughout an interval, it must hold on every subinterval as well. The proof term corresponding to this (trivial) derivation is π, where π is also the name for the assumption $s \circ [u_1', u_2']$. The rule (claims) is similar, except that it allows us to conclude $s \circ [u_1, u_2]$ from the assumption k' claims $s \circ [u_1', u_2']$. In this case, it must also be shown, among other things, that k' is stronger than the principal k in the view (premise $\models k' \succeq k$).

(saysI) is the only rule which changes the view. The notation $\Gamma|$ in this rule denotes the subset of Γ that contains exactly the claims of principals, i.e. the set $\{(k' \text{ claims } s' \circ [u_1', u_2']) \in \Gamma\}$. The rule means that $(k \text{ says } s) \circ [u_1, u_2]$ holds in any view α if $s \circ [u_1, u_2]$ holds in the view k, u_1, u_2 using only claims of principals. Assumptions of the form $s' \circ [u_1', u_2']$ are eliminated from Γ in the premise because they may have been added in the view α (using other rules

not shown here), but may not hold in the view k, u_1, u_2.

($\supset$E) is a variant of the common rule of modus ponens. It means that if $s_1 \supset s_2$ holds during an interval $[u_1, u_2]$, and s_1 holds during a *subinterval* $[u'_1, u'_2]$, then s_2 must hold during any interval $[u''_1, u''_2]$, which is contained in both. ($\forall$E) states that if $\forall x{:}\sigma.s$ holds during some interval $[u_1, u_2]$, then $s[t/x]$ holds during the same interval for any term t.

The rule (interI) states that an interpreted atomic formula i is provable if it holds in the abstract logical representation of the environment E. The proof term `sinjI` that appears in this rule has no specific structure; it is merely a marker to indicate that a decision procedure must be invoked to check i in the prevailing environment. The rule (consI) is similar but it is used to establish constraints. It should be noted that the view α is relevant only for assumptions in Γ and is not used in the rules (interI) and (consI).

Meta-theory: A meta-theorem is a theorem about the proof system in general. For an access control logic like BL, whose application is based on verification of proofs, meta-theory of the proof system is an important foundational justification [22]. We state below two important meta-theorems about BL's proof system (substitution and subsumption). Structural theorems such as weakening for the hypotheses also hold, but we do not state them explicitly. $M[M'/\pi]$ denotes the capture-avoiding substitution of proof term M' for the name π in the proof term M.

Theorem III.1 (Substitution)**.** *Suppose the following hold:*

1) $M' :: E; \Gamma \xrightarrow{\alpha} s \circ [u_1, u_2]$
2) $M :: E; \Gamma, \pi : s \circ [u_1, u_2] \xrightarrow{\alpha} r \circ [u'_1, u'_2]$

Then, $(M[M'/\pi]) :: E; \Gamma \xrightarrow{\alpha} r \circ [u'_1, u'_2]$

Theorem III.2 (Subsumption)**.** *Suppose the following hold:*

1) $M :: E; \Gamma \xrightarrow{\alpha} s \circ [u_1, u_2]$
2) $\models u_1 \leq u'_1$ *and* $\models u'_2 \leq u_2$

Then, $M :: E; \Gamma \xrightarrow{\alpha} s \circ [u'_1, u'_2]$

Several other meta-theorems about BL, its sequent calculus, and proofs of the above theorems are available in a separate report [20, Chapter 4].

B. Connection to Enforcement

Representation of files and principals: The logic BL does not mandate how files and users are concretely represented. However, from the perspective of an implementation, making this choice is important. In PCFS, files and directories are represented by their full path names, relative to the path where PCFS is mounted. Thus, if PCFS is mounted at `/path/to/mountpoint`, then the file `/foo/bar` in any formula refers to the file `/path/to/mountpoint/foo/bar` in the file system. Principals are represented in one of two ways: either as symbolic constants, or by their Linux user ids. The former is used to represent principals that do not correspond to any real users (e.g., organizational roles), while the latter is used for principals that do (e.g., users that run programs and access files). Permissions are given on a per-file (or per-directory) basis to real users.

Representation of policy in BL: If an administrator k creates a rule represented by formula s, and puts it in a certificate that is valid from time u_1 to time u_2, then this rule is reflected in BL as the assumption k claims $s \circ [u_1, u_2]$. In addition, we require that each rule be accompanied by a unique name (a string), which is written in the certificate with the rule. This name is used to refer to the assumption in proofs. The whole policy has the form $\Gamma = \{\pi_i : k_i \text{ claims } s_i \circ [u_i, u'_i] \mid 1 \leq i \leq n\}$, where k_i's are administrators, and π_i's are unique names for the rules of the policy.

What should be proved?: We assume the existence of one distinguished administrator, symbolically denoted admin, who is the ultimate authority on access. In order to get permission η on file f at time u, user k must prove that the policy in effect entails the defined judgment $\texttt{auth}(k, f, \eta, u)$, where:

$$\texttt{auth}(k, f, \eta, u) \triangleq (\text{admin says } \texttt{may}(k, f, \eta)) \circ [u, u]$$

`may` is a fixed uninterpreted predicate taking three arguments, and u is the time of access ($[u, u]$ is a singleton set on the integer line containing exactly the time point u).

When we start constructing a proof in BL at the top level, the exact view α does not matter. As a result, to get access, it must be shown that: $E; \Gamma \xrightarrow{\alpha} \texttt{auth}(k, f, \eta, u)$, where α is a view made of fresh constants, and Γ is the policy. When E is fixed, we often abbreviate this hypothetical judgment to $\Gamma \vdash \texttt{auth}(k, f, \eta, u)$.

Usually, admin delegates part of its authority to other administrators through rules. Also, in most policies, admin may have authority over the predicate `may` but not other predicates. For this reason, it is advisable to keep `admin` distinct from ℓ, the strongest principal whom everyone believes on every predicate.

Interpreted Predicates: The implementation of BL in PCFS natively supports two interpreted predicates, although support for other predicates can be added easily through a programming API provided for this purpose. These two predicates are: $\mathbf{owner}(f, k)$, which means that file f has owner k, and $\mathbf{has\_xattr}(f, a, v)$, which means that file f has value v for the extended attribute `user.#pcfs.a`. Extended attributes beginning with the prefix `user.#pcfs.` are specially protected by PCFS – a special permission called "govern" is needed to change them. By limiting this permission to trusted individuals only, policies may use these attributes to label files in a secure manner, as illustrated in Section IV. Interpreted predicates are written in **boldface** to distinguish them from others.

C. Expressiveness of BL

Besides the realistic case study described in Section IV, we evaluate expressiveness of BL in two ways. First, on

the practical side, we have examined common access policy policy idioms that can be expressed in BL and those that cannot be expressed in BL. We mention some of these idioms briefly (details may be found in [20, Sections 3.1.2 and 4.1.2]). Access control matrices, administrative roles and groups, and delegation of authority can be represented in BL using the says modality and first-order quantifiers, as in other access control frameworks [11, 17, 22]. Further, using the @ connective, we can express certificate expiration directly. Using the @ connective we can also express policies that rely on explicit points of time in the past. As an example of this class of policies, the following rule, created by an administrator A, states that if principal k becomes an alumnus at time T and has access to file f at time T, then k will continue to have access to f for 180 days after T.

$$\begin{aligned}&\mathsf{A}\ \mathsf{says}\ \forall k, f, T.\\ &\quad((\texttt{alumni}\ k\ T) \wedge ((\texttt{mayaccess}\ k\ f)\ @\ [T, T]))\\ &\qquad\supset ((\texttt{mayaccess}\ k\ f)\ @\ [T, T + 180d]))\end{aligned}$$

Such policies cannot be represented in access control logics that treat time as a part of the state and, consequently, allow policies to refer to the time of access but not other points of time (e.g., [9, 11]). This increased expressiveness justifies the separation of time from other forms of state in BL.

BL's novel interpretation of the says modality through views in hypothetical judgments also improves expressiveness as compared to other access control logics, particularly those that admit the axiom $s \supset (k\ \mathsf{says}\ s)$ [4, 7, 21, 22] (BL does not admit this axiom). In BL's proof system $k'\ \mathsf{says}\ ((k\ \mathsf{says}\ s) \supset s')$ and $k\ \mathsf{says}\ s$ imply $k'\ \mathsf{says}\ s'$, but $k'\ \mathsf{says}\ ((k\ \mathsf{says}\ s) \supset s')$ and $k'\ \mathsf{says}\ s$ do not, in general, imply $k'\ \mathsf{says}\ s'$. As a result, $k'\ \mathsf{says}\ ((k\ \mathsf{says}\ s) \supset s')$ represents a transfer of authority over formula s from k' to k, wherein k' does not retain control of s. This is useful in representing many realistic access policies, including several parts of our case study (Section IV). In logics that admit the axiom $s \supset (k\ \mathsf{says}\ s)$, $k'\ \mathsf{says}\ ((k\ \mathsf{says}\ s) \supset s')$ and $k'\ \mathsf{says}\ s$ imply $k'\ \mathsf{says}\ s'$, so representing this form of delegation is difficult.

Using constraints that test for set membership, policies that require approval of k out of n designated principals can be expressed easily in BL (see [11, Section 5] for details). In common with other access control logics, it is difficult to express in BL policies for which order of rules matters, as well as policies that are contradictory. However, it should be noted that, like many other access logics, $k\ \mathsf{says}\ \bot$ does not imply $\bot$ in BL, so contradictory statements by individuals do not make the policy inconsistent as a whole.

To further establish the expressiveness of BL as a logic for representing access policies, we have translated two existing policy frameworks into BL, and proved soundness and completeness of the translations. The two frameworks are an access control logic used in a significant amount of prior work [4, 8, 21, 22, 28] and the authorization framework Soutei [36]. We refer the reader to another report for details of the translations [20, Section 3.5].

IV. Case Study: Classified Information in U.S.

Based on information obtained from public government documents, and through an industrial collaborator, we have completed a realistic case study that formalizes several policies for access to sensitive (classified or potentially classifiable) information in the intelligence community in the U.S. The study is quite extensive and contains approximately 50 fixed rules that exercise both explicit time as well as system state in BL. Extended attributes of files, manifest in BL through interpreted predicates, are used to represent the classification status of files (classified vs unclassified) and their classification level. Attributes of individuals are specified in certificates issued by administrators, many of which expire at fixed points of time. For example, some background checks expire every 5 years. These expirations are represented using the @ connective in BL.

Whereas the entire case study is available in a separate technical report [23], in this section we present two policy rules adapted from this case study as an illustration of the use of BL. Access to classified information in practice is more complicated than indicated by our illustrative example and involves background checks, credit checks, and polygraph tests among other things, all of which we omit here to keep the illustration simple.

Each sensitive file in an intelligence agency is assumed to have a classification level, which is an element of the ordered set $\mathsf{confidential} < \mathsf{secret} < \mathsf{topsecret}$. Dually, each individual affiliated with such an agency has a clearance level, also from the same set. This clearance level indicates the highest level of classified files the individual may read. For the purpose of representation in BL, we assume that the classification level of a file is written in an extended attribute $\texttt{user.\#pcfs.level}$ on the file. We also assume one distinguished administrator, hr, who is responsible for deciding attributes of users, e.g., giving them classification levels and employment certifications.

Let us assume that principal k may read file f only if three conditions are met: (a) k is an employee of the intelligence organization (predicate $\texttt{employee}(k)$), (b) k has a classification level above the file (predicate $\texttt{hasLevelForFile}(k, f)$), and (c) k gets permission from the owner of the file. Suppose that this rule came in effect in 2000, and will remain in effect till 2010. The following credential (created by admin) captures this rule in BL. For readability, we omit sort annotations from quantifiers.

(1) $$\begin{aligned}&\mathsf{admin}\ \mathsf{claims}\ \forall k, k', f.\\ &\quad(((\mathsf{hr}\ \mathsf{says}\ \texttt{employee}(k)) \wedge\\ &\qquad(\texttt{hasLevelForFile}(k, f)) \wedge\\ &\qquad(\mathbf{owner}(f, k')) \wedge\\ &\qquad(k'\ \mathsf{says}\ \texttt{may}(k, f, \mathsf{read}))) \supset \texttt{may}(k, f, \mathsf{read}))\\ &\quad\circ\ [2000, 2010]\end{aligned}$$

The predicate $\texttt{hasLevelForFile}(k, f)$ may further be defined by admin in terms of classification levels of k and f.

(2) $\mathsf{admin}\ \mathsf{claims}\ \forall k, f, l, l'.$
$(((\mathbf{has\_xattr}(f, \texttt{level}, l) \wedge$
$(\mathsf{hr}\ \mathsf{says}\ \texttt{levelPrin}(k, l')) \wedge$
$(\texttt{below}(l, l'))) \supset \texttt{hasLevelForFile}(k, f))$
$\circ\ [2000, 2010]$

It is instructive to observe how the interpreted predicate **owner** has been used to capture the actual owner of the file as written in its meta data, and how **has_xattr** has been used to reflect the file's classification level in the logic. The predicate $\texttt{below}(l, l')$ captures the order $l < l'$ between classification levels. We assume that all principals believe this order. Hence it is stated by the strongest principal ℓ.

(3) $\ell\ \mathsf{claims}\ \texttt{below}(\mathsf{confidential}, \mathsf{secret}) \circ [2000, 2010]$
(4) $\ell\ \mathsf{claims}\ \texttt{below}(\mathsf{secret}, \mathsf{topsecret}) \circ [2000, 2010]$
(5) $\ell\ \mathsf{claims}\ \texttt{below}(\mathsf{confidential}, \mathsf{topsecret}) \circ [2000, 2010]$

As an illustration of the use of this policy, let us assume that file /secret.txt is owned by Alice (user id 1003) and classified at the level secret. Thus, the following must hold in the file system state E at the time of access:

(A) $E \models \mathbf{owner}(/\mathsf{secret.txt}, \mathsf{uid}\ 1003)$
(B) $E \models \mathbf{has\_xattr}(/\mathsf{secret.txt}, \texttt{level}, \mathsf{secret})$

Suppose further that Bob (user id 1500) is an employee cleared at level topsecret from 2007 to 2009, and further that Alice wants to let Bob read file /secret.txt from 2008 to 2009. This information may be captured by the following formulas (signed by the respective principals).

(6) $\mathsf{hr}\ \mathsf{claims}\ \texttt{employee}(\mathsf{uid}\ 1500) \circ [2007, 2009]$
(7) $\mathsf{hr}\ \mathsf{claims}\ \texttt{levelPrin}(\mathsf{uid}\ 1500, \mathsf{topsecret}) \circ [2007, 2009]$
(8) $(\mathsf{uid}\ 1003)\ \mathsf{claims}\ \texttt{may}(\mathsf{uid}\ 1500, \mathsf{secret.txt}, \mathsf{read}) \circ [2008, 2009]$

Let Γ denote the set of policy rules (1)–(8) (with corresponding names p1–p8). Then using the rules of Figure 2 we can show that there is a proof term M such that $M :: E; \Gamma \xrightarrow{\alpha} (\mathsf{admin}\ \mathsf{says}\ \texttt{may}(\mathsf{uid}\ 1500, /\mathsf{secret.txt}, \mathsf{read})) \circ [2008, 2009]$, if E satisfies the conditions (A) and (B). From Theorem III.2, it follows that $M :: E; \Gamma \xrightarrow{\alpha} \texttt{auth}(\mathsf{uid}\ 1500, /\mathsf{secret.txt}, \mathsf{read}, u)$ whenever $u \in [2008, 2009]$, and hence Bob should be able to read /secret.txt from 2008 to 2009. This is what we may intuitively expect because the intersection of the validity of all certificates issued here is exactly $[2008, 2009]$. Further, $M :: E; \Gamma \xrightarrow{\alpha} \texttt{auth}(\mathsf{uid}\ 1500, /\mathsf{secret.txt}, \mathsf{read}, u)$ will not hold if $u \notin [2008, 2009]$ or E does not satisfy either (A) or (B). Therefore, the proof M correctly reflects the dependence of policy rules on both time as well as system state.

V. PCFS Front End: Proof Search and Verification

Next, we describe the front end of the PCFS implementation. We start by describing the proof search tool briefly, and then turn to a formal description of proof verification and the structure of procaps.

A. Automatic Proof Search

Even though users are free to construct proofs of access by any means they like, PCFS provides a command line tool called `pcfs-search` for performing this task automatically. As discussed in Section III, the objective is to prove a judgment of the form $E; \Gamma \xrightarrow{\alpha} (\mathsf{admin}\ \mathsf{says}\ \texttt{may}(k, f, \eta)) \circ [u, u]$, where u is the expected time of access. In almost all cases, it is unreasonable to expect that the time of access can be predicted in advance to the precision of seconds (which is the precision at which enforcement of time works in PCFS), so instead of an exact time u, the user is expected to provide a range of time $[u_1, u_2]$ during which she desires access. The user must also provide the parameters k, f, η and the policy Γ in the form of certificates obtained from administrators. The output of the tool is the proof term M such that $M :: E; \Gamma \xrightarrow{\alpha} (\mathsf{admin}\ \mathsf{says}\ \texttt{may}(k, f, \eta)) \circ [u_1, u_2]$. By Theorem III.2 it follows that $M :: E; \Gamma \xrightarrow{\alpha} \texttt{auth}(k, f, \eta, u)$ for every $u \in [u_1, u_2]$, so this proof term M can be used for access at any time in the interval $[u_1, u_2]$.

Proof search in BL is, in general, an undecidable problem because BL extends first-order intuitionistic logic, which is itself undecidable. However, as past work on languages and logics for authorization shows [11, 13, 17, 31, 36], most access policies in practice fit into a restricted fragment of logic on which logic programming techniques can be used for proof construction. Although logic programming methods work fast, extending them from fragments of first-order logic (where they are well understood) to BL's two modalities – $k\ \mathsf{says}\ s$ and $s\ @\ [u_1, u_2]$ – is a challenging task. The second modality is particularly difficult to handle since it interacts with all other connectives of BL in non-trivial ways. Due to a lack of space, we omit details of the proof search method, but mention only that it is based on a fragment of BL over which goal-directed search (e.g., [32]) is complete. The prover searches for proofs depth-first like Prolog for reasons of efficiency, although it is also possible to search breadth-first, which would result in a semi-decision procedure for the fragment.

A salient point about the prover is that it must have access to decision procedures for interpreted predicates I and constraints c. When the prover needs to establish one of these, it calls the corresponding decision procedure. A point of concern is that the file system state E at the time of proof search may not be what the user expects it to be at the time of access. As a result, a proof search may fail, when a proof would actually exist at the time that the user desires access. To construct such "optimistic" proofs, `pcfs-search` can

be run in interactive mode, where the tool asks for user input about expected file system state if it fails to construct a proof in the prevailing state.

B. Proof Verification and Procaps

The proof verifier checks proofs that a user constructs and issues procaps in return. Since these procaps can be directly used for access, the proof verifier is a trusted program. Briefly, the proof verifier is invoked with a command line tool `pcfs-verify`. It is given as input the policy Γ (in the form of signed certificates), the parameters k, f, η, and a proof term M. The verifier first checks that the policy is correct, i.e. all its certificates have authentic digital signatures. For this, the verifier must have access to some public key infrastructure (PKI) that maps public keys to principals that own them. We use a simple PKI, with a single certifying authority (CA) that certifies all keys. The public key of the CA is stored in a specially protected section of the file system itself as described in Section VI.

Second, the verifier checks the logical structure of the proof term, i.e. $M :: E; \Gamma \xrightarrow{\alpha} \mathtt{auth}(k, f, \eta, u)$. This verification is performed bidirectionally to minimize the need for annotating proof terms with formulas [35]. Although this is mostly standard, subtleties arise in the PCFS architecture due to two reasons.

1) The proof should be checked in the file system state E which would prevail when the procap is used in future, but it may be impossible to predict this state at the time of proof verification.
2) It may not be possible to predict u, the time(s) at which the procap generated from the proof will be used for access.

In PCFS, problem (1) is addressed by *never checking interpreted predicates* during proof verification. Instead, when the verifier encounters the proof term `pf_sinjI`, which corresponds to an application of the rule (interI) from Figure 2, the verifier writes the interpreted predicate i to be checked in the output procap. This predicate must then be checked by the file system back end when the procap is used. As a result, any interpreted predicates on which the validity of the proof depends are transferred as is to the procap, and are checked at the time of access in the state prevalent at that time.

To address problem (2), the verifier uses a *special symbolic constant* ctime, which has sort time, in place of u: the verifier tries to check that $M :: \cdot; \Gamma \xrightarrow{\alpha} \mathtt{auth}(k, f, \eta, \mathsf{ctime})$. During verification, many constraints of the form $u_1 \leq u_2$ may fail to check, e.g., in the rules (hyp), (claims), ($\supset$E), and (consI), because either u_1 or u_2 contains the unresolved constant ctime. If this happens, then instead of verifying the constraint using the decision procedure, the constraint is written into the output procap. During file access, the file system back end *substitutes* the actual time of access for the constant ctime in the procap and checks the resulting ground constraint.

$$\frac{\vec{c_1} \bowtie \vec{c_2} = (u_1' \leq u_1, u_2 \leq u_2') \qquad \models \vec{c_2}}{\pi :: \cdot; \Gamma, \pi : s \circ [u_1', u_2'] \xRightarrow{\alpha} s \circ [u_1, u_2] \searrow \vec{c_1}; \cdot}\text{hyp}$$

$$\frac{\begin{array}{c}\alpha = k, u_b, u_e \qquad \models \vec{c_2} \\ \vec{c_1} \bowtie \vec{c_2} = (u_1' \leq u_1, u_2 \leq u_2', u_1' \leq u_b, u_e \leq u_2', k' \succeq k)\end{array}}{\pi :: \cdot; \Gamma, \pi : k' \ \mathsf{claims}\ s \circ [u_1', u_2'] \xRightarrow{\alpha} s \circ [u_1, u_2] \searrow \vec{c_1}; \cdot}\text{claims}$$

$$\frac{M :: \cdot; \Gamma |\ ^{k, u_1, u_2}\!\!\Longleftarrow s \circ [u_1, u_2] \searrow \vec{c}; \vec{i}}{(\mathtt{pf\_saysI}\ M) :: \cdot; \Gamma \xLeftarrow{\alpha} (k\ \mathsf{says}\ s) \circ [u_1, u_2] \searrow \vec{c}; \vec{i}}\text{saysI}$$

$$\frac{\begin{array}{c}M_1 :: \cdot; \Gamma \xRightarrow{\alpha} s_1 \supset s_2 \circ [u_1, u_2] \searrow \vec{c_1}; \vec{i_1} \\ M_2 :: \cdot; \Gamma \xLeftarrow{\alpha} s_1 \circ [u_1', u_2'] \searrow \vec{c_2}; \vec{i_2} \\ \vec{c_3} \bowtie \vec{c_4} = (u_1 \leq u_1' \leq u_1'', u_2'' \leq u_2' \leq u_2) \qquad \models \vec{c_4}\end{array}}{\begin{array}{c}(\mathtt{pf\_impE}\ M_1\ M_2\ u_1'\ u_2') :: \cdot; \Gamma \\ \xLeftarrow{\alpha} s_2 \circ [u_1'', u_2''] \searrow \vec{c_1}, \vec{c_2}, \vec{c_3}; \vec{i_1}, \vec{i_2}\end{array}}\supset\text{E}$$

$$\frac{}{(\mathtt{pf\_sinjI}) :: \cdot; \Gamma \xLeftarrow{\alpha} i \circ [u_1, u_2] \searrow \cdot; i}\text{interI}$$

$$\frac{\vec{c_1} \bowtie \vec{c_2} = c \qquad \models \vec{c_2}}{(\mathtt{pf\_cinjI}) :: \cdot; \Gamma \xLeftarrow{\alpha} c \circ [u_1, u_2] \searrow \vec{c_1}; \cdot}\text{consI}$$

Figure 3. BL proof verification in PCFS (Selected rules)

Interpreted predicates and constraints written to a procap as described above together constitute the *conditions* of the procap.

Formal description of PCFS proof verification: We present a simplified formalization of the verification procedure of PCFS and prove that successful execution of the procedure on a proof followed by checking of the resulting conditions at the time of access is equivalent to checking the entire proof at the time of access.[3] This implies that the formal guarantees obtained from PCFS are equal to those from proof-carrying authorization without procaps.

Let $\vec{c}$ and $\vec{i}$ denote multisets of constraints and interpreted predicates respectively. Proof verification in PCFS is formalized by two hypothetical judgments: the *checking* judgment $M :: \cdot; \Gamma \xLeftarrow{\alpha} s \circ [u_1, u_2] \searrow \vec{c}; \vec{i}$ and the *inference* judgment $M :: \cdot; \Gamma \xRightarrow{\alpha} s \circ [u_1, u_2] \searrow \vec{c}; \vec{i}$. The intent of both judgments is that if $\models \vec{c}$ and $E \models \vec{i}$, then $M :: E; \Gamma \xrightarrow{\alpha} s \circ [u_1, u_2]$ in BL's natural deduction system (Figure 2). As is standard in bidirectional proof verification [35], $s \circ [u_1, u_2]$ is an *input* to the verification procedure in the checking judgment, and an *output* of the procedure in the synthesis judgment. M, E, Γ, α are inputs in both cases. The novelty here are the multisets $\vec{c}, \vec{i}$ which are outputs of both judgments and form the conditions of

[3] As in Section III, the simplification here is that we do not explicitly record parameters that may appear in hypothetical judgments, nor do we consider hypothetical constraints.

the procap generated after proof verification.

Selected rules for establishing these two judgments are shown in Figure 3. The notation $\vec{c_1} \bowtie \vec{c_2} = \vec{c}$ in these rules means that $\vec{c_1}$ and $\vec{c_2}$ are a disjoint partition of $\vec{c}$ such that $\vec{c_1}$ has all constraints of $\vec{c}$ which contain ctime and do not hold in that generality, and $\vec{c_2}$ has all the remaining constraints of $\vec{c}$. The rules of verification are similar to those of natural deduction. The difference is that in each rule of proof verification, constraints that would have been verified in the premise of the corresponding rule of natural deduction but cannot be checked because they contain ctime are added to the output $\vec{c}$ in the conclusion of the rule (from where they get written to the procap generated, and are checked whenever the procap is used to authorize access), whereas the remaining constraints are checked immediately. In the rule (interI), instead of checking the interpreted atom i as in natural deduction, we write i to the output conditions.

The rules, when applied backwards, can be interpreted as a decision procedure for checking proof terms. The following theorem establishes formally the soundness and completeness of this verification procedure. For the proof, see [20, Section 5.2].

Theorem V.1 (Correctness of PCFS Verification)**.** *Let M, Γ, α, s not contain the constant* ctime*. Then the following hold.*

1) *(Soundness) If $M :: \cdot; \Gamma \overset{\alpha}{\Longleftarrow} s \circ [\mathsf{ctime}, \mathsf{ctime}] \searrow \vec{c}; \vec{i}$, $\models \vec{c}[u/\mathsf{ctime}]$, and $E \models \vec{i}$, then $M :: E; \Gamma \overset{\alpha}{\longrightarrow} s \circ [u, u]$.*
2) *(Completeness) If $M :: E; \Gamma \overset{\alpha}{\longrightarrow} s \circ [u, u]$ then there exist $\vec{c}$ and $\vec{i}$ such that $M :: \cdot; \Gamma \overset{\alpha}{\Longleftarrow} s \circ [\mathsf{ctime}, \mathsf{ctime}] \searrow \vec{c}; \vec{i}$, $\models \vec{c}[u/\mathsf{ctime}]$, and $E \models \vec{i}$.*

Soundness in the above theorem means that if a proof term M is verified in the PCFS verifier as being a witness for s if the conditions $\vec{c}$ and $\vec{i}$ hold, then in any system state E and at any time u such that $\models \vec{c}[u/\mathsf{time}]$ and $E \models \vec{i}$, M is indeed a correct proof of $s \circ [u, u]$. Completeness is dual: if M is a proof term that correctly shows $s \circ [u, u]$ for some time u in some system state E, then PCFS verification on M (at any other time and in any state) will succeed in outputting some constraints $\vec{c}$ and some interpreted predicates $\vec{i}$, both of which will check at time u in state E. Together, the two clauses of the theorem imply that procap based enforcement in PCFS has the same formal guarantees as proof-carrying authorization (the latter would check the proof term M entirely at the time of access).

C. Procap structure

Formally, a procap contains four-tuple $\langle \psi, \vec{i}, \vec{c}, \Sigma \rangle$, where

- $\psi = \langle k, f, \eta \rangle$ is a triple that lists the principal, file, and permission that the procap authorizes.
- $\vec{i}$ is a list of interpreted predicates on which the verified proof depends.
- $\vec{c}$ is a list of constraints that contain the constant ctime, and on which the proof depends.
- Σ is a cryptographic signature over the first three components (written in ASCII syntax). This guarantees the procap's integrity.

The procap $\langle \psi, \vec{i}, \vec{c}, \Sigma \rangle$ is generated by the proof verifier when it successfully checks the certificates that constitute a policy Γ and the judgment $M :: \cdot; \Gamma \overset{\alpha}{\Longleftarrow} \mathtt{auth}(k, f, \eta, \mathsf{ctime})$ for some M and a view α composed of fresh constants. The signature Σ is generated using a symmetric secret key that is stored in a specially marked file in the PCFS file system (Section VI). The file system back end ensures that *only a specific user id* (called pcfssystem) has read access to this file. The verification tool `pcfs-verify`'s disk file is owned by this user, and executes with a set-uid bit. As a result, when a user invokes this program, it runs with pcfssystem's user id and, hence, gets access to the secret key. Other users do not have access to the key and, therefore, cannot forge procaps.

Before admitting a procap, the file system back end must check not only the procap's signature Σ, but also the interpreted predicates $\vec{i}$ in the state prevalent at the time of access and the constraints in the list $\vec{c}$ with ctime substituted by the actual time of access.

Example 1: At the end of Section IV, we constructed a proof term M which established $E; \Gamma \overset{\alpha}{\longrightarrow} (\mathsf{admin\ says}\ \mathtt{may}(\mathsf{uid}\ 1500, /\mathsf{secret.txt}, \mathsf{read})) \circ [2008, 2009]$, where E was required to satisfy the two conditions (A) and (B). If we give this proof term to the PCFS proof verifier, the resulting procap will have the structure $\langle \psi, \vec{i}, \vec{c}, \Sigma \rangle$, where

- $\psi = \langle \mathsf{uid}\ 1500, /\mathsf{secret.txt}, \mathsf{read} \rangle$
- $\vec{i} = \mathbf{owner}(/\mathsf{secret.txt}, \mathsf{uid}\ 1003)$, $\mathbf{has\_xattr}(/\mathsf{secret.txt}, \mathtt{level}, \mathsf{secret})$
- $\vec{c} = 2008{:}01{:}01{:}00{:}00{:}00 \leq \mathsf{ctime}$, $\mathsf{ctime} \leq 2009{:}12{:}31{:}23{:}59{:}59$

The predicates in list $\vec{i}$ imply that the procap is valid only in a state where the file /secret.txt is owned by Alice, and the file has extended attribute `user.#pcfs.level` set to secret. These correspond exactly to conditions (A) and (B) from Section IV. The list $\vec{c}$ means that the time of access (ctime) must lie in the interval $[2008, 2009]$, which is also what we expect from the policy rules.

Certificate Revocation: A revocation refers to the withdrawal of a signed policy rule or fact after it has been created but before it expires. Revocations are an issue for enforcement because proofs and procaps depending on a revoked statement may already have been generated. There are two simple ways to enforce revocations using procaps, both of which we describe briefly. The methods have an efficiency vs accuracy trade-off and one of these may be selected depending on deployment requirements. The current implementation of PCFS does not support revocation, but either of these methods is easy to add.

- A list of unique ids of certificates on which a proof depends can be included in the procap generated from the proof. Before admitting a procap, the file system back end can compare the list of certificate ids in it to a list of revoked certificates provided by administrators. If there is an overlap, the procap can be rejected. Although this would enforce revocation perfectly, it would also slow down file access because an additional check would be performed on each procap.
- Alternatively, the list of revoked certificates can be provided to the proof verifier instead of the file system back end. The verifier can then refuse to accept any proof that depends on revoked certificates. If the verifier issues a procap, it can be short lived, i.e. its validity can be restricted to a short duration T using a constraint on ctime. Although the effect of revocation in this method is not immediate (it can lag by a time T), the back end performance is not affected.

VI. PCFS Back end

Whereas the front end of PCFS is used to generate procaps from proofs of access, the back end grants access to files and directories, using procaps to check access rights. Since the back end is invoked at every file access, it has to be extremely efficient. In this section, we discuss the design and implementation of the PCFS back end.

Overall architecture: The back end of PCFS is implemented as a process server, which listens to upcalls made from the kernel. The implementation relies on the Linux kernel module Fuse [2]. When a call is made to a file system resource within the mount path of the PCFS file system, this module makes an upcall to the process server, and depending on the operation requested, a specific handling function is invoked. There is roughly one function for every POSIX file system operation (open, read, write, stat, unlink, rmdir, mkdir, etc.). Depending on the operation, this handling function first finds and checks procaps corresponding to permissions needed to perform the operation. If these checks succeed, it invokes an identical file system call, but on a different mount path, which is actually an ext3 file system. In order to bypass any access checks during the call to the ext3 file system, the process server runs with superuser privileges. Further to prevent users from directly using the ext3 file system to access the data, we give ownership of the root directory on the ext3 file system to the superuser, and turn off all access on it.[4] Using an ext3 file system for I/O makes our back end simpler since we need to implement only the code to look up the procaps and to check them.

[4]A more secure method to prevent access via the underlying file system is to keep data encrypted on it, and to decrypt data in the process server. We have not implemented this design since our objective here is to measure the performance of access checks.

Organization of the file system: For the purpose of illustration let us assume that PCFS is mounted at `/pcfs`, and that it makes calls to the ext3 file system at `/src`. Then the file structure visible inside `/pcfs` is the same as that in `/src`, except that all calls on the former are subject to procap based checks. A special directory `/pcfs/#config` contains configuration data for the file system, including procaps and the secret key used to sign them. This directory is protected by the file system with strict rules that do not use procaps. We list below some of the important files and directories in this special directory, as well their contents and protections.

`/pcfs/#config/config-file`: File containing configuration options, including the user ids of the principals admin and pcfssystem. (Recall from Section V that pcfssystem is the only user who has access to the secret key needed to sign procaps.) Anyone can read this file, but only pcfssystem can change this file.

`/pcfs/#config/shared-key`: Contains the shared key used to sign procaps. Only pcfssystem may read or write this file.

`/pcfs/#config/ca-pubkey.pem`: Contains the public key of the certifying authority who signs associations between other public keys and users. Anyone may read this file, but only pcfssystem may write to it.

`/pcfs/#config/procaps/`: This directory contains the procaps. Its organization is discussed below. The user pcfssystem has full access to this directory and other users have access to specific subdirectories only.

Within the directory `/pcfs/#config/procaps/`, there is a subdirectory for each user that stores procaps relevant to that user. In general, the procap giving the right $\langle k, f, \eta \rangle$ is stored in the file `/pcfs/#config/procaps/<k>/<f>.perm.<η>`. Here <k> is the user id of the user k, <f> is the path of the file f (relative to the mount point), and <η> is a textual representation of the permission. Thus each procap is stored in a separate file and, further, for each right $\langle k, f, \eta \rangle$ there is at most one procap that authorizes the right. While this may be restrictive, it makes procap look up easy since the path where a procap is to be found can be determined simply by knowing the PCFS mount point and the right $\langle k, f, \eta \rangle$. The PCFS server ensures that only user k can access (read, write, or delete) files inside `/pcfs/#config/procaps/<k>/`. This is done to prevent denial of service attacks by other users.

The user pcfssystem has complete access to all files and directories within `/pcfs/#config/`, and is expected to act as the maintainer of configuration in a PCFS file system. In particular, it may periodically delete obsolete procaps. At the same time, the user account of pcfssystem may be a very attractive target for attack: If an attacker gains control

Operation	Permissions needed
stat /foo	execute on /foo
open /foo in read mode	read on /foo
open /foo in write mode	write on /foo
create /bar/foo	write on /bar
delete /bar/foo	identity on /bar/foo
rename /bar to /foo	identity on /bar, write on /foo
getxattr on /foo	execute on /foo
setxattr on /foo	govern on /foo if attribute starts with user.#pcfs., write otherwise
chown on /foo	govern on /foo

Figure 4. Permissions needed to perform some operations

of this user account, it can obtain the secret key used to sign and verify procaps, and then inject fake procaps to access other files. To prevent this, the PCFS process server denies pcfssystem all rights in *other* directories inside the file system. Thus, to attack the file system through this mechanism, the attacker must break into at least one more account in addition to pcfssystem.

Permissions: PCFS uses five distinct permissions on any file or directory: read, write, execute, identity, and govern. In contrast, POSIX mandates only the first three permissions. The read and write permissions are the obvious ones; they are needed to read and to change the contents of a file/directory respectively. As usual, execute is the permission to read the meta data of a file or directory. The identity permission is needed to delete a file or directory, or to rename it. This permission is separated from others because in many settings, administrators may not want to allow users to be able to delete or rename shared files, but perform other operations on them. The govern permission is needed to change the owner of a file and to change extended attributes starting with the prefix user.#pcfs. Because of this special protection, these attributes can be used by administrators to give classification or security labels to files, on which policy rules can depend, as illustrated in Section IV. Figure 4 lists the permissions needed to perform some common file system operations. During a file system call, this table is used to determine the procaps that must be looked up and checked. By separating the govern permission from write, we allow for the possibility of mandatory administration of file attributes as, for instance, happens with classification labels in Section IV. Such administration is difficult with POSIX permissions because POSIX allows anyone with write permission on a file to also modify all of its attributes.

Default Permissions: When a program first creates a file, it cannot be assumed that any policy rules apply to the file, since that usually requires creation of certificates by administrators. Yet, many programs create temporary files, to which they must get access. To allow such programs to execute correctly, when a new file or directory is created, PCFS automatically creates and stores default procaps giving the creator of the file read, write, execute, and identity permissions for a fixed period of time (this period can be adjusted at mount time). This allows programs to create and use temporary files. In addition the user admin is given execute and govern rights on the new file. After this period of time elapses, the administrators must create policy rules to control access to the file, since the default procaps expire.

In-memory procap cache: Since procaps are stored in files, and one or more of them must be read to authorize almost every operation on a PCFS file system, it is helpful to cache commonly used procaps in memory to improve performance. Accordingly, PCFS uses a least recently used (LRU) in-memory cache for procaps, whose size can be adjusted through an option in the file /pcfs/#config/config-file. The cache stores parsed procaps, whose signatures have already been verified. The only cost involved in using a cached procap is checking its conditions ($\vec{c}$ and $\vec{i}$ from Section V). This is fast and usually takes only 10–100μs. In contrast, seeking the procap on disk may take a few milliseconds, and parsing it often takes up to 70μs. We evaluate the effect of the cache with different hit rates in Section VII. The PCFS back end automatically marks a cached procap dirty if its corresponding file on disk changes or is deleted. This forces the cached procap to be read again from the disk whenever it is needed next.

VII. Performance of the Back End

We report the results of some performance benchmarks on the back end of our prototype implementation of PCFS. Specifically, we evaluate the overhead of access checks during read, write, stat, create, and delete operations, and measure the effectiveness of the in-memory procap cache through microbenchmarks. To evaluate performance in practice, we also present the results of two simple macrobenchmarks. Since we are primarily interested in measuring the overhead of procap-based access checks, our baseline for comparing performance is a Fuse-based file system that does not perform the corresponding checks, but otherwise runs a server process and uses an underlying ext3 file system, just as PCFS does. We call this file system Fuse/Null. For macrobenchmarks we also compare with a native ext3 file system. All measurements reported here were made on a 2.4GHz Core Duo 2 machine with 3GB RAM and a 7200RPM 100GB hard disk drive, running the Linux kernel 2.6.24-23.

Read and write throughput: By default, PCFS does not make any access checks when read or write operations are performed on a previously opened file. (Access checks on each read and write call can be turned on using an option in the configuration file.) As a result its read and write throughput is very close to that of Fuse/Null. The following table summarizes the read and write throughput of PCFS

and Fuse/Null based on reading and writing a 1GB file sequentially using the Bonnie++ test suite [1].

Operation	PCFS (MB/s)	Fuse/Null (MB/s)
Read	538.69	567.47
Write	73.18	76.05

Even if access checks on every read and write are enabled, the read and write throughput do not show a significant change as long as required procaps remain cached in memory.

File stats and effectiveness of caching: Besides read and write, two other common file operations are open and stat (reading a file's meta-data). In terms of access checks, both are similar, since usually one procap must be checked in each case. (Two procaps must be checked when a file is opened in read and write modes simultaneously.) We report in the table below the speed of the stat operation and the effect of the in-memory procap cache with different hit rates. All measurements are reported in number of operations per second, as well as time taken per operation. The row label $n\%$ indicates a measurement with a cache hit rate of $n\%$. For comparison, performance of Fuse/Null is also shown. The figures are based on choosing a random file 20,000 times in a directory containing exactly 20,000 files, and calling the stat function on it. To get a hit rate of $n\%$, the cache size is set to $n/100 \times 20000$, and the cache is warmed a priori with random procaps. It is easy to prove that for an LRU cache this results in a hit rate of exactly $n\%$ when subsequent files (procaps) are also chosen at random. All procaps used here are default procaps, whose conditions include two constraints of the form $u_1 \leq u_2$, and one interpreted predicate of each of the two forms $\mathbf{has\_xattr}$ and $\mathbf{owner}$.

Cache hit rate	Stats per second	Time per stat (μs)
0%	5774	173.2
50%	7186	139.2
90%	8871	112.7
95%	9851	101.5
98%	11879	84.2
100%	23652	42.2
Fuse/Null	36042	27.7

As can be seen from this table, the procap cache is helpful for fast performance. The difference of the time values in the last two rows is an estimate of the time it takes to check a cached procap, i.e. the time needed to check the conditions in a procap. In this case, this time is $42.2 - 27.7 = 14.5\mu$s. This estimate is rough, and the actual time varies with the complexity of the conditions in the procap. In other experiments, we have found that this time varies from 10 to 100μs. By taking the difference of the time values in the first and last rows, we obtain an estimate of the time required to read a procap, check its signature, parse the procap, and check its conditions. In this experiment, this time is $173.2 - 27.7 = 145.5\mu$s. Additional time may be needed to seek to the procap on disk, which was most likely not counted here, since the procaps used were in a single directory in the underlying file system, hence making the latter's cache very effective. Nonetheless, this suggests that, in general, procap checking is dominated by reading and parsing times. The signatures we use for procaps are message authentication codes, which can be verified in 1 to 2μs each.

File creation and deletion: The table below lists the number of create and delete operations per second that are supported by PCFS and Fuse/Null. These are measured by creating and deleting 10,000 files in a single directory.

Operation	PCFS (op/s)	Fuse/Null (op/s)
Create	1386	4738
Delete	1989	15429

The reason why PCFS is approximately 3.5 times slower than FUSE/Null in creating files is that in this experiment PCFS also created six default procaps for every file created. As a result, the PCFS numbers measure creation of seven times as many files in three separate directories. Similarly, deletion in PCFS in this experiment is nearly 7.7 times slower than that in Fuse/Null because when a file is deleted all procaps related to the file are deleted to prevent useless procaps from accumulating. In this case, each file deletion in PCFS corresponds to seven file deletions on the ext3 file system in three different directories. The effect of the procap cache is negligible during these experiments, since the cache size was kept very small as compared to the number of files. Due to its high cost, automatic deletion of procaps can be turned off using an option in the configuration file.

Macrobenchmarks: To understand the performance of PCFS in practice, we also ran two simple macrobenchmarks, both of which are reasonably intensive in file operations. The first (called OpenSSL in the table below), untars the OpenSSL source code, compiles it and deletes it. The other (called Fuse in the table below), performs similar operations for the source of the fuse kernel module five times in sequence. As can be seen, the performance penalty for PCFS as compared to Fuse/Null is approximately 10% for OpenSSL, and 2.5% for Fuse. The difference arises because the OpenSSL benchmark depends more on file creation and deletion as compared to the Fuse benchmark.

Benchmark	PCFS (s)	Fuse/Null (s)	Ext3 (s)
OpenSSL	126	114	94
Fuse $\times$ 5	79	77	70

Summary: In summary, assuming a low rate of cache misses, the performance of PCFS on common file operations like read, write, stat, and open is comparable to that of Fuse/Null. On the other hand, less common operations like create and delete are slower because procaps must be managed. We believe that the efficiency attained even by our

prototype implementation is good enough for use in practice. A realistic file system based on the PCFS architecture would probably not use a process server architecture, which would improve its performance further.

VIII. Related Work

The use of certificates to represent policies expressed in logic as well as the use of logical inference to authorize access was initiated in the Taos operating system [39]. It is interesting that the authors of Taos suggest that caching of authorizations at several levels would be critical to high performance. PCFS procaps and their in-memory cache in the back end realize this suggestion and show that the suggestion works well in practice. The idea of removing proof search from the trusted computing base by requiring the reference monitor to check user-provided proofs originated in proof-carrying authorization (PCA) [7, 9, 10]. PCFS differs from PCA in that it off-lines proof verification as well, and uses conditional capabilities generated from it for authorizing access. This design is motivated by the need to have quicker access in PCFS than has been the requirement in previous deployments of PCA [9, 10].

Many logics and logic-based languages have been proposed in the past for representing access control policies (e.g., [4, 5, 11, 17, 22, 26, 36]). The k says s modality in BL is most closely related to a similar modality in Binder [17]. Although our treatment of explicit time draws on prior work by DeYoung and the authors [18], we believe that a logical combination of time and interpreted predicates is novel to BL. The treatment of interpreted predicates in BL is similar to that in the work of Dougherty et al [19] and that in the Nexus Authorization Logic (NAL; Schneider et al. [38]). Independent of our work, PCA based in NAL has been implemented in low-level interfaces, including a file system, in the Nexus Operating System. Although similar in spirit, the NAL implementation differs from ours in that it follows a traditional PCA architecture where proofs are checked directly by the reference monitor without mediating capabilities.

The STRONGMAN project [29] includes an implementation of a file system, and several other applications that rely on the KeyNote trust management system [14] for authorization. Many prior file systems have used capabilities to authorize access (e.g., [6, 25, 33, 37]), but the use of proofs to generate capabilities is novel to our work. The overall design of PCFS was inspired by an intriguing paper by Chaudhuri [15] that considers formal analysis of bisimulation-based correctness of implementations of authorization through cryptographic capabilities in the face of dynamic policies. That paper also considers many strategies for enforcing time-based and state dependent policies, but the mechanism used to represent policies is treated abstractly. In contrast, in Theorem V.1, we show our enforcement correct with respect to a concrete logic and proof system.

IX. Conclusion

PCFS combines strong logical foundations for access policies with an efficient enforcement based on proofs and cryptographic capabilities. Owing to a very expressive logic for policies and conditions in capabilities, PCFS can enforce time and state dependent policies rigorously and efficiently. A significant contribution of our work is Theorem V.1 which shows that enforcement of policies using procaps is identical to one with proofs directly (as in PCA). We believe that owing to its modular design and implementation, the PCFS architecture can be used in both centralized and decentralized settings without significant modification.

One interesting direction for future work is to consider access control with logical proofs in operation-intensive settings by deriving *all possible* permissions from the access policies periodically, thus freeing the user from the burden of generating proofs, having them verified, and maintaining capabilities. Another area for future work may be to build an optimized version of PCFS that does not rely on Fuse. Other avenues for future work may be to construct tools for authoring policies to complement the PCFS front end, and user studies to further validate usefulness of the framework.

Acknowledgments: This work was supported in part by the iCAST project sponsored by the National Science Council, Taiwan, under grant NSC97-2745-P-001-001, the Air Force Research Laboratory under grant FA87500720028, and CyLab at Carnegie Mellon under grant DAAD19-02-1-0389 from the Army Research Office. The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of ARO, AFRL, CMU, CyLab, NSF, or the U.S. Government or any of its agencies.

References

[1] "Bonnie++, a file system benchmark," available from http://www.coker.com.au/bonnie++/.

[2] "FUSE: Filesystem in Userspace," available from http://fuse.sourceforge.net/.

[3] "OpenSSL: The open source toolkit for SSL/TLS," See http://www.openssl.org.

[4] M. Abadi, "Access control in a core calculus of dependency," *Electronic Notes in Theoretical Computer Science*, vol. 172, pp. 5–31, April 2007, *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin*.

[5] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, "A calculus for access control in distributed systems," *ACM Transactions on Programming Languages and Systems*, vol. 15, no. 4, pp. 706–734, 1993.

[6] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath, "Block-level security for network-attached disks," in *Proceedings of the 2nd Conference on File and Storage Technologies (FAST)*, 2003, pp. 159–174.

[7] A. W. Appel and E. W. Felten, "Proof-carrying authentication," in *Proceedings of the 6th ACM Conference on Computer and Communications Security (CCS)*, 1999, pp. 52–62.

[8] K. Avijit, A. Datta, and R. Harper, "Distributed programming with distributed authorization," in *Proceedings of the Fifth ACM Workshop on Types in Language Design and Implementation (TLDI)*, 2009, pp. 27–38.

[9] L. Bauer, "Access control for the web via proof-carrying authorization," Ph.D. dissertation, Princeton University, November 2003.

[10] L. Bauer, S. Garriss, J. M. McCune, M. K. Reiter, J. Rouse, and P. Rutenbar, "Device-enabled authorization in the Grey system," in *Information Security: 8th International Conference (ISC'05)*, September 2005, pp. 431–445.

[11] M. Y. Becker, C. Fournet, and A. D. Gordon, "Design and semantics of a decentralized authorization language," in *20th IEEE Computer Security Foundations Symposium*, 2007, pp. 3–15.

[12] M. Y. Becker, J. F. Mackay, and B. Dillaway, "Abductive authorization credential gathering," in *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, July 2009, pp. 1–8.

[13] M. Y. Becker and P. Sewell, "Cassandra: Flexible trust management applied to health records," in *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW)*, 2004, pp. 139–154.

[14] M. Blaze, J. Fiegenbaum, and J. Ioannidis, "The Keynote trust-management system version 2," See http://www.ietf.org/rfc/rfc2704.txt, 1999.

[15] A. Chaudhuri, "On secure distributed implementations of dynamic access control," in *Proceedings of the Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis, and Issues in the Theory of Security (FCS-ARSPA-WITS)*, 2008, pp. 93–107.

[16] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest, "Certificate chain discovery in SPKI/SDSI," *Journal of Computer Security*, vol. 9, no. 4, pp. 285–322, 2001.

[17] J. DeTreville, "Binder, a logic-based security language," in *Proceedings of the IEEE 2002 Symposium on Security and Privacy (S&P'02)*, May 2002, pp. 105–113.

[18] H. DeYoung, D. Garg, and F. Pfenning, "An authorization logic with explicit time," in *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF-21)*, Jun. 2008, pp. 133–145, extended version available as Carnegie Mellon University Technical Report CMU-CS-07-166.

[19] D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Specifying and reasoning about dynamic access control policies," in *Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR)*, 2006, pp. 632–646.

[20] D. Garg, "Proof theory for authorization logic and its application to a practical file system," Ph.D. dissertation, Carnegie Mellon University, 2009, available as Technical Report CMU-CS-09-168.

[21] D. Garg and M. Abadi, "A modal deconstruction of access control logics," in *Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS 2008)*, April 2008, pp. 216–230.

[22] D. Garg and F. Pfenning, "Non-interference in constructive authorization logic," in *Proceedings of the 19th Computer Security Foundations Workshop (CSFW '06)*, July 2006, pp. 283–293.

[23] D. Garg, F. Pfenning, D. Serenyi, and B. Witten, "A logical representation of common rules for controlling access to classified information," Carnegie Mellon University, Tech. Rep. CMU-CS-09-139, 2009.

[24] G. Gentzen, "Untersuchungen über das logische Schließen," *Mathematische Zeitschrift*, vol. 39, pp. 176–210, 405–431, 1935, english translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

[25] H. Gobioff, G. Gibson, and D. Tygar, "Security for network attached storage devices," Carnegie Mellon University, Tech. Rep. CMU-CS-97-185, 1997.

[26] Y. Gurevich and I. Neeman, "DKAL: Distributed-knowledge authorization language," in *Proceedings of the 21st IEEE Symposium on Computer Security Foundations (CSF-21)*, 2008, pp. 149–162.

[27] R. Housley, W. Ford, W. Polk, and D. Solo, "Internet X.509 public key infrastructure," See http://www.ietf.org/rfc/rfc2459.txt, 1999.

[28] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic, "Aura: A programming language for authorization and audit," in *Proceedings of the International Conference on Functional Programming (ICFP)*, 2008, pp. 27–38.

[29] A. D. Keromytis, S. Ioannidis, M. B. Greenwald, and J. M. Smith, "The STRONGMAN architecture," in *Proceedings of the DARPA Information Survivability Conference and Exposition*, vol. 1, 2003, pp. 178–188.

[30] B. Lampson, M. Abadi, M. Burrows, and E. Wobber, "Authentication in distributed systems: Theory and practice," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 265–310, November 1992.

[31] N. Li and J. C. Mitchell, "Datalog with constraints: A foundation for trust management languages," in *PADL '03: Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, 2003, pp. 58–73.

[32] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, "Uniform proofs as a foundation for logic programming," *Annals of Pure and Applied Logic*, vol. 51, pp. 125–157, 1991.

[33] C. Olson and E. L. Miller, "Secure capabilities for a petabyte-scale object-based distributed file system," in *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, 2005, pp. 64–73.

[34] F. Pfenning and R. Davies, "A judgmental reconstruction of modal logic," *Mathematical Structures in Computer Science*, vol. 11, pp. 511–540, 2001.

[35] B. C. Pierce and D. N. Turner, "Local type inference," *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 1, pp. 1–44, 2000.

[36] A. Pimlott and O. Kiselyov, "Soutei, a logic-based trust-management system," in *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, 2006, pp. 130–145.

[37] B. C. Reed, E. G. Chron, R. C. Burns, and D. D. E. Long, "Authenticating network-attached storage," *IEEE Micro*, vol. 20, no. 1, pp. 49–57, 2000.

[38] F. B. Schneider, K. Walsh, and E. G. Sirer, "Nexus Authorization Logic (NAL): Design rationale and applications," Cornell University, Tech. Rep., Sep. 2009, online at http://ecommons.library.cornell.edu/handle/1813/13679.

[39] E. Wobber, M. Abadi, M. Burrows, and B. Lampson, "Authentication in the Taos operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 1, pp. 3–32, Feb. 1994.