Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 171 (2007) 3–21

www.elsevier.com/locate/entcs

An End-To-End Approach to Distributed Policy Language Implementation (Extended Abstract)

Tom Chothia[1]

Centrum voor Wiskunde en Informatica (CWI)
Kruislaan 413, 1098 SJ
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands.

Dominic Duggan[2]

Dept of Computer Science
Stevens Institute of Technology
Hoboken, NJ 07040, USA.

Ye Wu[3]

Dept of Computer Science
Stevens Institute of Technology
Hoboken, NJ 07040, USA.

Abstract

Language-based security approaches to access control and information flow control must at some point rely on a language for expressing policies. However there will in general be several choices for the correct policy language for any given application, and several choices for the implementation of a policy language in a given domain. This article considers an approach to implementing the policy language at the application level, relying on trusted cryptographic libraries whose interface security guarantees are used to verify the correctness of the policy language implementation.

Keywords: Access control, policy language, cryptographic libraries, abstract data types.

1 Introduction

Many software applications run over untrusted networks. Programmers use libraries of cryptographic operations to secure the communications over these networks. In

[1] Email: `T.Chothia@cwi.nl`

[2] Email: `dduggan@cs.stevens.edu`

[3] Email: `ywu1@cs.stevens.edu`

doi:10.1016/j.entcs.2007.03.010

using these libraries, they are implicitly enforcing security policies for the data is being exchanged. This policy is often unstated, or only stated informally in program comments. If these security policies are stated in a suitably formal policy language, then adherence to these policies may be verified, often using automatic or semi-automatic techniques.

One particular approach, exemplified by language-based security, is to express the policies in a policy language that is incorporated into the type system. Type-checking then provides an avenue for checking the correct manipulation of data according to the security policies expressed in program types. These security policies can then be related to the cryptographic libraries used to secure network communications, via typed APIs for cryptographic operations that express the security guarantees these operations are intended to enforce (e.g., encryption for secrecy, digital signing for integrity).

An obvious issue that needs to be addressed is the form of the policy language that is used in the type system to express security policies. The problem is that any policy language will inevitably (and in fact fairly quickly) run into applications for which is it inadequate. For example, the JIF language [26,25,28] incorporates a policy language based on sets of access control lists (ACLs), augmented with a form of delegation of principal rights. An obvious extension to consider for the JIF policy language is some form of role-based access control (RBAC), including some notion of delegation of role-based rights [7]. But even if one just considers the extension with role-based access control, there are many variations on RBAC that could be added. The RT family of languages by Li et al [22,23] is a good candidate for such a policy language, since it appears to be the closest to a "universal" language for distributed RBAC (i.e., with delegation). But there are numerous variations in the RT family, and presumably more variations still to be proposed. Furthermore there are new policy languages being constantly proposed that go beyond RBAC and delegation, such as policy languages where principals share control of access to resources.

Rather than committing to a single policy language for all applications, an "end-to-end" approach would allow individual applications to define their own policy languages, or to choose from a library of possible policy languages. Rather than committing to a particular policy language in the type system, instead arbitrary policy languages could be loaded, as abstract data types, on a per-application basis. This approach has been used successfully in the past, in several applications: logical frameworks for theorem-proving environments [21,32], certified compilation environments and proof-carrying code [4,36,4,12], and proof-carrying authentication [3,5].

In this article we consider an approach that relates application-defined policy languages to application-defined network security using cryptographic libraries. This approach combines two avenues of study:

(i) Type-based cryptographic operations that used typed APIs to relate the operations to the security properties they are intended to enforce across networks.

(ii) Frameworks for defining languages and logics, particularly the approach of

$$\begin{array}{rcl}
K \in \text{Kind} & ::= & \textsf{Type} \;\mid\; \textsf{Prop} \;\mid\; \textsf{Prin} \;\mid\; k \;\mid\; K_1 \Rightarrow K_2 \\
T, P, F \in \text{Type, Prin, Prop} & ::= & t \;\mid\; tc \;\mid\; T_1 \rightarrow T_2 \;\mid\; \forall t : K.T \;\mid\; (T_1\; T_2) \;\mid\; P \;\textsf{says}\; F \;\mid\; \textsf{rep}\; F \\
e \in \text{Exp} & ::= & x, y, z \;\mid\; a, b, c \;\mid\; \lambda x : T.e \;\mid\; (e_1\; e_2) \;\mid\; \Lambda t : K.e \;\mid\; (e\; T) \;\mid \\
 & & \textsf{case}\; e \;\textsf{of}\; \overline{c(\bar{t}, \overline{x}) \Rightarrow e}
\end{array}$$

Fig. 1. Syntax of Minilanguage

$$\begin{array}{rcl}
\mathit{Th} \in \text{Theory Signature} & ::= & (\{\overline{k}\}, \{\overline{t : \overline{K} \Rightarrow \textsf{Prop}}\},\; \{\overline{c : T_c}\},\; \{\overline{c} : \overline{\forall \bar{t} : \overline{K}.\overline{T} \rightarrow \textsf{rep}\; F}\}, \mathit{CE}) \\
\mathit{CE} \in \text{Constants Env} & ::= & (\{\bar{t} : \overline{K}\}, \{\overline{c} : \overline{\forall \bar{t} : \overline{K}.\overline{T} \rightarrow \textsf{rep}\; P}\}) \\
D \in \text{Datatype} & ::= & \textsf{datatype}\; tc : \overline{K} \Rightarrow \textsf{Type with}\; \overline{c : \forall \bar{t} : \overline{K_c}.\overline{T} \rightarrow tc(\overline{T'})} \\
\mathit{TE} \in \text{Type Env} & ::= & \{\} \;\mid\; \{t : K\} \;\mid\; \mathit{TE}_1 \cup \mathit{TE}_2 \\
\mathit{VE} \in \text{Value Env} & ::= & \{\} \;\mid\; \{x : T\} \;\mid\; \mathit{VE}_1 \cup \mathit{VE}_2
\end{array}$$

$$\begin{array}{ll}
\text{Well-Formed Context} & \overline{\mathit{Th}}, \overline{D}, \mathit{TE}, \mathit{VE} \vdash \textbf{context} \\
\text{Kind Formation} & \overline{\mathit{Th}} \vdash K \\
\text{Type Formation} & \overline{\mathit{Th}}, \overline{D}, \mathit{TE} \vdash T : K \\
\text{Value Formation} & \overline{\mathit{Th}}, \overline{D}, \mathit{TE}, \mathit{VE} \vdash e : T
\end{array}$$

Fig. 2. Syntax of Type Judgements

proof-carrying authentication [3,5].

As a major example, we consider a particular policy language implemented as a library in such a framework. The policy language in question is chosen because it has two disparate implementations, involving quite different sets of cryptographic operations. This demonstrates the wisdom of moving the implementation of the policy language out of the language itself and into libraries. Not only does it appear impractical to expect a single universal policy language that will satisfy the needs of all applications, but it is also questionable if there is a single universal implementation of a policy language that matches all applications.

We present a specification of the metalanguage as an explicitly typed functional language, to simplify the presentation. This allows us to give equational specifications of the semantics of policy languages. We are then able to relate this treatment of policy language implementation to classic work in the algebraic semantics of abstract data types, ensuring that an implementation is correct with respect to an equational specification [17,20]. "Equational correctness" in this setting means that an authentication operation will extract well-formed "evidence" for policy statements from digitally signed credentials.

We give a description of the type system in Sect. 2. The policy language implementation example is provided in Sect. 3–Sect. 6. We relate this to type-based access control and information flow control in Sect. 7. Sect. 8 considers related work while Sect. 9 provides some conclusions.

$$\frac{\overline{Th}, \overline{D}, TE, VE \vdash \textbf{ context} \quad (x : T) \in VE}{\overline{Th}, \overline{D}, TE, VE \vdash x : T} \quad \text{(Val Var)}$$

$$\frac{\overline{Th}, \overline{D}, TE, VE \cup \{x : T_1\} \vdash e : T_2}{\overline{Th}, \overline{D}, TE, VE \vdash (\lambda x : T_1.e) : (T_1 \rightarrow T_2)} \quad \text{(Val Fun)}$$

$$\frac{\overline{Th}, \overline{D}, TE, VE \vdash e_1 : (T_2 \rightarrow T_1) \quad \overline{Th}, \overline{D}, TE, VE \vdash e_2 : T_2}{\overline{Th}, \overline{D}, TE, VE \vdash (e_1\ e_2) : T_1} \quad \text{(Val App)}$$

$$\frac{\overline{Th}, \overline{D}, TE \cup \{t : K\}, VE \vdash e : T}{\overline{Th}, \overline{D}, TE, VE \vdash (\Lambda t : K.e) : (\forall t : K.T)} \quad \text{(Val TFun)}$$

$$\frac{\overline{Th}, \overline{D}, TE, VE \vdash e : (\forall t : K.T) \quad \overline{Th}, \overline{D}, TE \vdash\ T : K}{\overline{Th}, \overline{D}, TE, VE \vdash (e\ T) : \{T/t\}T} \quad \text{(Val TApp)}$$

$$\frac{\begin{array}{c}\overline{Th}, \overline{D}, TE, VE \vdash e : (tc\ \overline{T}) \quad D_i = (\mathsf{datatype}\ tc : \overline{K} \Rightarrow \mathsf{Type\ with}\ \overline{c : \forall \overline{t_1} : \overline{K_1}.\forall \overline{t_2} : \overline{K_2}.\overline{T'} \rightarrow tc(\overline{T''})}) \\ \overline{K} = \overline{K_1} \quad \overline{T} = \overline{\{\overline{T}/\overline{t_1}\}T''} \quad \overline{\overline{Th}, \overline{D}, TE \cup \{\overline{t_2 : K_2}\}, VE \cup \{\overline{x : \{\overline{T}/\overline{t_1}\}T'}\} \vdash e' : T} \quad \{\overline{t_2}\} \cap ftv(T) = \{\}\end{array}}{\overline{Th}, \overline{D}, TE, VE \vdash (\mathsf{case}\ e\ \mathsf{of}\ \overline{c(\overline{t_2}, \overline{x}) \Rightarrow e'}) : T} \quad \text{(Val Case)}$$

Fig. 3. Type System

$$(\lambda x : T.e)v \longrightarrow \{v/x\}e \quad \text{(Red Fun)}$$

$$(\Lambda t : K.e)T \longrightarrow \{T/t\}e \quad \text{(Red TFun)}$$

$$(\mathsf{case}\ c_i(\overline{T}, \overline{v})\ \mathsf{of}\ \overline{c(\overline{t}, \overline{x}) \Rightarrow e}) \longrightarrow \{\overline{T}/\overline{t_i}, \overline{v}/\overline{x_i}\}e_i \quad \text{(Red Case)}$$

$$\frac{e_1 \longrightarrow e_2}{E[e_1] \longrightarrow E[e_2]} \quad \text{(Red Cong)}$$

Fig. 4. Evaluation Rules

2 Type System

The syntax of our mini-language is provided in Fig. 1. We use the notation $\overline{\ldots}$ to denote a sequence, e.g., $\overline{T}$ denotes the sequence of types $T_1, \ldots, T_n$ for some n. The type system comprises a three-level system of values, types and kinds. Kinds are necessary to check the well-formedness of types, because of the richness of the latter. The important forms of types are:

(i) Ordinary types are used to check the well-formedness of values. Besides function and polymorphic types, we include datatypes for basic structuring. These datatypes go beyond ML datatypes in two significant ways: they allow non-regular recursive type descriptions, and they allow free (implicitly existentially quantified) type parameters in the types of data constructors. Both facilities are critical for examples we have developed.

(ii) *Propositions* are the basis of application-specific policy languages. Essentially access control restrictions are stated in terms of propositions, where the propositions themselves are formed using *predicates*. Each predicate is defined in a *signature* for an application-specific policy language.

Kinds include kind constants k, introduced by signatures for policy languages. This is because we model security policies at the type level, and therefore need type-level witnesses for the entities referred to in policies. For example, a policy language might introduce kind constants for principals and roles, say Prin and Role respectively. We add a primitive notion of principals for the purposes of the examples, although as discussed in Sect. 7 there are alternatives for how to represent principals. To support parameterized types and predicates, we also have a kind for type operators $K_1 \Rightarrow K_2$. This also allows the representation of for example parameterized roles.

At the type level, we add a primitive predicate P says F representing access policy "statements" made by principals. There is an interesting issue of how to generate such statements in the language. This involves two issues: how to represent such type-level statements at the value level, and how to ensure the integrity of such statements. For representation, we add a representation type $\mathsf{rep}\ F$ for the value-level representation of the type-level proposition F. This also provides a representation type for value-level representatives of policy-language entities. For example, if P is a principal (at the type level), $\mathsf{rep}\ P$ is the value representation of this principal (for example, a principal may be represented at the value level by its public key).

The value-level language is moderately conventional, aside from the fact that the case construct disguises the use of existential (locally abstract) types. This is crucial , for example, in abstracting the identity of intermediate signing principals in the type of a credential chain. We assume a defined let construct:

$$(\mathsf{let}\ x : T = e_1\ \mathsf{in}\ e_2) \equiv (\lambda x : T.e_2)\ e_1.$$

An application-specific policy language is specified by a *theory signature Th*. The policy language specification includes:

(i) The declaration of new *kind constants* (for example, Role for roles).

(ii) The declaration of type-level *predicate constants*, which should be more properly considered as constructors for forming statements in a metalanguage representation of the policy language.

(iii) The declaration of value-level constructors for the *inference rules* in the policy language, the rules that allow new credentials to be derived from existing credentials.

(iv) The declaration of value-level constructors for representations of the *statements* of the policy language. In general a representation of a proposition F has type $\mathsf{rep}\ F$.

(v) A constants environment CE for value-level representations of entities that are modelled in the policy language. This constants environment includes some collection (possibly infinite) of names $\{\bar{t}\}$ for entities modelled at the type level (for example, names of principals and roles). The constants environment also includes some collection (possibly infinite) of constructors for value-level representations of these entities. In many cases these are simple constants (for example, public keys identify principals and strings identify roles), but in some cases structured representations may be necessary (for example, for parameterized roles). We treat rep as a special type constructor rather than introduce kind polymorphism, which technically is necessary because the form of the argument of rep may range over different kinds (propositions, principals, roles, etc).

It is important to understand the difference between the types F and $\mathsf{rep}\ F$. The former denotes the type of a certificate that verifies that a statement in the policy language is valid. Such a certificate is built using the constructors for the inference rules of the language. The latter denotes the type of a value-level representation for a policy language statement. It is only checked for its well-formedness, not for its veracity.

For the purposes of the example in this article, we add a kind Prin for principals, as well as a proposition form $P\ \mathsf{says}\ F$ for statements in an arbitrary policy language. The rules of a policy language based on this will include rules for combining certificates, so that the combination can be checked and the resulting statement verified. An interesting question then arises: how to generate *primitive* policy language statements while ensuring their integrity. One approach is to rely on private signing keys, e.g., an operation of type

$$\mathit{PrivateKey}(P) \rightarrow (\mathsf{rep}\ F) \rightarrow P\ \mathsf{says}\ F.$$

This is effectively the approach of proof-carrying authentication. We consider another approach, based on type-based access control and information flow control, in Sect. 7.

The forms of the judgements are provided in Fig. 2 and the type rules are provided in Fig. 3. The judgements include several contexts:

(i) There is a collection of *theory signatures* Th.

(ii) There is a collection of datatypes, which for simplicity we assume are declared globally and mutually recursively with theory signatures.

(iii) There is a type environment mapping type parameters to their kinds.

(iv) There is a value environment mapping variables x, y, z and constants a, b, c (which may include cryptographic types) to their types.

Define values in the language by:

$$v ::= x \mid c(\overline{T}, \overline{v}) \mid \lambda x : T.e \mid \Lambda t : K.e$$

Values include constants and expressions constructed using data constructors. Define evaluation contexts by:

$$\begin{aligned} E[\,] ::= [\,] \mid (E[\,]\ e) \mid (v\ E[\,]) \mid (E[\,]\ T)\\ (\mathsf{case}\ E[\,]\ \mathsf{of}\ \overline{c(\overline{t}, \overline{x}) \Rightarrow e}) \end{aligned}$$

The evaluation rules are provided in Fig. 4.

Lemma 2.1 (Subject Reduction) *If* $\overline{\mathit{Th}}, \overline{D}, \mathit{TE}, \mathit{VE} \vdash e : T$ *and* $e \longrightarrow e'$, *then*

$\overline{Th}, \overline{D}, TE, VE \vdash e' : T$.

Lemma 2.2 (Progress) *If* $\overline{Th}, \overline{D}, TE, VE \vdash e : T$*, then there does not exist a sequence of reductions*

$$e = e_0 \longrightarrow \cdots \longrightarrow e_n$$

where e_n *does not contain any redices and is not a value.*

3 Role-Based Policy Language Implementation

Our main example illustrates the general approach of bringing aspects of the policy language outside the trusted computing base (TCB) and implementing them as libraries. As explained in Sect. 1, part of the motivation for this approach is the myriad possibilities for the policy language. Even if one were to fix on a policy language, there may be several possible implementation strategies associated with it. The example in this and the following sections illustrates an end-to-end approach based on the policy language itself being implemented outside the TCB.

General delegation has been criticized in some circles because of the expense of credential chain discovery and of checking credential chains [38]. Cascaded delegation is a more limited form of delegation that only allows credential chains to be extended in one direction [29,37]. Cascaded delegation was explicitly developed for delegation in distributed systems. Role-based cascaded delegation (RBCD) synthesizes role-based access control and cascaded delegation [38].

Two implementation strategies have been proposed for RBCD:

(i) The simple naive approach uses credential chains and a conventional signing algorithm such as RSA or DSA [15,34].

(ii) Motivated by the cost of credential checking with the naive approach, an alternative approach is to use the algorithms for hierarchical certificate-based encryption (HCBE) for signing credential chains [16]. Rather than continually recheck credential chains, this approach allows a chain of signed credentials to be merged (aggregated) into a single credential [38].

It should be emphasized that our approach is completely agnostic on the issue of whether the HCBE approach is in fact faster than the naive RSA approach. The point here is to show how the approach to incorporating policy languages into language-based security can accomodate different languages with alternative implementation strategies.

In what follows, we distill the essence of RBCD, RSA and HCBE into simple functional operations. We refer to the language with the *CascDel* ADT as the source level, and the language into which it is translated as the target level.

4 Role-Based Cascaded Delegation

Fig. 5 provides the specification for the policy language for RBCD. In keeping with the formal system outlined in Sect. 2, a theory signature consists of some collection

$\mathsf{predicate}\ A.r \overset{\text{ap}}{\Longrightarrow} D$
$\mathsf{predicate}\ D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r$
$\mathsf{predicate}\ D_0.priv \overset{\text{dp}}{\Longrightarrow} D$

$\mathsf{rule}\ authorize : (D_0\ \mathsf{says}\ D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r) \to (A\ \mathsf{says}\ A.r \overset{\text{ap}}{\Longrightarrow} D) \to (D_0\ \mathsf{says}\ D_0.priv \overset{\text{dp}}{\Longrightarrow} D)$
$\mathsf{rule}\ delegate : (D_0\ \mathsf{says}\ D_0.priv \overset{\text{dp}}{\Longrightarrow} D) \to (D\ \mathsf{says}\ D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r) \to (D_0\ \mathsf{says}\ D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)$

$\mathsf{type}\ CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)$

$$\begin{aligned}
INITIATE &: (SimpleCert(D_0, (D_0.priv \overset{\text{ar}}{\Longrightarrow} A_1.r_1))) \\
&\quad \to CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A_1.r_1) \\
EXTEND &: CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A_1.r_1) \\
&\quad \to (SimpleCert(A_1, (A_1.r_1 \overset{\text{ap}}{\Longrightarrow} D))) \\
&\quad \to SimpleCert(D, (D_0.priv \overset{\text{ar}}{\Longrightarrow} A_2.r_2)) \\
&\quad \to CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A_2.r_2) \\
SAY &: CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r) \\
&\quad \to (SimpleCert(A, (A.r \overset{\text{ap}}{\Longrightarrow} D))) \\
&\quad \to (D_0\ \mathsf{says}\ (D_0.priv \overset{\text{ap}}{\Longrightarrow} D)) \\
\\
sayCert &: SimpleCert(D, F) \to (D\ \mathsf{says}\ F)
\end{aligned}$$

$$\begin{aligned}
SAY(INITIATE(c), rc) &= authorize(sayCert(c), sayCert(rc)) \\
SAY(EXTEND(ch, rc, c), rc_0) &= authorize(delegate(SAY(ch, rc), sayCert(c)), sayCert(rc_0))
\end{aligned}$$

Fig. 5. Signatures of RBCD Operations

of predicates $\{\overline{t : \overline{K} \Rightarrow \mathsf{Prop}}\}$, specified by predicate clauses in the policy language specification. We define the predicates in the example languuage as infix constructors, since that is how they are normally depicted in the literature. There are several access predicates defined: $A.r \overset{\text{ap}}{\Longrightarrow} D$ for role membership, $D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r$ for granting a privilege $D_0.priv$ (controlled by D_0) to the role $A.r$, and $D_0.priv \overset{\text{dp}}{\Longrightarrow} D$ for granting a privilege to the principal D.

There is also some collection of inference rule constructors $\{\overline{c : \forall t : \overline{K}.\overline{T} \to \mathsf{rep}\ F}\}$, specified by rule clauses in the policy language specification. There are two rules for chaining credentials (in cleartext):

(i) *authorize* grants a privilege to a principal based on a role membership certificate.

(ii) *delegate* delegates a privilege to a role by a principal that has already obtained that privilege.

The *authorize* and *delegate* rules rely on static compile-time credential checking based on type-checking. The correctness of the credentials built using these rules is based on the types of the arguments, which are unsigned cleartext. Credentials based on these rules are amenable to forgery and tampering when transmitted across address spaces. So there should be some way to build credential chains, using cryptographic signing, that prevents these forms of attacks.

$$
\begin{aligned}
&\mathsf{type}\ \mathit{Signed}(F)\\
&\qquad \mathit{sign} : \mathit{PrivateKey}(D) \to (D\ \mathsf{says}\ F) \to \mathit{Signed}(F)\\
&\qquad \mathit{auth} : \mathit{PublicKey}(D) \to \mathit{Signed}(F) \to (D\ \mathsf{says}\ F)\\
&\mathit{auth}(k^{+}, \mathit{sign}(k^{-}, v)) = v
\end{aligned}
$$

Fig. 6. RSA Signing and Authentication

$$
\begin{aligned}
&\mathsf{datatype}\ \mathit{SimpleCert}(D, F)\\
&\mathsf{with}\ \mathit{scert} : (D\ \mathsf{says}\ F) \to \mathit{Signed}(F) \to \mathit{SimpleCert}(D, F)\\
&//\ \text{Derived operations}\\
&\mathit{makeCert} : (D\ \mathsf{says}\ F) \to \mathit{PrivateKey}(D) \to \mathit{SimpleCert}(D, F)\\
&\mathit{authCert} : \mathit{Signed}(F) \to \mathit{PublicKey}(D) \to \mathit{SimpleCert}(D, F)\\
&\mathit{sayCert} : \mathit{SimpleCert}(D, F) \to D\ \mathsf{says}\ F\\
&\mathit{getCert} : \mathit{SimpleCert}(D, F) \to \mathit{Signed}(F)\\
\\
\mathit{makeCert} &= \lambda c.\ \lambda k.\ \mathit{scert}(c, \mathit{sign}(k, c))\\
\mathit{authCert} &= \lambda \mathit{cphtxt}.\ \lambda k.\ \mathsf{let}\ \mathit{clrtxt} = \mathit{auth}(k, \mathit{cphtxt})\ \mathsf{in}\ \mathit{scert}(\mathit{clrtxt}, \mathit{cphtxt})\\
\mathit{sayCert} &= \lambda \mathit{sc}.\ \mathsf{case}\ \mathit{sc}\ \mathsf{of}\ \mathit{scert}(\mathit{clrtxt}, \_) \Rightarrow\ \mathit{clrtxt}\\
\mathit{getCert} &= \lambda \mathit{sc}.\ \mathsf{case}\ \mathit{sc}\ \mathsf{of}\ \mathit{scert}(\_, \mathit{cphtxt}) \Rightarrow\ \mathit{cphtxt}
\end{aligned}
$$

Fig. 7. Simple Certificates

The *CascDel* abstract data type is a specification for credential chains that include ciphertext. This type is parameterized by four arguments, so we write it as $\mathit{CascDel}(D_0.\mathit{priv} \overset{\text{ar}}{\Longrightarrow} A.r)$: it specifies a credential chain that demonstrates the grant of access to $D_0.\mathit{priv}$ to principals in the role $A.r$. There are three operations for this ADT:

(i) *INITIATE* initiates a credential chain based on the inital grant of the privilege to a role.

(ii) *EXTEND* extends such a credential chain, based on a principal in the granted role delegating the privilege to another role.

(iii) *SAY* uses the credential chain, and a role membership certificate for the granted role, to generate a proof that a principal has been granted the privilege.

These operations take credentials as arguments. Providing cleartext credentials of type D says F is insufficient: the combined credentials will typically be bundled up and retransmitted across unsafe networks, and the digital signatures for the original credentials from which the cleartext credentials are derived must be included in this bundling. Therefore instead of cleartext credentials of type D says F, we use combined cleartext and ciphertext credentials of type $\mathit{SimpleCert}(D, F)$. A value of such a type is a pair of a ciphertext credential (with the signing principal elided) and the underlying cleartext credential (with the signing principal D made explicit in the type). It is straightforward to map between ciphertext credentials and these credential pairs during marshalling and unmarshalling.

```
datatype $Chain(D_0.priv, A_1.r_1 \overset{\text{dr}}{\Longrightarrow} A_2.r_2)$
with $emptyChain$ : $Chain(D_0.priv, A.r \overset{\text{dr}}{\Longrightarrow} A.r)$
and $simpleChain$ : $SimpleCert(A_1, A_1.r_1 \overset{\text{ap}}{\Longrightarrow} D)$
                  $\rightarrow SimpleCert(D, D_0.priv \overset{\text{ar}}{\Longrightarrow} A_2.r_2)$
                  $\rightarrow Chain(D_0.priv, A_1.r_1 \overset{\text{dr}}{\Longrightarrow} A_2.r_2)$
and $combineChain$ : $Chain(D_0.priv, A_1.r_1 \overset{\text{dr}}{\Longrightarrow} A.r)$
                   $\rightarrow Chain(D_0.priv, A.r \overset{\text{dr}}{\Longrightarrow} A_2.r_2)$
                   $\rightarrow Chain(D_0.priv, A_1.r_1 \overset{\text{dr}}{\Longrightarrow} A_2.r_2)$
```

Fig. 8. Chains of Role Delegations

```
datatype $RBCD(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)$
with $rbcd$ : $SimpleCert(D_0, D_0.priv \overset{\text{ar}}{\Longrightarrow} A_0.r_0)$
        $\rightarrow Chain(D_0.priv, A_0.r_0 \overset{\text{dr}}{\Longrightarrow} A.r)$
        $\rightarrow RBCD(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)$

$[\![CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)]\!] = RBCD(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)$

$[\![INITIATE]\!] = \lambda c : (SimpleCert(D_0, (D_0.priv \overset{\text{ar}}{\Longrightarrow} A_1.r_1))).$
                 $rbcd(c, emptyChain)$

$[\![EXTEND]\!] = \lambda ch : [\![CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A_1.r_1)]\!].$
               $\lambda rc : (SimpleCert(A_1, (A_1.r_1 \overset{\text{ap}}{\Longrightarrow} D))).$
               $\lambda c : SimpleCert(D, (D_0.priv \overset{\text{ar}}{\Longrightarrow} A_2.r_2)).$
               case $ch$ of $rbcd(cert, chain) \Rightarrow$
                  $rbcd(cert, combineChain(chain, simpleChain(rc, c)))$

$[\![SAY]\!] = \lambda ch : [\![CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)]\!].$
            $\lambda rc : (SimpleCert(A, (A.r \overset{\text{ap}}{\Longrightarrow} D))).$
            $authorize(loop(ch), rc)$

            where $loop = \lambda ch : [\![CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)]\!].$
                          case $ch$ of $rbcd(cert, chain) \Rightarrow loop0(sayCert(cert), chain)$

            and $loop0 = \lambda cert : (D_0$ says $D_0.priv \overset{\text{ar}}{\Longrightarrow} A_0.r_0).$
                         $\lambda chain : Chain(D_0.priv, A_0.r_0 \overset{\text{dr}}{\Longrightarrow} A.r)$
                         case $chain$ of
                           $emptyChain \Rightarrow cert$
                         | $simpleChain(rc_0, c_0) \Rightarrow delegate(authorize(cert,$
                                                                   $sayCert(rc_0)),$
                                                      $sayCert(c_0))$
                         | $combineChain(chn_1, chn_2) \Rightarrow$ let $cert_0 = loop0(cert, chn_1)$ in
                                                      $loop(cert_0, chn_2)$
```

Fig. 9. Translation of RBCD to Chained Delegation

5 Implementation in RSA

Fig. 6 gives the operations for RSA digital signing. The signing operation elides the identity of the principal making a statement F, and authentication re-exposes this principal identity after a runtime cryptographic check. The signing operation maps from a cleartext credential to the ciphertext equivalent, and the authentication operation is a checked operation for mapping back again. The notations k^+ and k^- denote the public and private parts, respectively, of a public-private key pair.

Fig. 7 gives the specification for simple certificates, that bundle a ciphertext credential and its cleartext credential together as a pair. The *makeCert* operation is used by the originator of a cleartext credential to generate the ciphertext equivalent, using his or her private signing key. The *authCert* operation is used by receivers of ciphertext credentials to pair them with their cleartext equivalents, using signature authentication. Finally *sayCert* and *getCert* extract the cleartext and ciphertext parts, respectively, of a simple certificate.

The RSA implementation of RBCD uses chains of digitally signed credentials. The *Chain* datatype is used to implement these chains. In general such a chain of credentials denotes the delegation of access rights (for privilege $D_0.priv$) from role $A_1.r_1$ to role $A_2.r_2$, therefore we denote it as $Chain(D_0.priv, A_1.r_1 \overset{\mathrm{dr}}{\Longrightarrow} A_2.r_2)$.

Fig. 9 provides the implementation of RBCD using chained credentials signed using RSA. The credentials are represented as a pair of the credential granting initial access to a role $A_0.r_0$, and thereafter a chain of delegations to some role $A.r$. There is an implicit use of existential types to elide the intermediate role $A_0.r_0$ in the type. The *INITIATE* operation creates an empty chain, the *EXTEND* operation extends the chain by one more role, and the *SAY* operation recurses over the resulting chain to build a cleartext credential using the inference rules of the access control logic. There is no digital signature checking during the operation of *SAY*; it is assumed that this will have been done using unmarshalling, creating a pair of type $SimpleCert(D, F)$ from a marshalled credential of type $Signed(F)$ using an authentication key of $PublicKey(D)$.

With the addition of (RSA) cryptographic operations, we must allow for stuck computation due to failure of authentication:

Theorem 5.1 (Modified Progress) *If $\overline{Th}, \overline{D}, TE, VE \vdash e : T$, then there does not exist a sequence of reductions*

$$e = e_0 \longrightarrow \cdots \longrightarrow e_n$$

where e_n does not contain any redices and is not a value, or e_n has the form $E[auth(k_1^+, sign(k_2^-, v))]$, where $k_1 \neq k_2$.

As noted, this does not affect the implementations of the RBCD operations, since they do not perform authentication. Authentication is performed in the process of unmarshalling signed certificates and building simple certificates. If simple certificate formation is restricted to the *makeCert* (make a new simple certificate) and *authCert* (build a new simple certificate from ciphertext) operations, then authentication failures are isolated to the latter.

We verify the following by induction on the original type derivation:

Theorem 5.2 (Type Preservation) *If $\overline{Th}, \overline{D}, TE, VE \vdash e : T$ at the source level, then $\overline{Th}, \overline{D}, TE, [\![VE]\!] \vdash [\![e]\!] : [\![T]\!]$ at the target level.*

The following result is a simple verification based on the size of the input to the *loop0* function in the definition of $[\![SAY]\!]$:

Theorem 5.3 (Termination) *The translations of the INITIATE, EXTEND and SAY operations are guaranteed to terminate on all arguments.*

This ensures that the result of the *SAY* operation is a well-formed certificate type $HSigned(F)$

$$\begin{array}{rcl}
hsign & : & HPrivateKey(D_0) \to (D_0 \text{ says } F) \to HSigned(F) \\
hauth & : & HPublicKey(D_0) \to HSigned(F) \to (D_0 \text{ says } F) \\
[\_]^- & : & PrivateKey(D) \to HPrivateKey(D) \\
[\_]^+ & : & PublicKey(D) \to HPublicKey(D) \\
\_@^-\_ & : & HPrivateKey(D_1) \to HPrivateKey(D_2) \to HPrivateKey(D_1) \\
\_@^+\_ & : & HPublicKey(D_1) \to HPublicKey(D_2) \to HPublicKey(D_1) \\
haggregate & : & SimpleCert(D_1, F_1) \\
 & & \to SimpleCert(D_2, F_2) \\
 & & \to ((D_1 \text{ says } F_1) \to (D_2 \text{ says } F_2) \to (D_1 \text{ says } F)) \\
 & & \to SimpleCert(D_1, F)
\end{array}$$

datatype $SimpleCert(D, F)$ with $hcert : (D \text{ says } F) \to HSigned(F) \to SimpleCert(D, F)$

$$\begin{array}{l}
hauth([k_1^+, \ldots, k_n^+]^+, hsign([k_1^-, \ldots, k_n^-]^-, v)) \ = v \\
haggregate(hcert(hsign(k_1, c_1), c_1'), hcert(hsign(k_2, c_2), c_2'), p) \\
\qquad\qquad = hcert(hsign((k_1 @^- k_2), p(c_1, c_2)), p(c_1', c_2'))
\end{array}$$

// Derived operations

$$\begin{array}{rcl}
makeHCert & : & (D \text{ says } F) \to HPrivateKey(D) \to SimpleCert(D, F) \\
authHCert & : & HSigned(F) \to HPublicKey(D) \to SimpleCert(D, F) \\
sayHCert & : & SimpleCert(D, F) \to D \text{ says } F \\
getHCert & : & SimpleCert(D, F) \to HSigned(F)
\end{array}$$

Fig. 10. HCBE Signing, Authentication and Aggregation

$$[\![CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)]\!] = SimpleCert(D_0, D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)$$

$$INITIATE = \lambda c : (SimpleCert(D_0, (D_0.priv \overset{\text{ar}}{\Longrightarrow} A_1.r_1))).\ c$$

$$\begin{array}{rl}
EXTEND = & \lambda ch : [\![CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A_1.r_1)]\!]. \\
 & \lambda rc : (SimpleCert(A_1, (A_1.r_1 \overset{\text{ap}}{\Longrightarrow} D))). \\
 & \lambda c : SimpleCert(D, (D_0.priv \overset{\text{ar}}{\Longrightarrow} A_2.r_2)). \\
 & haggregate(haggregate(ch, rc, authorize), c, delegate)
\end{array}$$

$$\begin{array}{rl}
SAY = & \lambda ch : [\![CascDel(D_0.priv \overset{\text{ar}}{\Longrightarrow} A.r)]\!]. \\
 & \lambda rc : (SimpleCert(A, (A.r \overset{\text{ap}}{\Longrightarrow} D))). \\
 & haggregate(ch, rc, authorize)
\end{array}$$

Fig. 11. Translation of RBCD to HCBE

validating the policy language statement, constructed using the inference rules of the policy language logic.

We verify the following by induction on the size of the credentials chain constructed using the *EXTEND* operation:

Theorem 5.4 *The equations of the CascDel ADT are satisfied by its implementation in RSA.*

6 Implementation Using HCBE

Fig. 10 provides operations based on HCBE for signing. The algorithms for digital signing now are different and therefore the implementation of simple certificates $SimpleCert(D, F)$ is also different. The distinguishing feature of these signing algorithms is the *haggregate* operation that aggregates or merges two signed credentials into one. We define this aggregation operation for simple certificates (pairs of cleartext and ciphertext credentials), rather than just ciphertext credentials, in order to continue doing well-formedness checks on the underlying cleartext credential without having to digitally authenticate. The aggregation operation takes as its third argument an inference rule for combining the underlying cleartext credentials when ciphertext credentials are merged.

The new concept introduced by the HCBE operations is that of combined signing and authentication keys. The aggregation operation of HCBE builds a new ciphertext, from two signed ciphertexts, by signing the underlying cleartexts with the combination of the original private signing keys. The combination of the corresponding public keys is then used to authenticate. An advantage of the HCBE approach over the RSA approach is that the identities of the signing principals, beyond the first principal in the chain, does not need to be revealed, thus providing some notion of privacy in credential chain checking. We represent key combination using operations for injecting simple keys into compound keys ($[\_]^-$ and $[\_]^+$ for private and public keys, respectively), as well as operations for combining compound keys ($\_@^-\_$ and $\_@^+\_$ for private and public keys, respectively). The types of these operations reflect the fact that only the identity of the first signing principal in a chain is revealed by the signatures resulting from these compound keys. We assume the combination operations $@^-$ and $@^+$ are left-associative and abbreviate:

$$[k_1^-, \ldots, k_n^-]^- \equiv [k_1^-]^- @^+ \cdots @^+ [k_n^-]^-$$
$$[k_1^+, \ldots, k_n^+]^+ \equiv [k_1^+]^+ @^+ \cdots @^+ [k_n^+]^+$$

These signing operations are used in Fig. 11 to implement the operations of RBCD. Unlike the RSA implementation, there is no longer any use of credential chains and hence no need for a recursive algorithm to verify a digital credential chain. Instead the aggregation operation of HCBE is used to combine credentials, during the *EXTEND* operation and during the *SAY* operation (combining the last role membership credential with the aggregated credential chain).

Theorem 6.1 (Modified Progress) *If $\overline{Th}, \overline{D}, TE, VE \vdash e : T$, then there does not exist a sequence of reductions*

$$e = e_0 \longrightarrow \cdots \longrightarrow e_n$$

where e_n does not contain any redices and is not a value, or e_n has the form $E[hauth(k_1^+, hsign(k_2^-, v))]$, where k_1 and k_2 are compound keys, and $k_1 \neq k_2$.

As before, verify the following by induction on the original type derivation:

Theorem 6.2 (Type Preservation) *If $\overline{Th}, \overline{D}, TE, VE \vdash e : T$ at the source level, then $\overline{Th}, \overline{D}, TE, [\![VE]\!] \vdash [\![e]\!] : [\![T]\!]$ at the target level.*

Theorem 6.3 (Termination) *The translations of the INITIATE, EXTEND and SAY operations are guaranteed to terminate on all arguments.*

The verification of the following requires induction on the size of the compound keys in the *haggregate* operation that is the underlying representation for a RBCD certificate in HCBE:

Theorem 6.4 *The equations of the CascDel ADT are satisfied by its implementation in RSA.*

7 Relating to Language-Based Information Flow Control

In Sect. 2 we discussed the issue of how to generate primitive policy language statements, of the form P says F, while ensuring the integrity of the statement. Simply creating a data structure of representation type rep $(P$ says $F)$ is obviously insufficient. The approach mentioned in Sect. 2, similar to the approach of proof-carrying authentication, relies on private signing keys for principals to generate these statements. In this section we consider another approach, related to integrity-checking in language-based security.

The JIF language [26,25,28] demonstrates an approach to access-checking and information flow control checking in a programming language type system. In addition to *types*, program variables have *labels* that control (read and write) access to those variables. A label is a set of *policies*. A policy is essentially an access control list, written $\{A : \{B_1, \ldots, B_m\}\}$ allowing access to principals $B_1, \ldots, B_m$. The principal A has control of the policy, in the sense that A may declassify information protected by that policy. JIF allows several distinct security policies to be defined for a program variable, in order to support secure information sharing: Different principals may have independent control on the sharing of information stored in a program variable. In general a program variable has a *labelled type*

$$T^{L_1,L_2}$$

where L_1 is a *secrecy* label, restricting the principals that may read from that variable, while L_2 is an *integrity* label, restricting the principals that may write to that variable.

In proof-carrying authentication, higher-order logic is used as a metalanguage for policy languages. In an object language with the `say` predicate, there is a rule that allows logical conclusions to be derived from signed statements:

for all principals `P`, `(P signed F)` $\Rightarrow$ `(P says F)`

where `F` is a formula in logic. As an alternative, in a language such as JIF with labelled types (or any kind of integrity constraints encoded in the type system), we can specify the following rule in the specification of the theory for a policy language:

$$\forall L.\forall A.\forall B.(\mathsf{rep}\ F^{L,\{A:\{B\}\}} \rightarrow (B\ \mathsf{says}\ F)^{L,\{A:\{B\}\}}.$$

The integrity constraint in the labelled type ensures that this is a valid primitive statement in the policy language. Unlike the aforesaid approaches, it does not rely on cryptographic libraries. Therefore expensive cryptographic signing operations, and some assumptions about key management, are avoided for credential creation and manipulation as long as the credentials stay within a process address space. Going one step further, we may treat P says F as an abbreviation:

$$P\ \mathsf{says}\ F \equiv (\mathsf{rep}\ F)^{\{\},\{P:\{P\}\}}.$$

So a primitive policy language statement F corresponds to a value-level representation for that formula, with integrity checking that the term was generated by (on behalf of) the corresponding principal.

Now we can go one step further and generalize this abbreviation to:

$$L \ \mathsf{says}\ F \equiv (\mathsf{rep}\ F)^{\{\},L}.$$

In this case we generalize the notion of who expresses a policy language statement completely, from some primitive notion of the principal that generated the statement, to a more general notion of any principals allowed to make that statement according to the integrity policy specified by the security label L.

Labels and policies still rely on some notion of principals, at least in the JIF framework. However alternatives may be explored where the form of policies is generalized from ACLs, and where principals then become derived concepts (derived from the access policies). This is an interesting line of further work that we are currently pursuing.

8 Related Work

The motivation for this work has been the need for proper programming abstractions for developing applications that must manage some or all of the task of securing their communication in a network environment. Abadi [1] considers a type system for ensuring that secrecy is preserved in security protocols implemented in that type system. Gordon and Jeffrey [18,19] have developed a type-based approach to verifying authentication protocols. Abadi and Blanchet [2,6] pursue an approach to analyzing security protocols, initially for secrecy properties but later generalizing it to integrity properties. All of these works are focused on verifying secrecy and integrity properties of security protocols. As such the type systems that they use are far more sophisticated than the average programmer will use, while at the same time they give very strong guarantees of secrecy and integrity. The focus of our work is not protocol verification, but building accountable systems: engineering a system where accesses can be logged, but doing it in such a way that the performance of the system is not killed by the demands of credentials checking. So for example we make no attempt to cope with replay attacks.

Other work on security in programming languages has focused on ensuring safety properties of untrusted code [31,30,24] and preventing unwanted security flows in programs [13,27,42,33]. Sabelfeld and Myers [35] provide an excellent overview of work in language-based information-flow security.

As already mentioned, this work essentially combines two strands of work:

(i) Type-based cryptographic APIs that relate cryptographic operations to the security policies that they are intended to enforce [14].

(ii) Frameworks for encoding policy languages (signatures and credential formation rules) so that particular policy languages are placed outside of the TCB [3,5].

Appel, Baujer and Felten's proof-carrying authentication uses higher-order logic to encode the inference rules of policy languages, basing the consistency of the latter on the consistency of the former. Our approach instead is closer to the LF approach of defining policy languages as abstract data types in a theory signature, rather than as encodings in HOL. The system can be viewed as a dependent type system, where types may depend on values. Our approach maintains a "phase distinction" between values and types, so that types do not depend on values, relying on type-level names for value-level entities (e.g., principals), and singleton types that relate the two. Part of the motivation for this is that, for purposes that go beyond the current account [10], it is useful to have some notion of defined equality at the type level (equating principal names, for example), and for this it appears preferable to keep the levels distinct.

As discussed, we are interested in relating this approach to type-based approaches to enforcing security policies, such as for example the decentralized label model of the JIF language [26,25,28]. Distributed versions of JIF have been implemented. The J/Split system [43] partitions a sequential JIF program into a distributed system, where portions of the program run on hosts that are trusted for the principals for whom the code runs. For two mutually distrustful principals engaged in a distributed game, a trusted third party (the board) is responsible for communicating data from one party to the other. Network security (cryptographic operations) is implicitly part of the TCB. Chothia et al [9] introduce Key-Based Decentralized Label Model (KDLM), which combines the idea of typed-based cryptographic APIs expressing security properties, with many of the ideas of the JIF type system. However they only consider a policy language involving ACLs. Tse and Zdancewic [40,39] consider a language based on certificate-based declassification. This is essentially a variation of the JIF policy language (based on ACLs and delegation of principal rights): rather than defining a policy as an ACL, it is defined by a set of certificates given out for that policy that grant access to principals. Although they report that their first-class principals can be mapped down to PKI, it appears that network security remains part of the language runtime. They provide a monadic semantics that allows them to reason modularly about non-interference for various extensions of the language.

9 Conclusions

We have described an approach for building distributed implementations of policy languages over untrusted networks, building on trusted cryptographic libraries and using the language type system to carry correctness guarantees through the implementation. At the same time, we have deliberately omitted aspects of the implementations, for the purpose of simplifying the exposition. We deliberately treat the cryptographic operations as black boxes and do not consider attacks based on exploiting properties of the algorithms. So this approach is very much based on Dolev-Yao assumptions about the algorithms. Additionally we do not consider network attacks based on protocol weaknesses, for example, exploiting attacks based on repeated uses of nonces. There has been a great deal of good work on type-based approaches to security protocol verification [18,19], essentially defining domain-specific languages in which it is impossible to implement incorrect protocols. These approaches make explicit notions of nonces, with type systems that prevent their repeated use and track correspondence assertions to verify aspects of the protocols. It appears to be plausible that our approach can be combined with these other approaches, at the cost of some increase in complexity (primarily the addition of effect types). The main conceptual overhead in this approach is a ubiquitious use of parameterized types to model security policies in the type system. Since there is successful experience with the use of genericity in ML, Haskell, Ada, increasingly Java generics and to some extent C++ templates, we are confident are our approach could be adopted by reasonably competent software developers. We are in the process of implementing this approach to test this assertion.

An obvious question that arises, is how the correctness of the implementations relates to security guarantees in programs that use these library implementations. The implementation correctness guarantees well-formed certificates in the policy language for a particular application. Our approach is to relate these to statements to information flow guarantees based on a security type system, as alluded to in Sect. 7. The trustworthiness of the statements then justifies some modifications to the standard notions of non-interference underlying information flow control We intend to report further on this in a subsequent article.

The equational specifications are relatively simple and can be implemented as rewrite systems in environments such as ASF, ELAN and Maude [41,8,11]. However encoding the implementations in these environments should not miss an important part of the encoding, the typing that ensures that "cleartext" credentials are well-formed. A more interesting use of a rewriting environment would be to augment the equational specifications of cryptographic operations with specifications of security protocols, for example for key exchange. This would yield stronger security properties for policy language implementations than we consider in this paper. This is an interesting avenue that we intend to pursue for future work.

Another issue is generalizing the forms of specifications of these abstract data types, to include not only the semantics of the cryptographic operations but also the semantics of protocols in which these operations are used. This appears to be a potentially fruitful application of on-going research in rewriting-based techniques for reasoning about security policies.

References

[1] Abadi, M., *Secrecy by typing in security protocols*, in: *Theoretical Aspects of Computer Science*, 1997, pp. 611–638.

[2] Abadi, M. and B. Blanchet, *Analyzing security protocols with secrecy types and logic programs*, in: *Proceedings of ACM Symposium on Principles of Programming Languages*, 2002, pp. 33–44.

[3] Appel, A. and E. Felten, *Proof-carrying authentication*, in: *ACM Symposium on Security*, 2000.

[4] Appel, A. W., *Foundational proof-carrying code*, in: *Proceedings of IEEE Symposium on Logic in Computer Science*, 2001.

[5] Bauer, L., "Access Control for the Web via Proof-carrying Authorization," Ph.D. thesis, Princeton University (2003).

[6] Blanchet, B., *From secrecy to authenticity in security protocols*, in: *9th International Static Analysis Symposium (SAS'02)*, 2002, pp. 242–259.

[7] Blaze, M., J. Feigenbaum and J. Lacy, *Decentralized trust management*, in: *IEEE Symposium on Security and Privacy*, 1996.

[8] Borovansky, P., C. Kirchner, H. Kirchner, P. E. Moreau and M. Vittek, *ELAN: A logical framework based on computational systems*, in: *Proc. of the First Int. Workshop on Rewriting Logic*, Lecture Notes in Computer Science **4** (1996).

[9] Chothia, T., D. Duggan and J. Vitek, *Type-based distributed access control*, in: *Computer Security Foundations Workshop* (2003).

[10] Chothia, T., D. Duggan and J. Vitek, *Principals, policies and keys in a secure distributed programming language*, in: *Foundations of Computer Security*, 2004.

[11] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada, *The Maude system*, Lecture Notes in Computer Science **1631** (1999), pp. 240–243.

[12] Crary, K. and S. Sarkar, *Foundational certified code in a metalogical framework*, in: *Conference on Automated Deduction*, 2003.

[13] Denning, D. E. and P. J. Denning, *Certification of programs for secure information flow*, Communications of the ACM (1977).

[14] Duggan, D., *Type-based cryptographic operations*, Journal of Computer Security (2003).

[15] FIPS, *Fips 186-2 digital signature standard*, Technical report (2000).

[16] Gentry, C., *Certificate-based encryption and the certificate revocation problem*, in: *Eurocrypt'03*, 2003, pp. 272–293.

[17] Goguen, J., *An initial algebra approach to the specification, correctness and implementation of abstract data types*, Current Trends in Programming Methodology **4: Data Structuring**, Prentice-Hall, Englewood Cliffs, New Jersey, 1978 pp. 80–149.

[18] Gordon, A. D. and A. Jeffrey, *Authenticity by typing for security protocols*, in: *IEEE Computer Security Foundations Workshop (CSFW)*, 2001.

[19] Gordon, A. D. and A. Jeffrey, *Types and effects for asymmetric cryptographic protocols*, in: *IEEE Computer Security Foundations Workshop (CSFW)*, 2002.

[20] Guttag, J. and J. Horning, *The algebraic specification of abstract data types*, Acta Informatica **10** (1978), pp. 27–52.

[21] Harper, R., F. Honsell and G. Plotkin, *A framework for defining logics*, Journal of the ACM **40** (1993), pp. 143–184.

[22] Li, N., J. C. Mitchell and W. H. Winsborough, *Beyond proof-of-compliance: Security analysis in trust management*, Journal of the ACM (2005), to appear.

[23] Li, N., W. H. Winsborough and J. C. Mitchell, *Distributed credential chain discovery in trust management*, Journal of Computer Security **11** (2003), pp. 35–86.

[24] Morrisett, G., D. Walker, K. Crary and N. Glew, *From System F to typed assembly language*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 528–569.

[25] Myers, A. C., *Jflow: Practical mostly-static information flow control*, in: *Proceedings of ACM Symposium on Principles of Programming Languages*, 1999, pp. 228–241.

[26] Myers, A. C. and B. Liskov, *A decentralized model for information flow control*, in: *Symposium on Operating Systems Principles*, 1997.

[27] Myers, A. C. and B. Liskov, *Complete, safe information flow with decentralized labels*, in: *IEEE Symposium on Security and Privacy*, 1998.

[28] Myers, A. C. and B. Liskov, *Protecting privacy using the decentralized label model*, ACM Transactions on Software Engineering and Methodology **9** (2000).

[29] Nagaratnam, N. and D. Lea, *Secure delegation for distributed object environments*, in: *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998.

[30] Necula, G., *Proof-carrying code*, in: *ACM Symposium on Principles of Programming Languages*, 1997, pp. 106–119.

[31] Necula, G. and P. Lee, *Safe kernel extensions without run-time checking*, in: *Operating Systems Design and Implementation*, 1996.

[32] Pfenning, F., *Logic programming in the LF logical framework*, in: G. Huet and G. Plotkin, editors, *Logical Frameworks*, Cambridge University Press, 1990 pp. 149–181.

[33] Pottier, F. and S. Conchon, *Information flow inference for free*, in: *Proceedings of ACM International Conference on Functional Programming*, 2000.

[34] Rivest, R., A. Shamir and L. Adleman, *A method for obtaining digital signatures and public key cryptosystems*, Communications of the ACM **21** (1978), pp. 120–126.

[35] Sabelfeld, A. and A. Myers, *Language-based information-flow security*, IEEE Journal on Selected Areas in Communications (2002).

[36] Shao, Z., V. Trifonov, B. Saha, and N. Papaspyrou, *A type system for certified binaries*, ACM Transactions on Programming Languages and Systems **27** (2005), pp. 1–45.

[37] Sollins, K. R., *Cascaded authentication*, in: *IEEE Symposium on Security and Privacy*, 1988, pp. 156–163.

[38] Tamassia, R., D. Yao and W. Winsborough, *Role-based cascaded delegation*, in: *SACMAT;04*, IBM Yorktown Heights, New York, USA, 2004.

[39] Tse, S. and S. Zdancewic, *Run-time principals in information-flow type systems*, in: *IEEE Symposium on Security and Privacy*, 2004.

[40] Tse, S. and S. Zdancewic, *Designing a security-typed language with certificate-based declassification*, in: *European Symposium on Programming*, 2005.

[41] van den Brand, M., A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser and J. Visser, *The ASF+SDF meta-environment: A component-based language development environment*, in: *Computational Complexity*, Lecture Notes in Computer Science, 2001, pp. 365–370.

[42] Volpano, D. and G. Smith, *A type-based approach to program security*, in: *Proceedings of the International Joint Conference on Theory and Practice of Software Development* (1997).

[43] Zdancewic, S., L. Zheng, N. Nystrom and A. C. Myers, *Secure program partitioning*, Transactions on Computer Systems **20** (2002), pp. 283–328.