Proof carrying authorization in ML5

Romain Brenguier
Internship report
Supervisor : Frank Pfenning
Carnegie Mellon University

April - August 2008

Contents

1 Introduction **4**

2 Proof carrying authorization **4**
- 2.1 Access control 4
- 2.2 Authorization logic 4
- 2.3 Proof carrying authorization architecture 6

3 ML5 **6**
- 3.1 A modal logic for distributed computing 6
  - 3.1.1 IS5 7
  - 3.1.2 Lambda 5 natural deduction 7
- 3.2 Validity 7
- 3.3 Implementation 8
- 3.4 Example 9
- 3.5 Security issues 10

4 ML5 with Proof Carrying Authorization **10**
- 4.1 Introduction 10
- 4.2 Architecture 11
- 4.3 Authorization logic 11
- 4.4 Syntax of the language 12
- 4.5 Explicit polymorphism 14
- 4.6 Certificates 16

5 Implementation **16**
- 5.1 Runtime system 16
- 5.2 Identity 16
- 5.3 Certificates 17
- 5.4 Resources 17

6 Application **17**

7 Conclusion **18**

Synthesis

General context

This internship stands in the field of logic and programming languages. The proof carrying authorization architecture is an efficient way to control access to resources and has already been successfully used in some applications. It is for instance used by the Grey project at CMU [BGM+05]. This system uses a resource monitor to protect ressources, proofs have to be submitted to this resource monitor when one wants to have access to the resource. The aim of this research is a programming language in which we can easily write these proofs and in which the type system makes sure that these proofs are correct before even submitting them to the resource monitor. Such a type system would make a resource monitor redundant, as when we are executing a well-typed program we know that every access attempt will succeed.

Problem Studied

One of the goal of the Manifest Security project [CHP+07], is to develop a programming language, in which the fact that a given program respects some policies is made explicit by type checking.

The problem we are addressing here is authorization, we do so by using a proof carrying authorization architecture.

ML5 is a distributed language in which location where the code is executed is explicit in the type system. I have studied a way to add proof carrying authorization into this language.

It is a first step towards a manifestly secure language.

Proposed contribution

We extend ML5 with proof carrying authorization. Taking this distributed programming language as a basis, we added a way to represent propositions and proofs of an authorization logic into the language. Our goal is to eliminate incorrect authorization proofs by typechecking proof terms written in the language.

The basic idea was to introduce proposition and truth term at the same level than types in the language. We thought about what primitives a programmer will need, how they would be integrated to the language and then really added them to the implementation of the compiler we had.

Our implementation for this language consists of a compiler and a runtime system incorporated in a HTTP server.

We also tried to implement some applications using this programming language.

Validity arguments

We are developing a compiler for our language and hoping to have applications programmed with it shortly.

Our work is based on ML5, we added proof terms at the same level than types in the language. It gives a way to extend existing languages with an authorization policy, in an elegant way. What we have done here, can easily be extended or adapted for other logics.

Achievements and Prospects

Thanks to our work, we now have a distributed language in which proof of authorization can directly be expressed, and checked at compile time, preventing the use of wrong proofs at run time.

This idea can be generalized to other languages, and also to other logics and not only authorization logics.

By now the language allows us to write some useful and secure applications, but there is several additions we would like to make.

1 Introduction

The goal of this report will be to show how we added proof carrying authorization into the ML5 language. Proof carrying authorization is a technique used for access control, based on an authorization logic. We will start by presenting it and its advantages.

We will then describe the ML5 language and its type system. This language was our starting point, it is a distributed language and incorporate the notion of location in its type system. I find it necessary to present it before going into the changes we made to it.

The language we have designed integrates an authorization logic in its type system, and we will go through a complete description of it.

And finally I will discuss the implementation of our compiler and runtime system for the language and the applications we are thinking about.

2 Proof carrying authorization

2.1 Access control

The problem we are addressing is access control. We want to restrict access to resources to specific users. This can be for instance a file on a local filesystem or a database on a distant computer. To enforce access control we generally have a *reference monitor* that verifies that a request is performed by a user who has the permission to do so. The set of rules determining who should be granted access is called the *policy*.

Specifying policies will be done by writing proposition in an authorization logic and access to a resource by a specific user will be allowed when a proof in this logic has for conclusion that this user can access the resource.

We will start by presenting an authorization logic, and giving an example of proof in this logic. And we then introduce proof carrying authorization architecture.

2.2 Authorization logic

An authorization logic [GP06, PD99] allows us to specify clearly what the policies concerning access to a ressource are. We give here the syntax of formulas for a simple authorization logic.

$$A ::= P | A \supset A | \forall K\ A | K \text{says} A$$

Figure 1: Syntax of formulas, P denotes an arbitrary atomic formula

We reason with this formulas by constructing proofs. Proofs are constructed by using the inference rules given in figure 2.2.

The first rule is the use of a hypothesis, impI and impE are introduction and elimination rules for the implication construct $\supset$, and allpI and allpE for the universal quantification $\forall$, we will not detail them as they are commonly used in natural deduction.

The last three rules are the most interesting. truaff states that if a proposition is true, then anybody affirms it. saysI is the rule that internalizes the affirmation judgment as a proposition. saysE states that if the truth of a proposition A makes K believe in B, then it is enough that K says A to make K affirm B, it implies that when reasoning from the point of view of K, it is the same if something is true or if we think it is.

$$\textsc{use}\ \frac{}{\Gamma, \textsc{true}(A) \vdash \textsc{true}(A)}$$

$$\textsc{imp}\text{I}\ \frac{\Gamma, \textsc{true}(A) \vdash \textsc{true}(B)}{\Gamma \vdash \textsc{true}(A \supset B)} \qquad \textsc{imp}\text{E}\ \frac{\Gamma \vdash \textsc{true}(A \supset B) \qquad \Gamma \vdash \textsc{true}(A)}{\Gamma \vdash \textsc{true}(B)}$$

$$\textsc{allp}\text{I}\ \frac{\Gamma \vdash \textsc{true}(A)}{\Gamma \vdash \textsc{true}(\forall x A)} \qquad \textsc{allp}\text{E}\ \frac{\Gamma \vdash \textsc{true}(\forall x A)}{\Gamma \vdash \textsc{true}(A[x \leftarrow K])}$$

$$\textsc{says}\text{I}\ \frac{\Gamma \vdash \textsc{aff}(K, A)}{\Gamma \vdash \textsc{true}(K \text{says} A)}$$

$$\textsc{truaff}\ \frac{\Gamma \vdash \textsc{true}(A)}{\Gamma \vdash \textsc{aff}(K, A)} \qquad \textsc{says}\text{E}\ \frac{\Gamma \vdash \textsc{true}(K \text{says} A) \qquad \Gamma, A \vdash \textsc{aff}(K, B)}{\Gamma \vdash \textsc{aff}(K, B)}$$

Figure 2: Inference rules

Example

We give the example of a grade sheet, a teacher use to publish grades for his students. He wants to be able access them in order to write them and read them and he want his students to be able to read them. Policies would be :

$$p1 : \mathit{Alice}\,\text{says}\,\textsc{maywt}(\mathit{Sheet}, \mathit{Alice})$$

$$p2 : \mathit{Alice}\,\text{says}\,\textsc{mayrd}(\mathit{Sheet}, \mathit{Alice})$$

$$p3 : \mathit{Alice}\,\text{says}\,\forall x(\textsc{student}(x, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, x))$$

The three predicate we have here are $\textsc{maywt}(\mathit{Sheet}, K)$ meaning that the principal K may write on the sheet, $\textsc{mayrd}(\mathit{Sheet}, K)$ meaning that K may read it and $\textsc{student}(B, A)$ meaning that B is a student of A.

Here we suppose that *Alice* is the principal controlling the ressource that means that

$$\forall x(\mathit{Alice}\,\text{says}\,\textsc{maywt}(\mathit{Sheet}, x)) \supset \textsc{maywt}(\mathit{Sheet}, x)$$

and

$$\forall x(\mathit{Alice}\,\text{says}\,\textsc{mayrd}(\mathit{Sheet}, x)) \supset \textsc{mayrd}(\mathit{Sheet}, x)$$

.

We also have to specify in the policy who the students are, for instance :

$$p4 : \textsc{student}(\mathit{Bob}, \mathit{Alice})$$

Now, we can see that *Bob* should be able to read the sheet, for that we give a proof of $\mathit{Alice}\,\text{says}\,\textsc{mayrd}(\mathit{Sheet}, \mathit{Bob})$. Let Γ be the policies, this would be the hypothesis for our proof.

We first begin by proving $\mathit{Alice}\,\text{says}\,\textsc{mayrd}(\mathit{Sheet}, \mathit{Bob})$ under the hypothesis

$$\Gamma' = \Gamma, \textsc{true}(\textsc{student}(\mathit{Bob}, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))$$

. We will refer to this proof as Π_1.

$$\textsc{truaff}\ \dfrac{\textsc{imp}E\ \dfrac{\dfrac{\textsc{use}\ \dfrac{}{\Gamma' \vdash \textsc{true}(\textsc{student}(\mathit{Bob}, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))}}{\vdots} \qquad \dfrac{}{\Gamma' \vdash \textsc{true}(\textsc{student}(\mathit{Bob}, \mathit{Alice}))}\ \textsc{use}}{\Gamma' \vdash \textsc{true}(\textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))}}{\Gamma' \vdash \textsc{aff}(\mathit{Alice}, \textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))}$$

We then prove $\textsc{student}(\mathit{Bob}, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, \mathit{Bob})$ under the hypothesis $\Gamma'' = \Gamma, \textsc{true}(\forall x(\textsc{student}(x, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, x)))$.

$$\textsc{allp}E\ \dfrac{\textsc{use}\ \dfrac{}{\Gamma'' \vdash \textsc{true}(\forall x(\textsc{student}(x, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, x)))}}{\Gamma'' \vdash \textsc{true}(\textsc{student}(\mathit{Bob}, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))}$$

And we conclude by using the rule *saysE* on the to preceding proofs.

$$\textsc{trueaff}\ \dfrac{\textsc{says}E\ \dfrac{\textsc{says}I\ \dfrac{\textsc{says}E\ \dfrac{\textsc{use}\ \dfrac{}{\Gamma \vdash p3} \qquad \dfrac{\Pi_2}{\Gamma'' \vdash \textsc{aff}(\mathit{Alice}, \textsc{student}(\mathit{Bob}, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))}\ \textsc{trueaff}}{\Gamma \vdash \textsc{aff}(\mathit{Alice}, \textsc{student}(\mathit{Bob}, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))}}{\Gamma \vdash \textsc{true}(\mathit{Alice}\,\text{says}\,\textsc{student}(\mathit{Bob}, \mathit{Alice}) \supset \textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))} \qquad \Pi_1}{\Gamma \vdash \textsc{aff}(\mathit{Alice}, \textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))}}{\Gamma \vdash \textsc{true}(\mathit{Alice}\,\text{says}\,\textsc{mayrd}(\mathit{Sheet}, \mathit{Bob}))}$$

2.3 Proof carrying authorization architecture

In general looking for a proof is a difficult problem, undecidable in the logic we consider. Therefore it is not a good idea to rely on the ressource monitor to automatically find a proof of authorization. On the other hand the user making the request is more likely to know why the access should be alloweded to him. In the preceding example *Bob* would know that he have access because he is a student.

It is a better idea to rely on the user to construct a proof of authorization, so that the ressource monitor only have to check the proof, which is a much simpler task. This is the proof carrying authorization architecture that we want to setup in the context of the ML5 programming language.

3 ML5

3.1 A modal logic for distributed computing

We present ML5, a high level programming language for distributed computing.

In distributed computing, a program execution takes place at different locations, but these locations can have different capabilities. To address this difficulty, ML5 propose a solution by enriching the ML type system with locations [Mur08, MCH07].

With functional languages, types systems are strongly connected with logic. With the Curry-Howard isomorphism we interpret types as propositions and programs become proofs of the propositions corresponding to the types they inhabit.

To introduce a type system with a notion of location, modal logic was chosen, because it allows reasoning from different point of view.

3.1.1 IS5

IS5 is the modal logic on which on which ML5 type system is based. The logic we present here is in fact a particular case of this logic.

$$A, B ::= \Box A \,|\, \Diamond A \,|\, A \supset B \,|\, A \, at \, w \,|A \vee B \,|\, A \wedge B \,|\, p$$

Figure 3: Syntax of propositions in the IS5 logic

In this logic judgements take the form

$$\Gamma \vdash A@w$$

meaning that under the assumption Γ, A is true at the world w. Hypothesis in Γ take the form $B@w'$.

The rules of this logic are given in figure 4 The connectives $\vee$, $\wedge$ and $\supset$ (for implication) have the the same meaning than in non-modal intuitionistic logic when reasoning at the same location . For instance if we know $A@w$ and $B@w$ then we can conclude $A \wedge B@w$.

$\Box A$ means that A is true in all world, therefore if we know $\Box A@w$, we can conclude $A@w'$. In order to prove $\Box A$ we need to prove $A@\omega$, without knowing anything about the world ω.

$\Diamond A$ means that A is true at some world. To prove $\Diamond A$ we only have to give a proof of it at one location. What we can conclude from it, is what we can conclude of knowing $A@\omega$ without knowing anything about ω.

$A \, at \, w$ is the internalization in the logic of the judgement $A@w$.

3.1.2 Lambda 5 natural deduction

Lambda 5 is the lambda calculus corresponding to proof terms of the IS5 logic. In order to make location changes explicit, a new rule is added, it's the GET rule. Therefore the logic presented in figure 5 is different from IS5 but remains equivalent.

Judgments now take the form

$$\Gamma \vdash M : A@\text{w}$$

meaning that M is a proof of $A@\text{w}$ under the hypothesis Γ.

We constrain the rules to act locally and add the GET rule, we give here a fragment of the calculus , some of the rules are not mentioned as they are not modified.

We restrict the **get** construct to **mobile** types. These are types of the values that can be safely communicated between worlds.

3.2 Validity

In general, a large amount of the code written in an application is not particular to any location, so a validity judgment is added. This judgment takes the form $u \sim A$ meaning

$$\wedge\text{ I}\ \frac{\Gamma \vdash A@\mathrm{w} \qquad \Gamma \vdash B@\mathrm{w}}{\Gamma \vdash A \wedge B@\mathrm{w}} \qquad \wedge\mathrm{E}_1\ \frac{\Gamma \vdash A \wedge B@\mathrm{w}}{\Gamma \vdash A@\mathrm{w}} \qquad \wedge\mathrm{E}_2\ \frac{\Gamma \vdash A \wedge B@\mathrm{w}}{\Gamma \vdash B@\mathrm{w}}$$

$$\vee\mathrm{I}_1\ \frac{\Gamma \vdash A@\mathrm{w}}{\Gamma \vdash A \vee B@\mathrm{w}} \qquad \vee\mathrm{I}_2\ \frac{\Gamma \vdash B@\mathrm{w}}{\Gamma \vdash A \vee B@\mathrm{w}}$$

$$\vee\mathrm{E}\ \frac{\Gamma \vdash A \vee B@\mathrm{w}' \qquad \Gamma, A@\mathrm{w}' \vdash C@\mathrm{w} \qquad \Gamma, B@\mathrm{w}' \vdash C@\mathrm{w}}{\Gamma \vdash C@\mathrm{w}}$$

$$\supset\text{ I}\ \frac{\Gamma, A@\mathrm{w} \vdash B@\mathrm{w}}{\Gamma \vdash A \supset B@\mathrm{w}} \qquad \supset\text{ E}\ \frac{\Gamma \vdash A \supset B@\mathrm{w} \qquad \Gamma \vdash A@w}{\Gamma \vdash B@\mathrm{w}}$$

$$\text{AT}\mathrm{I}\ \frac{\Gamma \vdash A@\mathrm{w}}{\Gamma \vdash A\,at\,\mathrm{w}@\mathrm{w}'} \qquad \text{AT}\mathrm{E}\ \frac{\Gamma \vdash A\,at\,\mathrm{w}''@\mathrm{w}' \qquad \Gamma, A@\mathrm{w}'' \vdash B@\mathrm{w}}{\Gamma \vdash B@\mathrm{w}}$$

$$\Box\text{ I}\ \frac{\Gamma, \omega\ \ world \vdash A@\omega}{\Gamma \vdash \Box A@\mathrm{w}} \qquad \Box\text{ E}\ \frac{\Gamma \vdash \Box A@\mathrm{w}}{\Gamma \vdash A@\mathrm{w}'}$$

$$\Diamond\text{ I}\ \frac{\Gamma \vdash A@\mathrm{w}'}{\Gamma \vdash \Diamond A@\mathrm{w}} \qquad \Diamond\text{ E}\ \frac{\Gamma \vdash \Diamond A@\mathrm{w}' \qquad \Gamma, \omega\ \ world, A@\omega \vdash B@\mathrm{w}}{\Gamma \vdash B@\mathrm{w}}$$

$$\text{HYP}\ \frac{}{\Gamma, A@\mathrm{w}, \Gamma' \vdash A@\mathrm{w}}$$

Figure 4: IS5

that u can have the type A at any world. The way we introduce validity judgment in a program is by using the **put** declaration. This judgment is internalized in the logic with the {} (“shamrock”) modality. Type inference make declaration valid when possible.

3.3 Implementation

A compiler : ml5pgh

ml5pgh is an implementation for an ML5 compiler. It’s specialized for web applications involving a web browser and a HTTP server. The code produced consists of Javascript for the web browser side and bytecode for the server side. Javascript can be executed by most web browsers , but for the bytecode produced, a specific server which can interpret the bytecode has been implemented. Although it can only produce this two kind of code, most of the step in the compiler are independent from this particular setting, and in order to add new output languages , only the step of code generation would have to be added.

$$\textsc{at}\mathrm{I}\ \frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathbf{hold} M : A\,at\,\mathrm{w}@\mathrm{w}} \qquad \textsc{at}\mathrm{E}\ \frac{\Gamma \vdash M : A\,at\,\mathrm{w}'@\mathrm{w} \qquad \Gamma, A@\mathrm{w}' \vdash N : B@\mathrm{w}}{\Gamma \vdash \mathbf{leta} M\,in\,N : B@\mathrm{w}}$$

$$\Box\ \mathrm{I}\ \frac{\Gamma, \omega\ \ world \vdash M : A@\omega}{\Gamma \vdash \mathbf{box} M : \Box A@\mathrm{w}} \qquad \Box\ \mathrm{E}\ \frac{\Gamma \vdash M : \Box A@\mathrm{w}}{\Gamma \vdash \mathbf{unbox} M : A@\mathrm{w}}$$

$$\Diamond\ \mathrm{I}\ \frac{\Gamma \vdash M : A@\mathrm{w}}{\Gamma \vdash \mathbf{here} M : \Diamond A@\mathrm{w}} \qquad \Diamond\ \mathrm{E}\ \frac{\Gamma \vdash M : \Diamond A@\mathrm{w}' \qquad \Gamma, \omega\ \ world, A@\omega \vdash N : B@\mathrm{w}}{\Gamma \vdash \mathbf{letd}\omega, x = M\,in\,N : B@\mathrm{w}}$$

$$\textsc{get}\ \frac{A\mathbf{mobile} \qquad \Gamma \vdash M : A@\mathrm{w}'}{\Gamma \vdash \mathbf{get}[w']M : A@\mathrm{w}}$$

Figure 5: Lambda 5 natural deduction

$$\frac{}{\Box\ \mathbf{mobile}} \qquad \frac{}{\Diamond \mathbf{mobile}}$$

$$\frac{A\ \mathbf{mobile} \qquad B\mathbf{mobile}}{A \vee B\ \mathbf{mobile}} \qquad \frac{A\ \mathbf{mobile} \qquad B\mathbf{mobile}}{A \wedge B\ \mathbf{mobile}}$$

$$\frac{}{A\,at\,\mathrm{w}\ \mathbf{mobile}}$$

Figure 6: The *mobile* judgment

Web Features

ML5 can be extended by external functions. On the server side we typically have a database library to access some file that we can read and write. On the browser side we can use every Javascript primitive. In particular a library manipulating DOM objects is furnished, DOM (Document Object Mode) is a standard model for representing HTML or XML. This allows ML5 programs to dynamically modify the structure of the page they are executed in, it also gives other usefull capabilities like access to cookies or redirection to another website.

3.4 Example

We present an example in figure 7, which illustrates some ML5 primitives.

In this example there are two different worlds *server* and *home*. In the first line, we make valid the result of a piece of code executed on the server. *hold* is used to obtain a pointer to the result of *checksig sig*, this is *mobile* but we will need to be at the location *server* to access to find its original value. The typing judgment for *checked_sig* is here $checked_sig \sim string\,option\,at\,server$.

In the second part, we are at the server location and can find the original value of the result of *checksig sig* using *leta*.

We then, according to this result, change the executing location to home in order to

Figure 7: A piece of ML5 source code

```
put checked_sig = from server get hold (checksig sig)

val _ =
  from server get
    let leta cs = checked_sig
    in
      case cs of
        NONE => from home get setstring ([you are not logged in])
      | (SOME log) => from home get setstring ([ OK ])
    end
```

display a string in the web browser used by the client.

3.5 Security issues

The problem of ML5 we now want to address is the lack of security, one part of the code is executed on the client side, therefore it cannot be trusted, however arbitrary continuations can be passed from the client to the server, giving the client potential access to any field in the database. We address this issue by using the ideas of proof carrying authorization.

4 ML5 with Proof Carrying Authorization

4.1 Introduction

Starting from the ML5 programming language we now want to integrate a logic into the language.

Classically in a functional programming language, we distinguish two levels : terms and types. Types classify terms.

Here, we would like to add propositions for example, at the same level than types. But we have to keep a distinction between the two, to do so we need a third level. We now have three levels :

- terms
- constructors , including types, which classify terms
- kinds , which would classify constructors

The ML5 language basically consists of two kinds : TYPE and WORLD, on the level of constructors we have type constructors,type variables and world variable, and finally on the level of terms we have the terms which have been described in section 3.

The idea will be to add propositions of the logic on the level of constructors, on this level we will also have proof terms and therefore judgments will be on the level of kinds as they classify proofs. We don't add anything on the level of terms except for the ACQUIRE_CERT primitive.

As there is nothing yet in the language that allow us to manipulate constructors, it requires us to add an explicit polymorphism instantiation for the kinds we added.

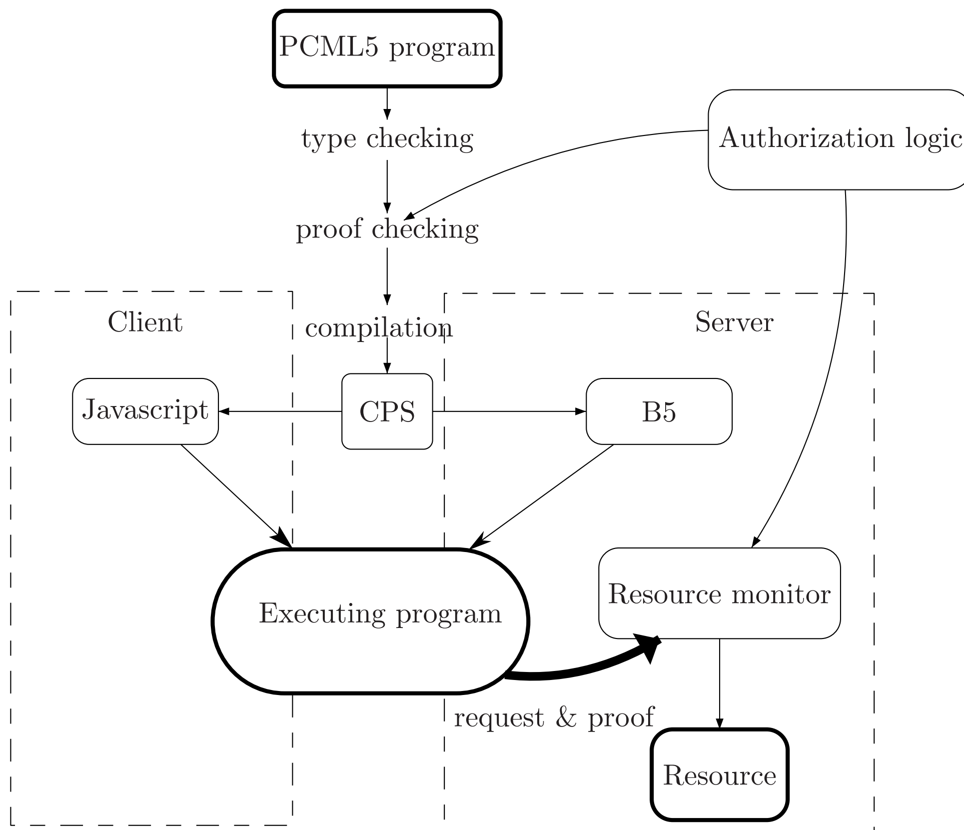
We are going to discuss all these points one by one.

4.2 Architecture

Lets first describe the architecture we are focusing on.

We have two worlds : server and client. The program is compiled to a continuation-passing-style language, and from here, two different kinds of code are generated. On the client side Javascript is executed and on the server side we execute a bytecode language called B5.

The resource we are trying to protect is a database on the server side, it is protected by a ressource monitor that requires proofs of access.



4.3 Authorization logic

The authorization logic we are using is quite simple. It only has two predicates : *mayrd* and *maywt*.

$$
\begin{array}{lcl}
A & ::= & \text{mayrd}(R, K) \mid \text{maywt}(R, K) \\
 & \mid & \langle K \rangle A \mid A \supset A \mid \forall_p A \mid \forall_r A
\end{array}
$$

Table 1: Syntax for propositions

We don't give here the rules for this logic, as it is really close to the one described in 2.2. The difference is that we distinguish resources and principals so we have two universal quantifications. We will give inference rules for proof terms in subsection 4.4.

4.4 Syntax of the language

We give in the next table the details of the syntax of our language.

$$
\begin{array}{llcl}
\text{Kinds} & L & ::= & \text{TYPE} \mid \text{WORLD} \mid \text{DATABASE} \mid \text{PRINCIPAL} \\
 & & \mid & \text{PROP} \mid \text{TRUE}(\text{A}) \mid \text{AFF}(\text{K,A}) \\
 & & & \\
\text{Constructors} & N & ::= & \alpha \mid K \mid W \mid A \to A \mid A \wedge A \mid A \vee A \\
 & & \mid & \Box A \mid \Diamond A \mid A@W \\
 & & \mid & \text{handle } (A) \mid \text{identity } (K) \\
\text{(Propositions)} & & \mid & \text{mayrd}(A, K) \mid \text{maywt}(A, K) \\
 & & \mid & \langle K \rangle A \mid A \supset A \mid \forall_p \alpha A \mid \forall_r \alpha A \\
\text{(Proof terms)} & & \mid & \text{truaff}(A, K, A) \mid \text{saysI}(K, A, A) \mid \text{saysE}(A, A, A) \\
 & & \mid & \text{impI}(A, A, A) \mid \text{impE}(A, A) \mid \\
 & & \mid & \text{allpI}(A, A) \mid \text{allpE}(A, A) \mid \text{allrI}(A, A) \mid \text{allrE}(A, A) \\
 & & & \\
\text{Normal constructors} & A & ::= & N \mid \forall \alpha :: L.A \\
\text{Terms} & M & ::= & x \mid \lambda x : A.M \mid M_1 M_2 \mid \langle M_1, M_2 \rangle \\
 & & \mid & \#_1 M \mid \#_2 M \mid \text{inl } M \mid \text{inr } M \mid \text{case } M \ \{x.M_1 | x.M_2\} \\
 & & \mid & \text{hold } (M) \mid \text{leta } x = M \text{ in } M \mid \text{box } \alpha \ . \ M \mid \text{unbox } M \\
 & & \mid & \text{here } (M) \mid \text{letd } x, \alpha = M \text{ in } M \mid \text{get } [A] \ M \\
 & & \mid & (MN) \\
 & & \mid & \text{acquire_cert } [A] \ \{\alpha.M | M\}
\end{array}
$$

Table 2: Syntax

We describe how types, proposition and proof terms are formed.

We added to ML5 syntax new kinds : databases, principals, propositions, and judgments of the form TRUE(A) and AFF(K,A) where K is a principal and A a proposition. This judgment kinds will be used to classify proof terms according to the statement they are proving.

We add two new types : *identity* and *handle*, terms of this types can be considered as the equivalent of principals and database but at the level of terms. The difference is that we don't want terms to appear in proof terms, because proof terms have to be checked by a resource monitor. The resource monitor is not an interpreter, it is not able to execute terms.

The other types are similar to ML5.

Propositions are just the internalization of propositions of the logic in the language as constructors.

Figure 8: Kinds

$$\frac{}{\Delta \vdash \textsc{type}\,\mathrm{kind}} \qquad \frac{}{\Delta \vdash \textsc{world}\,\mathrm{kind}}$$

$$\frac{}{\Delta \vdash \textsc{database}\,\mathrm{kind}} \qquad \frac{}{\Delta \vdash \textsc{principal}\,\mathrm{kind}} \qquad \frac{}{\Delta \vdash \textsc{proposition}\,\mathrm{kind}}$$

$$\frac{\Delta \vdash A :: \textsc{proposition}}{\Delta \vdash \textsc{true}(A)\,\mathrm{kind}} \qquad \frac{\Delta \vdash A :: \textsc{proposition} \qquad \Delta \vdash K :: \textsc{principal}}{\Delta \vdash \textsc{aff}(K, A)\,\mathrm{kind}}$$

Figure 9: Types

$$\frac{\Delta \vdash P :: \textsc{principal}}{\Delta \vdash \mathrm{identity}(P) :: \textsc{type}} \qquad \frac{\Delta \vdash D :: \textsc{database}}{\Delta \vdash \mathrm{handle}(D) :: \textsc{type}}$$

$$\frac{\Delta \vdash A :: \textsc{type} \qquad \Delta \vdash B :: \textsc{type}}{\Delta \vdash A \to B :: \textsc{type}}$$

$$\frac{\Delta \vdash A :: \textsc{type} \qquad \Delta \vdash B :: \textsc{type}}{\Delta \vdash A \wedge B :: \textsc{type}} \qquad \frac{\Delta \vdash A :: \textsc{type} \qquad \Delta \vdash B :: \textsc{type}}{\Delta \vdash A \vee B :: \textsc{type}}$$

$$\frac{\Delta \vdash A :: \textsc{type}}{\Delta \vdash \Box A :: \textsc{type}} \qquad \frac{\Delta \vdash A :: \textsc{type}}{\Delta \vdash \Diamond A :: \textsc{type}} \qquad \frac{\Delta \vdash A :: \textsc{type} \qquad \Delta \vdash W :: \textsc{world}}{\Delta \vdash A@W :: \textsc{type}}$$

Figure 10: Propositions

$$\frac{\Delta \vdash D :: \textsc{database} \qquad \Delta \vdash K :: \textsc{principal}}{\Delta \vdash \mathrm{mayrd}(D, K) :: \textsc{proposition}}$$

$$\frac{\Delta \vdash D :: \textsc{database} \qquad \Delta \vdash K :: \textsc{principal}}{\Delta \vdash \mathrm{maywt}(D, K) :: \textsc{proposition}}$$

$$\frac{\Delta \vdash K :: \textsc{principal} \qquad \Delta \vdash P :: \textsc{proposition}}{\Delta \vdash \langle K \rangle P :: \textsc{proposition}}$$

$$\frac{\Delta \vdash A :: \textsc{proposition} \qquad \Delta \vdash B :: \textsc{proposition}}{\Delta \vdash A \supset B :: \textsc{proposition}}$$

$$\frac{\Delta, x :: \textsc{principal} \vdash A :: \textsc{proposition}}{\Delta \vdash \forall_p x\, A :: \textsc{proposition}} \qquad \frac{\Delta, x :: \textsc{database} \vdash A :: \textsc{proposition}}{\Delta \vdash \forall_r x\, A :: \textsc{proposition}}$$

Proof terms are somehow more interesting because their kinds reflect the proposition they

are a proof of. These proofs are checked at compile time so that we are sure that compiled code contains only correct proofs. Thanks to that programs cannot fail in the execution because the ressource monitor refused access being submitted an incorrect proof. However the ressource monitor remains necessary as we do not have control on what is executed at other worlds and some malicious computers could execute ill-typed code.

$$\textsc{use}\ \frac{}{\Delta, x :: \textsc{true}(A) \vdash x :: \textsc{true}(A)}$$

$$\textsc{imp}\mathrm{I}\ \frac{\Delta, x :: \textsc{true}(A) \vdash y :: \textsc{true}(B)}{\Delta \vdash \textsc{imp}\mathrm{I}(x, A, y) :: \textsc{true}(A \supset B)} \qquad \textsc{imp}\mathrm{E}\ \frac{\Delta \vdash x :: \textsc{true}(A \supset B) \qquad \Delta \vdash y :: \textsc{true}(A)}{\Delta \vdash \textsc{imp}\mathrm{E}(x, y) :: \textsc{true}(B)}$$

$$\textsc{allp}\mathrm{I}\ \frac{\Delta \vdash y :: \textsc{true}(A)}{\Delta \vdash \textsc{allp}\mathrm{I}(x, y) :: \textsc{true}(\forall_p x.A)} \qquad \textsc{allp}\mathrm{E}\ \frac{\Delta \vdash y :: \textsc{true}(\forall_p x.A) \qquad \Delta \vdash K :: \textsc{principal}}{\Delta \vdash \textsc{allp}\mathrm{E}(y, K) :: \textsc{true}(A[x \leftarrow_p K])}$$

$$\textsc{allr}\mathrm{I}\ \frac{\Delta \vdash y :: \textsc{true}(A)}{\Delta \vdash \textsc{allr}\mathrm{I}(x, y) :: \textsc{true}(\forall_r x.A)} \qquad \textsc{allr}\mathrm{E}\ \frac{\Delta \vdash y :: \textsc{true}(\forall_p x.A) \qquad \Delta \vdash D :: \textsc{database}}{\Delta \vdash \textsc{allr}\mathrm{E}(y, \alpha) :: \textsc{true}(A[x \leftarrow_r D])}$$

$$\textsc{says}\mathrm{I}\ \frac{\Delta \vdash x :: \textsc{aff}(K, A)}{\Delta \vdash \textsc{says}\mathrm{I}(K, A, x) :: \textsc{true}(K \mathrm{says} A)}$$

$$\textsc{truaff}\ \frac{\Delta \vdash x :: \textsc{true}(A)}{\Delta \vdash \textsc{truaff}(A, K, x) :: \textsc{aff}(K, A)}$$

$$\textsc{says}\mathrm{E}\ \frac{\Delta \vdash x :: \textsc{true}(K \mathrm{says} A) \qquad \Delta, \alpha :: A \vdash y :: \textsc{aff}(K, B)}{\Delta \vdash \textsc{says}\mathrm{E}(\alpha, x, y) :: \textsc{aff}(K, B)}$$

Figure 11: Proof terms

4.5 Explicit polymorphism

Proofs need to be passed to some functions, so that they can verify the proofs and, for instance, allow the access to a database. This is similar to polymorphism, since such functions will takes constructors as argument. For instance we would like to be able to declare something like this

$$\text{read} : \forall p :: \textsc{principal}.\forall x :: \textsc{true}(mayrd(db, p)).\text{identity}(p) \to \text{string}$$

In the same way we could represent a polymorphic type

$$\text{id} : \forall a :: \textsc{type}.a \to a$$

The difference with polymorphism over types is that we already have type inference, making explicit instantiation unnecessary. In the case of proofs we want the programmer to manually write them, as their is no way to automatically infer them.

The typing rules for this instantiation follow in figure 12.

Figure 12: Instantiation rules

$$\frac{\Delta \vdash M : \forall x :: \textsc{database}.T \qquad \Delta \vdash D :: \textsc{database}}{\Delta \vdash (MD) : T}$$

$$\frac{\Delta \vdash M : \forall x :: \textsc{principal}.T \qquad \Delta \vdash P :: \textsc{principal}}{\Delta \vdash (MP) : T}$$

$$\frac{\Delta \vdash M : \forall x :: \textsc{proposition}.T \qquad \Delta \vdash P :: \textsc{proposition}}{\Delta \vdash (MP) : T}$$

$$\frac{\Delta \vdash M : \forall x :: \textsc{true}(A).T \qquad \Delta \vdash P :: \textsc{true}(A)}{\Delta \vdash (MP) : T}$$

$$\frac{\Delta \vdash M : \forall x :: \textsc{aff}(K, A).T \qquad \Delta \vdash P :: \textsc{aff}(K, A)}{\Delta \vdash (MP) : T}$$

We have an application of polymorphic terms to kinds, but we choose not to have a corresponding abstraction. Consequently, only values declared as externals can introduce this kind of polymorphism. Read and write are two examples of these functions.

Read and Write

With polymorphism we can now express that a function needs a proof, directly in its type. A principal can read a resource only if he can prove that the principal controlling this resource, says he may read it. This can directly be expressed in our language through the type of the read function.

$$\begin{aligned}\text{read} : \forall d :: \textsc{database}\, \forall p :: \textsc{principal}\, \forall t :: \textsc{true}(\textsc{mayrd}(d, p))\\ \text{handle}(d) \rightarrow \text{identity}(p) \rightarrow \text{string}\end{aligned}$$

and the write function is similar

$$\begin{aligned}\text{write} : \forall d :: \textsc{database}\, \forall p :: \textsc{principal}\, \forall t :: \textsc{true}(\textsc{maywt}(d, p))\\ \text{handle}(d) \rightarrow \text{identity}(p) \rightarrow \text{string} \rightarrow \text{unit}\end{aligned}$$

Here is the actual declaration of this function in the language :

```
extern val (d:database , p:principal ,  prf : TRUE mayrd d p) read :
      d dbhandle -> p identity -> string @ server
```

```
extern val (d:database , p:principal , prf : TRUE maywt d p) write :
      d dbhandle -> p identity -> string -> unit @ server
```

4.6 Certificates

The proofs we construct need hypothesis, they would be the access policies. When the reference monitor checks the proofs, he also need to verify that hypothesis are correct. These hypothesis would take the form of digitally signed certificates, this way the resource monitor can check them by knowing the public key of a principal.

For instance we can use the hypothesis $K \text{says} P$ in a proof if we have a certificate asserting the proposition P, signed by the principal K.

We need to dynamically access these certificates. That is what provides the *acquire_cert* primitive. Here is the typing rule associated with *acquire_cert*.

$$\frac{\Delta, \alpha :: \textsc{true}(A); \Gamma \vdash M_1 : A'@l \qquad \Delta; \Gamma \vdash M_2 : A'@l}{\Delta; \Gamma \vdash \text{acquire_cert}[A]\{\alpha.M_1|M_2\} : A'@l}$$

In this rule, M_1 represent the term that is to be executed if we succeed at runtime in finding a certificate for A. In that case the variable α is bind and is of kind $\textsc{true}(A)$, it can be used in access proof terms as a proof of A.

If we fail to get a certificate then M_2 is executed.

5 Implementation

5.1 Runtime system

Our implementation is based on the *ml5pgh* compiler and the *server5*, so the runtime system is almost similar. The compiler is designed for web applications and produce Javascript and bytecode. Both are kept on the server side.

When someone use his web browser to access an application on the server, he received the Javascript part of the program. The browser should then starts the execution of the script it received, and the server starts to execute the bytecode part of the program.

5.2 Identity

The terms of type identity(p), should in fact be evidence that the principal executing the program is the principal p. In our implementation it is represented by a certificate digitally signed by an authentication authority. This certificate is checked by the runtime system along with proofs and other certificates.

To provide a certificate of identity the server can trust we are using the authentication system used by Carnegie Mellon University's websites. This system is called pubcookie and is an implementation of WebISO, it's used in several universities in the United States. The goal is to provide an authentication server common for every web-applications in the university, this way the users do not have to type their logins and passwords everytime they access a new application, and they only do so when asked by the webiso login server, which they can trust.

The application server side of pubcookie is written as an apache module, and as we are not using apache but our own server implementation (called server5) we needed to reimplement some of the primitives provided by this module.

We provide this identification by functions declared as external, as it is implemented in the server runtime. This is the way we get an identity type.

Here are the actual declaration in our language.

```
val webiso : unit -> string @ home

extern val (p:principal) checksig : string -> p identity option @ server
```

The webiso function request that the user identify himself to the webiso server, the string returned is a cookie generated by this login server containing the login of the user, signed with private key of the login server, and encrypted with a symmetric key that webiso shares with our application server. The checksig function converts this string to a identity if it can verify its signature is valid.

The signature is conserved in the identity value, as it would have to be checked by the resource monitor when we access a resource.

5.3 Certificates

Certificates are mainly managed by the server, there is no primitives in our language yet to dynamically add or remove certificates, so for now they are read by the server in files.

This certificates are propositions, along with a digitally signed hash of the string representation of this propositions.

5.4 Resources

In our implementation resources are only on the server side. So we implemented a simple database linked with the server side which uses a resource monitor, requiring a proof everytime somebody want to access its data.

The way we obtain a handle value for a database is by using the gethandle function.

```
extern val (d:database) gethandle : unit -> d handle
```

6 Application

As an application, we propose to build a grade sheet in this programming language.

We are planning to use it for more complex applications such as a wiki, but we will need to dynamically create ressources which is not yet possible, we will discuss this point in the conclusion.

Example : grade sheet

The policies we want to have in this application are very similar to the ones in 2.2.

We have one database, lets call it *gradedb*. Policies would be

$$p1 : \textsc{maywt}(gradedb, Alice)$$

$$p2 : \textsc{mayrd}(gradedb, Alice)$$

if *Alice* is the teacher and for each student we will need to add a proposition of the form

$$\textsc{mayrd}(gradedb, Bob)$$

The first thing we need is the user to login so that we have a value of type *identity*. This may redirect the user to a login server where he would have to enter his login and password.

We then try to acquire certificates saying that we may read and write the database. If we have both then we are the teacher and we can edit the sheet, if we only have read permissions then we can just display the grades.

```
extern database gradedb
principal user

val cookie = webiso ()
val id = checksig {user} cookie
val gradehandle = gethandle {gradedb} ()

val _ = match (id,gradehandle) with
   (SOME id, SOME gradehandle) =>
       acquire_cert{mayrd gradedb user}
       then p1 .
           acquire_cert{maywt gradedb user}
             then p2 . edit {user,gradedb,p1,p2,id,gradehandle} ()
             else display {user,gradedb,p1,id,gradehandle} ()
       else ()
   | _ => ()
```

edit and display would be two functions that I will not detail. Their types are

```
val (u:principal,db:database,
     p1:TRUE mayrd db u, p2:TRUE maywt db u,
     id: u identity, h: db handle) edit : unit -> unit
val (u:principal,db:database,
     p1:TRUE mayrd db u,
     id: u identity, h: db handle) display : unit -> unit
```

And in these functions a call to read will for instance be performed like this

```
val grades = read {db,u,p1} h id
```

7 Conclusion

Thanks to our work, we now have a distributed language in which proof of authorization can directly be expressed, and checked at compile time, preventing the use of wrong proofs at run time. We will have to build more applications in this language to illustrate how it can be

used and why is type system is useful in practice. Their is also several points that we would like to think about.

It could be a good allow arbitrary predicates, for instance in the sheet grade application, it would be more convenient to have a *student* predicate. More generally, we could be using more expressive logic, for instance with consumable credentials [BBPR06, GBB+06] or a notion of time [DGP08].

We could have primitives in the language to manage certificates, because their is no easy way yet to add certificates or to revoke them, the AURA programming language [JVM+08] for instance has a primitive for certificate signature.

We can notice that the checking of proofs by the ressource monitor is useless for well typed programs. We could think of getting rid of it. For a distributed language this is not so evident because we do not know if parts of the program that are executed on other world are well typed, but at least on the server side we know that proofs constructed at our worlds are correct so we can allow access to our database without checking these proofs but just looking at their kinds.

Adding existential types could also be useful, for instance in the type of the checksig function, it could prevent some runtime errors.

References

[ABLP93] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15:706–734, 1993.

[BBPR06] Lujo Bauer, Kevin D. Bowers, Frank Pfenning, and Michael K. Reiter. Consumable credentials in logic-based access control. Technical report, Carnegie Mellon University, 2006.

[BGM+05] 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 2005*, volume 3650 of *Lecture Notes in Computer Science*, pages 431–445, September 2005.

[CHP+07] Karl Crary, Robert Harper, Frank Pfenning, Benjamin C. Pierce, Stephanie Weirich, and Stephan Zdancewic. Project summary manifest security, 2007.

[DGP08] Henry DeYoung, Deepak Garg, and Frank Pfenning. An authorization logic with explicit time. In *Proceedings of the 21st IEEE Symposium on Computer Security Foundations (CSF-21)*, 2008.

[GBB+06] Deepak Garg, Lujo Bauer, Kevin Bowers, Frank Pfenning, and Michael Reiter. A linear logic of authorization and knowledge. In *Proceedings of the 11th European Symposium on Research in Computer Security*, pages 297–312. Springer, 2006.

[GP06] 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.

[JVM+08] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: A programming language for authorization and audit. 2008. To appear in International Conference on Functional Programming.

[MCH07] Tom Murphy, VII, Karl Crary, and Robert Harper. Type-safe distributed programming with ML5. In *Trustworthy Global Computing 2007*, November 2007.

[Mur08] Tom Murphy, VII. *Modal Types for Mobile Code.* PhD thesis, Carnegie Mellon, January 2008. (draft).

[PD99] Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. In *Mathematical Structures in Computer Science*, page 2001, 1999.

[TZ05] Stephen Tse and Steve Zdancewic. Designing a security-typed language with certificate-based declassification. In *Proc. European Symp. on Programming, volume 3444 of LNCS*, pages 279–294. Springer-Verlag, 2005.