# Principals, Policies and Keys in a Secure Distributed Programming Language

Tom Chothia
Department of Computer Science
Stevens Institute of Technology
Hoboken, New Jersey 07030, USA.
tomc@cs.stevens-tech.edu

Dominic Duggan
Department of Computer Science
Stevens Institute of Technology
Hoboken, New Jersey 07030, USA.
dduggan@cs.stevens-tech.edu

Jan Vitek
Dept of Computer Science
Purdue University
West Lafayette, IN 47907, USA.
jv@cs.purdue.edu

**Abstract**

KDLM is a model of information flow control for distributed programming languages, introducing a distributed notion of selective declassification. KDLM uses type-level notions of both principals and policies to model security policies incorporating access control and information flow control, with run-time representatives based on cryptographic keys. This article addresses the question of how this control is enforced when programs cross address spaces and networks. An object calculus is introduced that extends KDLM. This uses a notion of locally opaque principal and key names in objects to transmit and enforce restrictions across address spaces, with key equality used as a form of runtime type discrimination to reveal "opaque" principal names at the receiver.

## 1 Introduction

Language-based security is becoming an increasingly important aspect of computer security. Large amounts of data are being processed by computer programs and there is a need for control and accountability in the way these programs handle the information they glean from the data. The computer security community has focussed on access control mechanisms for databases of sensitive data. The programming language community has focussed on information flow control: making sure that sharing of data is controlled in some manner to ensure that it does not leak in undesirable ways. The programming language community has focussed on information flow control because language technology such as type systems can be fruitfully applied to this problem.

The Decentralized Label Model (DLM) is a model of information flow control that was introduced by Myers and Liskov [19]. This model avoids one undesirable aspect of classical information flow control, the need for some centrally defined lattice of information levels, by implicitly defining a lattice based on access control. It also introduced *selective declassification*, allowing a principal to relax its own access control and in this way relax information flow control. This relies critically on the notion of the principal for the currently executing code.

More recently the authors proposed the Key-based Decentralized Label Model (KDLM) as a model that combines information flow control and application-based cryptography for secure network programming [6]. The argument for this approach is the usual end-to-end argument in system design: it is ultimately unrealistic to expect a single one-size-fits-all solution to network security in the runtime. The application must be able to build its own network security stack for any approach to scale, yet the type system must prevent the application from violating the information flow guarantees in the type system while establishing the network security. KDLM combined ideas from DLM and type-based cryptographic APIs [9]. In particular KDLM introduced the notion of key names, type level entities representing encryption and signing keys, as well as principals as in DLM. A particular motivation for KDLM was *distributed declassification*.

A question that is posed by KDLM is, what is the nature of principal names and key names in programs that may cross address spaces? All modelling for security purposes is done in the type system, with any runtime checks (beyond the static type-based checking) based on keys. We can certainly formulate examples with principals such as `Alice` and `Bob`, but these merely refer to compile-time type names within an address space. How do two separate programs come to agreement on the identities of their principals?

In this article we extend KDLM with an object calculus, motivated by its ultimate application in an object-oriented programming language. We call this language Jeddak, it extends Java. Since principals (and key names) must be modelled as type-level entities, objects carry type-level fields via an adaptation of the form of generics that have been proposed for Java and are expected to appear in JDK 1.5 [4, 5, 26]. In this approach, the names of keys and principals are treated as abstract data types for transmission of data, with key equality used as a form of runtime type discrimination to reveal "opaque" principal names at the receiver.

In Sect. 2 we explain with some examples how our approach can be used to support key distribution while enforcing access control across address spaces. In Sect. 3 we provide a formal type system, and in Sect. 4 a dynamic semantics. Finally Sect. 5 considers related work and conclusions.

## 2 Key Distribution

The Jeddak type system models notions of key-based access control. So it includes notions of *principal names* and *policy names* that are analogous to types. Whereas the latter may be loosely interpreted as sets of values, the former represent abstract modelling entities whose runtime representatives are actual keys. The motivation for this approach is that much access control can be checked at compile-time. This in turn enables lightweight access control checking that facilitates a fine-grained audit trail for checking accountability.

However this compile-time checking cannot be extended across a distributed system, where values cross process address spaces. An analogy may be made here with RMI programming. Code executing within an address space is fully type-checked. Dynamic typing is used to ensure the type-safety of values that are transmitted between address spaces. Unmarshalling ensures that values are well-formed, and checked downcasting ensures that values have the application-specific types required at receiving sites.

Jeddak includes analogous mechanisms for "hiding" access information at a sending site and "exposing" that access information at the receiving site. The exposure operation requires runtime checking, but based on key identity rather than checked downcasting. We illustrate this by showing how keys with limited distribution may be shared between processes in a safe manner.

Types for values include *labelled types*, of the form

$$\{K_1,\ldots,K_n\}T.$$

The type $T$ is a type in the normal sense. The set $\{K_1,\ldots,K_n\}$ is a *label*. In DLM, a label is a set of security policies. In KDLM, a label is a set of *policy names*, i.e., names for security policies. This extra level of indirection is motivated by mechanisms for distributed declassification, as explained in [6]. A policy is essentially an access control list, of the form

$$[P:\{P_1,\ldots,P_k\}]$$

where $P,P_1,\ldots,P_k$ are *principal names*. This policy is controlled by the principal $P$. This notion of control of a policy by a principal is useful for reclassification purposes: a principal can allow information controlled by one policy to "flow" (via assignment) to a variable controlled by a different policy, provided that principal controls both policies. The policy itself restricts access to a variable to the principals $\{P_1,\ldots,P_k\}$. A policy may be either to restrict "read" rights (to control secrecy) or to restrict "write" rights (to control integrity) on a variable. Since access to a variable is controlled by a set of such policies, access to that variable is restricted to the principals in the intersection of the policies making up the label.

For example if we have

```
K1  specifies policy  [Joe:{Joe,Mary,Sam}]
K2  specifies policy  [Mary:{Mary,Sue,Sam}]
{K1,K2}int x;
```

Then the variable `x` is only accessible to the principals `Mary` and `Sam`.

Policy names only exist at compile-time, for modelling the security policies of the program. At run-time policy names may be represented by run-time witnesses, given by public-private key pairs. For a secrecy policy name, such a pair corresponds to a public encryption key and its corresponding private decryption key. For an integrity policy name, such a pair corresponds to a public authentication key and its corresponding signing key. A policy name that has an associated key pair representing it at run-time is declared as an "actual" policy name (as opposed to a "virtual" policy name that is used only for modelling purposes at compile-time). Consider the following example:

```
spolicy MyPolicy [ThisPrin:{Mary,Sue}];
pubkey⟨MyPolicy⟩ myPublicKey;
{MyPolicy}privkey⟨MyPolicy⟩ myPrivateKey;
```

This example declares a policy names, `MyPolicy`, and two keys, `myPublicKey` and `myPrivateKey`. The keyword `spolicy` identifies this policy name as a secrecy policy name; dually the language has integrity policy names, identified by the keyword `ipolicy`. The policy `MyPolicy` is controlled by the principal for the currently executing code, identified by the keyword `ThisPrin`. We assume as in the Java security model that code is signed by a principal. The policy `MyPolicy` has a (unique) associated public-private key pair. Two variables are declared as part of an actual policy name declaration, and these variables are initialized with the public and private parts of the associated key pair. The parameterized types `pubkey⟨_⟩` and `privkey⟨_⟩` denote the types of encryption and decryption keys, respectively (where the policy name specifies a secrecy policy). These types are parameterized by the associated policy name. So the public encryption key for the policy name `MyPolicy` has type `pubkey⟨MyPolicy⟩`.

The decryption key has type `privkey⟨MyPolicy⟩`. It must also have a label that is consistent with the policy of the associated policy name. The policy of the policy name `MyPolicy` restricts read access to variables to the principals `Mary` and `Sue` (and implicitly `ThisPrin`). If data with this access restriction is subsequently encrypted for transmission over a network, we must ensure that the private decryption key is not distributed to principals other than `Mary`, `Sue` and `ThisPrin`.

Now we have an apparent problem with actual policy names. Since such policy names can only be declared locally in a program, never at top level, how can we ever return a result whose type involves actual policy names? Consider for example returning a newly generated public-private key pair as the result of a method. To do this, we must return not only the key pair, but also the key name that they are associated with. We have so far talked of methods that take policy names as inputs, similarly to type parameters. Now we must allow methods to return policy names as outputs[1].

The following example defines a class for objects that contain public-private key pairs. Each such object has a `publicKey` field and a `privateKey` field. The class abstracts over the policy name that these fields share, and the principal and principal set in the policy for the key name, and the private policy name that specifies access restrictions on the private key:

```
class KeyPair ⟨prin Owner,
               prinSet Objects,
               spolicy K [Owner:Objects] ⟩ {
   pubkey⟨K⟩ publicKey;
   {K}privkey⟨K⟩ privateKey;
   KeyPair (pubkey⟨K⟩ pub, {K}privkey⟨K⟩ priv) auth Owner {
               this.publicKey = pub;
               this.privateKey = priv;
   }
}
```

An instantiation of this class must include instantiations for all of the type-level parameters. We extend the positional Java syntax for instantiation with a field-based syntax that is useful for what follows, so an example instantiation is:

```
new KeyPair⟨Owner=O,Objects=Os,K=K⟩
          (publicKey,privateKey)
```

for some `O`, `Os` and `K`, and some public and private keys `publicKey` and `privateKey`, respectively. The type of the resulting object is:

```
KeyPair⟨Owner=O,Objects=Os,K=K⟩
```

Now there is a problem with this type. Suppose a new policy name is generated as part of the generation of a new pair, in a local context, and this object is to be returned from this context. But this return is not possible because the surrounding context must already know the name of the newly generated policy name `K`, which is impossible. Therefore Jeddak adopts a refinement of the JDK 1.5 notion of raw types: we allow some of the type parameters in a generic instantiation to be hidden. This partial hiding is the motivation for the field-based syntax for generic instantiation. We have the subtyping

```
KeyPair⟨Owner=O,Objects=Os,K=K⟩  ≤
   KeyPair⟨Owner=O,Objects=Os⟩
```

So the identity of the locally generated policy name is hidden, but the access restrictions are still visible in the type of the resulting object. In the following example, a nullary method `newPair` generates a new key pair, as part of declaring a new local (actual) policy name, and returns the result in a `KeyPair` object:

[1] For the reader cognizant with type theory, input policy names are universally quantified while output policy names are existentially quantified [16].

```
KeyPair⟨Owner,Objects⟩ newPair
      ⟨prin Owner, prinSet Objects⟩ () auth Owner {
    spolicy K [Owner:{Objects}];
    pubkey⟨K⟩ publicKey;
    {K}decKey⟨K⟩ privateKey;
    return new KeyPair ⟨Owner=Owner,Objects=Objects,K=K⟩
                        (publicKey, privateKey);
}
```

Now we can say:

```
KeyPair⟨ThisPrin,{Mary,Sue}⟩ x =
    newPair ⟨ThisPrin,{Mary,Sue}⟩();
KeyPair⟨ThisPrin,{Mary,Sue}⟩ y =
    newPair ⟨ThisPrin,{Mary,Sue}⟩();
```

Now we have the danger that the keys in x and y may be considered interchangeable, since x and y have the same types. This is because we have "hidden" the policy names for the keys of x and y. In Jeddak the hidden types are considered to be *local abstract types*, that can actually be referred to as fields of the object. So we have the typings:

$$\texttt{x.publicKey} \in \texttt{pubkey}\langle\texttt{x.K}\rangle$$

$$\texttt{y.publicKey} \in \texttt{pubkey}\langle\texttt{y.K}\rangle$$

$$\texttt{x.privateKey} \in \{\texttt{x.K}\}\texttt{privkey}\langle\texttt{x.K}\rangle$$

So the policy names for x and y are distinguished (x.K and y.K, respectively). The encryption and decryption keys cannot be referred to independently of the policy name.

If a public key must be transmitted, it must be bundled up in another object that has a copy of the policy name:

```
class PubKey ⟨prin Owner,
              prinSet Objects,
              spolicy K [Owner:Objects] ⟩
    pubkey⟨K⟩ publicKey;
    PubKey (pubkey⟨K⟩ pub) auth Owner {
       this.publicKey = pub;
    }
}
```

Then we can make a copy of the public key for distribution as follows:

```
PubKey⟨ThisPrin,{Mary,Sue}⟩ z =
    new PubKey ⟨Owner=ThisPrin,Objects={Mary,Sue},K=x.K⟩
               (x.publicKey);
```

The typing of this is subtle. We have the following subtyping rule:

$$\texttt{PubKey}\langle\texttt{Owner=ThisPrin,Objects=\{Mary,Sue\},K=x.K}\rangle \leq \texttt{PubKey}\langle\texttt{Owner=ThisPrin,Objects=\{Mary,Sue\}}\rangle.$$

The first type reveals that the K type field of the PubKey object is actually an alias for x.K. This is reflected by the type equality K=x.K in the type of the object. Such a type cannot escape the scope of the variable x. On the other hand, the second type does not reveal this alias. This type is less informative, but an object with this type can escape the scope of x. So the general strategy for initializing an object with type fields is to first give it the very precise type resulting from its instantiation, then use subtyping to "hide" the bindings of the type fields.

For example, assuming a default nullary constructor for the above clause, the above code can be elaborated (say for bytecode generation) as:

```
PubKey ⟨Owner=ThisPrin,Objects={Mary,Sue},K=x.K⟩ temp =
     new PubKey ⟨Owner=ThisPrin,Objects={Mary,Sue},K=x.K⟩ ();
temp.init (x.publicKey);
PubKey⟨Owner=ThisPrin,Objects={Mary,Sue}⟩ z = temp;
```

The argument type of temp.init is pubkey⟨temp.K⟩, but the type of temp also exports the type equality K=x.K, so we have temp.K=x.K. It is only because of this type equality that the constructor application is able to type check, since the argument has type pubkey⟨x.K⟩. So equality constraints between policy names are a crucial component of ensuring that copy constructors typecheck.

An object of type

$$\texttt{KeyPair}\langle\texttt{Owner=ThisPrin,Objects=\{Mary,Sue\}}\rangle$$

still cannot be transmitted to another address space, since the principal names ThisPrin, Mary and Sue only make sense within an address space. We can certainly elide this type to

$$\texttt{KeyPair}\langle\rangle$$

This type has no free variables in it, and therefore it can be transmitted across address spaces. However there is then the problem that the resulting object is useless at the receiving site. Because the principals in the policy are locally abstract, there is no way to relate them to principals at the receiving site, and therefore no way to allow principals at the receiving site to access the private parts of the object according to the policy. Because the owner of the policy and the principals allowed read access by the policy, are completely abstract, no principal can ever access the key.

We need a way of hiding some of the principal identities in an object type, transmitting that object to another address space, then safely re-exposing the principal identities at the receiving site. To make this exposure safe, it must be based on checking the identity of authentication keys that identify principals. The following class accomplishes this:

```
class DistKeyPair ⟨prin Owner,
                   prinSet Objects,
                   prin Reader ∈ Objects,
                   spolicy K [Owner:Objects]⟩ {
    final pubkey⟨Reader⟩ rdrKey;
    pubkey⟨K⟩ publicKey;
    KeyPair (pubkey⟨Reader⟩ rdr,
             pubkey⟨K⟩ pub,
             {K}privkey⟨K⟩ priv) {
       this.rdrKey = rdr;
       this.publicKey = pub;
       this.privateKey = priv;
    }
}
```

The generator of the key pair creates an object of this type and elides its type to DistKeyPair⟨⟩. This key is then transmitted to another process. The receiving process uses the rdrKey field to justify exposing the identity of the reader principal name. If the rdrKey field is equal to ThisPrin, then the local principal name Reader can be exposed to ThisPrin at the receiving site. Although the set of allowed readers Objects is still opaque, the set constraint Objects⊇{Reader} justifies allowing the receiving process to access the key. This checked exposure requires a new construct in Jeddak, the keyassert statement that asserts an equality between keys to justify an equality between policy names. If the key assertion succeeds, then the executing process is allowed to access the private key in the received object, using the reflected equality between key names and the inclusion constraint in the object type. This is summarized in Fig. 1.

## 3 Type System

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

```
// Server side:
Object req = reqCh.readObject();
PubIKey kx = (PubIKey)req;

spolicy [ThisPrin:kx.Owner] Kshare;
pubkey⟨Kshare⟩ ksPub;
privkey⟨Kshare⟩{Kshare} ksPriv;

DistKeyPair reply =
  new DistKeyPair⟨...,Reader=kx.K⟩
    (kx.publicKey,ksPub,ksPriv);
replyCh.writeObject(reply);
```

```
// Client side:
PubIKey req = new PubIKey ⟨...,K=ThisPrin⟩ (thisPrin);
reqCh.writeObject(req);

Object reply = replyCh.readObject();
DistKeyPair ky = (DistKeyPair)reply;
keyassert ky.rdrKey==thisPrin;
// Now we know:  ky.Reader=ThisPrin
// So we know:  ThisPrin∈ky.Objects
...  ky.privateKey ...
```

Figure 1: Enforcing Access Restrictions Across Address Spaces

| | | | |
|---|---|---|---|
| $A \in$ Arity, Kind | $::=$ | $\mathsf{Prin}$ | Principal |
| | $\mid$ | $\mathsf{SPolicy}(P:\overline{P})$ | Secrecy policy name |
| | $\mid$ | $\mathsf{IPolicy}(P:\overline{P})$ | Integrity policy name |
| | $\mid$ | $\mathsf{Type}$ | Type |
| $K,P,T \in$ Key, Prin, Type | $::=$ | $k,p,t$ | Variable, name |
| | $\mid$ | $x.t,\ a.t$ | Type field |
| | $\mid$ | $\mathsf{PrivKey}(K)$ | Private key type |
| | $\mid$ | $\mathsf{PubKey}(K)$ | Public key type |
| | $\mid$ | $C\langle \overline{t=OT} \rangle$ | Class |
| $OT \in$ Optional Type | $::=$ | $T \quad \mid \quad \top$ | |
| $L \in$ Label | $::=$ | $\{K_1,\ldots,K_m\}$ | |
| $LT \in$ Labelled type | $::=$ | $[T]^{L_1,L_2}$ | |

Figure 2: Syntax of Kinds and Types

arities or kinds organizes the well-formed types, principal names and policy names. Besides the kind of types ($\mathsf{Type}$) and principal names ($\mathsf{Prin}$), there are two kinds for policy names: encryption (secrecy) policy names $\mathsf{SPolicy}(P:\overline{P})$ and signing (integrity) policy names $\mathsf{IPolicy}(P:\overline{P})$. This kind identifies the creator of the policy ($P$) and the principals who may have access to the private part of the public-private key pair for that policy ($\{\overline{P}\}$).

The types then consist of key types, indexed by policy names, classes, and opaque types resulting from accessing type fields in objects. The latter type has the form $x.t$ or $a.t$ where $x$ is a program variable and $a$ is an object identity. This restriction on the form of such opaque types only permits a limited dependency of types on values, maintaining the phase distinction between values and types. A class type has the form $C\langle \overline{t=OT} \rangle$ where each $OT$ is either null $\top$ or a type definition $T$. The former denotes a type in the instantiation of a class that has been made opaque, the latter a type whose instantiation is revealed in the object type.

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 are 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 several forms of environments:

$$\begin{array}{rcl}
TE \in \text{Type Env} & ::= & \varepsilon \mid (t:A) \mid TE_1,TE_2 \\
VE \in \text{Value Env} & ::= & \varepsilon \mid (x:LT) \mid (a:LT) \mid VE_1,VE_2 \\
CE \in \text{Class Env} & ::= & \{\} \mid \{C \mapsto Cl\} \mid CE_1 \cup CE_2 \\
EE \in \text{Constraint Env} & ::= & \{\} \mid \{K_1 = K_2\} \mid EE_1 \cup EE_2
\end{array}$$

The sequence concatenation operation (,) is assumed to be associative with $\varepsilon$ (empty sequence) as its unit. The type environment $TE$ binds type, principal and policy names to their kinds. The value environment $VE$ binds variables, keys and object identifiers to their types. The class environment maps from a class name to the definition of that class. Since this environment is invariant throughout both the static and dynamic semantics, we leave it implicit in the semantic rules. Finally the constraint environment $EE$ records hypothetical equalities between policy names, based on runtime key equality checks. This latter environment is only necessary because of the keyassert construct.

The type environment $TE$ is used in the definition of a meta-function that, given a label (set of policy names), computes the set of principals that is defined by that label in that type environment:

$$\begin{array}{rcl}
PRINS_{TE;VE}(\{\}) & = & \{\} \\
PRINS_{TE;VE}(L_1 \cup L_2) & = & PRINS_{TE;VE}(L_1) \cap PRINS_{TE;VE}(L_2) \\
PRINS_{TE;VE}(\{k\}) & = & \{\overline{P}\} \\
& & \text{if } \exists TE_1,TE_2.\ TE = (TE_1, k:\mathsf{SPolicy}(P:\overline{P}), TE_2) \\
PRINS_{TE;VE}(\{x.k\}) & = & \{\overline{P}\} \\
& & \text{if } \left\{ \begin{array}{l} \exists VE_1,VE_2.\ VE = (VE_1, x:[C\langle \overline{t=OT} \rangle]^{L_1,L_2}, VE_2) \\ CE(C) = (\mathsf{class}\ C\langle \overline{t:A} \rangle \ldots) \\ k = t_i \text{ and } \{\overline{x.t/t}\}A_i = (\mathsf{SPolicy}(P:\overline{P})) \end{array} \right.
\end{array}$$

An equivalent definition can be given for the set of principals allowed "write" access according to integrity labels.

The formation rules for environments and kinds are omitted for lack of space. The formation rules for (labelled) types are also omitted. These rules check the obvious well-formedness conditions, e.g., that in an encryption key $\mathsf{PubKey}(K)$, the argument $K$

| | | | |
|---|---|---|---|
| $e \in$ Expression | $::=$ | $w, x, y, z$ | Variable |
| | $\vert$ | $a, b, c, n$ | Name (Object id, Key) |
| | $\vert$ | $\mathsf{newKey}\langle k : A\rangle(a^+ : LT_1, a^- : LT_2)\{e\}$ | New policy name |
| | $\vert$ | $\mathsf{new}\ [C\langle\overline{t = T}\rangle]^{L_1,L_2}(\overline{e})$ | Create object |
| | $\vert$ | $v.x$ | Variable access |
| | $\vert$ | $v.x(\overline{e})$ | Method invocation |
| | $\vert$ | $(LT\ x = e_1;\ e_2)$ | Local variable |
| | $\vert$ | $\mathsf{upcast}_{L_1,L_2} e$ | Upcast on labels |
| | $\vert$ | $(\mathsf{keyassert}\ e_1 == e_2\ \mathsf{in}\ e;\ \mathsf{else}\ e')$ | Assert key equality |
| | $\vert$ | $v$ | Value |
| $v \in$ Value | $::=$ | $x \quad \vert \quad \langle\!\langle a\rangle\!\rangle^{L_1,L_2}_{P,()}$ | |
| $K \in$ Constructor | $::=$ | $C(\overline{LT_2}\ \overline{x_2}, \overline{LT_1}\ \overline{x_1})\{\mathsf{super}(\overline{x_2});\ \overline{\mathsf{this}.x_1} = \overline{x_1};\}$ | |
| $M \in$ Method | $::=$ | $LT\ f(\overline{LT}\ \overline{x})\{\mathsf{return}\ e;\}$ | |
| $Cl \in$ Class | $::=$ | $\mathsf{class}\ C\langle\overline{t_2}, \overline{t_1} : \overline{A_2}, \overline{A_1}\rangle\ \mathsf{extends}\ C_2\langle\overline{t_2} = \overline{t_2}\rangle\ \{\overline{LT}\ \overline{x};\ \overline{M}\ K\}$ | |

Figure 3: Syntax of Expressions and Classes

denotes a policy name with an encryption policy 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.

The syntax of values, expressions and processes is provided in Fig. 3. For type preservation purposes, we also introduce a notion of *annotated values* $\langle\!\langle a\rangle\!\rangle^{L_1,L_2}_{P,()}$, where $a$ is a name, either an object identifier or a key. The motivation for this is declassification, discussed in the conclusions.

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

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

to check that the expression (code) $e$ is well-formed with annotated type $[T]^{L_1,L_2}$ (secrecy label $L_1$ and integrity label $L_2$), under the assumption that it will be evaluated (executed) under the authority of the principal $P$, in the corresponding type, value and constraint environments $TE$, $VE$ and $EE$, respectively.

The type rule for policy generation is given by the (VAL NEWKEY) rule in Fig. 4. Actual key generation introduces not only a new policy name, but also a public-private key pair for that policy name. This key pair is denoted by the pair of dual names $(a^+, a^-)$. The decryption key $a^-$ has an encryption key type indexed by the new policy name, and the compatibility restrictions between the kind $A$ of this policy name and the labels of the key type are enforced before the key and its type are added to the value environment.

Aliasing of type fields in objects, reflected by type sharing constraints in object interfaces, is a fundamentally important part of our mechanism for sharing keys, since policy names are locally opaque in objects. Consider for example:

$C\langle k = \top\rangle\ x$;   // $k$ is local opaque key name
$C\langle k = x.k\rangle\ x$;   // by (Val Obj Self)
$C\langle k = x.k\rangle\ y = x$;
$\texttt{pubkey}\langle C\langle k\texttt{=x.}k\rangle\rangle\ z = e\ ?\ x : y$;

Allowing either $x$ or $y$ to be assigned to $z$ is allowed because of the type sharing constraint in the types of $z$ and $y$.

Because of this treatment of type sharing, parameterized types are handled differently from GJ [4]. Whereas the latter allows class extensions of the form:

$$\mathsf{class}\ C_1\langle\overline{t}\rangle\ \mathsf{extends}\ C_2\langle\overline{T}\rangle\ \{\ldots\},$$

we restrict the forms of such extensions to

$$\mathsf{class}\ C_1\langle\overline{t_2}, \overline{t_1} : \overline{A_2}, \overline{A_1}\rangle\ \mathsf{extends}\ C_2\langle\overline{t_2} = \overline{t_2}\rangle\ \{\ldots\}.$$

In other words, the extending class can only add type parameters to the extended class, and the first $|\,\overline{t_2}\,|$ type parameters are aliases for the type parameters to the extended class. This is necessary because of our treatment of type equality, and particularly the (VAL OBJ SELF) rule for allowing an object to refer to its own identity in the definition of its type exports. To see why the restriction on class extensions is necessary, consider the following example:

$\mathsf{class}\ C_1\langle t_1 : \mathsf{Type}, t_2 : \mathsf{Type}\rangle\ \mathsf{extends}\ C_2\langle t_1 = t_2, t_2 = t_1\rangle\ \{\ldots\}$
$C_1\langle t_1 = T_1, t_2 = T_2\rangle\ x$;
$C_1\langle t_1 = x.t_1, t_2 = x.t_2\rangle\ x$;   // by (Val Obj Self)
$C_2\langle t_1 = x.t_2, t_2 = x.t_1\rangle\ x$;   // by (Val Subsumption)
$C_2\langle t_1 = x.t_1, t_2 = x.t_2\rangle\ x$;   // by (Val Obj Self)

From this it is necessary to conclude that $x.t_1 = x.t_2$ and $x.t_2 = x.t_1$, which in general will not be true.

The type rules for expressions are provided in Fig. 4. In the (VAL ACCESS) and (VAL INVOKE) rules for field access and method invocation, respectively, the definitions of the type parameters $\overline{t}$ are relativized to the object itself ($\overline{x.t}$ if the object is referenced through the variable $x$, for example). This gives the field or method the most general type possible, with other equivalent types derivable using type equality and the type definitions $\overline{t} = \overline{T}$ exported in the object interface.

The (VAL ACCESS), (VAL INVOKE) and (VAL KEY ASSERT) rules all involve accessing a value, so any result from the ensuing computation carries with it the restrictions on the label for the value. This is the point of the merge that is done on the labelled type of the result, incorporating the restrictions from the value's labels into the result.

The subtyping and equality rules for types are provided in Fig. 4. The (VAL OBJ SELF) rule allows any object's type to be redefined to relativize the identity of its type fields, some of which may be opaque, to the reference to the object itself. For example the $t$ field of an object referenced by variable $x$ can be relativized to have the definition $x.t$, reflecting the identity of the type that is relative to $x$.

The (VAL SUBSUMPTION) rule in Fig. 4 allows any value to have its type to be subsumed to some other type. This includes

$$\frac{\begin{array}{c} A = \mathsf{SPolicy}(P : \overline{P}) \qquad TE;VE \vdash A\ \mathbf{kind} \\ LT_1 = [\mathsf{PubKey}(k)]^{L_1,L_1'} \qquad (TE,k:A);VE \vdash LT_1 : \mathsf{Type} \\ LT_2 = [\mathsf{PrivKey}(k)]^{L_2,L_2'} \qquad (TE,k:A);VE \vdash LT_2 : \mathsf{Type} \\ (TE,k:A);(VE,a^+ : LT_1, a^- : LT_2);EE \vdash e \in^P [T]^{L,L'} \qquad k \notin tv([T]^{L_1,L_2}) \end{array}}{TE;VE;EE \vdash^O \mathsf{newKey}\langle k:A\rangle(a^+ : LT_1, a^- : LT_2)\{e\} \in^P [T]^{L,L'}} \quad \text{(VAL NEWKEY)}$$

$$\frac{TE;VE;EE \vdash e \in^P [T]^{L_1,L_2} \qquad P \in PRINS_{TE;VE}(L_1)}{TE;VE;EE \vdash^I e \in^P [T]^{L_1,L_2}} \quad \text{(VAL SECRECY)}$$

$$\frac{TE;VE;EE \vdash^O e \in^P [T]^{L_1,L_2} \qquad P \in PRINS_{TE;VE}(L_2)}{TE;VE;EE \vdash e \in^P [T]^{L_1,L_2}} \quad \text{(VAL INTEGRITY)}$$

$$\frac{TE \vdash VE\ \mathbf{venv} \qquad TE;VE \vdash P : \mathsf{Prin} \qquad VE = (VE_1, x : [T]^{L_1,L_2}, VE_2)}{TE;VE;EE \vdash x \in^P [T]^{L_1,L_2}} \quad \text{(VAL VAR)}$$

$$\frac{TE \vdash VE\ \mathbf{venv} \qquad TE;VE \vdash P : \mathsf{Prin} \qquad VE = (VE_1, n : [T]^{L_1,L_2}, VE_2)}{TE;VE;EE \vdash n \in^P [T]^{L_1,L_2}} \quad \text{(VAL NAME)}$$

$$\frac{\begin{array}{c} TE;VE;EE \vdash v \in^P [C\langle \overline{t} = \overline{OT}\rangle]^{L_1,L_2} \qquad VARTYS(C) = (\overline{x} : \overline{LT}) \\ LT = \{\overline{STRIP(v).t}/\overline{t}\}LT_i \end{array}}{TE;VE;EE \vdash v.x_i \in^P MERGE(LT, L_1, L_2)} \quad \text{(VAL ACCESS)}$$

$$\frac{\begin{array}{c} TE;VE;EE \vdash v \in^P [C\langle \overline{t} = \overline{OT}\rangle]^{L_1,L_2} \\ METHODTY(C,f) = (\overline{LT} \to LT) \qquad TE;VE;EE \vdash \overline{e} \in^P \overline{\{\overline{STRIP(v).t}/\overline{t}\}LT} \\ LT' = \{\overline{STRIP(v).t}/\overline{t}\}LT \end{array}}{TE;VE;EE \vdash v.f(\overline{e}) \in^P MERGE(LT', L_1, L_2)} \quad \text{(VAL INVOKE)}$$

$$\frac{\begin{array}{c} TE;VE \vdash C\langle \overline{t} = \overline{T}\rangle : \mathsf{Type} \qquad VARSTY(C) = (\overline{x} : \overline{LT}) \\ TE;VE;EE \vdash \overline{e} \in^P \overline{\{\overline{T}/\overline{t}\}LT} \end{array}}{TE;VE;EE \vdash \mathsf{new}\ [C\langle \overline{t} = \overline{T}\rangle]^{L_1,L_2}(\overline{e}) \in^P [C\langle \overline{t} = \overline{T}\rangle]^{L_1,L_2}} \quad \text{(VAL NEW OBJECT)}$$

$$\frac{\begin{array}{c} TE;VE;EE \vdash e_i \in^P [\mathsf{PubKey}(K_i)]^{L_i,L_i'} \\ TE;VE;EE \cup \{K_1 = K_2\} \vdash e \in^P LT \qquad TE;VE;EE \vdash e' \in^P LT \\ LT' = MERGE(LT, L_1 \cup L_2, L_1' \cap L_2') \end{array}}{TE;VE;EE \vdash (\mathsf{keyassert}\ e_1 == e_2\ \mathsf{in}\ e\ \mathsf{else}\ e') \in^P LT'} \quad \text{(VAL KEY ASSERT)}$$

Figure 4: Formation Rules for Expressions

strengthening the secrecy restrictions and weakening the integrity restrictions in the outermost labels, as well as various forms of type equality and subtyping. There are two sources of type equality hypotheses. One is equations in the constraint environment *EE*, added by keyassert; such constraints are considered using the (CON CONS) type equality rule. The other source of type equality hypotheses is in type definitions exported in object interfaces, considered by the (CONEQ PATH) type equality rules. The type equality rule (CONEQ OBJ) is simply a congruence rule for equality on object interfaces.

There are two subtyping rules, both for object types. The (CONSUB CLS) rule allows some of the type definitions in an object type to be elided or made opaque in the object interface, as given by the (CONSUB CLS HIDE) rule that may be used for some of the exported type definitions, replacing type definitions with $\top$. The (CONSUB EXTENDS) rule is the classic subtype rule based on named inheritance between classes. In this case the rule must also take cognizance of extra type parameters added by the subclass, that must be elided when widening the original type to the base class type.

## 4 Operational Semantics

In this section we provide the operational semantics for our language. To maintain the phase distinction between types and values, even though types may include types of the form $x.t$ where $x$ is bound to a value of object type, we give a semantics where objects are allocated on an implicit heap and are referred to by references. In other words, we give a pointer-based rather than a functional semantics. An object pointer or identifier is just a name bound in the surrounding context. We extend the syntax of expressions with binders for newly generated policy names and value names (keys or object identifiers), and with structural congruence rules that allow such binders to be moved out of an evaluation context (modulo scoping of free variables).

We also extend the syntax of expressions with object definitions of the form $a = \mathsf{new}\ C\langle \overline{t} = \overline{T}\rangle(\overline{v})$, denoting the definition of an object that has been allocated on the heap at address $a$. The semantics includes structural congruence rules that allow such an object definition to be moved inside any evaluation context where the object's definition is required (a field is being accessed or a method is being invoked). In general any expression will have the form

$$e \quad | \quad a_1 = \mathsf{new}\ C_1\langle \overline{t_1} = \overline{T_1}\rangle(\overline{v_1}) \quad | \quad \ldots \quad | \quad a_k = \mathsf{new}\ C_k\langle \overline{t_k} = \overline{T_k}\rangle(\overline{v_k})$$

where $k$ object definitions have been brought into the local context by structural congruence rules. This invariant is maintained by the extensions to the type system to type such configurations. We will say that an expression is *irreducible* if in the example above $e = v$, i.e., generalizing the notion of a value to this situation where expressions may contain object definitions. The full set of extensions to expressions to describe evaluation is given in Fig. 5.

To provide the semantics, we define an *evaluation context* $\mathbb{E}[\ .\ ]$ for expressions that allows evaluation within a particular configuration. The semantics is provided as a labelled transition system (LTS) in Fig. 6, using the following reduction relations:

$$\begin{array}{ll} e \xrightarrow{TE;VE} e' & \text{"Internal" computation} \\ e \xrightarrow{TE;VE;P} e' & \text{Computation under a principal} \end{array}$$

The first of these relations, for internal computation, is for computation that is independent of the principal for whom the code is executing, and does not offer interaction to any external observer. The (RED INTERN NEWACTKEY) allocates a new policy name and the corresponding key pair. We only give the cases for secrecy policy names, for brevity. The (RED PRIN NEWOBJ) rule allocates a new object on the heap, generating a new name for the reference for that object. The (RED PRIN ACCESS) and (RED PRIN INVOKE) rules, for accessing a field and invoking a method, respectively, use this object definition, which must be brought into the context of the access or invocation using structural congruence. The (RED PRIN UPCAST) rule adds security restrictions that follow from the control path that was followed to produce a particular value. For example, when a field is accessed in an object, the value retrieved has not only the restrictions on the field value itself, but also the access restrictions on the object pointer that was used to access that field in the first place. Similar remarks apply to the result of

| | | | |
|---|---|---|---|
| $e \in$ Expression | $::=$ | $a = \mathsf{new}\ C\langle\overline{t = T}\rangle(\overline{v})$ | Object definition |
| | $\mid$ | $\mathsf{new}(k : A)\{e\}$ | Policy name |
| | $\mid$ | $\mathsf{new}(a : LT)\{e\}$ | Object id, key |
| | $\mid$ | $(e_1 \mid e_2)$ | Parallel composition |
| | $\mid$ | $P[e]$ | Execute as principal |

Figure 5: Syntax of Expressions extended with Object Definitions

a method invocation. The (RED PRIN RETURN) rule discards the context that was allocated when the code for a method signed by a particular principal started executing. All object definitions should have been moved out of the context before this is done.

**Theorem 1 (Type Preservation).** *The following propositions are true:*

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

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

**Theorem 2 (Progress).** 1. *Suppose* $TE; VE; \{\} \vdash e \in^P [T]^{L,L'}$. *Then either e is irreducible or* $e \xrightarrow{TE;VE} e'$ *for some* $e'$.

2. *Suppose* $TE; VE; \{\} \vdash e \in^P [T]^{L,L'}$. *Then either e is irreducible or* $e \xrightarrow{TE;VE;P} e'$ *for some* $e'$.

The semantics incorporates access control checks in various places: an integrity check in (RED PRIN NEWOBJ), and secrecy checks in (RED PRIN ACCESS), (RED PRIN INVOKE), (RED PRIN KEYASST SUCC) and (RED PRIN KEYASST FAIL). The progress result demonstrates that evaluation can never "get stuck" because of an access check failure, and so the runtime access checking in the semantics is unnecessary.

## 5 Conclusions

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 [10, 11] have developed a type-based approach to verifying authentication protocols. Abadi and Blanchet [2, 3] 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 [21, 20, 17] and preventing unwanted security flows in programs [7, 18, 27, 23]. Sabelfeld and Myers [24] provide an excellent overview of work in language-based information-flow security. Pottier and Conchon [23] 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 [22] has extended this to the pi-calculus. That approach can be applied to the object calculus introduced in the current article. In the presence of declassification, it is a well-known problem to define what safety guarantees are provided by information flow control [24, 28].

We have not had space to consider declassification in this account, particularly the notion of distributed declassification introduced by KDLM. It appears that the notion of *robust declassification* can be adapted to the framework of KDLM; we intend to report on this in a subsequent paper. A fuller report on Jeddak, including applications and an implementation, are also a topic of ongoing research.

The current article borrows heavily from work on type systems for module languages [13, 12, 14, 15, 25, 8]. In effect objects in our object calculus are first-class modules, with interface inclusion based on the inheritance hierarchy. However module languages do not normally consider re-exposing a type field that has been made opaque, as we do with `keyassert`. Also of course module languages do not normally enforce access control and information flow control. The semantics of module languages are normally functional; our object-calculus based semantics appears to be novel.

## References


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

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

[4] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, pages 183–200, Vancouver, British Columbia, 1998. ACM Press.

[5] Robert Cartwright and Guy Steele. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, pages 201–215, Vancouver, British Columbia, 1998. ACM Press.

$$\frac{e_1 \xrightarrow{(TE,k:A);VE} e_2}{\mathsf{new}(k:A)\{e_1\} \xrightarrow{TE;VE} \mathsf{new}(k:A)\{e_2\}} \qquad \text{(Red Intern NewTy)}$$

$$\frac{e_1 \xrightarrow{TE;(VE,a:LT)} e_2}{\mathsf{new}(a:LT)\{e_1\} \xrightarrow{TE;VE} \mathsf{new}(a:LT)\{e_2\}} \qquad \text{(Red Intern NewVal)}$$

$$\frac{e_1 \xrightarrow{TE;VE} e_1'}{(e_1 \mid e_2) \xrightarrow{TE;VE} (e_1 \mid e_2)} \qquad \text{(Red Intern Par)}$$

$$\frac{e_1 \xrightarrow{TE;VE;P} e_2}{P[e_1] \xrightarrow{TE;VE} P[e_2]} \qquad \text{(Red Intern Cfg)}$$

$$\frac{e = \mathsf{new}(k:A)\{\mathsf{new}(a^+ : LT_1)\{\mathsf{new}(a^- : LT_2)\{e\}\}\}}{\mathsf{newKey}\langle k:A\rangle(a^+ : LT_1, a^- : LT_2)\{e\} \xrightarrow{TE;VE} e'} \qquad \text{(Red Intern NewKey)}$$

$$\frac{e_1 \xrightarrow{TE;VE;P_0} e_2}{P_0[e_1] \xrightarrow{TE;VE;P} P_0[e_2]} \qquad \text{(Red Prin Cfg)}$$

$$\frac{e_1 \xrightarrow{TE;VE;P} e_2}{\mathbb{E}[e_1] \xrightarrow{TE;VE;P} \mathbb{E}[e_2]} \qquad \text{(Red Prin Ctxt)}$$

$$\frac{e = \mathsf{new}\ [C\langle \overline{t} = \overline{T}\rangle]^{L_1,L_2}(\overline{v}) \qquad P \in PRINS_{TE;VE}(L_2)}{e \xrightarrow{TE;VE;P} \mathsf{new}(a : [C\langle \overline{t} = \overline{T}\rangle]^{L_1,L_2})\{\langle\!\langle a\rangle\!\rangle^{L_1,L_2}_{P,()} \mid a = \mathsf{new}\ C\langle \overline{t} = \overline{T}\rangle(\overline{v})\}} \qquad \text{(Red Prin NewObj)}$$

$$\frac{\begin{array}{c} v = \langle\!\langle a\rangle\!\rangle^{L_1,L_2}_{P_0,()} \qquad P \in PRINS_{TE;VE}(L_1) \\ e = (a = \mathsf{new}\ C\langle \overline{t} = \overline{T}\rangle(\overline{v})) \qquad VARS(C) = (\overline{x}) \end{array}}{(e \mid v.x_i) \xrightarrow{TE;VE;P} (e \mid \mathsf{upcast}_{L_1,L_2} v_i)} \qquad \text{(Red Prin Access)}$$

$$\frac{\begin{array}{c} v = \langle\!\langle a\rangle\!\rangle^{L_1,L_2}_{P_0,()} \qquad P \in PRINS_{TE;VE}(L_1) \qquad e = (a = \mathsf{new}\ C\langle \overline{t} = \overline{T}\rangle(\overline{v})) \\ METHOD(C,f) = (\overline{x}.e) \qquad SIGNER(C) = P_0 \end{array}}{(e \mid v.f(\overline{v'})) \xrightarrow{TE;VE;P} (e \mid P_0[\mathsf{upcast}_{L_1,L_2} e\{P_0/ThisPrin, v/this, \overline{v'}/\overline{x}\}])} \qquad \text{(Red Prin Invoke)}$$

$$\mathsf{upcast}_{L_1',L_2'} \langle\!\langle a\rangle\!\rangle^{L_1,L_2}_{P_0,()} \xrightarrow{TE;VE;P} \langle\!\langle a\rangle\!\rangle^{L_1 \cup L_1', L_2 \cap L_2'}_{P_0,()} \qquad \text{(Red Prin Upcast)}$$

$$P_0[v] \xrightarrow{TE;VE;P} v \qquad \text{(Red Prin Return)}$$

$$\frac{v = \langle\!\langle a\rangle\!\rangle^{L_1,L_2}_{P_0,()} \qquad P \in PRINS_{TE;VE}(L_1)}{(\mathsf{keyassert}\ v == v\ \mathsf{in}\ e;\ \mathsf{else}\ e') \xrightarrow{TE;VE;P} \mathsf{upcast}_{L_1,L_2} e} \qquad \text{(Red Prin KeyAsst Succ)}$$

$$\frac{fv(v_1) = fv(v_2) = \{\} \qquad v_1 \neq v_2 \qquad v_i = \langle\!\langle a_i\rangle\!\rangle^{L_i,L_i'}_{P_i,()} \qquad P \in PRINS_{TE;VE}(L_1 \cup L_2)}{(\mathsf{keyassert}\ v_1 == v_2\ \mathsf{in}\ e;\ \mathsf{else}\ e') \xrightarrow{TE;VE;P} \mathsf{upcast}_{L_1 \cup L_2, L_1 \cap L_1'} e'} \qquad \text{(Red Prin KeyAsst Succ)}$$

Figure 6: Computation Rules


[6] Tom Chothia, Dominic Duggan, and Jan Vitek. Type-based distributed access control. In *Computer Security Foundations Workshop*, Asilomar, California, June 2003. IEEE.

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

[8] Dominic Duggan. Type-safe dynamic linking with recursive DLLs and shared libraries. *ACM Transactions on Programming Languages and Systems*, November 2002.

[9] Dominic Duggan. Type-based cryptographic operations. *Journal of Computer Security*, 2003.

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

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

[12] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994. ACM Press.

[13] Robert Harper, John Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In Paul Hudak, editor, *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 341–354. Association for Computing Machinery, 1990.

[14] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994. acmp.

[15] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 154–163, San Francisco, California, January 1995. ACM Press.

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

[17] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, 1999.

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

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

[20] George Necula. Proof-carrying code. In *ACM Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[21] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Operating Systems Design and Implementation*, 1996.

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

[23] Francois Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of ACM International Conference on Functional Programming*, 2000.

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

[25] Zhong Shao. Transparent modules with fully syntactic signatures. In *Proceedings of ACM International Conference on Functional Programming*, Paris, France, September 1999.

[26] Jose H. Solorzano and Suad Alagic. Parametric polymorphism for Java: A reflective solution. In Craig Chambers, editor, *Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications*, Vancouver, British Columbia, 1998.

[27] 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.

[28] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Computer Security Foundations Workshop*, 2001.


## A Miscellaneous Definitions

$$\frac{TE;VE;EE \vdash v \in^P [C\langle \overline{t} = \overline{OT}\rangle]^{L_i,L_i'}}{TE;VE;EE \vdash v \in^P [C\langle \overline{t} = \overline{STRIP(v).t}\rangle]^{L_i,L_i'}} \quad (\text{VAL OBJ SELF})$$

$$\frac{\begin{array}{c} TE;VE;EE \vdash e \in^P [T]^{L_1,L_2} \quad L_1 \subseteq L_1' \quad L_2' \subseteq L_2 \\ TE;VE \vdash L_i' \textbf{ label} \quad TE;VE \vdash T' : \mathsf{Type} \quad TE;VE;EE \vdash T \leq T' \end{array}}{TE;VE;EE \vdash e \in^P [T']^{L_1',L_2'}}$$
$$(\text{VAL SUBSUMPTION})$$

$$\frac{TE;VE \vdash EE \textbf{ eenv}}{TE;VE;EE \vdash T = T} \quad (\text{CONEQ REFL})$$

$$\frac{TE;VE;EE \vdash T_2 = T_1}{TE;VE;EE \vdash T_1 = T_2} \quad (\text{CONEQ SYM})$$

$$\frac{TE;VE;EE \vdash T_1 = T_2 \quad TE;VE;EE \vdash T_2 = T_3}{TE;VE;EE \vdash T_1 = T_3} \quad (\text{CONEQ TRANS})$$

$$\frac{TE;VE \vdash EE \textbf{ eenv} \quad (K_1 = K_2) \in EE}{TE;VE;EE \vdash K_1 = K_2} \quad (\text{CONEQ CONS})$$

$$\frac{TE;VE \vdash C\langle \overline{t} = \overline{OT}\rangle : \mathsf{Type} \quad TE;VE;EE \vdash \overline{OT} = \overline{OT'}}{TE;VE;EE \vdash C\langle \overline{t} = \overline{OT}\rangle = C\langle \overline{t} = \overline{OT'}\rangle}$$
$$(\text{CONEQ OBJ})$$

$$\frac{TE;VE \vdash EE \textbf{ eenv} \quad VE = (VE_1, x : C\langle \overline{t} = \overline{OT}\rangle, VE_2) \quad OT_i = T_i}{TE;VE;EE \vdash x.t_i = \{\overline{x.t}/\overline{t}\}T_i}$$
$$(\text{CONEQ PATH})$$

$$\frac{TE;VE \vdash EE \textbf{ eenv} \quad VE = (VE_1, a : C\langle \overline{t} = \overline{OT}\rangle, VE_2) \quad OT_i = T_i}{TE;VE;EE \vdash a.t_i = \{\overline{a.t}/\overline{t}\}T_i}$$
$$(\text{CONEQ PATH})$$

$$\frac{TE;VE;EE \vdash T_1 = T_2}{TE;VE;EE \vdash T_1 \leq T_2} \quad (\text{CONSUB EQ})$$

$$\frac{TE;VE;EE \vdash T_1 \leq T_2 \quad TE;VE;EE \vdash T_2 \leq T_3}{TE;VE;EE \vdash T_1 \leq T_3} \quad (\text{CONSUB TRANS})$$

$$\frac{TE;VE \vdash EE \textbf{ eenv} \quad TE;VE;EE \vdash \overline{OT} \leq \overline{OT'}}{TE;VE;EE \vdash C\langle \overline{t} = \overline{OT}\rangle \leq C\langle \overline{t} = \overline{OT'}\rangle} \quad (\text{CONSUB CLS})$$

$$\frac{\begin{array}{c} TE;VE \vdash EE \textbf{ eenv} \\ CE(C_1) = (\mathsf{class}\ C_1\langle \overline{t_2}, \overline{t_1} : \overline{A_2}, \overline{A_1}\rangle\ \mathsf{extends}\ C_2\langle \overline{t_2} = \overline{t_2}\rangle\{\ldots\}) \end{array}}{TE;VE;EE \vdash C_1\langle \overline{t_2}, \overline{t_1} = \overline{OT_2}, \overline{OT_1}\rangle \leq C_2\langle \overline{t_2} = \overline{OT_2}\rangle}$$
$$(\text{CONSUB EXTENDS})$$

$$TE;VE;EE \vdash T \leq \top \quad (\text{CONSUB CLS HIDE})$$

$$TE;VE;EE \vdash OT \leq OT \quad (\text{CONSUB CLS REFL})$$

Figure 7: Subtype and Type Equality Rules

| | |
|---|---|
| $TE \vdash VE$ **venv** | Well-formed type and value environment |
| $TE; VE \vdash EE$ **eenv** | Well-formed constraint environment |
| $TE; VE \vdash A$ **kind** | Well-formed kind |
| $TE; VE \vdash LT : A$ | Well-formed type |
| $TE; VE \vdash L$ **label** | Well-formed label |
| $TE; VE; EE \vdash T_1 = T_2$ | Type equality |
| $TE; VE; EE \vdash T_1 \leq T_2$ | Type inclusion |
| $TE; VE; EE \vdash^I e \in^P [T]^{L_1,L_2}$ | Type judgement (inputs checked) |
| $TE; VE; EE \vdash^O e \in^P [T]^{L_1,L_2}$ | Type judgement (outputs to be checked) |
| $TE; VE; EE \vdash e \in^P [T]^{L_1,L_2}$ | Type judgement (outputs checked) |

Figure 8: Judgements for Type System

$$
\begin{array}{rcl}
\mathit{STRIP}(\langle\!\langle a \rangle\!\rangle_{P,()}^{L_1,L_2}) & = & a \\
\mathit{STRIP}(x) & = & x \\
\mathit{MERGE}([T]^{L_1,L_2}, L_1', L_2') & = & [T]^{L_1 \cup L_1', L_2 \cap L_2'} \\
\mathit{VARTYS}(\mathsf{Object}) & = & (() : ()) \\
\mathit{VARTYS}(C_1) & = & (\overline{x_2}, \overline{x_1} : \overline{LT_2}, \overline{LT_1}) \\
 & & \quad_{\text{where}} \quad CE(C_1) = (\mathsf{class}\ C_1 \langle \overline{t_2}, \overline{t_1} : \overline{A_2}, \overline{A_1} \rangle \mathsf{extends}\ C_2 \langle \overline{t_2} = \overline{t_2} \rangle \{ \overline{x_1} : \overline{LT_1};\ \overline{M}\ K \}) \\
 & & \quad\quad\quad\ \ \mathit{VARTYS}(C_2) = (\overline{x_2} : \overline{LT_2}) \\
\mathit{METHODTY}(C_1, x) & = & (\overline{LT} \to LT) \\
 & & \quad_{\text{where}} \quad CE(C_1) = (\mathsf{class}\ C_1 \langle \overline{t_2}, \overline{t_1} : \overline{A_2}, \overline{A_1} \rangle \mathsf{extends}\ C_2 \langle \overline{t_2} = \overline{t_2} \rangle \{ \overline{x_1} : \overline{LT_1};\ \overline{M}\ K \}) \\
 & & \quad\quad\quad\ \ (LT\ x(\overline{LT}\ \overline{y}) \{ \mathsf{return}\ e \}) \in \overline{M} \\
\mathit{METHODTY}(C_1, x) & = & \mathit{METHODTY}(C_2, x)\ _{\text{otherwise}} \\
 & & \quad_{\text{where}} \quad CE(C_1) = (\mathsf{class}\ C_1 \langle \overline{t_2}, \overline{t_1} : \overline{A_2}, \overline{A_1} \rangle\ \mathsf{extends}\ C_2 \langle \overline{t_2} = \overline{t_2} \rangle \{ \overline{x_1} : \overline{LT_1};\ \overline{M}\ K \})
\end{array}
$$

Figure 9: Metafunctions for Semantics