Mobile Functions and Secure Information Flow

Dilsun Kırlı*

1. Introduction

Programming languages which facilitate mobility of code are becoming widely used since the mobile agents paradigm, also known as the mobile computation paradigm, appears to be well-suited for computation in the highly distributed networks of our day. Mobile agents can move from site to site and can be executed remotely using the local resources of the sites they visit. The fact that these resources may be of different sensitivity levels raises the question of providing secrecy so that high sensitivity data is not leaked to parties which may use it in a potentially harmful way. This problem does not only concern controlling access to local resources but also tracing the flow of information through computation. Several authors point out the benefits of addressing such security concerns in the design phase of a mobile code language as opposed to delegating them entirely to the lower-levels of the system. Recently, the approach of formulating secure flow analyses as type systems has earned attention [8, 2, 3]. We contribute to the field of type-based approaches to security by bringing into attention a computational model which has not yet been studied in this context.

We focus on the problem of providing secrecy within the framework of a higher-order mobile code language called Mobile-λ which has its roots in languages such as Concurrent ML [5] and Facile [1]. Mobile-λ supports mobile computation by combining a functional core with constructs for transmitting and receiving values on channels across remote sites. A characteristic feature is that functions and channels are first class values and mobile agents can simply be coded as mobile functions. We discuss an appropriate notion of secrecy for Mobile-λ focusing on information flow aspects rather than access control, and motivate a property which we wish to enforce. Following the type-based approach to security we then propose a type system to enforce

*Laboratory for Foundations of Computer Science, The University of Edinburgh, King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, Scotland, UK. Fax : +44 131 667 72 09, Email : zdk@dcs.ed.ac.uk

this property. Our type system is a direct descendant of the former type and effect systems proposed for ML [6] and Facile [7] which dealt with the problem of providing polymorphism in the presence of features for imperative and concurrent computation and those which were used for communication analyses [4]. A significant advantage of using the type and effect discipline is that providing ML-style polymorphism in this new context, namely allowing the expression of code which behaves uniformly over values of different secrecy levels, becomes a natural extension of the earlier work in this field.

2. System Model

We consider a simple system which consists of two remote sites each of which has a trusted Mobile-λ compiler. The security of the system relies on the combination of authentication and static checks on Mobile-λ programs. This approach to security is in line with the work on a Web navigator with applets written in CAML, a strongly typed functional language of the ML family [3].

As a Mobile-λ program is executed, values which flow in the system can be of different sensitivity levels: secret or public. Likewise, the users of the system are classified into two security classes. High-level users can observe the results of Mobile-λ programs without any restriction while low-level users can observe only those results which are public. In Section 4, we discuss an appropriate notion of secure information flow so that low-security users cannot exploit Mobile-λ programs to deduce secret values.

Each site in the system provides a standard set of values as a part of its computational environment. Binding to these *ubiquitous* values occurs dynamically so that mobile code can use the local copy of a ubiquitous value at the site it is visiting. Ubiquitous values include services coded as functions. Throughout our discussion, we assume that ubiquitous functions do not introduce non-determinism to the system or have side-effects which violate secrecy.

The concurrency in the system is limited to the concurrent evaluation of two sequential threads, one at each site. The security property which we enforce through strong static typing has been formulated with the assumption that the system is closed, that is the communication is internal to the system. However, we are also investigating the possibility of enforcing a similar property in a system where communication between multiple-sites is allowed.

3. Mobile-λ

The abstract syntax of expressions follows closely those of functional languages which support channel-based communication such as Facile and Concurrent ML, the difference being in that expressions in canonical form are labelled with secrecy levels: H for secret, L for public. The writer of the code declares the secrecy level of constants, channels and functions. Values of these expressions constitute the values of the language.

$$
\begin{array}{lrcl}
\text{(label)} & l & ::= & H \mid L \\
\text{(type)} & t & ::= & \mathtt{unit} \mid \mathtt{int} \mid \mathtt{bool} \mid \mathtt{chan}(t) \mid t \rightarrow t \\
\text{(expression)} & e & ::= & scon^l \mid x \mid \mathtt{chan}^l(t) \mid \mathtt{fn}^l\, x \Rightarrow e \mid \mathtt{rec}^l\, f(x) \Rightarrow e \\
& & & \mid e_1\, \mathtt{op}\, e_2 \mid e_1\, e_2 \mid e_1; e_2 \mid \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 \\
& & & \mid \mathtt{if}\, e_1\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3 \mid e_1? \mid e_1!e_2
\end{array}
$$

We have defined a big-step operational semantics for Mobile-λ in natural style. Our semantics is instrumented with annotations which represent communication events that may occur during evaluation. Mobile-λ adopts synchronous communication. Synchronisation between the expressions is specified by a collection of matching rules between communication event sequences. Communication events within the system need to match up for a successful evaluation which yields a value at each site.

4. Information Flow in Mobile-λ

It is a common approach to define security requirements of a system based on an interface model which specifies restrictions on a system's input-output relation. A frequently sought secrecy property is *noninterference* which requires that high-level user inputs cannot interfere with low-level outputs. In Mobile-λ, a datum of information can flow into a function through its environment and argument. The appropriate notion of noninterference would require that the following do not depend on the secret values which flow into a function: i) a public result returned by the function and ii) the set of the public values which flow out of the function on communication channels.

A challenging point in the enforcement of the noninterference property arises from the presence of conditional statements. There always is an implicit flow from the guard of a conditional expression to its branches. Requiring the value returned by the branch of a conditional expression to be at least as secret as its guard is not always sufficient to prevent secrecy violations as is illustrated by Example 1 below.

Example 1: The client requests a function for taking the square of an integer on channel `sq`. It applies it to integer 500 which it declares to be secret. The server which is to provide the function requested by the client, first allocates a public boolean channel `gt100` and defines the function `square` so that it includes a test on the argument of the function. After sending the function to the client the server waits for a boolean result on channel `gt100`.

```
Server:  let   gt100 = chan^L(bool)
         in    let square = fn^L x      => if x > 100
                                           then ( gt100 ! true^L; x * x )
                                           else ( gt100 ! false^L; x * x )
               in ( sq !  square ;      gt100?  )

Client:  let   secretInt = 500^H
         in    (sq?)  secretInt
```

The function `square` returns the desired result to the client. However, it would also allow a low-level user who knows the code of the server to infer whether the secret integer at the client site is greater than 100 by inspecting the boolean value returned on `gt100`. Detecting the possibility of insecure information flow of this nature and rejecting some programs on this basis is an essential contribution of the static type system introduced in Section 5.

There may be cases where a user wants to apply the same function to arguments of different secrecy levels. Our type system facilitates this by allowing types to be parametric in secrecy levels.

Example 2: The client below applies the same function to take the square of a secret and a public integer.

```
Server:  let   square = fn^L x => x*x
               (in sq !  square; sq !  square)

Client:  let   secretInt = 100^H
         in    let publicInt = 200^L
               in ( (sq?)  secretInt; (sq?)  publicInt )
```

5. Type System

Our type system views types τ as a pair consisting of two components: a raw type $\bar{\tau}$ and a secrecy label l. Raw types classify values in the conventional sense whereas secrecy labels associate with them a particular secrecy level. The secrecy labels can be considered as elements of a lattice with least element L and greatest element H where the ordering is denoted by the symbol $\sqsubseteq$ such that $L \sqsubseteq H$. We also use communication effects κ as annotations on function types. These stand for the communication events which may be incurred by a function when it is applied. A limited form of polymorphism is incorporated through variable secrecy labels and universal quantification over these.

(Secrecy label)	l	::=	$H \mid L \mid \gamma$
(Raw type)	$\bar{\tau}$	::=	$\mathsf{Unit} \mid \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{chan}[\bar{\tau}] \mid \tau \xrightarrow{\kappa} \tau$
(Annotated type)	τ	::=	$(\bar{\tau}, l)$
(Effect)	κ	::=	$\emptyset \mid \mathsf{send}(l, \bar{\tau}) \mid \mathsf{recv}(l, \bar{\tau}) \mid \kappa \cup \kappa$
(Type scheme)	σ	::=	$\forall \gamma_1 ... \gamma_n . \tau$

The static semantics for the language assigns a type and an effect (τ, κ) to each expression. This is represented by a judgement of the form $\Gamma \vdash e : \tau, \kappa$ where Γ represents a typing context. We present a subset of the typing rules below. The typing rule **(if)** makes use of the predicate called *Safe*. We use it to enforce that branches of an expression can only send on channels which are at least as secret as the guard of that expression. As is the case for ML-style typing, the generalisation of types to type schemes is performed by the **(let)** rule. The generalisation is allowed only on those variables which are not free in the type environment or the effect associated with the expression.

Definition 1 (Safe Flow) *$Safe(\kappa, l)$ if for all $\mathsf{send}(l', \bar{\tau}) \in \kappa$ it is the case that $l \sqsubseteq l'$.*

(chan) $\quad \Gamma \vdash \mathtt{chan}^l(t) : (\mathsf{chan}[\bar{\tau}], l), \emptyset$

(let)
$$\frac{\Gamma \vdash e_1 : \tau, \kappa \quad \Gamma[x \mapsto Gen(\kappa, \Gamma)(\tau)] \vdash e_2 : \tau_2, \kappa'}{\Gamma \vdash \mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2 : \tau_2, \kappa \cup \kappa'}$$

(if)
$$\frac{\begin{array}{l}\Gamma \vdash e_1 : (\mathsf{Bool}, l), \kappa, \quad \Gamma \vdash e_2 : (\bar{\tau}, l'), \kappa' \quad \Gamma \vdash e_3 : (\bar{\tau}, l'), \kappa'' \\ l \sqsubseteq l' \quad Safe(\kappa', l) \quad Safe(\kappa'', l)\end{array}}{\Gamma \vdash \mathtt{if}\, e_1 \,\mathtt{then}\, e_2 \,\mathtt{else}\, e_3 : (\bar{\tau}, l'), \kappa \cup \kappa' \cup \kappa''}$$

(receive)
$$\frac{\Gamma \vdash e : (\mathsf{chan}[\bar{\tau}], l), \kappa}{\Gamma \vdash e? : (\bar{\tau}, l), \kappa \cup \mathsf{recv}(l, \bar{\tau})}$$

(send)
$$\frac{\Gamma \vdash e_1 : (\mathsf{chan}[\bar{\tau}], l), \kappa \quad \Gamma \vdash e_2 : (\bar{\tau}, l), \kappa'}{\Gamma \vdash e_1 ! e_2 : (\mathsf{Unit}, l), \kappa \cup \kappa' \cup \mathsf{send}(l, \bar{\tau})}$$

We have proved the consistency of the static and dynamic semantics by adopting the proof method of [6] to our setting. This involved defining a consistency judgement which relates objects of the dynamic semantics to objects of the static semantics with respect to a given channel environment. We made use of this judgement also in the formalisation of our notion of noninterference and the proof that it holds for computation in Mobile-λ. The informal statement of noninterference is as follows.

Theorem 1 *If we evaluate a well-typed expression at a given site against two different initial environments which are the same except at bindings for secret values and both evaluations terminate yielding a public value, these values are guaranteed to be equal. Moreover, the two evaluations agree on the values transmitted on public channels.*

Acknowledgements I am grateful to Stephen Gilmore for his helpful comments on this work. My research is funded by a University of Edinburgh studentship.

References

[1] A. Giacalone, P. Mishra, and S. Prasad. Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, April 1989.

[2] N. Heintze and J. G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, CA, USA, 1998.

[3] X. Leroy and F. Rouaix. Security properties of typed applets. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 391–400. ACM Press, 1998.

[4] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proceedings of 21rd ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1994.

[5] J. Reppy. *Higher-order Concurrency*. PhD thesis, Department of Computer Science, Cornell University, 1992.

[6] J.P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111:245–296, 1994.

[7] B. Thomsen. Polymorphic sorts and types for concurrent functional programs. In J. Glauert, editor, *Proceedings of the 6th International Workshop on the Implementatin of Functional Languages*, Norwich,UK, 1994.

[8] D. Volpano and G. Smith. Language issues in mobile program security. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes of Computer Science*, pages 25–43. Springer Verlag, 1998.