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, 17, 18] 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 *proof* showing that the policy *entails* that the principal may perform the operation on the 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 and tested it with a PCA-based file system, PCFS [17].

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.

M. Backes and P. Ning (Eds.): ESORICS 2009, LNCS 5789, pp. 184–199, 2009.

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 formally prove 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. Thus, 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 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 [18,4]. 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 or type 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 static policies.
2. Programmer annotations in PCAL 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 *verify* any assumptions on the existence of authorization proofs and other constraints made by the compiler, thereby allowing sound optimizations.
3. We prove formally that the behavior of a compiled program is the same as that of the source program (Theorem 1) and that successfully compiled programs cannot fail due to access checks (Theorem 2). The proofs of these theorems require a precise characterization of assumptions 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.

There are two other notable aspects of PCAL's implementation, that we mention only briefly (details are presented in a technical report [10]). First, the PCAL compiler sometimes constructs proofs which are *parametric* over program variables whose values are not known at compile time. These variables are substituted at run time to obtain ground proofs. Second, functions and predicates are treated at different levels of abstraction in different parts of our implementation. Whereas in a script functions and predicates may have concrete implementations, the compiler only partially interprets them with abstract rewrite rules, so that

the script can be analyzed with symbolic techniques. Further, calls to the theorem prover are simplified, so that proof search need not interpret functions at all. This makes PCAL compatible with many different provers.

The rest of this 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. Details of the language, its compilation, correctness theorems, and implementation are covered in Section 4. Section 5 concludes the paper.

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 [18] and PCML_5 [4], where PCA as well as a logic for expressing policies are embedded in the type system, and proofs are data or type 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 [14, 8]. These languages use an external logic like PCAL, but the objective is to express logical conditions. 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). 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 [13], especially as applied to recent security type systems [9, 11]. Indeed, PCAL departs from previous lines of work in that it 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 the file system PCFS [17]. The latter implementation is the basic test bench for PCAL. The specific logic used for writing policies in this paper (and PCFS) is BL [15, 17]. It is related to, but more expressive than, many other logics and languages for writing access policies (*e.g.*, [1, 2, 16, 7, 12]).

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, 17, 18] 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 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. 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. The system interface must provide a mechanism by which users can submit proofs either prior to or along with an access request. Even though users are free to construct proofs by any means they like, it is convenient to have an automatic theorem prover to perform this task.

Assumptions. PCAL's compiler supports rich logics for writing policies, in which proofs may depend not only on the formulas constituting the policy, but also on system state (*e.g.*, meta-data of files and clock time). Let H denote the system state. 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. (In particular, s may be $\mathbf{auth}(A, \eta, t)$.)

PCAL assumes that an automatic theorem prover for the logic is available, both through an API and as a command line tool. 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, $H; \mathcal{L} \vdash s \not\searrow$ 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
```

3 Overview of PCAL

In this section, we work through a small example to demonstrate the steps of our compilation. (PCAL is formalized 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 with the convention that implication $\Rightarrow$ is right associative.

Policy

$\forall A.\forall x.$ **auth**$(A,$ `write`$,$ `path("/tmp", x)`$)$.
$\forall A.\forall x.\forall y.$ `member`$(x, y) \Rightarrow$ **auth**$(A,$ `read`$, y) \Rightarrow$
(**auth**$(A,$ `read`$, x) \wedge$
(`extension(`x`, "log")` $\Rightarrow$ **auth**$(A,$ `write`$, x)$)).

Informally, the policy asserts the following:

- 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.

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"

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.).

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

Theory

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

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. (A description of partial evaluation modulo equational theories is deferred to the related technical report [10]; remaining details are presented in Section 4.)

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) $\texttt{z} = \texttt{path(foo, x)}$.
(3) $\texttt{x} = \texttt{base}(x')$.
(4) $\texttt{member}(x', \texttt{foo})$.
(5) The statement `assert (read, foo)` in line 2 succeeds.

From condition (5), we can conclude that

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

Simplifying conditions (2), (3), and (4) using the given theory, we have

(7) $\texttt{z} = x'$.

Now from conditions (1), (4), (6), and (7) and the given policy, the theorem prover can conclude that $\textbf{auth}(\texttt{\_PRIN}, \texttt{write}, z)$, 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 uninterpreted functions and predicates `path`, `base`, `extension` occurring in P. The implementations of such functions and predicates are sound with respect to the equational theory used by the compiler. The value of `_PRIN` is provided by the user at the time of compilation (see Section 4).

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

For simplicity of presentation, we abstract various details of the implementation. 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 is bound to 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).

Extended syntax

$e ::=$	statements
...	
$\alpha =$ prove (η, t)	prove that principal χ has permission η on path t and bind the proof to α
inject (η, t) γ	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 is 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 inject (η, t) γ 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'. The reduction rules are shown in Figure 2. $H, S \overset{n(t_1,\ldots,t_k)}{\blacktriangleright} H', S'$ 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.

- **(Reduct for)** unrolls a loop P for each file x in a directory t. **(Reduct test)** simplifies test φ $\{P\}; Q$ 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 passes it to the system interface by placing 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 $\alpha =$ prove (η, t); inject (η, t) α is exactly the same as the command assert (η, t). However, assert (η, t) occurs only in source programs whereas prove (η, t) and inject (η, t) γ 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

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

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

(Reduct test)
$$\frac{\rho = (H, \_, \_) \qquad H \vDash \varphi}{\rho, \mathsf{test}\ \varphi\ \{P\}; Q \xrightarrow{\theta,\chi} \rho, P; Q} \qquad \frac{\rho = (H, \_, \_) \qquad H \nvDash \varphi}{\rho, \mathsf{test}\ \varphi\ \{P\}; Q \xrightarrow{\theta,\chi} \rho, Q}$$

(Reduct assign)
$$\rho, x = t; Q \xrightarrow{\theta,\chi} \rho, Q\{t/x\}$$

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

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

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

Fig. 2. Reduction rules

at the program point that the compiler is looking at. These facts are formally represented by $\Gamma = (\sigma, \Phi, \Xi)$.

- σ is a list of substitutions of the form $\{t/x\}$. The latter means that program variable x is bound to term t.
- Φ is a list of interpreted predicates φ that can be assumed to hold at a program point. These are gathered from commands $\mathsf{test}\ \varphi\ \{\ldots\}$ and $\mathsf{for}\ x\ \mathsf{in}\ t\ \{\ldots\}$. In particular, φ may be of the form $\mathtt{member}(t', t)$, meaning that path t' is in directory t; and we assume that $\mathtt{members}(H, t) = t_1, \ldots, t_k$ implies $H \vDash \mathtt{member}(t_1, t) \wedge \ldots \wedge \mathtt{member}(t_k, t)$.
- Ξ is a partial function from triples (A, η, t) to authorization proofs that the compiler has already constructed.

Figure 3 shows the rules to derive 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 $\sigma\backslash\widetilde{x}$ denotes the restriction of σ that removes the mappings for all variables in $\widetilde{x}$. Finally, $|\Xi|$ and $\langle\Xi\rangle$ extract the formulas and proofs in Ξ ($\prod$ denotes tupling of proofs):

$$|\Xi| = \bigwedge_{(A,\eta,t)\in\mathsf{dom}(\Xi)} \mathbf{auth}(A,\eta,t) \qquad \langle\Xi\rangle = \prod_{\gamma\in\mathsf{rng}(\Xi)} \gamma$$

- **(Comp end)** terminates compilation when end is seen.
- **(Comp for)** compiles for x in t $\{P\}$; Q by compiling P to P' under the added assumption $\mathtt{member}(x, t\sigma)$ (which must hold inside the body of the loop), and compiling Q to Q'. In each case, any prior substitutions for variables $\widetilde{x}$ assigned in P are removed from σ, because they may be invalidated during the execution of the loop (premises $\widetilde{x} = \mathcal{W}(P)$ and $\sigma' = \sigma\backslash\widetilde{x}$).
- **(Comp test)** is similar to **(Comp for)**; in this case the assumption $\varphi\sigma$ is added when compiling the body P of the test branch.
- **(Comp assign)** records the effect of assignment $x = t$ by augmenting substitution σ with $\{t\sigma/x\}$. This augmented substitution is used to compile the remaining program.
- **(Comp shell)** checks that there is a proof in the set of previously constructed proofs Ξ to authorize each permission needed to execute a shell command $n(t_1, \ldots, t_k)$. (Proofs are added to this set in the next two rules).
- **(Comp static)** and **(Comp 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 $|\Xi| \Rightarrow \mathbf{auth}(\chi, \eta, t\sigma)$ by calling the theorem prover. The context in which the proof is constructed not only contains H and the policy $\mathcal{L}$, but also information about directory memberships and predicates tested in outer scopes ($\Phi\sigma$). If a proof γ' can be constructed, rule **(Comp static)** is used: assert (η, t) is replaced by inject (η, t) γ, which passes the statically generated proof $\gamma = \gamma'\ \langle\Xi\rangle$ to the system interface at run time. ($\gamma'\ \langle\Xi\rangle$ is the proof of $\mathbf{auth}(\chi, \eta, t\sigma)$ obtained by eliminating the connective $\Rightarrow$ from $|\Xi| \Rightarrow \mathbf{auth}(\chi, \eta, t\sigma)$). Also, the fact that the new proof exists is recorded by updating Ξ to $\Xi' = \Xi[(\chi, \eta, t\sigma) \mapsto \gamma]$, and using Ξ' to compile the remaining program P. If the proof construction fails, rule **(Comp 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 $\mathbf{auth}(\chi, \eta, t\sigma)$ will exist when P executes (else $\alpha =$ prove (η, t) will block at run time), so Ξ is updated to $\Xi' = \Xi[(\chi, \eta, t\sigma) \mapsto \alpha]$.

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

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

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

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

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

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

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

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

Fig. 3. Compilation rules

Formal Guarantees. We close this section by stating two theorems that guarantee correctness of compilation. Proofs of these theorems can be found in the related technical report [10]. We begin by defining a preorder $\leq$ on system states. Roughly, $H \leq H'$ if any formula that holds under H also holds under H'.

Definition 1 ($\leq$). *For any H and H', let $H \leq H'$ if for all φ, γ, $\mathcal{L}$, and s, (1) $H \vDash \varphi$ implies $H' \vDash \varphi$, and (2) $\gamma :: H; \mathcal{L} \vdash s$ implies $\gamma :: H'; \mathcal{L} \vdash s$.*

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), (3), and (4) state that verification of proofs must

be closed under substitution, modus ponens, and product. Axiom (5) states that the theorem prover produces only verifiable proofs (*i.e.*, the theorem prover is sound). Axiom (6) states that the theorem prover always produces a proof if some proof exists (*i.e.*, the theorem prover is complete).

Axioms

(1) if $H, S \overset{n(t_1\sigma,\ldots,t_k\sigma)}{\blacktriangleright} H', S'$ then $H \leq H'$
(2) if $\gamma :: H; \mathcal{L} \vdash s$ then $\gamma\sigma :: H\sigma; \mathcal{L} \vdash s\sigma$
(3) if $\gamma :: H; \mathcal{L} \vdash s$ and $\gamma' :: H; \mathcal{L} \vdash s \Rightarrow s'$ then $(\gamma'\ \gamma) :: H; \mathcal{L} \vdash s'$
(4) if $\gamma_i :: H; \mathcal{L} \vdash s_i$ for each $i \in 1..n$, then $(\prod_{i\in 1..n} \gamma_i) :: H; \mathcal{L} \vdash \bigwedge_{i\in 1\ldots n} s_i$
(5) if $H; \mathcal{L} \vdash s \searrow \gamma$ then $\gamma :: H; \mathcal{L} \vdash s$
(6) if $\gamma :: H; \mathcal{L} \vdash s$ then $H; \mathcal{L} \vdash s \searrow \gamma'$ for some γ'.

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 state.

Theorem 1 (Compilation correctness). *Suppose that Axioms (1–6) hold, and* $(\varnothing, \varnothing, \varnothing) \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'$. *Then for all* A *and* $\rho = (H', \_, \_)$ *such that* $H \leq H'$, *we have* $\rho, P \overset{\theta,A}{\longrightarrow}{}^{\star} \rho', Q$ *for some* Q *if and only if* $\rho, P' \overset{\theta,A}{\longrightarrow}{}^{\star} \rho', Q'$ *for some* Q'. ($\overset{\theta,A}{\longrightarrow}{}^{\star}$ denotes the reflexive-transitive closure of $\overset{\theta,A}{\longrightarrow}$)

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, which differs from the earlier version in that its premises do not mention any proofs.*

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

Theorem 2 (Access control redundancy). *Suppose that Axioms (1–6) hold, and* $(\varnothing, \varnothing, \varnothing) \vdash P \overset{H,\theta,\chi}{\rightsquigarrow} P'$. *Then for all* A *and* $\rho = (H', \_, \_)$ *such that* $H \leq H'$, *we have* $\rho, P' \overset{\theta,A}{\longrightarrow}{}^{\star} \rho', Q$ *for some* Q *if and only if* $\rho, P' \overset{\theta,A}{\Longrightarrow}{}^{\star} \rho', Q'$ *for some* Q'.

Before we close this section, let us point out some consequences of our axioms. Axioms (2), (3), (4), (5) represent standard expectations from the proof system

and the theorem prover. Axiom (6) is required to prove soundness of the compiler ("if" direction of Theorem 1) since, in its absence, there is no guarantee that a statically provable authorization will be successfully proved in the rule **(Reduct assert)** when executing the source program directly. Axiom (1) is needed for a similar purpose; without this axiom, the compiler must throw away assumptions on the system state in the continuation of any shell command. However, the axiom may seem too strong and invalid in practice. Fortunately, weaker versions of this axiom suffice to prove our theorems for specific programs. In particular, the definition of $H \leq H'$ may be qualified to require that $H \vDash \varphi$ imply $H' \vDash \varphi$ for only those φ that appear in a program of interest (and their substitution instances).

Implementation. We have implemented a prototype PCAL compiler and tested it on the proof-carrying file system PCFS [17]. The specific logic currently used in our implementation is BL [15, 17]. Further details of the implementation and some additional examples of use may be found in our technical report [10].

5 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.*, [18, 4]).

Acknowledgments. Avik Chaudhuri was supported by DARPA under grant no. ODOD.HR00110810073. Deepak Garg was supported partially by the iCAST project sponsored by the National Science Council, Taiwan, under grant no. NSC97-2745-P-001-001, and partially by the Air Force Research Laboratory under grant no. FA87500720028.

References

1. Abadi, M.: Access control in a core calculus of dependency. Electronic Notes in Theoretical Computer Science 172, 5–31 (2007); Computation, Meaning, and Logic: Articles dedicated to Gordon Plotkin
2. Abadi, M., Burrows, M., Lampson, B., Plotkin, G.: A calculus for access control in distributed systems. ACM Transactions on Programming Languages and Systems 15(4), 706–734 (1993)
3. Appel, A.W., Felten, E.W.: Proof-carrying authentication. In: ACM Conference on Computer and Communications Security (CCS 2009), pp. 52–62. ACM Press, New York (1999)
4. Avijit, K., Datta, A., Harper, R.: $PCML_5$: A language for ensuring compliance with access control policies (2009); Draft, personal communication
5. Bauer, L.: Access Control for the Web via Proof-Carrying Authorization. PhD thesis, Princeton University (2003)
6. Bauer, L., Garriss, S., McCune, J.M., Reiter, M.K., Rouse, J., Rutenbar, P.: Device-enabled authorization in the grey system. In: Zhou, J., López, J., Deng, R.H., Bao, F. (eds.) ISC 2005. LNCS, vol. 3650, pp. 431–445. Springer, Heidelberg (2005)
7. Becker, M.Y., Fournet, C., Gordon, A.D.: Design and semantics of a decentralized authorization language. In: IEEE Computer Security Foundations Symposium (CSF 2007), pp. 3–15. IEEE Computer Society Press, Los Alamitos (2007)
8. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A., Maffeis, S.: Refinement types for secure implementations. In: IEEE Computer Security Foundations Symposium (CSF 2008), pp. 17–32. IEEE, Los Alamitos (2008)
9. Chaudhuri, A., Abadi, M.: Secrecy by typing and file-access control. In: IEEE Computer Security Foundations Workshop (CSFW 2006), pp. 112–123. IEEE, Los Alamitos (2006)
10. Chaudhuri, A., Garg, D.: PCAL: Language support for proof-carrying authorization systems. Technical Report CMU-CS-09-141, Carnegie Mellon University (2009)
11. Chaudhuri, A., Naldurg, P., Rajamani, S.: A type system for data-flow integrity on Windows Vista. In: ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2008), pp. 89–100. ACM, New York (2008)
12. DeTreville, J.: Binder, a logic-based security language. In: IEEE Symposium on Security and Privacy (S&P 2002), pp. 105–113. IEEE, Los Alamitos (2002)
13. Flanagan, C.: Hybrid type checking. In: ACM Symposium on Principles of Programming Languages (POPL 2006), pp. 245–256. ACM, New York (2006)
14. Fournet, C., Gordon, A., Maffeis, S.: A type discipline for authorization in distributed systems. In: IEEE Computer Security Foundations Symposium (CSF 2007), pp. 31–48. IEEE, Los Alamitos (2007)
15. Garg, D.: Proof search in an authorization logic. Technical Report CMU-CS-09-121, Carnegie Mellon University (2009)
16. Garg, D., Pfenning, F.: Non-interference in constructive authorization logic. In: IEEE Computer Security Foundations Workshop (CSFW 2006), pp. 283–293. IEEE, Los Alamitos (2006)
17. Garg, D., Pfenning, F.: A proof-carrying file system. Technical Report CMU-CS-09-123, Carnegie Mellon University (2009)
18. Jia, L., Vaughan, J.A., Mazurak, K., Zhao, J., Zarko, L., Schorr, J., Zdancewic, S.: Aura: A programming language for authorization and audit. In: ACM International Conference on Functional Programming (ICFP 2008). ACM, New York (2008)