Type-Based Distributed Access Control

(Extended Abstract)

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

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

Jan Vitek
Dept of Computer Sciences
Purdue University
West Lafayette, IN, USA.

Abstract

The Key-Based Decentralized Label Model (KDLM) is a type system that combines a weak form of information flow control, termed distributed access control in the article, with typed cryptographic operations. The motivation is to have a type system that ensures access control while giving the application the responsibility to secure network communications, and to do this safely. KDLM introduces the notion of declassification certificates *to support the declassification of encrypted data.*

1. Introduction

The notion of *distributed access control* defined in this paper is a weak form of information flow control that associates access restrictions to data

Like information flow control, it is important that distributed access control be enforced statically, at compile-time, though for performance reasons in the case of distributed access control[1]. The motivation for distributed access control is to enforce *accountability*, tracking which principals are responsible for allowing access to particular data.

A challenge with static checking of access control is how to reconcile type-based access control specifications with application-based network security. In current systems that do this static enforcement, network communications are secured by hiding the network inside the trusted computing base (TCB), using remote method invocation (RMI) or some similar mechanism to make the network transparent. In general this is unsatisfactory for the following three reasons: (*i*) The TCB is greatly enlarged by pushing the middleware into it, increasing the risk of security flaws due to bugs in the TCB code. (*ii*) In Internet programming, there is some skepticism that the network can be made transparent, and many systems (e.g. some banking systems) that are required to be secure and fault-tolerant are implemented using lower-level mechanisms than RMI because of this. (*iii*) Relying on the TCB to secure all communications violates the end-to-end principle in systems design. In many cases it will be more appropriate to have the application ensure network security rather than relying on the communication system.

The contribution of this paper is to show how network security can be moved out of the TCB into the application, in a system where distributed access control is enforced through the type system. All current approaches to statically checked distributed access control and information flow control require the network to be secured in the TCB. In short we want to:

1. Use the type system to enforce access control, including after we have handed off data to others, in a distributed environment.

2. Allow applications to secure network communication themselves using cryptographic techniques.

3. Ensure that the requirements of the access control specifications are not violated by the application-based network security (without considering covert channels).

1.1. Distributed Access Control

The traditional approach to access control is to provide the data on a database on the network, restrict access using access control lists (ACLs), and use authentication to verify the allowability of access requests from remote principals. An example of this is the Taos operating system [8, 7]. In recent years a trend has been to replace or supplement ACLs with delegation certificates, which do not require all accessing principals to be named in an ACL, but puts some limit on the extent to which access rights can be delegated.

[1]Distributed access control is completely impractical, as explained below, unless some large part of it can be done statically, with dynamic checking left to the point where data is unmarshalled.

An example of this is the Simple Public Key Infrastructure (SPKI) [16], as well as various systems for decentralized trust management (PolicyMaker, Keynote, etc) [11, 10].

We refer to these approaches to access control as *local access control*. It is local in the sense that the data being protected resides on a server somewhere on the network, and a reference monitor mediates access requests to that data based on an ACL and credentials authenticating the accessor, or based on authorization certificates delegated to the accessor. Essentially the accessor has a secure pointer (a capability) pointing to where the data is located on the network. The accessor can retrieve the data over the network using the secure pointer, and access control is enforced locally at the server.

We advocate an alternative approach to access control in a distributed system, *distributed access control*. With this approach, data may not reside centrally on some server. Rather the data itself is copied through the network, transmitted by applications through communication channels. Access control is distributed because it is not enforced at one point in the network where the data is accessed, but may be enforced at myriad points in the network where a copy of the data is accessed. Although there are several possible motivations for this approach, the particular motivation we exploit is *accountability*. The data is stored with certain access rights associated with it. If the data should be leaked, there should be a means for tracking backwards to the source of the leak. If the data is simply provided to the accessor and then forgotten about, there is no way to perform this forensic investigation. With distributed access control, on the other hand, the data is copied to the accessor along with a full specification of the access restrictions. If the accessor provides the data, or some result based on the data, to another party, then the justification for this is recorded for subsequent diagnosis.

1.2. Assumptions about the Network Environment

With enough assumptions about the network environment, distributed access control and accountability are trivial. On the other hand, such assumptions may entail requirements on the operating environment that are so onerous to be for all practical purposes unimplementable. In this work we make the following assumptions:

1. *Intermittent connectivity.* Communications may be sufficiently erratic that the application itself may need to handle the task of transferring data across administrative boundaries. Access decisions may need to be performed at a different network location from where the data was originally stored, *e.g* data cached on a disconnected machine should remain protected by access control policies.

2. *Trusted hosts.* We assume that there is sufficient control over the local operating environments to enforce access restrictions. In an implementation we could for example assume that the applications run on virtual machines that have not been tampered with.

3. *Insecure network communication.* We allow arbitrary hostile hosts to interfere with traffic between secure hosts. We expose the insecurity of the network to the application and require the application to deal with it. The environment to prevent secure data from being accidentally leaked on insecure communication channels. We do not consider covert communication channels, our motivation is purely to build an audit trail for finding leaks due to improper declassification.

4. *No PKI assumptions.* There is no a single "one-size fits all" approach to PKI. While commercial platforms offer systems such as X509v3 and SPKI, other infrastructures are sure to emerge. We concentrate instead on providing a typed framework in which applications are responsible for safe key distribution. The operations for safe key distribution are a central part of the contribution of this paper.

2. Informal Motivation

We consider a core language for secure distributed systems equipped with primitives for encryption and digital signing. These primitives are needed to ensure the secrecy of data, including keys, communicated over the network and for ensuring integrity of messages. We provide a type system that expresses precisely the secrecy and integrity properties that these operations are intended to ensure.

2.1. DLM: Decentralized Label Model

There is already a proposal for a type system that enforces secrecy and integrity in the type system [25]. We call this approach the "Decentralized Label Model" (DLM). Our approach shares much of its basic design with DLM: types include labels that enforce secrecy and integrity policies. However whereas DLM enforces all policies based on principals, in our system we add a notion of *key names*, and all enforcement is mediated through key names. The format of labels is different from DLM to reflect this change of perspective. The motivation for this approach is exactly to relate type-passed secrecy and integrity to the safe use of cryptographic operations. We refer to our system as Key-Based DLM or KDLM.

In DLM a labelled type has the form $[T]^{L_1,L_2}$, where T is a simple type, L_1 is the label restricting read access to the data (who can consume it), and L_2 is the label restricting right access to the data (who can produce it). Each label (in the DLM) is effectively a set of access control lists (ACLs), each one "owned" or "controlled" by a principal and allowing read or write access to the principals listed in the ACL. If the secrecy label L_1 has the form

$$\{P_1 : \{\overline{P_1}\}, \ldots, P_m : \{\overline{P_m}\}\}$$

then a principal P has read access to the data (can access the data) only if

$$P \in \bigcap_{i=1}^{m} \{\overline{P_i}\}.$$

Similarly if the integrity label L_2 has the above form, then a principal P has write access to the data (can produce the data) only if the above membership condition is true. One way of viewing the DLM is as a decentralized form of information flow control, where the type system no longer relies on the centralized definition of labels ("high," "low," and so on). Decentralized labels also support a controlled form of *declassification*: the owner of an ACL may extend that ACL, so that with the cooperation of all of the owners of the policies in the label, a principal may be added to those allowed to read or write the protected data.

2.2. KDLM: Key-Based DLM

The central idea in KDLM is to add *key names* to the type system. A key name has a "type" that, similarly to an ACL in a label in DLM, identifies an owner principal and a set of principals that have access to protected data. A key name may be either for encryption or for signing. Each key name has an associated public-private pair of cryptographic keys. Here *key names* are entities in the type system, while *cryptographic keys* are values used for public-key cryptography. The "type" of the key name constrains which principals can access the private key for that key name. The private key in turn has a secrecy label that cannot allow access to any principals outside those listed in the key name's ACL.

Key names are purely compile-time entities used for static checking and are stripped along with all type information before a program runs. The "type" of a key name is a *kind*, a form of type for type-level expressions. An encryption key name K that is generated by the principal P and is accessible to principals $P_1 \ldots P_m$ has the kind

$$K : \mathsf{EKey}_F(P : P_1 \ldots P_m)$$

The flag F indicates whether this is a *virtual* or *actual* key name. Virtual keys are sufficient for purely static access checks, avoiding unnecessary public-private key generation; they are also necessary for technical reasons. Actual key names are represented by cryptographic keys written a. We assume that the set of primitive values is partitioned into at least five sets: the set of channel names (for message passing), the set of encryption keys, the set of decryption keys, the set of signing keys and the set of authentication keys. Furthermore we assume that for every encryption key there is exactly one other key, a decryption key, that is its inverse with respect to cryptographic encryption/decryption. Similarly for every signing key there is exactly one authentication key that is its inverse. We denote the public and private parts of such a key pair by a^+ and a^-, respectively. Then for the encryption key name K above, we have the typings:

$$a^+ : [\mathsf{EncKey}(K)]^{L_1, L_1'}, \qquad a^- : [\mathsf{DecKey}(K)]^{L_2, L_2'}$$

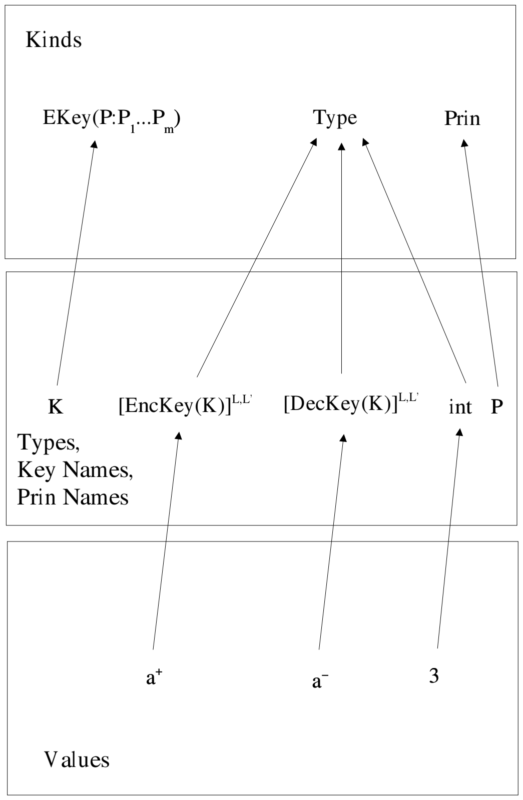

Figure 1. Values, types and kinds

The kind of the key name K enforces the restriction that the secrecy label L_2 of the private key a^- cannot allow any principal outside of $P_1 \ldots P_m$ to access the key. A diagram illustrating the relationships between values, types and kinds is provided in Fig. 1. We add integers for comparison. We emphasize that key names K and principal names P are purely compile-time entities, and only the actual public and private keys, a^+ and a^- respectively, exist at run-time. This stratification into values, types and kinds enforces this "phase distinction."

Purely static access control systems are too constraining for many practical applications. Declassification is a way to circumvent the rigidity of static type checking at the cost of potential security breaches. KDLM considers the issue of declassifying private keys. The constraints of key names prevent the granting of a private key to any principal outside those originally allowed in the key name's ACL. In KDLM we allow declassification by permitting the "owner" of the key name to issue a *declassification certificate*, of type

$$[K' \text{ reclassifies } K]^{L_3,L_3'}.$$

While the private key cannot be accessed by any principal outside the original set $P_1,\ldots,P_m$, the owner principal P may issue a declassification certificate that is accessible by some other principal. We describe below how these declassification certificates are then used to declassify data.

Declassification may appear to be an unnecessary complication, but in fact it is the most fundamental operation in our language, and leads to the most profound design decisions in the type system. When we combine labelled types with cryptographic operations, then we must consider the following question[2]:

> Should it be possible to declassify encrypted data, making it available to principals that did not have access to it beforehand, without having to decrypt the data, declassify it, and then re-encrypt under another key?

If the answer is no, then we can give our language a simple type system in which the principals mentioned in the decryption key name always are a subset of the ones allowed to encrypt the data. We take the view that this is too restrictive. For example a log server may archive an large number of messages encrypted under different keys depending on the sender, with the server not even necessarily a part of the applications that generated the messages. As a result, the log server will certainly have no part in decrypting messages. Yet if a new principal joins the system and wants to retrieve a particular message, some principal authorized to access that message will have to retrieve it from the server, decrypt it, and forward it re-encrypted under a new key. Without key declassification, the access control system is thus placing unreasonable constraints on the architecture and communication patterns of such distributed systems.

So we allow key declassification, via the mechanism of allowing principals to issue declassification certificates of the form K' reclassifies K, signifying that a principal with access to the decryption key for K' can decrypt values encrypted with the encrypted key for K. This is provided, of course, that the principal has access to the certificate. Furthermore the decrypted data is implicitly declassified at the point of decryption.

[2] Since everything that is said about secrecy and encryption can be "dualized" to integrity and digital signing, we only consider the former in our discussion.

Now we must ensure that the combination of encryption, key declassification, and then decryption cannot be used to perform an undesired declassification. Consider the following scenario:

1. There is a value of type $[T]^{L_1,L_1'}$. The principal P is not on the list of principals allowed to access this data (according to the secrecy label L_1).
2. There is an encryption key a^+ of (unlabelled) type $\mathsf{EncKey}(K)$, accessible to all. The kind of K is compatible with the secrecy label L_1, so a^+ can be used to encrypt the data.
3. There is a declassification certificate declassifying the key name K to some key name K'. This declassification certificate, and therefore the decryption key for K', are accessible to the principal P.
4. Putting this together, P can now declassify the data by encrypting it with the encryption key for K, then decrypting it with the declassification certificate.

The declassification certificate allows *any* data whose secrecy label is compatible with the encrypting key (i.e., where the value's secrecy label gives access to any principal that has access to the corresponding decrypting key) to be declassified.

So we must focus on what is the correct notion of "compatibility" between a secrecy label and a key name (identifying a public-private key pair for encryption). DLM is based on *structural equivalence* of labels and policies, that is, labels L and L' are equivalent if they are similar structurally, up to some obvious rearranging of terms. The analogy is with languages such as ML and Modula-3 where type equivalence is structural. Extending this structural notion of compability to key names would imply that a key can be used to encrypt data, provided that the set of principals with access to the key name's private key is included in the set of principals allowed access by the data's secrecy label.

However this notion of structural compability between key names and labels is too promiscuous. Suppose a principal has values v_1 and v_2 with structurally equivalent secrecy labels. The principal encrypts v_1 with a key with name K, then later declassifies it to K'. However the principal does not want the value v_2 to be subsequently declassified via the same route.

Declassification of keys creates a connection between key names K and K', delegating decryption rights for K to K'. In the DLM, there must then be a connection between the encryption key a^+ for K and secrecy labels for values that can be encrypted with a^+, and a connection between the decryption key b^- for K and secrecy labels for values that result from decrypting with a^-. This latter connection is necessary to allow the principal decrypting with the declassification certificate to access the data. The first connection is based on name inclusion (of key names). If the latter two connections are based on (a semantic form of) structural inclusion, then this will allow declassification certificates to be used to perform unintended declassification.

The alternative to structural equivalence is *name equivalence*. Name equivalence for types means that two types must have the same name; differently named types with the same structure are not considered equivalent. Examples include almost all popular languages, including C and Java. In KDLM, we avoid the above problems with unintended declassifications by removing some of the structural equivalence and inclusion between labels, but not all of it. To see the need for some form of structural equivalence, consider a hypothetical code fragment:

```
(e) ?  foo() :  bar()
```

Suppose `foo` and `bar` are procedures provided by different principals, with secrecy labels L_f and L_b respectively for their return types. If label inclusion is strictly name-based, then the result of this conditional must have a secrecy label L that includes both L_f and L_b. If label inclusions are name-based and therefore must be defined, who is responsible for this? Clearly the principal executing the conditional cannot, otherwise they would have the ability to declassify data provided by `foo` and `bar`. If we allow the executing principal to add this inclusion based on some structural containment condition, then we are back where we started, with the DLM. The principals for `foo` and `bar` cannot do this classification, since they know nothing of L (in general). This motivates keeping labels as *sets*, and the label on the result of the conditional will contain the label $L_f \cup L_b$.

We introduce name equivalence, but not name inclusion, on policies. In fact we already have a mechanism for naming policies: key names! Recall that for example for encryption keys, the kind of a key name identifies the principal that generated the key, and also identifies the set of principals that can access the decryption key. Our type system will have two forms of key names, actual and virtual. The former have associated encryption and decryption keys, the latter do not. The latter can be used for static access checking, while the latter are used when data must be encrypted (or of course signed, for integrity) for transmission over the network.

In general a type has the form $[T]^{L,L'}$ where L is a set of encryption key names and L' a set of signing key names. If a secrecy label L has the form $\{K_1,\ldots,K_m\}$ where we have the kindings

$$K_1 : \mathsf{EKey}_F(P_1 : \overline{P_1}), \ldots, K_m : \mathsf{EKey}_F(P_m : \overline{P_m}),$$

then a process executing for principal P cannot access this value unless $P \in \bigcap_{i=1}^{m}\{\overline{P_i}\}$. This is similar to the read access restriction under the DLM, except that our approach adds the extra level of indirection through the key names and their kinds.

To encrypt a value with this secrecy label, one must have public keys for all of the key names in L, of types

$$[\mathsf{EncKey}(K_1)]^{L_1,L_1'}, \ldots, [\mathsf{EncKey}(K_m)]^{L_m,L_m'}.$$

To decrypt the resulting value, a process executing for principal P' must have access to declassification certificates, declassifying the key names $\overline{K_m}$ under which the data is encrypted to key names $\overline{K_m'}$, so P' must have declassification certificates of types

$$[K_1' \text{ reclassifies } K_1]^{L_1'',L_1'''}, \ldots, [K_m' \text{ reclassifies } K_m]^{L_1'',L_m'''}.$$

Decrypting the value with these certificates produces a value with secrecy label $\{K_1',\ldots,K_m'\}$, and the process for P' has read access to this value if $P' \in \bigcap_{i=1}^{m}\{\overline{P_i'}\}$, where each K_i' has kind $\mathsf{EKey}_{\mathsf{Actual}}(P_i' : \overline{P_i'})$.

The type system requires that every secret key be secured by the secrecy label on its type, so that it cannot be accidentally leaked to any principals outside of those specified in the kind of its key name. Every secrecy label for a secret key must therefore contain one or more other secrecy key names, that must have already been defined before the secret key's own key name was introduced. There is the danger of an infinite regress here: how does one define the first key? This is part of the motivation for virtual key names: since such key names do not have any associated public-private key pairs, they can be introduced without the need for any other secrecy key names to protect their private keys. The other motivation for virtual key names is convenience and performance: we should not have to endure the expense of generating new public-private key pairs every time we introduce new key names that will only be used for statically checked access control.

Declassification certificates have two applications: The first of which is of course decrypting values that were encrypted under the declassified key. The other application is to declassify a value that has not been encrypted. Our language includes a declassify operation that requires a declassification certificate. When a value is so declassified, the result is annotated with the declassification certificate. This has both practical and theoretical motivation. It supports our original motivation for distributed access control, of building an audit trail; and it is necessary in order to ensure that types are preserved during computation.

3. Type System: Formal Description

We now provide a more detailed description of the type system. The syntax of types and kinds is provided in Fig. 2.

$A \in$ Arity, Kind	$::=$	Prin	Principal
	$\mid$	$\mathsf{EKey}_F(P : \overline{P})$	Encryption key
	$\mid$	$\mathsf{SKey}_F(P : \overline{P})$	Signing key
	$\mid$	Type	Type
$F \in$ Key Flag	$::=$	Actual	Actual key
	$\mid$	$\mathsf{Virtual}$	Virtual key
$K, P, T \in$ Key, Prin, Type	$::=$	k, p, t	Variable, name
	$\mid$	$\mathsf{DecKey}(K)$	Decryption key type
	$\mid$	$\mathsf{EncKey}(K)$	Encryption key type
	$\mid$	$\mathsf{AuthKey}(K)$	Authentication key type
	$\mid$	$\mathsf{SignKey}(K)$	Signing key type
	$\mid$	K_1 reclassifies K_2	Reclassification cert
	$\mid$	$\mathcal{E}\{LT\}$	Actually encrypted
	$\mid$	$\mathcal{S}\{LT\}$	Actually signed
	$\mid$	$\mathsf{Chan}(LT)$	Channel type
	$\mid$	$\langle t : A\rangle LT$	Package
$L \in$ Label	$::=$	$\{K_1, \ldots, K_m\}$	
$LT \in$ Labelled type	$::=$	$[T]^{L_1,L_2}$	

Figure 2. Syntax of Kinds and Types

The system of arities or kinds organizes the well-formed types, principal names and key names. Besides the kind of types (Type) and principal names (Prin), there are two kinds for key names: encryption (secrecy) key names $\mathsf{EKey}_F(P : \overline{P})$ and signing (integrity) key names $\mathsf{SKey}_F(P : \overline{P})$. This kind identifies the creator of the key (P) and the principals who may have access to the private part of the public-private key pair for that key ($\{\overline{P}\}$). Every key name kind also has a flag indicating if it is a virtual key (static checking only) or an actual key (dynamic checking may also be performed, using cryptographic operations).

The types then consist of key types, indexed by key names, the types of reclassification certificates, and the types of actually (i.e., cryptographically) encrypted or signed values, $\mathcal{E}\{LT\}$ and $\mathcal{S}\{LT\}$. The types are rounded out by the other (non-cryptographic) types of the base language, in this case channel types and package types. The latter are also known as *existential types* [22]; they are practically necessary for transmitting principal and key information when principal and key names are static second-class entities, at the same level as types. For example to send an encryption key to a process that requires an encryption key created by the principal *root*, no matter the underlying key name, we send it a value of type[3]

$$\langle k : \mathsf{EKey}_F(root : (root))\rangle[\mathsf{EncKey}(k)]^{L_1,L_2}.$$

Kinds and types depend on each other, resulting in a dependent kind system. The formation rules for kinds and types must therefore be formulated in a mutually recursive fashion, and these formation rules intertwined with the rules for environment formation in the type system. An environment is a sequence of pairs, binding (variables or names) to (kinds or types). We have two forms of environments:

$$\begin{array}{rcl} TE \in \text{Type Env} & ::= & \varepsilon \mid (t : A) \mid TE_1, TE_2 \\ VE \in \text{Value Env} & ::= & \varepsilon \mid (x : LT) \mid (a : LT) \mid VE_1, VE_2 \end{array}$$

The sequence concatenation operation (,) is assumed to be associative with ε (empty sequence) as its unit.

The type environment TE is used in the definition of a metafunction that, given a label (set of key names), computes the set of principals that is defined by that label in that

[3]Even this trivial example suggests a need for some notion of parameterized types (generics) for parameterizing values by key names and principal names, and even sets of principal names because of key name kinds. Presumably some notion of bounded type quantification would also be useful. However we do not consider this avenue any further in this paper.

$e \in$ Expression	$::=$	w, x, y, z	Variable
	$\mid$	a, b, c, n	Name (Channel, Key)
	$\mid$	$\mathsf{newKey}\langle k : A \rangle \{e\}$	New virtual key
	$\mid$	$\mathsf{newKey}\langle k : A \rangle (a^{+} : LT_1, a^{-} : LT_2)\{e\}$	New actual key
	$\mid$	$\mathsf{reclassifyCert}_{K_1,K_2}()$	Reclassify cert (virtual key)
	$\mid$	$\mathsf{reclassifyCert}_{K_1,K_2}(e)$	Reclassify cert (actual key)
	$\mid$	$\mathsf{chain}_{K_1,K_2,K_3}(e_1, e_2)$	Chain certs
	$\mid$	$\mathsf{reclassify}_{K_1,K_2}(e_1, e_2)$	Reclassify value
	$\mid$	$\mathsf{encrypt}_{\overline{K}}(\overline{e_1}, e_2)$	Encrypt
	$\mid$	$\mathsf{decrypt}_{\overline{K_1};\overline{K_2}}(\overline{e_1}, e_2)$	Decrypt
	$\mid$	$\mathsf{sign}_{\overline{K_1};\overline{K_2}}(\overline{e_1}, e_2)$	Sign
	$\mid$	$\mathsf{auth}_{\overline{K}}(\overline{e_1}, e_2)$	Authenticate
	$\mid$	$\mathsf{new}(n : LT)\{e\}$	New channel
	$\mid$	$\mathsf{fork}\{e\}$	New process
	$\mid$	$\mathsf{send}(e_1, e_2)$	Message send
	$\mid$	$\mathsf{receive}(a)$	Message receive
	$\mid$	$\mathsf{pack}_{\langle t:A \rangle LT}(K, e)$	Package (key, data)
	$\mid$	$\mathsf{unpack}\ e_1\ \mathsf{to}\ \langle k : A \rangle (x : LT)\{e_2\}$	Open package
$R \in$ Process	$::=$	e	Sequential process
	$\mid$	$\mathsf{new}(k : A)\{R\}$	Key name
	$\mid$	$\mathsf{new}(a : LT)\{R\}$	Channel, key
	$\mid$	$(R_1 \mid R_2)$	Parallel composition

Figure 3. Syntax of Expressions and Processes

type environment:

$$\begin{array}{rcl}
PRINS_{TE}(\{\}) & = & \{\} \\
PRINS_{TE}(L_1 \cup L_2) & = & PRINS_{TE}(L_1) \cap PRINS_{TE}(L_2) \\
PRINS_{TE}(\{k\}) & = & \{\overline{P}\} \text{ if } \exists TE_1, TE_2. \\
 & & TE = (TE_1, k : \mathsf{EKey}_F(P : \overline{P}), TE_2) \\
PRINS_{TE}(\{k\}) & = & \{\overline{P}\} \text{ if } \exists TE_1, TE_2. \\
 & & TE = (TE_1, k : \mathsf{SKey}_F(P : \overline{P}), TE_2)
\end{array}$$

We make use of the fact that the only forms of key names in our system are atomic names. If some extension of the system were to consider more structured key names, then this metafunction would have to be defined mutually recursively (though the recursion would be well-founded) with the type formation rules.

The formation rules for (labelled) types check, e.g., that in an encryption key $\mathsf{EncKey}(K)$, the argument K denotes a key name with an encryption key name kind. These rules also check label constraints that are placed on public and private key types by the kinds of the corresponding key names. For example for encryption and decryption keys we have these type formation rules:

$$\frac{TE \vdash L_1, L_2 \ \mathbf{label} \qquad TE \vdash K : \mathsf{EKey}_F(P : \overline{P}) \qquad P \in PRINS_{TE}(L_2) \qquad PRINS_{TE}(L_1) \subseteq \{\overline{P}\}}{TE \vdash [\mathsf{DecKey}(K)]^{L_1, L_2} : \mathsf{Type}} \quad \text{(LCON DECKEY)}$$

$$\frac{TE \vdash L_1, L_2 \ \mathbf{label} \qquad TE \vdash K : \mathsf{EKey}_F(P : \overline{P}) \qquad P \in PRINS_{TE}(L_2)}{TE \vdash [\mathsf{EncKey}(K)]^{L_1, L_2} : \mathsf{Type}} \quad \text{(LCON ENCKEY)}$$

The first rule checks that the decryption key has the owning principal P in its integrity label, and that the set of principals allowed access to the decryption key by its secrecy label is contained in the principals allowed by the corresponding key name's kind. The second rule checks that the encryption key has the owning principal P in its integrity label. Analogous (dual) rules hold for signing and authentication keys.

The kinds of key names also place a constraint on the labels of declassification certificates. For example for declassification certificates for encryption keys, we have the following:

$$\frac{\begin{array}{c} TE \vdash L_1, L_2 \textbf{ label} \qquad TE \vdash K_1 : \mathsf{EKey}_{F_1}(P_1 : \overline{P_1}) \\ TE \vdash K_2 : \mathsf{EKey}_{F_2}(P_2 : \overline{P_2}) \\ P_2 \in \mathit{PRINS}_{TE}(L_2) \qquad \{\overline{P_1}\} \subseteq \mathit{PRINS}_{TE}(L_1) \end{array}}{TE \vdash [K_1 \ \mathsf{reclassifies}\ K_2]^{L_1,L_2} : \mathsf{Type}} \quad \text{(LCON RECL DECR)}$$

This rule requires that the integrity label for the certificate include the principal name owning the key name K_2 that is declassified; only this principal could have declassified the key name. The secrecy label for the certificate must define a set of principals that is contained in the set of principals that are allowed to access the private decryption key for K_1, the key name to which K_2 is declassified. Because of this latter condition, only principals that are already allowed to decrypt values encrypted with the public key for K_1 (using the latter's private key) will also be allowed to decrypt values encrypted with the public key for K_2 (using the declassification certificate). So reclassification certificates allow keys to be shared between principals, even in unanticipated ways, while certificates are the basis for building an audit trail of such key sharing.

The syntax of values, expressions and processes is provided in Fig. 3, where values V denote a subset of the expressions e where no further evaluation is necessary. For type preservation purposes, we also introduce a notion of *annotated values* $\langle\!\langle V \rangle\!\rangle_{P,C}^{;}$. The motivation is that for example an expression may have secrecy label $\{K\}$ but then be dynamically declassified to have secrecy label $\{K'\}$. With no relationship between the two key names, we cannot argue that the result of declassification has the same type as the original value, and therefore evaluation does not appear to preserve types. In our language, declassification is only possible when it is authorized by a declassification certificate. We restore the desired invariant of type preservation under evaluation by assuming that values are annotated by chains of such certificates. A declassification operation then adds its certificate chain to the annotation of the value it is declassifying. Annotations on values are not purely a fiction for preserving type preservation; they are consistent with the original motivation for distributed access control, *viz* building a fine-grain audit trail of accesses and declassifications.

The type system for expressions in general uses judgements of the form

$$TE; VE \vdash e \in^P [T]^{L_1,L_2}$$

to check that the expression (code) e is well-formed with annotated type $[T]^{L_1,L_2}$, under the assumption that it will be evaluated (executed) under the authority of the principal P, in the corresponding type and value environments TE and VE, respectively. We modularize the type rules for expressions with three judgement forms: $TE; VE \vdash^I e \in^P [T]^{L_1,L_2}$, $TE; VE \vdash^O e \in^P [T]^{L_1,L_2}$ and $TE; VE \vdash e \in^P [T]^{L_1,L_2}$. The first of these denotes a type judgement where we have checked that the principal P is allowed to access the result of the expression e, according to the secrecy label L_1. Typically such a judgement will be used for the premises for an expression type rule. The second judgement form is used as a conclusion for an expression type rule, denoting that the integrity condition on the conclusion has not yet been checked, i.e., that the principal P is allowed to compute the result of the expression e, according to the integrity label L_2. So the general form of a type derivation is

$$\dfrac{\dfrac{\dfrac{\vdots}{TE; VE \vdash e \in^P [T]^{L_1,L_2}}}{TE; VE \vdash^I e \in^P [T]^{L_1,L_2}} \quad \ldots \quad \dfrac{\dfrac{\vdots}{TE; VE \vdash e \in^P [T]^{L_1,L_2}}}{TE; VE \vdash^I e \in^P [T]^{L_1,L_2}}}{\dfrac{TE; VE \vdash^O e \in^P [T]^{L_1,L_2}}{TE; VE \vdash e \in^P [T]^{L_1,L_2}}}$$

In some cases the input and output checking may be circumvented, for example for variables, or for example where we sign a value with a private key and do not expect the access restrictions on the key to propagate to the resulting ciphertext (since we are ignoring covert channels and only considering access control, as far as we are concerned the key cannot be recovered from the ciphertext).

There are two key generation operations, one for virtual keys and one for actual keys. The type rule for virtual key generation is as follows:

$$\frac{\begin{array}{c} A = \mathsf{EKey}_{\mathsf{Virtual}}(P : \overline{P}) \qquad TE \vdash A \textbf{ kind} \\ (TE, k : A); VE \vdash e \in^P [T]^{L,L'} \qquad k \notin tv([T]^{L_1,L_2}) \end{array}}{TE; VE \vdash^O \mathsf{newKey}\langle k : A \rangle \{e\} \in^P [T]^{L,L'}} \quad \text{(VAL NEWKEY VIRT)}$$

This rule introduces a new encryption key name k, owned by the principal P that will evaluate this expression. This key name is local to the block e, but can escape this block if it is bundled up in a package. When used in a secrecy label, such a key name restricts accesses to some subset of the principals $\{\overline{P}\}$. The key is only useful for this form of static access checking.

$TE \vdash$ **tenv**	Well-formed type environment
$TE \vdash VE$ **venv**	Well-formed value environment
$TE \vdash A$ **kind**	Well-formed kind
$TE \vdash LT : A$	Well-formed type
$TE \vdash L$ **label**	Well-formed label
$TE; VE \vdash^I e \in^P [T]^{L_1,L_2}$	Type judgement (inputs checked)
$TE; VE \vdash^O e \in^P [T]^{L_1,L_2}$	Type judgement (outputs to be checked)
$TE; VE \vdash e \in^P [T]^{L_1,L_2}$	Type judgement (outputs checked)

Figure 4. Judgements for Type System

The type rule for actual key generation is as follows:

$$\frac{\begin{array}{c} A = \mathsf{EKey}_{\mathsf{Actual}}(P : \overline{P}) \quad TE \vdash A \textbf{ kind} \\ LT_1 = [\mathsf{EncKey}(k)]^{L_1,L_1'} \quad (TE, k : A) \vdash LT_1 : \mathsf{Type} \\ LT_2 = [\mathsf{DecKey}(k)]^{L_2,L_2'} \quad (TE, k : A) \vdash LT_2 : \mathsf{Type} \\ (TE, k : A); (VE, a^+ : LT_1, a^- : LT_2) \vdash e \in^P [T]^{L,L'} \\ k \notin tv([T]^{L_1,L_2}) \end{array}}{TE; VE \vdash^O \mathsf{newKey}\langle k : A\rangle(a^+ : LT_1, a^- : LT_2)\{e\} \in^P [T]^{L,L'}} \quad (\text{VAL NEWKEY ACT})$$

Actual key generation introduces not only a new key name, but also a public-private key pair for that key name. This key pair is denoted by the pair of dual names (a^+, a^-). The encryption key a^+ has an encryption key type indexed by the new key name, and the compatibility restrictions between the kind A of this key name and the labels of the key type are enforced before the key and its type are added to the value environment. Similarly for the decryption key a^-.

There are two forms of declassification for keys: declassifying a virtual key and declassifying an actual key. For the former we have the type rule:

$$\frac{\begin{array}{c} TE \vdash K_1 : \mathsf{EKey}_F(P_1 : \overline{P_1}) \quad TE \vdash K_2 : \mathsf{EKey}_{\mathsf{Virtual}}(P_2 : \overline{P_2}) \\ TE \vdash [K_1 \ \mathsf{reclassifies}\ K_2]^{L,L'} : \mathsf{Type} \\ TE \vdash VE \textbf{ venv} \quad TE \vdash P : \mathsf{Prin} \end{array}}{TE; VE \vdash^O \mathsf{reclassifyCert}_{K_1,K_2}() \in^P [K_1 \ \mathsf{reclassifies}\ K_2]^{L,L'}} \quad (\text{VAL RECL VIRT})$$

There is a little subtlety here. Suppose the principal P producing the certificate is not the same as the principal P_2 that owns the key being declassified? In that case we will have P included as one of the possible principals that created the certificate (after we check the integrity condition on the conclusion). Since this means that the certificate may have come from a principal other than the owner of the key being declassified, the declassification certificate will be effectively useless. So an attacker can always "spoof" a declassification certificate, but they must subvert the integrity checking to have this certificate taken seriously.

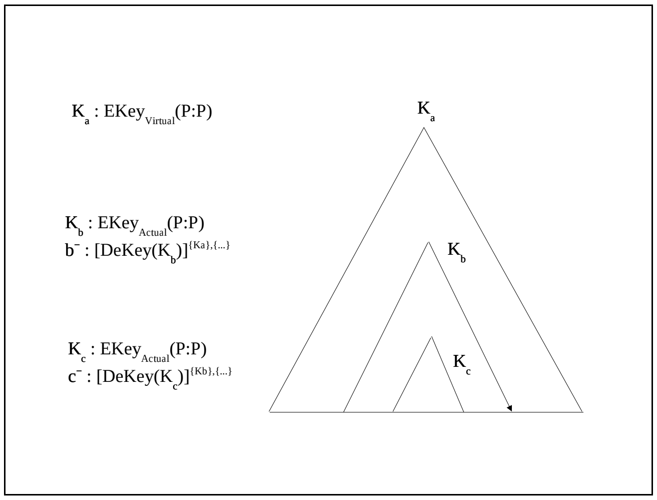
K_a : EKey$_{\text{Virtual}}$(P:P)
K_a
K_b : EKey$_{\text{Actual}}$(P:P)
b^- : [DeKey(K_b)]$^{\{Ka\},\{...\}}$
K_b
K_c : EKey$_{\text{Actual}}$(P:P)
c^- : [DeKey(K_c)]$^{\{Kb\},\{...\}}$
K_c

Figure 5. Key hierarchies and declassification

When declassifying an actual key name K_2, we must provide the associated decryption key. This is so that the principal receiving the declassification certificate can subsequently use it to declassify data encrypted under the encryption key for K_2. In this case we must also check that the decryption key could only have come from the principal that created the key name (P_2).

$$\frac{\begin{array}{c} TE \vdash K_1 : \mathsf{EKey}_F(P_1 : \overline{P_1}) \quad TE \vdash K_2 : \mathsf{EKey}_{\mathsf{Actual}}(P_2 : \overline{P_2}) \\ TE \vdash [K_1 \ \mathsf{reclassifies}\ K_2]^{L,L'} : \mathsf{Type} \\ TE; VE \vdash^I e \in^P [\mathsf{DecKey}(K_2)]^{L,L'} \quad PRINS_{TE}(L') = \{P_2\} \end{array}}{TE; VE \vdash^O \mathsf{reclassifyCert}_{K_1,K_2}(e) \in^P [K_1 \ \mathsf{reclassifies}\ K_2]^{L,L'}} \quad (\text{VAL RECL ACT})$$

Now there is a subtlety with the type rule for declassifying actual keys. The certificate that results from declassification has the same secrecy label L as the original decryption key, therefore the declassification certificate can only be accessed by the same set of principals $\{\overline{P_2}\}$ that already have access to the private key. Therefore this declassification certificate must itself be declassified.

We illustrate this point with an example, shown in Fig. 5. A principal P has created three key names, K_a, K_b and K_c. The private key for K_c is protected by K_b, while the private key for K_b is protected by K_a. The hierarchy stops here because K_a is a virtual key. Now the principal P wants to declassify K_c to the key K'_c of another principal P'. So P generates declassification certificates of types:

$$e_1 \in [K'_c \text{ reclassifies } K_c]^{\{K_b\},\{\ldots\}}$$

$$e_2 \in [K'_b \text{ reclassifies } K_b]^{\{K_a\},\{\ldots\}}$$

$$e_3 \in [K'_a \text{ reclassifies } K_a]^{\{K'_a\},\{\ldots\}}$$

The last certificate is allowed by the rule for declassifying virtual key names, above, that places no restrictions on the secrecy label for the certificate. Assuming that K'_a, K'_b and K'_c are keys of P' analogous to the keys K_a, K_b and K_c of P, then P' can use the last certificate to declassify the second certificate, and then use the second certificate to declassify the first certificate.

An obvious operation for declassification certificates is certificate chaining. This is provided by the $\mathsf{chain}_{K_1,K_2,K_3}(e_1,e_2)$ construct. In all of this, there is no problem with declassifying virtual key names to actual key names or vice versa. The only place where these certificates have computational import is in decryption, to which we turn below.

There are two applications of declassification certificates, as noted. The first application is in declassifying data, analogous to declassification in the DLM model:

$$\frac{\begin{array}{c} TE;VE \vdash^I e_1 \in^P [K' \text{ reclassifies } K]^{L_1,L'_1} \\ TE;VE \vdash^I e_2 \in^P [T]^{L_2 \uplus \{K\},L'_2} \\ TE \vdash K : \mathsf{EKey}_F(P_0 : \overline{P}) \quad PRINS_{TE}(L'_1) = \{P_0\} \\ TE \vdash [T]^{L_2 \uplus \{K'\},L'_2} : \mathsf{Type} \end{array}}{TE;VE \vdash^O \mathsf{reclassify}_{K',K}(e_1,e_2) \in^P [T]^{L_2 \uplus \{K'\},L'_2}}$$

(VAL RECLASSIFY)

In this rule the operation $\uplus$ denotes disjoint union. So declassification replaces the key name K in the secrecy label with the key name K'. The access restrictions on the declassification certificate do not propagate to the declassified value, since there is no construct in our language for extracting certificates from annotated values. We assume that all code within a single process is typed according to the type system described here, and that all communication over the network is encrypted in order avoid leaks due to low-level attackers that do not respect the type system and can intercept messages. We return to this point in the conclusions in Sect. 6.

The second application of declassification certificates is in declassifying data that has already been encrypted. In fact we fold this operation into the decryption operation itself. Rather than taking a sequence of decryption keys, corresponding to the keys under which the data is encrypted, this operation takes a sequence of declassification certificates. If a key name is not being declassified as part of decryption, then the corresponding certificate can simply declassify the key name to itself. The rules for encryption and decryption are as follows:

$$\frac{\begin{array}{c} TE;VE \vdash^I \overline{e} \in^P [\overline{\mathsf{EncKey}(K)}]^{\overline{L},\overline{L'}} \\ TE \vdash \overline{K} : \overline{\mathsf{EKey}_{\mathsf{Actual}}(P : \overline{P'})} \quad \overline{PRINS_{TE}(L')} = \overline{\{P\}} \\ TE;VE \vdash^I e_0 \in^P [T]^{L_0,L'_0} \quad L_0 = \{\overline{K}\} \quad TE \vdash L''_0 \textbf{ label} \end{array}}{TE;VE \vdash^O \mathsf{encrypt}_{\overline{K}}(\overline{e},e_0) \in^P [\mathcal{E}\{T\}]^{L''_0,L'_0}}$$

(VAL ENCRYPT)

$$\frac{\begin{array}{c} TE;VE \vdash^I \overline{e} \in^P [\overline{K' \text{ reclassifies } K}]^{\overline{L},\overline{L'}} \\ TE \vdash \overline{K} : \overline{\mathsf{EKey}_{\mathsf{Actual}}(P : \overline{P'})} \quad \overline{PRINS_{TE}(L')} = \overline{\{P\}} \\ TE;VE \vdash^I e_0 \in^P [\mathcal{E}\{T\}]^{L_0,L'_0} \quad L''_0 = \{\overline{K'}\} \end{array}}{TE;VE \vdash^O \mathsf{decrypt}_{\overline{K'};\overline{K}}(\overline{e},e_0) \in^P [T]^{L_0 \cup L''_0,L'_0}}$$

(VAL DECRYPT CERT)

The encryption rule checks that the encryption keys that are provided do in fact come from the principals that own the corresponding key names. These encryption keys must of course match the key names in the secrecy label for the data being encrypted. The secrecy label on the result is arbitrary (it might be empty). In a real sense encryption is a way of circumventing the type system, but in a "safe" fashion since the resulting value cannot be reintroduced without a dynamic check. This check is by the decryption operation. It again checks that the declassification certificates do indeed come from the principals that own the key names for the decryption keys being declassified by the certificates. The decryption operation asserts that the encrypted value was encrypted with the keys $\{\overline{K}\}$, but this can only be ascertained by a run-time check. If we implement encryption and decryption as iterated single-key operations, then we must assume some canonical ordering of the keys, so a sequence of keys is always applied in that canonical order to encrypt data under several keys. Additional assumptions may be necessary. The result of decryption has a secrecy label that includes the keys $\overline{K'}$ plus also any restrictions that propagate from the expression that produces the encrypted data.

There are analogous but essentially dual rules for integrity key names. We omit the obvious details in order to conserve space.

4. Operational Semantics

In this section we consider an operational semantics for our language. This semantics (perhaps surprisingly) does run-time access checking. We verify that types are

$V \in$ Value	$::=$	a, b, c, n	Name (Channel, Key)
	$\mid$	$\mathsf{reclassifyCert}_{K_1,K_2}()$	Reclassify cert (virtual key)
	$\mid$	$\mathsf{reclassifyCert}_{K_1,K_2}(v)$	Reclassify cert (actual key)
	$\mid$	$\mathsf{encrypt}_{\overline{K}}(v_1, v_2)$	Encrypted data
	$\mid$	$\mathsf{sign}_{\overline{K_1};\overline{K_2}}(v_1, v_2)$	Signed data
	$\mid$	$\mathsf{pack}_{\langle t:A\rangle LT}(K, v)$	Package (key, data)
$v \in$ Annotated Value	$::=$	$\langle\!\langle V\rangle\!\rangle^{L,L'}_{P,C}$	
$C \in$ Certificate chain	$::=$	$((P_1, v_1), \ldots, (P_m, v_m))$	
$e \in$ Expression	$::=$	$\langle e\rangle$	Add annotation
	$\mid$	v	Annotated value
	$\mid$	$P[e]$	Configuration

Figure 6. Additional Syntax for Semantics

preserved under evaluation. We use this result to verify progress: a process' evaluation can only block on a failure to decrypt, or a failure to authenticate a signature (or, when we consider message-passing operations, on a blocking receive). This verifies that the run-time access checking is unnecessary: the type system successfully prevents any access violations.

To describe and reason about the semantics, we extend the syntax of expressions as described in Fig. 6. The class of values V describes the result of evaluation, where no further evaluation steps are necessary. The language contains as values names (channels and keys), certificates, encrypted and signed data, and packages. We then annotate these values with a principal and a certificate chain. The principal and certificate chain annotations are used to ensure type preservation during evaluation. The value may have been computed by a principal different from the current principal, with data that are not accessible to the current principal. The certificate chain is extended every time the value is declassified, to justify giving it the declassified type it has even though the original value does not. The certificate is a sequence of (principal,declassification certificate) pairs, that is extended incrementally every time the annotated value is declassified.

The syntax of expressions is now extended with such annotated values v, and annotating expressions $\langle e\rangle$, and also configurations of the form $P[e]$. The latter denotes an expression (thread) e that is executing for the principal P. We allow such configurations to be nested; this happens when code executing for one principal does a method or procedure call to code supplied by another principal. The type rules for annotated values are provided in Fig. 7. Some of the computation rules are provided in Fig. 8.

We make one other extension over the language formulated in the previous section, and relax the require-

$$\frac{TE; VE \vdash V \in^{P'} [T]^{L,L'}}{TE; VE \vdash \langle\!\langle V\rangle\!\rangle^{L,L'}_{P',()} \in^{P} [T]^{L,L'}} \qquad (\textsc{Val AnnVal Base})$$

$$\frac{TE; VE \vdash \mathsf{reclassify}_{K,K'}(v, \langle\!\langle V\rangle\!\rangle^{L_0,L_0'}_{P_1,C}) \in^{P_2} [T]^{L,L'}}{TE; VE \vdash \langle\!\langle V\rangle\!\rangle^{L_0,L_0'}_{P_1,(C,(P_2,v))} \in^{P} [T]^{L,L'}} \qquad (\textsc{Val AnnVal Step})$$

Figure 7. Type Rules for Annotated Values

ments on the decryption operation: in an operation of the form $\mathsf{decrypt}_{\overline{K_1};\overline{K_2}}(\overline{e_1}, e_2)$, the sequence of expressions $\overline{e_1}$ can contain both declassification certificates and decryption keys. This is used by the operational semantics to decompose declassification certificates into decryption keys. The (VAL DECRYPT) type rule can be straightforwardly generalized, and we avoided doing this in the previous section only to avoid making the rule any more complicated.

The semantics is provided as a labelled transition system (LTS) in Fig. 8, using the following reduction relations:

$R \xrightarrow{TE;VE} R'$	"Internal" computation
$e \xrightarrow{TE;VE;P} e'$	Computation under a principal

There are other labelled transitions for message-passing and process forking, but we omit them from this extended abstract, since they do not contribute markedly to the results. The rules for the two reduction relations above are provided in Fig. 8.

The (RED PRIN ANN) operation sets things up by annotating a value with the principal responsible for its creation, and an empty certificate chain. This is the point where a value is allocated, and this is where we see the first access

$$\frac{TE;VE \vdash V \in^P [T]^{L,L'}}{\langle V \rangle \xrightarrow{TE;VE;P} \langle\!\langle V \rangle\!\rangle^{L,L'}_{P,()}} \quad \text{(RED PRIN ANN)}$$

$$\frac{TE;VE \vdash^I v_1 \in^P [T_1]^{L_1,L_1'} \quad TE;VE \vdash^I v_2 \in^P [T_2]^{L_3,L_3'}}{\mathsf{reclassify}_{K_1,K_2}(v_1,v_2) \xrightarrow{TE;VE;P} CEXTEND(v_2,((P,v_1)))} \quad \text{(RED PRIN RECL)}$$

$$\frac{v_1 = \langle\!\langle \mathsf{chain}_{K_1,K_0,K_2}(v_2,v_3) \rangle\!\rangle^{L_1,L_1'}_{P_1,C_1} \quad TE;VE \vdash^I v_1 \in^P [T]^{L_0,L_0'} \quad v_2' = CEXTEND(v_2,C_1) \quad v_3' = CEXTEND(v_3,C_1)}{\mathsf{decrypt}_{\overline{K_1},K_1;\overline{K_2},K_2}((\overline{v_1},v_1),v) \xrightarrow{TE;VE;P} \mathsf{reclassify}_{K_1,K_0}(v_2',\mathsf{decrypt}_{\overline{K_1},K_0;\overline{K_2},K_2}((\overline{v_1},v_3'),v))} \quad \text{(RED PRIN DECCH)}$$

$$\frac{v_1 = \langle\!\langle \mathsf{reclassifyCert}_{K_1,K_2}(v_0) \rangle\!\rangle^{L_1,L_1'}_{P',C} \quad TE;VE \vdash^I v_1 \in^P [T]^{L_2,L_2'} \quad v_0' = CEXTEND(v_0,C)}{\mathsf{decrypt}_{\overline{K_1},K_1;\overline{K_2},K_2}((\overline{v_1},v_1),v) \xrightarrow{TE;VE;P} \mathsf{reclassify}_{K_1,K_2}(v_1,\mathsf{decrypt}_{\overline{K_1},K_2;\overline{K_2},K_2}((\overline{v_1},v_0'),v))} \quad \text{(RED PRIN DECCRT)}$$

$$\frac{\overline{v} = \overline{\langle\!\langle a^- \rangle\!\rangle^{L_1,L_1'}_{P_1,C_1}} \quad v' = \langle\!\langle \mathsf{encrypt}_{\overline{K}}(\overline{\langle\!\langle a^+ \rangle\!\rangle^{L_2,L_2'}_{P_2,C_2}},v'') \rangle\!\rangle^{L,L'}_{P',C} \quad TE;VE \vdash^I \overline{v} \in^P [\overline{T}]^{\overline{L_1''},\overline{L_1'''}} \quad TE;VE \vdash^I v' \in^P [T']^{L'',L'''}}{\mathsf{decrypt}_{\overline{K};\overline{K}}(\overline{v},v') \xrightarrow{TE;VE;P} CEXTEND(v'',C)} \quad \text{(RED PRIN DECACT)}$$

Figure 8. Selected Computation Rules

check: the reduction rule makes sure that the principal of the executing process is included in the integrity label of the value's type. Because of some tricky scoping issues with certificate chains, we finesse this check by using the type rules in the previous section, specifically for conclusions of the form $TE;VE \vdash e \in^P [T]^{L,L'}$ where all "output" or integrity constraints (from the label L') have been checked.

The next rule, (RED PRIN RECL), performs declassification. To ensure type preservation with declassification, we add the certificate authorizing declassification to the certificate chain associated with the value being declassified. The following metafunction computes the extension of the certificate chain in an annotated value:

$$CEXTEND(\langle\!\langle V \rangle\!\rangle^{L,L'}_{P,C_1}, C_2) \quad = \quad \langle\!\langle V \rangle\!\rangle^{L,L'}_{P,(C_2,C_1)}$$

The (RED PRIN RECL) also checks the secrecy constraints, that the reclassification operation has authorization to access both of its arguments. Again we use the type system for perform this access check, this time with conclusions of the form $TE;VE \vdash^I e \in^P [T]^{L,L'}$ where all secrecy constraints (from the label L) have been checked.

The next three rules handle decryption, where the first input is a sequence of declassification certificates and decryption keys. The (RED PRIN DECCH) rule decomposes a declassification certificate chain in one of the arguments, moving the "left" certificate out of decryption by declassifying the result of decryption, and leaving the "right" certificate as an argument to decryption. This transition uses the type system to check that the executing principal has access to the declassification certificate that it is decomposing.

The (RED PRIN DECCRT) rule deals with the case where one of the declassification certificates in the arguments to decryption is declassifying a decryption key, i.e., it is a declassification certificate for an actual key name. In this case the transition replaces the declassification certificate, as an argument to decryption, with its underlying decryption key, and again inserts a declassification operation on the result of decryption (using the original declassification certificate that has been replaced as an argument to decryption).

Because of the typing of decryption, it cannot be the case that one of the arguments to decryption is a declassification certificate for a virtual key name. Therefore this handles all possible declassification certificate types.

The final step, rule (RED PRIN DECACT), performs the actual decryption, now that all declassification certificates have been "peeled off" from the decryption operation and moved outside as declassification operations on the result of decrypting.

The reason that each of the intermediate stages in decryption is well-typed, is because of annotated values v that "remember" the original owning principal that generated a certificate or key. Such data structures are typed with respect to the owning principal rather than the currently executing princpal. Annotated values are a purely formal notion for typing run-time values.

We have omitted the type rules for processes, but they are completely standard and straightforward.

Theorem 1 (Type Preservation) *The following propositions verify that types are preserved by computation:*

1. *Suppose* $TE;VE \vdash R_1$ **proc** *and* $R_1 \xrightarrow{TE;VE} R_2$, *then* $TE;VE \vdash R_2$ **proc**.

2. *Suppose $TE;VE \vdash e \in^P [T]^{L,L'}$ and $e \xrightarrow{TE;VE;P} e'$, then $TE;VE \vdash e' \in^P [T]^{L,L'}$.*

Define an *evaluation context* $\mathbb{C}[\,.\,]$ in the normal fashion. Define a *process context* by:

$$\begin{array}{rcl} \mathbb{P}[\,.\,] & ::= & \mathbb{C}[\,.\,] \;\mid\; (\mathbb{P}[\,.\,] \mid R) \;\mid\; (R \mid \mathbb{P}[\,.\,]) \\ & \mid & \mathsf{new}(k:A)\{\mathbb{P}[\,.\,]\} \;\mid\; \mathsf{new}(a:LT)\{\mathbb{P}[\,.\,]\} \end{array}$$

This is used to identify places in a process where there should be a redex, and if not then computation is blocked. The proof of the following uses the fact that the dynamic access checks are performed as type checks, and also of course using the previous theorem. Since we have not had space to present the semantics of message-passing, we only consider the subset of the language with the cryptographic operations.

Theorem 2 (Progress) *The following propositions verify that computation can only "get stuck" because of a mismatch between private and public keys in a decryption or authentication operation:*

1. *Suppose $TE;VE \vdash R$ **proc** and $R = \mathbb{P}[e]$ where e is neither a value nor a redex. Then e is of one of the two following forms:*

$$\mathsf{decrypt}_{\overline{K};\overline{K}}(\overline{v_1}, \langle\!\langle \mathsf{encrypt}_{\overline{K}}(\overline{v_2}, v)\rangle\!\rangle^{L,L'}_{P',C})$$

$$\mathsf{auth}_{\overline{K}}(\overline{v_1}, \langle\!\langle \mathsf{sign}_{\overline{K};\overline{K}}(\overline{v_2}, v)\rangle\!\rangle^{L,L'}_{P',C})$$

where $\overline{v_1} = \overline{\langle\!\langle a^-\rangle\!\rangle^{L_1,L_1'}_{P_1,C_1}}$ and $\overline{v_2} = \overline{\langle\!\langle b^+\rangle\!\rangle^{L_2,L_2'}_{P_2,C_2}}$ and $\overline{a^-} \neq \overline{b^+}$, i.e., an application of decryption or an application of authentication, where the private keys do not match the public keys.

2. *Suppose $TE;VE \vdash e \in^P [T]^{L,L'}$ and $e = \mathbb{P}[e']$ where e' is neither a value nor a redex. Then e' is of one of the two forms described in the first part of the theorem.*

In particular this justifies not performing the run-time access checks (formulated in our semantics as type checks), since these checks are guaranteed to always succeed.

5. Related Work

The motivation for this work has been the need for proper programming abstractions for applications that must manage the task of securing their own communication. Much of the work on wide-area languages has focused on security, for example, providing abstractions of secure channels [5, 4], controlling key distribution [13, 12], reasoning about security protocols [6, 1], tracking untrustworthy hosts in the system [20, 30], etc.

Abadi [1] considers a type system for ensuring that secrecy is preserved in security protocols. For securing communication over untrusted networks, he includes a "universal" type Un inhabited by encrypted values. His type system prevents "secrets" from being leaked to untrusted parties, but allows encrypted values to be divulged. In an analogous way, encrypted and signed values in our type system provide a way to temporarily subvert the access controls in the type system, with the secrecy and integrity properties enforced by labels reasserted when the ciphertext is decrypted/authenticated.

Gordon and Jeffrey [17, 18] have developed a type-based approach to verifying authentication protocols. Their dependent type system maintains a connection between messages and nonces. Nonce types and an effect type system use correspondence assertions to ensure the freshness of communications.

Abadi and Blanchet [2, 9] have worked on analyzing security protocols, they show how it is possible to guarantee secrecy properties and then generalizing this to guarantee integrity. Their system uses a type of "secret," and a type system that ensures that secret items are never put on channels that are shared with untrusted parties. They can translate types in their system into logic programs that can then be used to check protocols for correctness. The emphasis of this work is somewhat different, since Bruno and Blancet work in a more "black and white" environment where there are trusted parties and untrusted parties. In contrast our interest is in a more refined type system where we allow certain parties to access certain data. For reasons of space and exposition, we have for avoided the issue of prevented data from leaking on untrusted channels. It may be interesting to investigate this in future work.

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.

Sumii and Pierce [34] describe the cryptographic lambda calculus, an extension of the typed lambda calculus with symmetric cryptographic operations along the lines of the spi-calculus. They then use reasoning techniques for the polymorphic lambda calculus (in particular, relational parametricity) to verify properties of programs implemented using cryptographic techniques.

A number of other systems also combine elements of access control and information flow. Stoughton [33] presents a simple but elegant system that uses access control to assign the same piece of data different information flow *tags* for different groups of principles. This is done using a denotational semantics that makes dynamic checks. The SLam calculus of Heintze and Riecke [19] extents the lambda-calculus with types that record both the security levels that may have direct access to a piece of data and the levels that might have indirect access. Dynamic, access control checks in the semantics ensure that these types are never violated. The type system imposes enough information flow restrictions to enable a proof of noninterference. In more recent work Abadi and Fournet have developed a system for C$^{\#}$ that performs access control by dynamically monitoring the flow of a program [3]. As each routine executes it imposes any access restrictions that are associated with that routine on the following code. If a routine attempts to perform a restricted action the execution is stopped with a security exception.

Other work on security in programming languages has focused on ensuring safety properties of untrusted code [27, 26, 23] and preventing unwanted security flows in programs [14, 24, 35, 29]. Laud has developed a complexity-theoretical approach to information flow [21], that concentrates on data leaks that can be discovered in polynomial time. Sabelfeld and Myers [31] provide an excellent overview of work in language-based information-flow security. Our security concerns have largely been with access control, but there is a clear and obvious relationship to the decentralized label model of JIF [25]. This relationship exists because we initially based our type system on decentralized labels, as explained in Sect. 2.

Pottier and Conchon [29] have developed an interesting approach to encoding information flows in the lambda-calculus, allowing non-interference to be checked in an operational manner, and Pottier [28] has extended this to the pi-calculus. Because of the operational nature of their work, it appears plausible that it could be useful in proving some form of non-interference for our language. But in the presence of declassification, it is a well-known problem to define what safety guarantees are provided by information flow control [31, 36]. In any case, this has not been the focus of our concern, as indicated by Sect. 1.

6. Conclusions

This paper has combined two ideas:

1. First, the notion of type-based cryptographic operations [15], with the intention that some of the secrecy and integrity properties of those operations can be checked statically. This can sometimes avoid the expense of cryptographic operations at run-time. At the very least, it provides a way for specifying the security guarantees of a channel provided from lower layers in the protocol stack to upper layers.

2. Second, the notion of decentralized labels, that combine access control and some form of information flow control [24, 25]. We are strictly interested in access control and so emphasize this aspect of decentralized labels.

The payoff from this combination is that we can now see a way to move network security out of the TCB and into the application, with the type system making sure that the application does not violate the label constraints (modulo declassification). For example a distributed implementation of JIF [37] puts all network security in the TCB and makes the network transparent to the program. It appears implausible that this will scale over the network, or in mission-critical or fault-tolerant systems.

There are obviously many issues to explore further. Our focus due to space has been on parties interacting and preventing data from “flowing” (directly) to undesirable parties. What about untyped attackers on the network? For now we assume that all network communication is encrypted and signed, but there should be ways to interact safely with untrusted parties, declassifying data from the world and reading data from the world. We have some ideas on how to proceed with this.

Cryptographic types [15] were based on the idea that there would be subnets (or just OS buffers) where it would be safe to assume that data was secure without being encrypted and signed. That work did not consider how to delineate where on the network it would be secure to send data intended for certain parties, and where it would be necessary to build trusted channels from untrusted channels. The current work still does not address this issue. We expect to address it in the near future, but back in the context of cryptographic types where the language and the problems are simpler.

It would be good to develop more notions of correctness for our work. For example it appears plausible that some form of robust non-interference could be demonstrated for this language [36], based on simulation relations. However the language that we work with is rich and we do not expect such a result to be easy.

Another direction to consider is accountability: formalizing it, perhaps based on causality types [32], and verifying that our certificate chains for annotated values are in some sense a good measure of accounability. These are all areas for future work.

References

[1] M. Abadi. Security by typing in security protocols. In *Theoretical Aspects of Computer Science*, pages 611–638, 1997.

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

[3] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, February 2003.

[4] M. Abadi, C. Fournet, and G. Gonthier. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy*, 1999.

[5] M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2000.

[6] M. Abadi and A. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[7] M. Abadi, B. Lampson, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992. Also appeared as SRC Research Report 83.

[8] M. Abadi, E. Wobber, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, 1994. Also appeared as SRC Research Report 117.

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

[10] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

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

[12] L. Cardelli, G. Ghelli, and A. D. Gordon. Secrecy and group creation. In *Concurrency Theory (CONCUR)*. Springer-Verlag, 2000.

[13] G. Castagna and J. Vitek. A calculus of secure mobile computations. In *Internet Programming Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

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

[15] D. Duggan. Cryptographic types. In *Computer Security Foundations Workshop*, Nova Scotia, Canada, 2002. IEEE Press.

[16] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. Simple public key certificate. Internet Draft draft-ietf-spki-cert-structure-05.txt, 1998.

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

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

[19] N. Heintze and J. G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[20] M. Hennessy and J. Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[21] P. Laud. Handling encryption in an analysis for secure information flow. In *ESOP 2003*, pages 159–173, 2003.

[22] J. Mitchell and G. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[23] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.

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

[25] A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4), 2000.

[26] G. Necula. Proof-carrying code. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1997.

[27] G. Necula and P. Lee. Safe kernel extensions without runtime checking. In *Operating Systems Design and Implementation*, 1996.

[28] F. Pottier. A simple view of type-secure information flow in the pi-calculus. In *Computer Security Foundations Workshop*, 2002.

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

[30] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1999.

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

[32] P. Sewell and J. Vitek. Secure composition of untrusted code: Wrappers and causality types. In *Computer Security Foundations Workshop*, 2000.

[33] A. Stoughton. Access flow: A protection model which integrates access control and information flow. *IEEE Symposium on Security and Privacy*, pages 9–18, 1981.

[34] E. Sumii and B. C. Pierce. Logical relations for encryption. In *Computer Security Foundations Workshop*, June 2001.

[35] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Springer-Verlag, 1997.

[36] S. Zdancewic and A. C. Myers. Robust declassification. In *Computer Security Foundations Workshop*, 2001.

[37] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Symposium on Operating Systems Principles*, 2001.