PCAL: Language Support for Proof-Carrying Authorization Systems

Avik Chaudhuri[1] and Deepak Garg[2]

[1] University of Maryland, College Park
[2] Carnegie Mellon University

Abstract. By shifting the burden of proofs to the user, a proof-carrying authorization (PCA) system can automatically enforce complex access control policies. Unfortunately, managing those proofs can be a daunting task for the user. In this paper we develop a Bash-like language, PCAL, that can automate correct and efficient use of a PCA interface. Given a PCAL script, the PCAL compiler tries to statically construct the proofs required for executing the commands in the script, while re-using proofs to the extent possible and rewriting the script to construct the remaining proofs dynamically. We obtain a formal guarantee that if the policy does not change between compile time and run time, then the compiled script cannot fail due to access checks at run time.

1 Introduction

Proof-carrying authorization (PCA) [3, 5, 6, 19, 15, 17] is a modern access control technology, where an access control policy is formalized as a set of *logical formulas*, and a principal is allowed to perform an operation on a resource only if that principal can produce a *logical proof* showing that the policy *entails* that the principal may perform that operation on that resource. While this architecture allows automatic enforcement of complex access control policies, it substantially increases the burden of the user, since each request to perform an operation must be accompanied by one or more proofs. Furthermore, even if the user employs a theorem prover to construct the proofs, the user must still ensure that enough proofs are generated for each request to succeed, while minimizing the costs of proof construction at run time. In this paper we develop a programming language that can assist the user in performing such tasks correctly and automatically in a system with PCA. We have implemented a compiler for our language that works in conjunction with a PCA-based file system [15].

Our language, PCAL, extends the Bash scripting language with some PCA-specific annotations; the PCAL compiler translates programs with these annotations to ordinary Bash scripts, to be executed in a system with PCA. More precisely, PCAL annotations can specify what proofs the programmer expects to hold at particular program points. Based on these annotations, the compiler performs the following tasks.

1. It checks that the programmer's expectations about proofs suffice to allow successful execution of every shell command in the script. For this, the compiler needs to know what permissions are required to execute each shell command. We provide this information through a configuration file.

2. Next, the compiler uses a theorem prover and information about the access control policy to try to *statically* construct proofs corresponding to the programmer's annotations. In cases where static proof construction fails, because the annotations do not convey enough static information, the compiler generates code that constructs the proof at run time by calling the theorem prover from the command line.
3. Finally, the compiler adds code to pass appropriate proofs for each shell command to the PCA interface.

Thus, the output of the compiler is a Bash script which, beyond the usual commands, contains some code to generate proofs at run time (when it cannot generate such proofs at compile time), and some code to pass the proofs, generated either statically or dynamically, to the PCA interface.

Using PCAL offers at least two advantages over a naive approach, where a user generates and passes to the PCA interface enough proofs of access before running an unannotated script.

1. Because of the static checks and dynamic code generated by the compiler, it is guaranteed that the resulting script will *at least try to construct all necessary proofs of access*. Thus, the script can fail only if the user does not have enough privileges to run it, and not because the user forgot to create some proofs. Indeed, we prove a formal theorem which states that if compilation of a program succeeds and the policy does not change between compilation and program execution, then the program cannot fail due to an access check (Theorem 2). This is very significant for scripts where the user cannot determine a priori what operations the script will perform.
2. Since the compiler sees all commands that the script will execute, it re-uses proofs to the extent possible and reduces the proof construction overhead, which a naive user may not be able to do. This is particularly relevant for POSIX-like policies where accessing a file requires an "execute" permission on all its ancestor directories. If several files in a directory need to be processed, there is no need to construct proofs for the ancestor directories again and again. The PCAL compiler takes advantage of this and other similar structure in policies and combines it with information about a program's commands to minimize proof construction.

By design, PCAL and its compiler are largely independent of the logic used to express policies. The compiler requires a theorem prover compatible with the logic used, but it does not analyze formulas or proofs itself. As a result, the compiler can be (trivially) modified to use a different logic. Similarly, the compiler is parametric in the shell commands it supports. It assumes a map from each shell command to the permissions needed to execute it, and a single shell command to pass proofs to the PCA interface. By replacing this map and the command, the compiler can be used to support any PCA interface, not necessarily a file system.

PCAL is distinct from other work that combines PCA with a programming language [17, 4]. In all such prior work, the language is used to enforce access control *statically*. On the other hand, PCAL uses a *combination* of static checks and

dynamic code to ensure compliance with the requirements of the PCA interface. Static enforcement is a special case of this approach, where an input program is rejected unless the compiler can construct all required proofs at compile time. Furthermore, in all prior work proofs are data structures and programmers must write explicit code to construct them. In particular, programmers must understand the logic. In contrast, PCAL separates proofs from programs, and shifts the burden of constructing proofs (and understanding the logic) from programmers to an automatic theorem prover. We believe that this not only makes PCAL's design modular, but also easier to use.

Contributions We believe that we are the first to propose, design, and implement a language that uses a combination of static checks and dynamic code to optimize the proof burden of a PCA-compliant program. This setting presents some unique technical challenges, and our design and implementation require some novel elements to deal with those challenges.

1. While we would like to discharge as many proofs as possible statically, we must be concerned about possibly invalidating the assumptions underlying those proofs at run time. For instance, the state of the system may not remain invariant between compile time and run time. This requires a careful separation of (dynamic) state conditions from other (static) constraints.
2. Since the values of some program variables cannot be determined at compile time, the PCAL compiler constructs *quantified proofs* which are parametric over these program variables. These variables are substituted at run time to obtain ground proofs. (See Section 5 for details.)
3. Another notable feature of PCAL is that programmer annotations have both static and dynamic semantics. Statically, they *specify* authorization conditions and other constraints that should hold at run time, thereby aiding verification of correctness by the compiler. Dynamically, they *justify* any assumptions on the existence of authorization proofs and other constraints made by the compiler, thereby allowing sound optimizations.
4. For practical reasons, we must also be concerned about balancing the relative strengths and weaknesses of a theorem prover (to discharge proofs) and compiler (to analyze programs). We achieve such a balance by working at several levels of abstraction. While all functions and predicates used in a script have concrete implementations at run time, the compiler only partially interprets these functions and predicates with abstract rewrite rules, so that the program can be analyzed with appropriate precision by symbolic techniques. Furthermore, calls to the theorem prover are simplified, so that the theorem prover can treat all functions and predicates as uninterpreted, and thus can search for proofs efficiently. Tying these levels of abstraction together requires some care in the implementation. This is discussed further in Section 5.
5. We prove formally that the behavior of a compiled program is the same as that of the source program (Theorem 1). The proof of this theorem requires a precise characterization of assumptions made on the theorem prover, the

proof verifier, and the relation between the environment in which the program is compiled and that in which it is executed. We believe that this characterization is a significant contribution of this work, because it is fundamental to any architecture that uses a similar approach.

Organization The rest of the paper is organized as follows. After closing this section with a brief review of related work, in Section 2 we discuss some background material covering PCA, and the assumptions we make about the interface it provides. Section 3 introduces PCAL and its compiler through an example. Formal details of the language, its compilation, and correctness theorems are covered in Section 4. Some important implementation-related issues are discussed in Section 5.

Related Work There are two prior lines of work on combining proofs of authorization with languages. The first line of work includes the languages Aura [17, 21] and PCML_5 [4], where PCA as well as a logic for expressing policies are embedded in the type system, and proofs are data structures that programs can analyze. This contrasts with PCAL, where proofs cannot be analyzed. PCAL's approach is advantageous because it decouples the logic from the language, thus making it easy to use the same compiler with different logics. It also alleviates the programmer's burden of understanding the logic. On the other hand, in Aura and PCML_5, parts of proofs can be re-used in different places, thus allowing potentially more efficient proof construction than in PCAL. However, it is unclear whether this advantage extends when automatic theorem provers are used in either Aura or PCML_5.

The second line of work includes several languages that culminate in the most recent F7 [11, 8]. These languages use an external logic like PCAL, but the objective is to express logical invariants. The programmer can introduce logical assumptions at different program points, and check statically at other program points that those assumptions entail some other formula(s). However, in PCAL, it is not necessary that each programmer annotation about a proof succeed statically; if it fails, code to construct the proof at run time is automatically inserted. This approach is similar to hybrid typechecking [10]. Indeed, a significant philosophical difference between PCAL and previous lines of work is that PCAL does not try to enforce security on its own; instead it is meant as a tool to help programs comply with a PCA interface that enforces security.

PCA, the architecture that PCAL supports, was introduced by Appel and Felten [3]. It has been applied in different settings including authorization for web services [5], the Grey system [6], and a file system [15]. The latter implementation is the basic test bench for PCAL. The specific logic used for writing policies in this paper is BL [12, 13]. It is related to, but more expressive than, many other logics and languages for writing access policies (*e.g.*, [1, 2, 14, 7, 16, 20, 9]).

2 Background

In this section we provide a brief overview of PCA, and list particular assumptions that PCAL makes about the underlying PCA-based system interface.

PCA [3, 5, 6, 15, 17, 19] is a general architecture for enforcing access control in settings that require complex, rule-based policies. Policy rules are expressed as formulas in some fixed logic, and enforced automatically using *formal proofs*. Let $\mathcal{L}$ denote a set of formulas that represent the access policy (see Section 3 for an example). The system interface grants user A permission η (*e.g.*, read, write, etc) on a resource t (*e.g.*, a file) only if A produces a formal proof γ which shows that $\mathcal{L}$ entails a formula $\mathbf{auth}(A, \eta, t)$ in the logic's proof system, or in formal notation $\gamma :: \mathcal{L} \vdash \mathbf{auth}(A, \eta, t)$. The formula $\mathbf{auth}(A, \eta, t)$ means that A has permission η on resource t. Its exact form depends on the logic in use and the resources being protected, but is irrelevant for the purposes of this paper. (Here it suffices to assume that $\mathbf{auth}(A, \eta, t)$ is an atomic formula.) The system interface checks the proof that A provides to make sure that it uses the logic's inference rules correctly, and that it proves the intended formula. To this end, it is assumed that there is a fast procedure to verify proofs.

Although users are free to construct proofs by any means, it is convenient to have an automatic theorem prover to perform this task. Further, the system interface must provide a mechanism by which users can submit proofs either prior to or along with an access request. The goal of PCAL is to help programmers create programs (Bash scripts) that generate required proofs while minimizing performance overhead, and correctly pass these proofs to the system interface.

Assumptions PCAL's compiler supports rich logics for writing policies, in which proofs may depend not only on the formulas constituting the policy (denoted $\mathcal{L}$), but also on system state (*e.g.*, meta data of files and clock time). The latter is abstractly denoted by the letter H, and we write $\gamma :: H; \mathcal{L} \vdash s$ to mean that γ is a formal proof which shows that in the system state H, policies $\mathcal{L}$ entail formula s (*e.g.*, s may be $\mathbf{auth}(A, \eta, t)$).

PCAL also requires that an automatic theorem prover for the logic be available, both through an API and as a command line tool. The former is used by the compiler to construct as many of the required proofs as possible at compile time, while the latter is used by the output script to construct the remaining proofs at run time. A call to the theorem prover (either through the API or the command line) is formally summarized by the notation $H; \mathcal{L} \vdash s \searrow \gamma$, which means that asking the theorem prover to construct a proof for s from policy $\mathcal{L}$ in state H results in the proof γ. Dually, $H; \mathcal{L} \nvdash s \searrow$ means that the theorem prover fails to construct a corresponding proof. The latter *does not* imply the absence of a proof in the logic, since the theorem prover may implement an incomplete search procedure. The following command is assumed to invoke the prover from the command line and store in the file `.pf` a proof which establishes $\mathbf{auth}(A, \eta, t)$ from the policies in `/.pl` and the prevailing system state.

$$\texttt{prove}\ \mathbf{auth}(A, \eta, t)\ \ \texttt{/.pl > .pf}$$

For passing proofs to the system interface, we assume a simple protocol: a command `inject` is called from the command line to give a proof to the system interface, which puts it in a store that is indexed by the triple (A, η, t) authorized by the proof. During the invocation of a system API, relevant proofs

are retrieved from this store and checked. For example, the following command injects the proof in the file `.pf` into the interface's store.

```
inject .pf
```

While proofs required by the file system to execute commands at run time must be ground, proofs produced at compile time may contain free variables, which we assume are listed in order, and which need to be instantiated at run time. For such proofs, the run-time substitutions of such variables are also provided to the `inject` command (with option `-subst`), so that the injected proofs are always ground. For example, the following command substitutes the run-time values of some Bash variables (in order) for the free variables in the proof read from file `.pf` and stores the resulting proof in the system interface.

```
inject .pf -subst $_PRIN $z $x $y $bar $foo
```

3 Overview of PCAL

In this section, we run through a small example to demonstrate the steps of our compilation. (We formalize PCAL in Section 4.) For this example, let there be a predicate `extension` and functions `path` and `base`, such that (informally):

- `extension`(f, e) holds if file f has extension e;
- `path`$(d, x) = p$ if path p is the concatenation of directory d and name x;
- `base`$(p) = x$ if `path`$(d, x) = p$ for some directory d.

Consider the program P in Figure 1, written in PCAL. This program iterates through the files in some directory `foo` (unspecified), copying them to a directory `bar` (set to `"/tmp"`). Furthermore, it touches those files in `foo` that have extension `"log"`. The reader may ignore the `assert` statements (in lines 2, 8, 12, and 13) in a first reading; we explain their meaning below.

The system is configured to check, for any command, that certain permissions are held on certain paths in order to execute that command. Let us assume the following configuration:

Configuration

- Iterating over directory d requires permission `read` on d.
- Executing the shell command `touch`(f) requires permission `write` on file f.
- Executing the shell command `cp`(f_1, f_2) requires permission `read` on file f_1, and permission `write` on file f_2.

The `assert` statements in P serve to establish, at run time, that the principal running the script has particular permissions on particular paths. The compiler tries to statically identify `assert` statements that must succeed at run time, and eliminate them at compile time.

Assume that `member` is a predicate such that `member`(f, d) holds if file f is in directory d. Consider the following policy, written in a first-order logic.

Program P

```
1  bar = "/tmp";
2  assert (read, foo);
3  for x in foo {
4      y = x;
5      x = base(x);
6      z = path(foo, x);
7      test extension(z, "log") {
8           assert (write, z);
9           shell touch(z)
10     };
11     z = path(bar, x);
12     assert (write, z);
13     assert (read, y);
14     shell cp(y, z)
15 }
```

Program Q

```
1  bar = "/tmp";
2  assert (read, foo);
3  for x in foo {
4      y = x;
5      x = base(x);
6      z = path(foo, x);
7      test extension(z, "log") {
8           -- assert (write, z);
9           shell touch(z)
10     };
11     z = path(bar, x);
12     -- assert (write, z);
13     -- assert (read, y);
14     shell cp(y, z)
15 }
```

Script $\mathcal{S}$

```
!/bin/bash
function base { _RET=${1##*/} }
function path { _RET=$1/$2 }
function extension { if [ ${1##*.} = $2 ]; then _RET="ok"; fi }
_PRIN="User"
foo="/home"

1  bar="/tmp"
2  prove auth ($_PRIN, read, $foo) /.pl > .pf
   inject .pf
3  for x in `ls $foo`; do x=$foo/$x
4    y=$x
5    _RET="_"; base $x; x=$_RET
6    _RET="_"; path $foo $x; z=$_RET
7    _RET="_"; extension $z "log"; if [ $_RET = "ok" ]; then
8      inject /.pf/1 -subst $_PRIN $z $x $y $bar $foo
9      touch $z
10   fi
11   _RET="_"; path $bar $x; z=$_RET
12   inject /.pf/2 -subst $_PRIN $z $x $y $bar $foo
13   inject /.pf/3 -subst $_PRIN $z $x $y $bar $foo
14   cp $y $z
15 done
```

Fig. 1. Translation of an input program P, via an intermediate program Q, to an output script $\mathcal{S}$. (The configuration, policy, and rewrite theory provided to the compiler are shown elsewhere.)

Policy

$$\mathbf{auth}(\texttt{"User"}, \texttt{read}, \texttt{"/home"}).$$
$$\forall A.\forall x.\ \mathbf{auth}(A, \texttt{write}, \texttt{path("/tmp", x)}).$$
$$\forall A.\forall x.\forall y.\ \texttt{member}(x, y) \Rightarrow \mathbf{auth}(A, \texttt{read}, y) \Rightarrow \mathbf{auth}(A, \texttt{read}, x) \wedge \texttt{extension(}x\texttt{, "log")} \Rightarrow \mathbf{auth}(A, \texttt{write}, x).$$

Informally, the policy asserts the following:

- the principal `"User"` has permission `read` on directory `"/home"`
- any principal A has permission `write` on any file in the directory `"/tmp"`
- for any principal A, file x, and directory y, if x is in y and A has permission `read` on y, then A has permission `read` on x, and furthermore, if x has extension `"log"` then A has permission `write` on x.

Finally, consider the following theory on the function symbols `path` and `base`, that abstracts the concrete semantics of these functions.

Theory

$$\forall x.\forall y.\ \texttt{member}(x, y) \Rightarrow \texttt{path(}y\texttt{, base(}x\texttt{))} = x$$

Given the configuration, policy, and theory above, our compiler automatically translates P to the intermediate program Q in Figure 1. In Q, all `assert` statements except that in line 2 are eliminated, since the compiler can infer that they must succeed at run time. Such inference requires collection of path conditions, partial evaluation of terms modulo the given equational theory, and calls to the theorem prover. We describe the compiler in detail in Sections 4 and 5.

In particular, for the `assert` statement in line 8, the compiler reasons automatically as follows. Let `_PRIN` be the principal running the script. Line 8 is reached only if the following conditions hold for some `z`, `x`, x', and `foo`:

(1) `extension(z, "log")`.
(2) `z` = `path(foo, x)`.
(3) `x` = `base(`x'`)`.
(4) `member(`x'`, foo)`.
(5) The statement `assert (read, foo)` in line 2 succeeds.

From condition (5), we can conclude that

(6) $\mathbf{auth}(\texttt{\_PRIN}, \texttt{read}, \texttt{foo})$.

Simplifying conditions (2) and (3) using the given theory, we have

(7) `z` = x'.

Now from conditions (1), (4), (6), and (7) and the given policy, the theorem prover can conclude that $\mathbf{auth}(\texttt{\_PRIN}, \texttt{write}, z)$, which is sufficient to eliminate the `assert` statement in line 8.

Next, we want to be able to run the intermediate program Q on a file system that supports PCA. The compiler translates Q to the equivalent Bash script $\mathcal{S}$ in Figure 1. The commands `prove` and `inject` perform functions described in

Section 2. The header (the part of $\mathcal{S}$ before the numbered lines) defines all free variables (`_PRIN` and `foo`) and uninterpreted functions and predicates (`path`, `base`, `extension`) in P. The implementations of such functions and predicates are sound with respect to the equational theory used by the compiler.

We close this section by discussing our trust assumptions. A policy is trusted, so any interpreted predicates in a policy (such as `member` and `extension`) must have trusted implementations (provided by the system). In contrast, a program is not trusted. The compiler may or may not be trusted. If the compiler is trusted, then the system can trust scripts produced by the compiler, and run such scripts without checking the proofs that they inject. This is significant in implementations where proofs may be large and proof verification may be costly. However, such a compiler cannot assume semantic properties of the functions used in a program (such as `base` and `path`) unless those functions have trusted implementations that are provided by the system. On the other hand, if the compiler is not trusted then the system must run all scripts with access checks. We implicitly assume the latter scenario in the sequel, and provide additional guarantees for the scenario in which the compiler is trusted (Theorem 2).

4 PCAL: Syntax, Semantics, and Compilation

We now formalize the PCAL language and its compiler. We present the syntax of PCAL programs, define their operational semantics, and finally define our compilation procedure and show that it preserves the behavior of programs.

For simplicity of presentation, we abstract various details of the implementation. (See Section 5 for a more detailed discussion.) Instead of Bash, we consider an extension of PCAL as the target language for compilation; programs in this target language can be easily rewritten to Bash. We also treat all function symbols as uninterpreted, although in principle, equations over terms may be freely added in the run time semantics (to model concrete implementations) and in the compiler (to model abstract properties of such implementations).

We assume that η, x, and t range over permissions, variables, and terms whose grammars are borrowed from the logic used to represent policies. φ denotes a logical predicate whose truth depends only on the system state (i.e., a predicate that is not defined by logical rules). PCAL programs are sequences of statements e described by the grammar below. Directories, files, and paths are represented as terms, and χ is a special variable that denotes the principal running a program.

Syntax

$e ::=$	statements
$\mathsf{for}\ x\ \mathsf{in}\ t\ \{P\}$	for each file f in directory t, bind x to f and do P
$\mathsf{test}\ \varphi\ \{P\}$	if condition φ holds, do P
$x = t$	assign t to x
$\mathsf{shell}\ n(t_1, \ldots, t_k)$	call shell command n with parameters $t_1, \ldots, t_k$
$\mathsf{assert}\ (\eta, t)$	assert that principal χ has permission η on path t
$P, Q ::=$	programs

$e; Q$	run e, then do Q
end	skip/halt

We also consider below an extension of PCAL which acts as the target language for the compiler. $\alpha = \mathsf{prove}\ (\eta, t)$ and $\mathsf{inject}\ (\eta, t)\ \gamma$ are formal representations of the commands `prove` and `inject` from Section 2. γ ranges over proofs and α denotes a variable bound to a proof (which, in the actual implementation, is a temporary file that stores the proof; see Section 5).

Extended syntax

$e ::=$	statements
...	
$\alpha = \mathsf{prove}\ (\eta, t)$	prove that principal χ has permission η on path t and bind the proof to α
$\mathsf{inject}\ (\eta, t)\ \gamma$	inject proof γ that authorizes (χ, η, t)

Semantics A PCAL program runs in an environment θ of the form $(\Delta, \mathcal{L})$, where Δ is a function from shell command names to lists of permissions (configuration) and $\mathcal{L}$ is the set of logical formulas used to determine access (policy). Informally, if $\Delta(n) = \eta_1, \ldots, \eta_k$ then executing shell command $n(t_1, \ldots, t_k)$ requires permissions $\eta_1, \ldots, \eta_k$ on paths $t_1, \ldots, t_k$ respectively.

A state ρ is a triple (H, S, ξ), where H an abstract, logical representation of the part of the system state on which proofs of access depend, S is a function from paths to terms (data store), and ξ is a partial function from triples (A, η, t) to proofs (proof store). H must contain, at the least, information about members of directories. We write $\texttt{members}(H, t)$ to denote the list of files in directory t in the system state H. Proofs injected using $\mathsf{inject}\ (\eta, t)\ \gamma$ are added to ξ.

Reductions are of the form $\rho, P \xrightarrow{\theta, \chi} \rho', P'$, meaning that program P at state ρ, run by principal χ in environment θ, reduces to program P' at state ρ'. Reduction rules rely on the external judgment $H, S \overset{n(t_1, \ldots, t_k)}{\blacktriangleright} H', S'$, which means that executing the shell command $n(t_1, \ldots, t_k)$ updates the system state H and data store S to H' and S' respectively. $H \models \varphi$ means that φ holds in H, and $H \not\models \varphi$ means that φ does not hold in H. In practice, whether φ holds in H or not is decided using a trusted decision procedure provided by the system.

Reduction $\rho, P \xrightarrow{\theta, \chi} \rho', P'$

(Reduct for)

$$\frac{\rho = (H, \_, \_) \qquad \texttt{members}(H, t) = t_1, \ldots, t_k}{\rho, \mathsf{for}\ x\ \mathsf{in}\ t\ \{P\}; Q \xrightarrow{\theta, \chi} \rho, P\{t_1/x\}; \ldots; P\{t_k/x\}; Q}$$

(Reduct test)

$$\frac{\rho = (H, \_, \_) \qquad H \models \varphi}{\rho, \mathsf{test}\ \varphi\ \{P\}; Q \xrightarrow{\theta, \chi} \rho, P; Q} \qquad \frac{\rho = (H, \_, \_) \qquad H \not\models \varphi}{\rho, \mathsf{test}\ \varphi\ \{P\}; Q \xrightarrow{\theta, \chi} \rho, Q}$$

(Reduct assign)

$$\rho, x = t; Q \xrightarrow{\theta, \chi} \rho, Q\{t/x\}$$

(Reduct shell)
$$\frac{\theta = (\Delta, \mathcal{L}) \qquad \Delta(n) = \eta_1, \ldots, \eta_k \qquad \rho = (H, S, \xi) \qquad \xi(\chi, \eta_i, t_i) = \gamma_i \qquad \gamma_i :: H; \mathcal{L} \vdash \mathbf{auth}(\chi, \eta_i, t_i) \qquad H, S \overset{n(t_1,\ldots,t_k)}{\blacktriangleright} H', S' \qquad \rho' = (H', S', \xi)}{\rho, \mathsf{shell}\ n(t_1, \ldots, t_k); P \xrightarrow{\theta,\chi} \rho', P}$$

(Reduct assert)
$$\frac{\theta = (\_, \mathcal{L}) \qquad \rho = (H, S, \xi) \qquad H; \mathcal{L} \vdash \mathbf{auth}(\chi, \eta, t) \searrow \gamma \qquad \rho' = (H, S, \xi[(\chi, \eta, t) \mapsto \gamma])}{\rho, \mathsf{assert}\ (\eta, t); P \xrightarrow{\theta,\chi} \rho', P}$$

(Reduct prove)
$$\frac{\theta = (\_, \mathcal{L}) \qquad \rho = (H, \_, \_) \qquad H; \mathcal{L} \vdash \mathbf{auth}(\chi, \eta, t) \searrow \gamma}{\rho, \alpha = \mathsf{prove}\ (\eta, t); P \xrightarrow{\theta,\chi} \rho, P\{\gamma/\alpha\}}$$

(Reduct inject)
$$\frac{\rho = (H, S, \xi) \qquad \rho' = (H, S, \xi[(\chi, \eta, t) \mapsto \gamma])}{\rho, \mathsf{inject}\ (\eta, t)\ \gamma; P \xrightarrow{\theta,\chi} \rho', P}$$

- **(Reduct for)** unrolls a loop P for each file x in a directory t. **(Reduct test)** simplifies $\mathsf{test}\ \varphi\ \{P\}; Q$ to $P; Q$ if $H \models \varphi$, and to Q otherwise. **(Reduct assign)** is straightforward.
- **(Reduct shell)** finds proofs $\gamma_1, \ldots, \gamma_n$ needed to authorize the shell command $n(t_1, \ldots, t_k)$ in the proof store ξ. It then checks these proofs (premise $\gamma_i :: H; \mathcal{L} \vdash \mathbf{auth}(\chi, \eta_i, t_i)$), and executes the shell command (premise $H, S \overset{n(t_1,\ldots,t_k)}{\blacktriangleright} H', S'$).
- **(Reduct assert)** calls the theorem prover to construct a proof γ which shows that χ has permission η on path t (premise $H; \mathcal{L} \vdash \mathbf{auth}(\chi, \eta, t) \searrow \gamma$), and gives it to the system interface by putting it in the store ξ.
- **(Reduct prove)** constructs a proof γ and binds α to it. **(Reduct inject)** places a proof γ in the proof store ξ. By these rules, the effect of the command sequence $\alpha = \mathsf{prove}\ (\eta, t); \mathsf{inject}\ (\eta, t)\ \alpha$ is exactly the same as the command $\mathsf{assert}\ (\eta, t)$. However, $\mathsf{assert}\ (\eta, t)$ occurs only in source programs whereas $\mathsf{prove}\ (\eta, t)$ and $\mathsf{inject}\ (\eta, t)\ \gamma$ occur only in compiled programs.

Compilation Next, we formalize compilation of PCAL programs. As the compiler traverses a program, it maintains a database of facts that must be true at the program point that the compiler is looking at. These facts are formally represented by $\Gamma = (\sigma, M, \Phi, \Xi)$.

- σ is a list of substitutions of the form $\{t/x\}$. The latter means that program variable x is bound to term t.
- M is a list of predicates of the form $\mathtt{member}(x, t)$, meaning that x is in directory t (x is a variable representing a file system object).
- Φ is a list of interpreted predicates φ that can be assumed to hold at a program point. These are gathered from commands $\mathsf{test}\ \varphi\ \{\ldots\}$.
- Ξ is a set of triples (A, η, t) for which the compiler has constructed authorization proofs, or those for which it is certain proofs will exist at run time.

Below, we show compilation judgments of the form $\Gamma \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'$, meaning that under assumptions Γ, program P compiles to program P' in environment θ and system state H. χ is given to the compiler at the time of invocation; it represents the user who is expected to run the compiled program. H is the state of the system in which the compiled program is expected to run. It may either be the system state at the time of compilation (if it is expected that the compiled program will run in the same state), or it may be a state that the user provides. Both χ and H are needed to call the theorem prover during compilation.

For any syntactic entity $\mathbb{E}$, we write $\mathbb{E}\sigma$ to denote the result of applying the substitution σ to $\mathbb{E}$. $\mathcal{W}(P)$ denotes the variables that are assigned in the program P, and $\sigma\backslash\overline{x}$ denotes the restriction of σ that removes the mappings for all variables in $\overline{x}$. Finally, $|\Xi|$ denotes a logical representation of Ξ: $|\Xi| = \{\mathbf{auth}(A,\eta,t) \mid (A,\eta,t) \in \Xi\}$.

Compilation $\Gamma \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'$

(Comp end)
$$\Gamma \vdash \mathsf{end}; P \overset{H,\theta,\chi}{\rightsquigarrow} P \qquad \Gamma \vdash \mathsf{end} \overset{H,\theta,\chi}{\rightsquigarrow} \mathsf{end}$$

(Comp for)
$$\frac{\begin{array}{c}\Gamma = (\sigma, M, \Phi, \Xi) \qquad x \text{ fresh in } \Gamma \qquad \overline{x} = \mathcal{W}(P) \\ \sigma' = \sigma\backslash\overline{x} \qquad M' = M, \mathtt{member}(x,t) \\ (\sigma', M', \Phi, \Xi) \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P' \qquad (\sigma', M, \Phi, \Xi) \vdash Q \overset{H,\theta,\chi}{\rightsquigarrow} Q'\end{array}}{\Gamma \vdash \mathsf{for}\ x\ \mathsf{in}\ t\ \{P\}; Q \overset{H,\theta,\chi}{\rightsquigarrow} \mathsf{for}\ x\ \mathsf{in}\ t\ \{P'\}; Q'}$$

(Comp test)
$$\frac{\begin{array}{c}\Gamma = (\sigma, M, \Phi, \Xi) \qquad \overline{x} = \mathcal{W}(P) \qquad \sigma' = \sigma\backslash\overline{x} \qquad \Phi' = \Phi, \varphi \\ (\sigma', M, \Phi', \Xi) \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P' \qquad (\sigma', M, \Phi, \Xi) \vdash Q \overset{H,\theta,\chi}{\rightsquigarrow} Q'\end{array}}{\Gamma \vdash \mathsf{test}\ \varphi\ \{P\}; Q \overset{H,\theta,\chi}{\rightsquigarrow} \mathsf{test}\ \varphi\ \{P'\}; Q'}$$

(Comp assign)
$$\frac{\Gamma = (\sigma, M, \Phi, \Xi) \qquad \sigma' = \sigma, \{t/x\} \qquad (\sigma', M, \Phi, \Xi) \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'}{\Gamma \vdash x = t; P \overset{H,\theta,\chi}{\rightsquigarrow} x = t; P'}$$

(Comp shell)
$$\frac{\begin{array}{c}\theta = (\Delta, \_) \qquad \Delta(n) = \eta_1, \ldots, \eta_k \qquad \Gamma = (\sigma, \_, \_, \Xi) \\ (\chi, \eta_i\sigma, t_i\sigma) \in \Xi \text{ for each } i \qquad \Gamma \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'\end{array}}{\Gamma \vdash \mathsf{shell}\ n(t_1, \ldots, t_k); P \overset{H,\theta,\chi}{\rightsquigarrow} \mathsf{shell}\ n(t_1, \ldots, t_k); P'}$$

(Comp assert static)
$$\frac{\begin{array}{c}\Gamma = (\sigma, M, \Phi, \Xi) \qquad \theta = (\_, \mathcal{L}) \\ H, \Phi\sigma; \mathcal{L}, M\sigma, |\Xi| \vdash \mathbf{auth}(\chi, \eta\sigma, t\sigma) \searrow \gamma \\ \Xi' = \Xi, (\chi, \eta\sigma, t\sigma) \qquad \Gamma' = (\sigma, M, \Phi, \Xi') \qquad \Gamma' \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'\end{array}}{\Gamma \vdash \mathsf{assert}\ (\eta, t); P \overset{H,\theta,\chi}{\rightsquigarrow} \mathsf{inject}\ (\eta, t)\ \gamma; P'}$$

(Comp assert dynamic)
$$\frac{\begin{array}{c}\Gamma = (\sigma, M, \Phi, \Xi) \qquad \theta = (\_, \mathcal{L}) \\ H, \Phi\sigma; \mathcal{L}, M\sigma, |\Xi| \not\vdash \mathbf{auth}(\chi, \eta\sigma, t\sigma) \searrow \\ \Xi' = \Xi, (\chi, \eta\sigma, t\sigma) \qquad \Gamma' = (\sigma, M, \Phi, \Xi') \qquad \Gamma' \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'\end{array}}{\Gamma \vdash \mathsf{assert}\ (\eta, t); P \overset{H,\theta,\chi}{\rightsquigarrow} \alpha = \mathsf{prove}\ (\eta, t); \mathsf{inject}\ (\eta, t)\ \alpha; P'}$$

- **(Comp end)** terminates compilation when end is encountered.
- **(Comp for)** compiles for x in t $\{P\}; Q$ by compiling P to P' under the added assumption $\texttt{member}(x, t)$ (which must hold inside the body of the loop), and compiling Q to Q'. In each case, any prior substitutions for variables $\overline{x}$ assigned in P are removed from σ, because they may be invalidated during the execution of the loop (premises $\overline{x} = \mathcal{W}(P)$ and $\sigma' = \sigma \backslash \overline{x}$).
- **(Comp test)** is similar to **(Comp for)**, except that in this case the assumption φ is added when compiling the body of the branch P.
- **(Comp assign)** records the effect of assignment $x = t$ by augmenting substitution σ with $\{t/x\}$. This augmented substitution is used to compile the remaining program.
- **(Comp shell)** checks that there exists a proof to authorize each permission needed to execute a shell command $n(t_1, \ldots, t_k)$. For this it looks up the set of previously constructed proofs Ξ. (Proofs are added to this set in the next two rules).
- **(Comp assert static)** and **(Comp assert dynamic)** are used to compile the command assert (η, t) in different cases. To decide which rule to use, the compiler tries to statically prove $\mathbf{auth}(\chi, \eta\sigma, t\sigma)$ by calling the theorem prover (the application of σ to η and t propagates known constraints on equality to the proposition being proved; this increases the chances of finding a proof). The context in which the proof is constructed not only contains H and the policy $\mathcal{L}$, but also information about directory memberships ($M\sigma$), predicates tested in outer scopes ($\Phi\sigma$), and previously constructed proofs $|\Xi|$. If proof construction succeeds, **(Comp assert static)** is used: assert (η, t) is replaced by inject (η, t) γ, which gives the (statically) generated proof γ to the system interface at run time. Also, the fact that the new proof exists is recorded by modifying Ξ to $\Xi' = \Xi, (\chi, \eta\sigma, t\sigma)$, and using Ξ' to compile the remaining program P. If the proof construction fails, rule **(Comp assert dynamic)** is used: the compiler generates code both to construct the proof at run time and to inject it into the system interface. Accordingly, assert (η, t) is compiled to $\alpha =$ prove (η, t); inject (η, t) α. Even in this case, it is safe to assume that a proof of $\mathbf{auth}(\chi, \eta\sigma, t\sigma)$ will exist when P executes (else $\alpha =$ prove (η, t) will block at run time), so Ξ is changed as before.

Formal Guarantees We close this section by stating the formal guarantees of compilation. Proof sketches appear in the appendix.

We begin by defining an ordering $\leq$ on system states. Roughly, $H \leq H'$ if any formula that can be proved under H can also be proved under H'.

Definition 1 ($\leq$). *For any H and H', let $H \leq H'$ if for all $\mathcal{L}$, s, and γ, if $H; \mathcal{L} \vdash s \searrow \gamma$ then $H'; \mathcal{L} \vdash s \searrow \gamma'$.*

Next, we assume the following axioms for the various external judgments. Roughly, Axiom (1) states that system states are updated monotonically by shell-command executions. Axioms (2) and (3) are assumptions on the theorem prover: proof construction must be closed under substitution and cut. Finally,

Axiom (4) states that any proof produced by the theorem prover can be verified (i.e., the theorem prover is bug-free).

Axioms

(1) if $H, S \overset{n(t_1\sigma,\ldots,t_k\sigma)}{\blacktriangleright} H', S'$ then $H \leq H'$
(2) if $H; \mathcal{L} \vdash s \searrow \gamma$ then $H\sigma; \mathcal{L} \vdash s\sigma \searrow \gamma'$
(3) if $H; \mathcal{L} \vdash s \searrow \gamma$ and $H; \mathcal{L}, s \vdash s' \searrow \gamma'$ then $H; \mathcal{L} \vdash s' \searrow \gamma''$
(4) if $H; \mathcal{L} \vdash s \searrow \gamma$ then $\gamma :: H; \mathcal{L} \vdash s$

We can now show that compilation preserves the behavior of programs. More precisely, if a program P compiles to a program P' under a system state H, and the programs are run from a system state H' such that $H \leq H'$, then P and P' evaluate to the same data stores.

Theorem 1 (Compilation correctness). *Suppose that Axioms (1–4) hold, and* $(\varnothing, \varnothing, \varnothing, \varnothing) \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'$. *Then for all* A *and* $\rho = (H', \_, \_)$ *such that* $H \leq H'$, *we have* $\rho, P \xrightarrow{\theta,A}{}^{\star} \rho', Q$ *for some* Q *if and only if* $\rho, P' \xrightarrow{\theta,A}{}^{\star} \rho', Q'$ *for some* Q'.

Finally, we show that a compiled program can never fail due to an access check, if the policy does not change between compile time and run time. Formally, compilation preserves the behavior of programs even if the compiled programs are run without access checks.

Definition 2 ($\Longrightarrow$). *Let* $\Longrightarrow$ *be the same reduction relation as* $\longrightarrow$ *except that the rule* **(Reduct shell)** *is replaced by the following rule.*

$$\frac{\theta = (\Delta, \mathcal{L}) \qquad \Delta(n) = \eta_1, \ldots, \eta_k \qquad \rho = (H, S, \xi) \qquad H, S \overset{n(t_1,\ldots,t_k)}{\blacktriangleright} H', S' \qquad \rho' = (H', S', \xi)}{\rho, \mathsf{shell}\ n(t_1, \ldots, t_k); P \xrightarrow{\theta,\chi} \rho', P}$$

Theorem 2 (Access control redundancy). *Suppose that Axioms (1–4) hold, and* $(\varnothing, \varnothing, \varnothing, \varnothing) \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'$. *Then for all* A *and* $\rho = (H', \_, \_)$ *such that* $H \leq H'$, *we have* $\rho, P \xrightarrow{\theta,A}{}^{\star} \rho', Q$ *for some* Q *if and only if* $\rho, P' \overset{\theta,A}{\Longrightarrow}{}^{\star} \rho', Q'$ *for some* Q'.

5 Implementation

We have implemented the PCAL compiler to work in conjunction with PCFS [15]. We now discuss some implementation details that are left abstract in Section 4.

Rewrite rules A set of rewrite rules over terms, modeling abstract properties of the concrete implementations of function symbols, can be provided to the compiler to improve its precision. The compiler constructs a normalization function based on these rules, and applies this function eagerly to substitutions. This works well even in cases where it is not possible to interpret function symbols with directed clauses in the policy. (Modeling equations as clauses usually causes proof searches to loop.)

Quantified proofs Statically generated proofs may contain free variables, and as such they are *parametric* over those variables. In the formal semantics, such proofs are bound and carried as values in the language (in inject statements), so they get implicitly instantiated before injection at run time. However in our actual implementation, such proofs are output to temporary files with distinct names (under `/.pf`), and the names are carried in the language; so the free variables in such proofs must be explicitly substituted at run time. This explains why we considered an explicit `-subst` option to the `inject` command in Section 2.

We have tested our implementation on the file system PCFS [15], using policies written in the authorization logic BL [13, 12]. The interested reader can find one such example (involving homework management between instructors and students of various courses) in the appendix.

6 Conclusion

PCAL combines static checks and dynamic theorem proving to automate correct and efficient use of a PCA-based interface. PCAL's compiler is modular: it is parametric over both the shell commands (system interface) and the logic it supports. Although this makes the compiler flexible, the interaction between the core language, shell commands, and the logic is subtle and requires careful design. The compiler is made practical through a combination of simple user annotations, static constraint tracking, dynamically checked assertions, and run time support from a command line theorem prover. We prove formally that these ideas work well together. It is our belief that PCAL's design is novel, and that it will be a useful stepping stone for languages that support rule-based access control interfaces in future.

There are several interesting avenues for future work. An obvious one is to run realistic examples on PCAL, to determine what other features are needed in practice. Another possible direction is a code execution architecture where a trusted PCAL compiler is used to generate certified scripts that are run with minimal access control checks. Finally, it will be interesting to apply ideas from PCAL, particularly the use of an automatic theorem prover, in the context of language-based security for access control interfaces (*e.g.*, [17, 4]).

References

1. Martín Abadi. Access control in a core calculus of dependency. *Electronic Notes in Theoretical Computer Science*, 172:5–31, apr 2007. *Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin.*

2. Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.
3. Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In G. Tsudik, editor, *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 52–62, Singapore, November 1999. ACM Press.
4. Kumar Avijit, Anupam Datta, and Robert Harper. PCML_5: A language for ensuring compliance with access control policies, 2009. Draft conveyed through personal communication.
5. Lujo Bauer. *Access Control for the Web via Proof-Carrying Authorization*. PhD thesis, Princeton University, November 2003.
6. Lujo Bauer, Scott Garriss, Jonathan M. McCune, Michael K. Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the Grey system. In *Information Security: 8th International Conference (ISC '05)*, Lecture Notes in Computer Science, pages 431–445, September 2005.
7. Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Design and semantics of a decentralized authorization language. In *20th IEEE Computer Security Foundations Symposium*, pages 3–15, 2007.
8. Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew Gordon, and Sergio Maffeis. Refinement types for secure implementations. In *CSF '08: Proceedings of the 21st IEEE Computer Security Foundations Symposium*, pages 17–32. IEEE Computer Society, 2008.
9. John DeTreville. Binder, a logic-based security language. In M. Abadi and S. Bellovin, editors, *Proceedings of the 2002 Symposium on Security and Privacy (S&P'02)*, pages 105–113, Berkeley, California, May 2002. IEEE Computer Society Press.
10. Cormac Flanagan. Hybrid type checking. In *POPL'06: Proceedings of the 33rd ACM Symposium on Principles of Programming Languages*, pages 245–256. ACM, 2006.
11. Cédric Fournet, Andrew Gordon, and Sergio Maffeis. A type discipline for authorization in distributed systems. In *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium*, pages 31–48. IEEE Computer Society, 2007.
12. Deepak Garg. Principal-centric reasoning in constructive authorization logic. Technical Report CMU-CS-09-120, Carnegie Mellon University, 2009.
13. Deepak Garg. Proof search in an authorization logic. Technical Report CMU-CS-09-121, Carnegie Mellon University, 2009.
14. Deepak Garg and Frank Pfenning. Non-interference in constructive authorization logic. In J. Guttman, editor, *Proceedings of the 19th Computer Security Foundations Workshop (CSFW '06)*, pages 283–293, Venice, Italy, July 2006. IEEE Computer Society Press.
15. Deepak Garg and Frank Pfenning. A proof-carrying file system. Technical Report CMU-CS-09-123, Carnegie Mellon University, 2009.
16. Yuri Gurevich and Itay Neeman. DKAL: Distributed-knowledge authorization language. In *Proceedings of the 21st IEEE Symposium on Computer Security Foundations (CSF-21)*, 2008.
17. Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: A programming language for authorization and audit. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2008.

18. Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, November 1992.
19. Chris Lesniewski-Laas, Bryan Ford, Jacob Strauss, Robert Morris, and M. Frans Kaashoek. Alpaca: Extensible authorization for distributed services. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS-2007)*, Alexandria, VA, October 2007.
20. Andrew Pimlott and Oleg Kiselyov. Soutei, a logic-based trust-management system. In *Proceedings of the Eighth International Symposium on Functional and Logic Programming (FLOPS 2006)*, pages 130–145, 2006.
21. Jeffrey A. Vaughan, Limin Jia, Karl Mazurak, and Steve Zdancewic. Evidence-based audit. In *Proceedings of the 21st IEEE Symposium on Computer Security Foundations (CSF-21)*, 2008.

A Proof sketches

A.1 Proof of Theorem 1

The proof relies on the following lemma. Here, we write $\sigma < \sigma'$ when there exists σ'' such that $\sigma\sigma'' = \sigma'$.

Lemma 1. *Suppose that Axioms (1–4) hold. Let σ be any ground substitution, and $\overline{\sigma}$ be such that $\overline{\sigma} < \sigma$. Let $\theta = \_, \mathcal{L}$. Let $\overline{H}$, M, Φ, Ξ, H, and ξ be such that $\overline{H}, M\sigma, \Phi\sigma \leq H$ and for all $(\chi, \eta\overline{\sigma}, t\overline{\sigma}) \in \Xi$, we have $H, \mathcal{L} \vdash \mathbf{auth}(\chi, \eta\sigma, t\sigma) \searrow \xi(\chi, \eta\sigma, t\sigma)$. Suppose that $(\overline{\sigma}, M, \Phi, \Xi) \vdash P \overset{\overline{H},\theta,\chi}{\rightsquigarrow} P'$. We have:*

- *if $H, S, \xi, P\sigma \xrightarrow{\theta,A} H', S', \xi', Q\sigma$ for some Q then $H, S, \xi, P'\sigma \xrightarrow{\theta,A^\star} H', S', \xi', Q'\sigma$ for some Q' such that $(\overline{\sigma}', M', \Phi', \Xi') \vdash Q \overset{\overline{H},\theta,\chi}{\rightsquigarrow} Q'$, $\overline{\sigma}' < \sigma$, and $\overline{H}, M'\sigma, \Phi'\sigma \leq H'$, and for all $(\chi, \eta\overline{\sigma}', t\overline{\sigma}') \in \Xi'$, we have $H', \mathcal{L} \vdash \mathbf{auth}(\chi, \eta\sigma, t\sigma) \searrow \xi'(\chi, \eta\sigma, t\sigma)$.*
- *if $H, S, \xi, P'\sigma \xrightarrow{\theta,A} H', S', \xi', Q'\sigma$ for some Q' then $H, S, \xi, P\sigma \xrightarrow{\theta,A^\star} H', S', \xi', Q\sigma$ for some Q such that $(\overline{\sigma}', M', \Phi', \Xi') \vdash Q \overset{\overline{H},\theta,\chi}{\rightsquigarrow} Q'$, $\overline{\sigma}' < \sigma$, and $\overline{H}, M'\sigma, \Phi'\sigma \leq H'$, and for all $(\chi, \eta\overline{\sigma}', t\overline{\sigma}') \in \Xi'$, we have $H', \mathcal{L} \vdash \mathbf{auth}(\chi, \eta\sigma, t\sigma) \searrow \xi'(\chi, \eta\sigma, t\sigma)$.*

Proof. By induction on the derivation structure of the compilation judgment.

Lemma 2. *Lemma 1 implies Theorem 1.*

Proof. By induction on the length of $\xrightarrow{\theta,A^\star}$.

A.2 Proof of Theorem 2

The proof follows by observation of the proof of Lemma 1. In particular, for the case where P is a shell command, using the relaxed **(Reduct shell)** rule suffices to establish the required invariants.

B Example: Homework for Courses

Consider the following idealized scenario for homework management of various courses at an university. There is a directory, `"/courses"`, containing the directories of all courses. In each course directory, there is a directory named `"instructor"` and a directory named `"students"`, both containing directories named after all students. Furthermore, the `"instructor"` directory contains a file called `"homework"`, and each directory under `"students"` has a file called `"solution"`.

```
courses = "/courses";

assert (read, courses);
for course in courses {
  assert (read, course);
  for user in course {
    test suffix (user, "instructor") {
      instructor = user;
      assert (read, instructor);
      for fileinstr in instructor {
        test suffix (fileinstr, "homework") {
          assert (read, fileinstr);
          students = course/"students";
          assert (read, students);
          for studdir in students {
            hwstud = studdir/"homework";
            assert (write, hwstud);
            shell cp (fileinstr, hwstud)
          }
        }
      }
    };

    test suffix (user, "students") {
      assert (read, user);
      for userdir in user {
        studname = base (userdir);
        solninstr = course/"instructor"/studname/"solution";
        solnstud = userdir/"solution";
        assert read solnstud;
        assert write solninstr;
        shell cp (solnstud, solninstr)
      }
    }
  }
}
```

Fig. 2. PCAL program for homework example

Figure 2 shows a PCAL program that does the following. It navigates into the `"instructor"` directory, and copies the `"homework"` file into each directory under `"students"` in turn. Then, it navigates into those directories and copies the `"solution"` file of that student into the corresponding directory under `"instructor"`.

Next, we show the policy in effect. The policy is written in the authorization logic BL [12, 13]. In order to represent policies made by different principals, BL includes a modality A **says** s which means that administrator A states, or believes formula s (s usually expresses a policy rule). The **says** modality has been considered in prior work (*e.g.*, [2, 18, 14, 1]), but the inference rules defining its meaning vary. BL's rules are shown below. In addition, any complete axiomatization of first-order intuitionistic logic is also assumed.

$$\vdash (A \textbf{ says } (s \Rightarrow t)) \Rightarrow ((A \textbf{ says } s) \Rightarrow (A \textbf{ says } t)) \tag{K}$$

$$\frac{\vdash s}{\vdash A \textbf{ says } s} \tag{N}$$

$$\vdash (A \textbf{ says } s) \Rightarrow (A' \textbf{ says } A \textbf{ says } s) \tag{I}$$

$$\vdash A \textbf{ says } ((A \textbf{ says } s) \Rightarrow s) \tag{C}$$

In our specific policy, we assume that S and L are separate authorities. The formula $\textbf{auth}(A, \eta, t)$ is defined as S **says** $\textbf{may}(A, \eta, t)$; in other words, S represents the enforcer of the policy. On the other hand, L is a local authority that may certify some formulas that S relies on, for example, the validity of the `"/courses"` directory and the membership of certain principals in special groups for which certain policy rules may apply.

We focus on a detailed modeling of the relationship between the function symbol / (that concatenates directory paths with file names to to give file paths) and directory membership constraints. This allows the compiler to reason, for example, that if f is in directory d and the suffix of f is x then $f \equiv d/x$. Other rules allow principals in the group `special` to inherit `read` permissions from ancestor directories. Finally, there are rules that are specific to instructors and students, specifying which files in the others' directories they are allowed to read and write.

The policy rules are split into two parts. The first part contains the rules stated by L:

$$\forall c.\ (S \textbf{ says } \texttt{member}(c, \texttt{"/courses"})) \Rightarrow \texttt{course}(c)$$

$$\texttt{special}(\texttt{"User"})$$

The next part contains the rules stated by S. These include all the policy rules, plus rules that model equivalences between paths constructed using /, `base`, and `suffix`.

$$\forall f.\forall d.\forall x.\ \texttt{member}(f, d) \Rightarrow \texttt{suffix}(f, x) \Rightarrow f \equiv d/x$$
$$\forall f.\forall d.\forall x.\forall p.\ \texttt{member}(f, d) \Rightarrow \texttt{suffix}(f, x) \Rightarrow d \equiv p \Rightarrow f \equiv p/x$$

$$\forall A.\forall f.\forall p.\forall \eta.\ f \equiv p \Rightarrow \mathbf{may}(A, \eta, f) \Rightarrow \mathbf{may}(A, \eta, p)$$

$$\forall f.\ \texttt{suffix}(f, \texttt{base}(x))$$

$$\begin{aligned}\forall A.\forall c.\forall d.\ (L\ \mathbf{says}\ \texttt{course}(c)) \Rightarrow \mathbf{may}(A, \texttt{read}, c/\texttt{"instructor"}) \Rightarrow \\ \mathbf{may}(A, \texttt{read}, c/\texttt{"students"}) \\ \wedge\ \texttt{member}(d, c/\texttt{"students"}) \Rightarrow \mathbf{may}(A, \texttt{write}, d/\texttt{"homework"})\end{aligned}$$

$$\begin{aligned}\forall A.\forall c.\forall x.\forall d.\ (L\ \mathbf{says}\ \texttt{course}(c)) \Rightarrow \mathbf{may}(A, \texttt{read}, c/\texttt{"students"}/x) \Rightarrow \\ \mathbf{may}(A, \texttt{write}, c/\texttt{"instructor"}/x/\texttt{"solution"}) \\ \wedge\ d \equiv c/\texttt{"students"}/x) \Rightarrow \mathbf{may}(A, \texttt{read}, d/\texttt{"solution"})\end{aligned}$$

$$\forall A.\ \mathbf{may}(A, \texttt{read}, \texttt{"/courses"})$$
$$\begin{aligned}\forall A.\forall f.\forall d.\ (L\ \mathbf{says}\ \texttt{special}(A)) \Rightarrow \texttt{member}(f, d) \Rightarrow \\ \mathbf{may}(A, \texttt{read}, d) \Rightarrow \mathbf{may}(A, \texttt{read}, f)\end{aligned}$$

With this policy, the PCAL compiler can eliminate *all* `assert` statements in the program of Figure 2, if the program is run by principal `"User"`.