# Mobile Code for Key Propagation

Hock Kim Tan    Luc Moreau

Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ UK
{hkvt99r,L.Moreau}@ecs.soton.ac.uk

## ABSTRACT

We introduce the concept of keylets, which are mobile code used to control the propagation of keys in a system, as well as a technique for mobile agent code security that involves encryption of partitioned code components. Keylets are used to support this technique by directing the distribution of keys that decrypt the encrypted components. Formalisation for keylet operations are presented, and a scenario illustrating the use of keylets to implement the technique for mobile agent code security is detailed.

## 1. INTRODUCTION

One of the main concerns currently impeding the wider acceptance and use of mobile agents is the issue of security. Mobile agent security can be broadly divided into two areas : *host security* (protecting the host platform from a malicious agent) and *code security* (protecting the mobile agent from a malicious platform). A good overview is provided in [7], [3] and [9] of both areas and the current techniques available to address them. Mobile agents have also been proposed as a possible alternative in supporting certain aspects of network security. A recent example is the protection of distributed intrusion detection systems by modeling their components as mobile agents that randomly move around a network [8].

In this paper, we present a technique to address the mobile code security issue for mobile agents. Our approach involves partitioning mobile agent code and state information into self-contained components. These components are encrypted using symmetric keys which are then made available to platforms that will host the mobile agent in a network. The distribution of keys to platforms are determined through the execution of a specific type of mobile code that we term as a *keylet*.

The propagation of keys provides a possible mechanism to implement trust propagation as part of a overall trust model [10] for a mobile agent system. The development of such a trust model could provide an efficient way for existing code security techniques to be deployed appropriately (with due consideration to performance) on the basis of trust relationships between the different entities in a mobile agent system [13].

The main contributions of this paper are :

- Introducing the concept of keylets, which are small pieces of mobile code used to control the propagation of keys in a distributed environment.
- Formalisation of the basic operations required of a keylet.
- Demonstration of how keylets and partitioning of code into components could be used in tandem to address the code security aspect for mobile agents.

Section 2 fleshes out the details of code components and keylets and Section 3 presents a formalisation for keylets. Section 4 describes a sample scenario involving the use of keylets and code components, while Section 5 provides a summary and discussion of the work presented. This is rounded off by a conclusion in Section 6.

## 2. KEYLETS AND CODE COMPONENTS

We introduce the idea of keylets by first providing a motivation for their use in a network, which is the protection of code security. Our proposed appoach to code security involves encrypting mobile agent code using conventional symmetric key techniques to produce cipher text that is neither executable nor interpretable directly. Decryption of the cipher text using the same symmetric key produces the original code, which can then be executed directly. Mobile agents are propagated in an encrypted form; executing them necessitates possession of the corresponding symmetric key by the host platform.

### 2.1 Code and State Partitioning

Instead of encrypting the original code of the mobile agent as a whole, we could partition it into several main components, each of which is encrypted/decrypted using different symmetric keys. Code, data and state information resulting from code execution can be organised into such self-contained components based on their corresponding relevance to the correct functioning of the mobile agent. In effect, each component represents a different security level of the overall code and state that needs to be safeguarded. The use of components here is analogous to the idea of fine-grained access control found in some programming languages. Fine grained access control is used to safeguard resources on the host machine from potentially malicious code execution, while at the same time offering partial but flexible access to resources for different types of code. In our work, the resource to be safeguarded is not the machine but the code, and the components provide the granularity for flexible access to different parts of the code.

The initial process of encrypting code components is performed by the agent owner, while the task of partitioning is executed by a third party code producer who could suppy the mobile agent as a template to the agent owner. The generation of the encryption keys can either be undertaken by the agent owner directly or offloaded to another trusted third party. Once the components of the mobile agent are encrypted with their corresponding keys, the agent is then dispatched to the first platform in its itinerary. The only issue that remains is ensuring the correct keys are propagated to the correct platforms so that these platforms are able to decrypt and execute specific components of the mobile agent. If we consider the agent owner site as the sole source for keys in the system, then all distribution of keys must commence from there. This becomes impractical when there is a large number of platforms that will host the mobile agent, since the agent owner site now becomes the focal point (and possible bottleneck) for processing requests and distributing keys. A more feasible option would be to allow platforms already in possession of keys (from an initial distribution from the agent owner site) to duplicate and propagate these keys onwards to other platforms that may need them. We now have a requirement for a mechanism for specifying key distribution activities, as well as the need for this mechanism be applicable on different platforms throughout the system. Our proposal for implementing such a mechanism is through the use of mobile code.

The propagation of keys in our system as described above assumes the support of an available public-key infrastructure for the dissemination of public keys (of all the platforms in the system). The symmetric keys are distributed in an encrypted form (using the public keys of their intended recipients) in order to prevent possible interception and duplication of them.

## 2.2 Keylets

In order to oversee and coordinate the propagation process occurring on the various host platforms, we introduce the concept of keylets. Keylets are mobile code used solely to direct the propagation of keys in a distributed system. They are created and launched by the agent owner and will begin the initial process of distributing keys from the agent owner's site to selected platforms in the system. After finishing this initial distribution, they can be dispatched to these platforms to direct the selective propagation of the keys available there onwards to other platforms in turn. The manner in which keylets are deployed thus make them similar conceptually to code appended to packets found in active networks [4]. The main difference now is that while the processing of the appended code in active networks does not generally involve rerouting of the packets, keylets in turn are specifically involved in determining the propagation direction of the keys.

Since keylets now determine the availability of keys which are necessary to reveal and execute mobile agent code on any platform, their integrity now becomes an issue of concern. Addressing code security for keylets is however quite different from conventional mobile agent code for two reasons :

1. Keylet functionality is limited only to key propagation and is therefore unrelated to mobile agent functionality. The offshot of this is that keylets need not execute on the same host platforms as mobile agents. We could have a situation where keylets migrate within a closed, trusted network of platforms that function as key depositories. Keylets could then propagate keys internally within this network, as well as to an external, untrusted network hosting mobile agents. Since there is no need for keylets to communicate with mobile agents directly, the risk of compromise is minimized.
2. Even if keylets were hosted in untrusted environments, the limited functionality of keylets means that its operations are well defined and understood. Consequently, available mechanisms used to verify general agent code integrity ([15], [6]) could be refined to work on keylets to achieve better results. In effect, the burden of verifying general agent code integrity is simplifed to verifying keylet integrity.

Keylets are thus not a new technique per se for addressing code security, rather they provide a higher level framework in which the available techniques can be employed more simply and effectively. In the example scenario that we will develop in Section 4, keylets are deployed on the same untrusted platforms as the mobile agents, making it necessary to consider their integrity as well. We do not discuss how this issue is tackled in the example but note that such security considerations are likely to be simpler for keylets than for a general mobile agent.

### 2.2.1 Fragmenting keys

We suggest that keys be manipulated in specific ways in order to minimize the risk of incorrect propagation of keys (i.e. keys being propagated to untrusted sites) resulting from the subversion of a keylet. A key can be considered as a linear sequence of bytes which can be partitioned into several smaller byte sequences. We give these smaller sequences the term *fragments*. This partitioning is undertaken by the agent owner during or after the initial phase of key generation. Once the partitioning is complete, these fragments can then be propagated along separate paths before reaching some common destination, where they are eventually reconstituted to form the original key. Thus, incorrect propagation of a few fragments is insufficient to permit a hostile platform to reconstitute the original key and gain unauthorised access to the code component protected by that key.

Alternatively, if the size of the fragments becomes small enough to make brute-force cryptanalysis viable, other methods such as multiple encryption of a component key with other secondary keys could be used. Again the motivation is to ensure that several different keys (propagated along different paths) are necessary in order to gain access to a particular code component.

## 3. FORMALISATION OF KEYLETS

An initial formalisation of keylets is presented in this section. This work is still at an early stage but is representative of the formalisation methodology that we wish to develop

| | | | |
|---|---|---|---|
| $\mathcal{F}$ | $=$ | $\{\ f_0,\ f_1,\ \ldots\ \}$ | (Set of key fragments) |
| $\mathcal{K}$ | $=$ | $VectorOf(\mathcal{F})$ | (Set of keys) |
| $\mathcal{EC}$ | $=$ | $\{\ ec_0,\ ec_1,\ \ldots\ \}$ | (Set of encrypted codelets) |
| $\mathcal{VC}$ | $=$ | $\{\ vc_0,\ vc_1,\ \ldots\ \}$ | (Set of visible codelets) |
| $\mathcal{CN}$ | $=$ | $\{\ n^c_0, n^c_1,\ \ldots\ \}$ | (Set of codelet names) |
| $\mathcal{PEC}$ | $=$ | $\mathcal{CN} \times \mathcal{EC}$ | (Pair of encrypted codelets) |
| $\mathcal{PVC}$ | $=$ | $\mathcal{CN} \times \mathcal{VC}$ | (Pair of visible codelets) |
| $MobileCode$ | $=$ | $\mathcal{CN} \to \mathcal{EC}$ | (Encrypted Code) |
| $\mathcal{T}$ | $=$ | $\{\ t_0,\ t_1,\ \ldots\ \}$ | (Set of tags for mobile agents) |
| $\mathcal{PK}$ | $=$ | $\mathcal{K} \times \mathcal{T} \times \mathcal{CN}$ | (Set of key tuples) |
| $\mathcal{NF}$ | $=$ | $\mathcal{T} \times \mathcal{CN} \times Integer \times Integer$ | (Set of key fragment names) |
| $\mathcal{PF}$ | $=$ | $\mathcal{NF}\ \times\ \mathcal{F}$ | (Set of key fragment pairs) |
| $\mathcal{S}$ | $=$ | $\{\ s_0,\ s_1,\ \ldots\ \}$ | (Set of sites) |
| $Keylet$ | $=$ | $ListOf(KeyletOperation)$ | (List of keylet operations) |
| $KeyletOperation$ | $=$ | $\textsf{migrate} : \mathcal{S} \to KeyletOperation\ \vert$ | (Keylet operations) |
| | | $\textsf{wait} : \mathcal{NF} \times \ldots \times \mathcal{NF} \to KeyletOperation\ \vert$ | |
| | | $\textsf{send} : \mathcal{S} \times \mathcal{NF} \times \ldots \times \mathcal{NF} \to KeyletOperation$ | |
| $merge$ | $:$ | $\mathcal{F} \times \ldots \times \mathcal{F} \to \mathcal{K}$ | |
| $decrypt$ | $:$ | $\mathcal{EC} \times \mathcal{K} \to \mathcal{VC}$ | |
| $Config$ | $=$ | $\mathcal{S} \to SiteConfig$ | (Complete Configuration) |
| $SiteConfig$ | $=$ | $\mathbb{P}(\mathcal{PF}) \times \mathbb{P}(\mathcal{PK}) \times \mathbb{P}(\mathcal{PEC}) \times \mathbb{P}(\mathcal{PVC}) \times\ \mathbb{P}(\mathcal{VC}) \times \mathbb{P}(Keylet)$ | (Site Configuration) |

**Figure 1: State Space (Sets)**

in specifying keylets. A possible use for a complete formal specification would be the static analysis of keylets to identify security or system properties in a mobile agent system (or a general distributed system) that are preserved through their use. This could aid in the automated or partially automated generation of keylets that fulfill specific security requirements.

A keylet can be defined generally as a piece of code that controls the way a fundamental unit of cryptographic information (or a group of such fundamental units) is distributed within a system. In our work in this paper, we utilise the concept of key fragments, which are essentially portions of a complete key. A key can thus be reconstructed by concatenating its component fragments in the appropriate order. Keylets are therefore pieces of mobile code whose primary functionality is to direct the propagation of these fragments to the various platforms in the system.

In order to formally introduce keylets, we model the system that the keylets are deployed in as an abstract machine. The state space for this machine is presented in Figures 1 and 2, for which details will be provided in the following section. The execution of keylet operations are then simply modelled as transitions that change the state of this abstract machine. In addition, platforms hosting keylets also need to execute operations that complement the functionality of the keylet in protecting code security. These operations will be modelled as transitions causing state change as well. The transitions resulting from both types of operations (keylet and platform) are shown in Figure 3 and discussed intuitively in Section 3.2. The last section looks at additional considerations as well as future possibilities for the evolution of keylet operations.

## 3.1 State space of the abstract machine

The fundamental unit of cryptographic information that a keylet works with is a key fragment. In our system, there can be several keylets executing at any time, each of them controlling the distribution of one or more fragments. We therefore provide a set of fragments, $\mathcal{F}$, that represents all the fragments available in the system. A complete key is obtained simply by concatenating together its constituent fragments in the correct sequence. A key, $\mathcal{K}$, can therefore be defined as a vector of fragments.

A mobile agent is composed of several self-contained code components, which we shall label as *codelets*. Each codelet is identified by a unique name and is encrypted/decrypted by a corresponding key. The set of all encrypted codelets is represented by $\mathcal{EC}$, the corresponding set of decrypted codelets is represented by $\mathcal{VC}$, and the set of unique names for both encrypted/decrypted codelets is $\mathcal{CN}$. A link between these unique names and the encrypted/decrypted codelets is provided through the tuples $\mathcal{PEC}$ and $\mathcal{PVC}$ respectively. We can therefore consider a mobile agent as a function mapping a unique name onto an encrypted codelet. $\mathcal{T}$ is the set of tags used to identify all the mobile agents in the system. Thus a combination of a tag and a unique name maps to a specific encrypted codelet in the system. This information is required when using a key, since we need to identify the codelet that the key decrypts. We thus have an additional set, $\mathcal{PK}$, whose elements are tuples of these related items ($k$, $t$, $n^c$).

The tag/unique name combination is also required when fragments are used to reconstitute a complete key, in order to identify the encrypted codelet that the newly constructed key will decrypt. In addition, it is also necessary to specify the number of (predefined) fragments required to reconstitute a complete key as well as the order of any individual fragment in the sequence of constituent fragments. All of this information is provided in the set $\mathcal{NF}$, whose elements (fragment names, $n^f$) are tuples of these items ($t$, $n^c$, x, y). We also provide a set $\mathcal{PF}$ whose elements are pairs of a fragment name $n^f$ and an actual fragment $f$, in order to establish the link between the two.

$\mathcal{S}$ is the set of sites available in the system, while a keylet is represented as list of keylet operations consisting of three basic operations that will be elaborated on in the next section. A site possessing a specific key is able to execute the codelet corresponding to that key when the mobile agent arrives at the site; it is therefore the task of the keylet(s) to ensure that the right keys are made available to the right sites through the correct keylet operations. When a mobile agent migrates, the codelets that compose it are all distributed in an encrypted form. Upon arrival at a target site, some or all of these encrypted codelets will be decrypted depending on the keys available at that site. *merge* and *decrypt* are two functions executed by the sites in the system; *merge* performs the construction of a complete key from its constituent fragments, while *decrypt* simply implements the decryption of a codelet with a correct corresponding key.

$$\begin{array}{rcl}
f & \in & \mathcal{F} \\
k & \in & \mathcal{K};\ k \ = \ \langle\ f_0,\ f_1,\ \ldots,\ f_y \rangle \\
p^k & \in & \mathcal{PK};\ p^k \ ::= \ \langle\ k,\ t,\ n^c\ \rangle \\
n^f & \in & \mathcal{NF};\ n^f \ ::= \ \langle\ t,\ n^c,\ x,\ y\ \rangle \\
p^f & \in & \mathcal{PF};\ p^f \ ::= \ \langle\ n^f,\ f\ \rangle \\
s & \in & \mathcal{S} \\
t & \in & \mathcal{T} \\
ec & \in & \mathcal{EC} \\
n^c & \in & \mathcal{CN} \\
p^{ec} & \in & \mathcal{PEC};\ p^{ec} \ ::= \ \langle\ n^c,\ ec\ \rangle \\
vc & \in & \mathcal{VC} \\
p^{vc} & \in & \mathcal{PVC};\ p^{vc} \ ::= \ \langle\ n^c,\ vc\ \rangle \\
kt & \in & Keylet \\
\Theta & \in & SiteConfig \\
D & \in & Config \\
D & ::= & \{\ (s_0,\Theta_0),\ (s_1,\Theta_1),\ldots,(s_m,\Theta_m)\ \} \\
x,\ y & \in & \mathbb{N}
\end{array}$$

**Figure 2: State Space (Variables)**

## 3.2 Basic keylet and platform operations

With regards to Figure 3, keylet and platform operations can be modelled as a series of transitions that evolve the state of the abstract machine described in the previous section. We describe these operations intuitively below.

### 3.2.1 Keylet operations

#### 3.2.1.1 *migrate*($s_1$).

This essentially dispatches the keylet code from the current site to a target site $s_1$, which is an element in the set of sites, $\mathcal{S}$, in the system. The keylet resumes execution at $s_1$ with the next operation following the *migrate* operation. No copy of the keylet code will be retained on the current platform.

#### 3.2.1.2 *send*($s_1, n_1^f, n_2^f, \ldots n_q^f$).

*send* simply propagates a list of fragments (specified by $n_1^f, n_2^f, \ldots n_q^f$) available on the current platform to $s_1$ . Fragments are propagated with their names, $\mathcal{NF}$, which specify the exact codelet corresponding to the key of which a particular fragment is part of, as well as the order of that fragment in the key sequence and the number of fragments required to reconstituate the complete key. Fragment propagation is achieved by duplicating the fragment and dispatching the copy, with the current platform still retaining the original fragment. Keylet code may but need not accompany fragment propagation; if a keylet is to be dispatched, a separate *migrate* operation must be used. The *send* operation requires that the list of fragments to be propagated are present before the transition associated with it can fire. In this way, it subsumes the functionality of the *wait* operation (explained below).

#### 3.2.1.3 *wait*($n_1^f, n_2^f, \ldots n_q^f$) .

This operation does not fire a transition until the fragments identified by the fragment list $n_1^f, n_2^f, \ldots n_q^f$ become available at the current site as a result of *send* operations of keylets on other platforms. Although *wait* is implied in the functionality of *send* (as mentioned previously), there are situations where it may be required as a predecessor to *migrate*. For example, it is possible for a keylet to wait for a list of fragments which it does not intend to propagate; in such an instance, the blocking action serves to synchronize the reception of the key fragments with the migration of the keylet.

### 3.2.2 Platform operations

Platforms receive fragments propagated by keylets executing on other platforms. The two basic operations would be to recombine these key fragments to form a complete key and then decrypt the appropriate code component with this key.

#### 3.2.2.1 $JOIN(t, n^c)$.

Upon reception of a fragment originating from a *send* operation of a keylet executing on a different platform, a background process on the platform will need to ascertain, using the fragment name $n^f$, whether the newly arrived fragment completes any existing sequence of fragments already available at that platform. In the event of such an occurrence, a complete key can then be reconstituted from these fragments. This can be accomplished by simply concatenating the fragments in the correct sequence. $JOIN$ specifies explicitly the identity of the new key through the creation of a new key tuple, $p^k$, that contains the newly created key along with the $t$, $n^c$ pair (which identifies the codelet the key corresponds to). The constituent fragments will be retained at the platform following the construction of the new key; thus a keylet arriving at this platform at a later point of time will be able to propagate these fragments onwards in the manner described in the *send* operation.

#### 3.2.2.2 $DECRYPT(p^k)$.

Once a complete key is available, it can be used to decrypt the code component (codelet) it protects. $p^k$ provides the link between the key sequence and the corresponding codelet it protects. It is assumed that the necessary decryption algorithm that utilizes the key is already present on the platform prior to the commencement of key propagation. As is the case with the *send* operation of the keylet, this operation does not cause a transition to fire until the matching code component becomes available at the platform (i.e. the mobile agent containing that component migrates there). Upon successful decryption, the component is then executed immediately in a suitable environment on that platform.

## 3.3 Other considerations

The fundamental encryption unit that we associate with a keylet is a key fragment. However, there may be cases when partitioning of keys into fragments is not required, and keys will thus be propagated in their entirety. In such instances, we simply define a key as consisting of a single fragment and still maintain the (now redundant) distinction between key fragment names, $\mathcal{NF}$, and key tuples, $\mathcal{NF}$, for consistency. In a similar vein, if a keylet arriving at a platform wishes to propagate a complete key that was recently reconstituted

***Initial configuration***

$$\begin{aligned}
D_1(s_A) &= \Theta_A^1 ::= \langle\, p_i^{f*}, p_i^{k*}, p_i^{ec*}, p_i^{vc*}, vc_i^*, kt_i^* \,\rangle \\
D_1(s_x) &= \Theta_x^1 ::= \langle\, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing, \varnothing \rangle
\end{aligned}$$

where $s_A$ is the agent owner site and $s_x$ is any other site in the system

***State transitions (Keylets)***

$D_1 \quad \Rightarrow MIGRATE(s_1) \quad \Rightarrow D_2$

$$\begin{aligned}
D_1(s_0) &= \Theta_0^1 ::= \langle\, p_0^{f*}, p_0^{k*}, p_0^{ec*}, p_0^{vc*}, vc_0^*, \{[\mathsf{migrate}(s_1) : kt]\} \cup kt_0^* \,\rangle \\
D_1(s_1) &= \Theta_1^1 ::= \langle\, p_1^{f*}, p_1^{k*}, p_1^{ec*}, p_1^{vc*}, vc_1^*, kt_1^* \,\rangle \\
D_2(s_0) &= \Theta_0^2 ::= \langle\, p_0^{f*}, p_0^{k*}, p_0^{ec*}, p_0^{vc*}, vc_0^*, kt_0^* \,\rangle \\
D_2(s_1) &= \Theta_1^2 ::= \langle\, p_1^{f*}, p_1^{k*}, p_1^{ec*}, p_1^{vc*}, vc_1^*, \{kt\} \cup kt_1^* \,\rangle
\end{aligned}$$

$D_1 \quad \Rightarrow WAIT(n_1^f, n_2^f, \ldots n_q^f) \quad \Rightarrow D_2$

$$\begin{aligned}
D_1(s) &= \Theta^1 ::= \langle\, \{\langle n_1^f, f_1\rangle, \langle n_2^f, f_2\rangle, \ldots, \langle n_q^f, f_q\rangle\} \cup p^{f*}, p^{k*}, p^{ec*}, p^{vc*}, vc^*, \{[\mathsf{wait}(n_1^f, n_2^f, \ldots n_q^f) : kt]\} \cup kt^* \,\rangle \\
D_2(s) &= \Theta^2 ::= \langle\, \{\langle n_1^f, f_1\rangle, \langle n_2^f, f_2\rangle, \ldots, \langle n_q^f, f_q\rangle\} \cup p^{f*}, p^{k*}, p^{ec*}, p^{vc*}, vc^*, \{kt\} \cup kt^* \,\rangle
\end{aligned}$$

$D_1 \quad \Rightarrow SEND(s_1, n_1^f, n_2^f, \ldots n_q^f) \quad \Rightarrow D_2$

$$\begin{aligned}
D_1(s_0) &= \Theta_0^1 ::= \langle\, \{\langle n_1^f, f_1\rangle, \langle n_2^f, f_2\rangle, \ldots, \langle n_q^f, f_q\rangle\} \cup p_0^{f*}, p_0^{k*}, p_0^{ec*}, p_0^{vc*}, vc_0^*, \{[\mathsf{send}(s_1, n_1^f, n_2^f, \ldots n_q^f) : kt]\} \cup kt_0^* \,\rangle \\
D_1(s_1) &= \Theta_1^1 ::= \langle\, p_1^{f*}, p_1^{k*}, p_1^{ec*}, p_1^{vc*}, vc_1^*, kt_1^* \,\rangle \\
D_2(s_0) &= \Theta_0^2 ::= \langle\, \{\langle n_1^f, f_1\rangle, \langle n_2^f, f_2\rangle, \ldots, \langle n_q^f, f_q\rangle\} \cup p_0^{f*}, p_0^{k*}, p_0^{ec*}, p_0^{vc*}, vc_0^*, \{kt\} \cup kt_0^* \,\rangle \\
D_2(s_1) &= \Theta_1^2 ::= \langle\, \{\langle n_1^f, f_1\rangle, \langle n_2^f, f_2\rangle, \ldots, \langle n_q^f, f_q\rangle\} \cup p_1^{f*}, p_1^{k*}, p_1^{ec*}, p_1^{vc*}, vc_1^*, kt_1^* \,\rangle
\end{aligned}$$

***State transitions (Site daemons)***

$D_1 \quad \Rightarrow JOIN(t, n^c) \quad \Rightarrow D_2$

$$\begin{aligned}
D_1(s) &= \Theta^1 ::= \langle\, \{\langle n_1^f, f_1\rangle, \langle n_2^f, f_2\rangle, \ldots, \langle n_y^f, f_y\rangle\} \cup p^{f*}, p^{k*}, p^{ec*}, p^{vc*}, vc^*, kt^* \,\rangle \\
D_2(s) &= \Theta^2 ::= \langle\, \{\langle n_1^f, f_1\rangle, \langle n_2^f, f_2\rangle, \ldots, \langle n_y^f, f_y\rangle\} \cup p^{f*}, \{\langle k, t, n^c\rangle\} \cup p^{k*}, p^{ec*}, p^{vc*}, vc^*, kt^* \,\rangle
\end{aligned}$$

where $n_i^f = \langle t, n^c, x_i, y\rangle \qquad i = 1 \ldots y$ and $k = merge(f_1, f_2, \ldots, f_y)$

$D_1 \quad \Rightarrow DECRYPT(p^k) \quad \Rightarrow D_2$

$$\begin{aligned}
D_1(s) &= \Theta^1 ::= \langle p^{f*}, \{p^k\} \cup p^{k*}, \{\langle n^c, ec\rangle\} \cup p^{ec*}, p^{vc*}, vc^*, kt^* \,\rangle \\
D_2(s) &= \Theta^2 ::= \langle p^{f*}, \{p^k\} \cup p^{k*}, \{\langle n^c, ec\rangle\} \cup p^{ec*}, \{\langle n^c, vc\rangle\} \cup p^{vc*}, vc^*, kt^* \,\rangle
\end{aligned}$$

where $p^k = \langle k, t, n^c\rangle$ and $vc = decrypt(ec, k)$

**Figure 3: State Transitions**

from existing fragments on that platform, it would need to propagate all the constituent fragments of the key instead of the key itself. This would then be reconstituted again on the receiving platform.

A keylet may be used to direct propagation of fragments corresponding to a single key or different keys. We do not allow a key to be partitioned again (either by platform, keylet or mobile code) to produce fragments of different quantities and sizes. Each key may comprise of fragments of different quantities and sizes, however the number and size is predefined by the key generator (or agent owner) prior to the start of state transitions in the system. This simplifies the process of identifying fragments as the keylet migrates from one platform to another. A possible alternative would be to allow keylets to directly partition complete keys at platforms into new fragments of arbitrary number and length. Although this extends the flexibility of keylet operation, it also increases the complexity of identifying and recombining fragments properly within the system. Assuming the associated problems can be addressed properly, the fragment operation would be an ideal addition to the three basic operations of a keylet.

The *wait* , *send* and *decrypt* operations all fire a transition only on occurrence of certain events. There will therefore be a need to impose a time limit for this required events to occur in order to ensure keylets do not remain suspended indefinitely. This time limit could be predefined by the agent owner or agreed on with mutual cooperation from the sites hosting the keylet. We do not consider this issue in the example scenario following this section as it can be clearly seen that the required events for the operations will occur, but we note the need for such consideration when implementing a practical system.

In general, keylet code will be generated and signed by the agent owner (using the owner's private key), and is correspondingly authenticated by all receiving platforms before being executed. The keylet operations as presented here do not incorporate the idea of retaining state information beyond identifying the next operation to be executed upon migration to a new platform. Inclusion of state would permit more varied interactions with the platform environment as well as the mobile code that the keylet controls. Modeling state formally however would result in the development of a specification for a programming language, for which numerous calculi are already available. The design of such a programming language would also be subject to the same security constraints underlying the development of languages for mobile code in general. In particular, a programming language for keylets would need to ensure that resource consumption is predictable, which is an important requirement in language design for active networks as well [4]. Again, the discussion of this requirement is beyond the scope of this paper and we note here the need for any future language design to consider such issues.

## 4. KEYLET SCENARIO

We now detail an example scenario involving the use of keylets containing all of the operations described previously. To provide context for a practical implementation, we employ the typical scenario involving a mobile agent travelling to an itinerary of trade sites from an initial agent owner site in order to locate the best value for a given item (Figure 6). An additional site is also included, the arbitrator site, whose goal is to provide information to the mobile agent to aid it in selecting the site to purchase from. A mobile agent in such a scenario could be partitioned into four main components :

- Component A, which obtains the price for a particular item at the trade site the agent is currently on and may also decide on the next site to visit (based on information available at the current site)
- Component B, which decides on the site offering the best value for the item (taking into account other factors in addition to its price)
- Component C, which contains information which is required by a trade site in order to conclude a purchase agreement
- Component D is an optional component, which if included in the mobile agent, would contain sensitive information that is selectively revealed to specific sites.

The use of components here is intended to limit exposure of code functionality and state information to only parts of the code and state required by any particular site. An obvious case would be the need to protect the functionality and state of Component B from any of the trade sites, since knowledge of the criteria used to select a site would impart an unfair advantage to a trade site making an offer to sell an item. On the other hand, although we may have a higher level of trust in an arbitrator site compared to a trade site, we should not permit the arbitrator site to access the portion of the code typically executed by the trade site, since this portion has no relevance for the correct functioning of the arbitrator site. Thus, by careful partitioning of code into components, we can determine the component(s) that would need to be executed at any particular site and ensure that only these component(s) are visible at that site. This also has the important advantage of limiting the damage of potential compromise by a malicious site to only the code component visible to it.

A possible itinerary for the mobile agent would be as follows. The agent is dispatched from the agent owner site and executes Component A as it migrates through the various trade sites, then Component B as it goes on to the arbitrator site, and finally Component C, as it returns to a particular trade site that it intends to purchase from. Once this route is complete, the agent then migrates back to the owner site, where suitable verification and processing of information accumulated by the agent during its tour can be undertaken. In addition, it might also be possible for the arbitrator site to obtain access to Component D, depending on the trade site it selects while executing component B. A trade site might decide to attach certain sensitive information about itself in component D of the mobile agent, which it will reveal to the arbitrator site only in the event of it being selected by the arbitrator as the “best-value” site. An example of such information could be discounts or other sales-related benefits that it wishes to offer to mobile agents from a specific user. This information can then be retained for use by the arbitrator when selecting a trade site again. In such a situation, the agent itinerary will include an additional stop at the arbitrator site again before returning to the owner site.

Corresponding to such an itinerary, the trade sites should be given access only to Component A. Component B should be accessible to the arbitrator, and Component C accessible only to the single trade site from where the purchase will be made at. Component D may be made accessible to the arbitrator site according to the particular trade site that executes Component C (i.e the site from where the purchase will be initiated). In line with the concepts developed earlier, we now need to associate a symmetric key with each component and ensure that these keys are made available to the sites that need to execute their corresponding components. The agent owner site (or a trusted third party) will now generate four keys, K1, K2, K3 and K4, which will be used to encrypt Components A to D respectively. K4 is technically a public key pair; the public portion used to encrypt Component D is distributed to the trade site interested in depositing information in that component, while the private portion to be used for decryption is retained on the agent owner site and propagated to the arbitrator site when the occasion arises. K3 will be initially partitioned into two fragments, which we shall label as K31 and K32. K1, K2 and the public / private portion of K4 do not require fragmenting in our example, and can be considered as keys that are constituted from a single fragment (i.e. the fragment is the key itself). This provides consistency for the construction of keylets, which we have already defined as performing fragment rather than key propagation.

This itinerary could also have been achieved by dispatching the components separately from the agent owner site to the respective sites that are permitted to have access to them (i.e. in effect making each component a smaller mobile agent). However this introduces an overhead due to the need for communication between the different agents in order to preserve the sequence in which the different components are executed on the respective sites. Furthermore, there may be situations where communication channels between the agent owner site and some other trade sites become temporarily unavailable (for example, in bandwidth limited applications involving mobile devices). Thus instead of waiting for channel availability in order to dispatch the required components to these sites, it would be more efficient to dispatch the agent consisting of all its constituent components to a site for which a channel is available. The agent could then migrate on its own violition through the network as and when channels becomes available without the need for further involvement from the agent owner.

Having laid the foundation for our scenario, we now proceed to examine the functionality we can expect in the mobile agent and the keylet, before stepping through the sequence of key, keylet and agent migrations that would be involved in a single tour of the agent.

## 4.1 Mobile agent components

A sample of the functionality contained in the four main components of a mobile agent as well as the state information retained in the agent as it migrates between platforms is shown in pseudo-code form in Figure 4.

*Component A*

1. Obtain price from host platform
2. Append to price list
3. Obtain information about other platforms
4. If (discount information available) Append discount information to Component D
5. Migrate to next site in itinerary

*Component B*

1. Read items from price list
2. Obtain additional information on discounts, taxes, etc from host platform.
3. Determine the best value item and the site offering it
4. Send request to agent owner site to spawn keylet Q
5. Migrate to best site

*Component C*

1. Calculate final price for item offered
2. Add price to state information

*Component D*

1. Information appended from any one of the trade sites

*State information*

1. Site itinerary : Trade Site 1, 2 and 3, Arbitrator site, Agent Owner site.
2. Price List

**Figure 4: Codelets**

At the start, we assume that the mobile agent has a fixed site itinerary determined by its owner prior to migrating from the owner's site. Components A to C of the agent will be encrypted with their respective keys prior to migration, and will remain in encrypted form for the duration of the agent's traversal in the network. Any site wishing to execute a particular component would thus require the necessary key in order to do so successfully. We also assume that sites that wish to deposit information in Component D are already in possession of the public portion of K4.

## 4.2 Keylet specification

There are two keylets (Figure 5) in our scenario. The main keylet, which we shall call P, is launched from the agent owner's site and will commence execution after the mobile agent is launched. Keylet P contains the necessary key propagation operations to ensure that all the sites listed in the predefined tour itinerary for the mobile agent obtain the correct keys. Keylet Q is created and launched by the agent owner as a response to a specific request from a trade site that has deposited information in Component 4 and wishes to reveal this information, pending certain conditions, to the arbitrator site. The Send operation in keylet Q works on the private portion of K4 that is available on the agent owner site (the public portion is available with the trade site depositing information into Component D).

*Keylet P*

1. Send (trade site 1, K1)
2. Send (trade site 1, K31)
3. Send (arbitrator site, K31)
4. Send (arbitrator site, K32)
5. Migrate (trade site 1)
6. Send (trade site 2, K1)
7. Send (trade site 2, K31)
8. Migrate (trade site 2)
9. Send (trade site 3, K1)
10. Send (trade site 3, K31)
11. Migrate(arbitrator site)
12. Send (selected site, K32)

*Keylet Q*

1. Migrate (trade site 2)
2. Wait (K32)
3. Migrate (agent owner site)
4. Send (arbitrator site, K4)

**Figure 5: Keylets**

Working on the assumption of fail safe communication between sites (i.e. any message or key propagated between sites will arrive within a predefined time limit), we shall now commence to trace the path of the keylets and the mobile agent as they migrate through the network to accomplish their respective functions. Italic alphabetic numbering is provided in brackets throughout the explanation as references to Figure 6, where the movements of the mobile agent, keylet and keys / key fragments are illustrated.

## 4.3 Scenario explanation

At the start, keylet P propagates K1 and K31 to trade site 1 and K2 and K32 to the arbitrator site ($a$, $b$). The keylet then dispatches itself to trade site 1 ($c$). During this time, the mobile agent also migrates to trade site 1 ($d$), where component A is decrypted with K1 and executed. The requirement for a key to be present at a trade site in order to execute a corresponding component, provides a mechanism to synchronize the migration of the agent and the propagation of keys.

Keylet P, now resident on trade site 1, proceeds to propagate K1 and K31 forward to the next site in the itinerary($e$), before migrating there itself ($f$). Once at trade site 2, it propagates K1 and K31 onwards to the trade site 3 ($h$) and then dispatches itself to the arbitrator site ($i$), where it blocks on the Send operation (which requires the value of a selected site). During this period of time, the mobile agent migrates through its itinerary of trade sites ($g$, $j$) before arriving at the arbitrator site as well ($k$). Let us assume that while the agent is executing component A on trade site 2, the trade site decides it wishes to deposit information

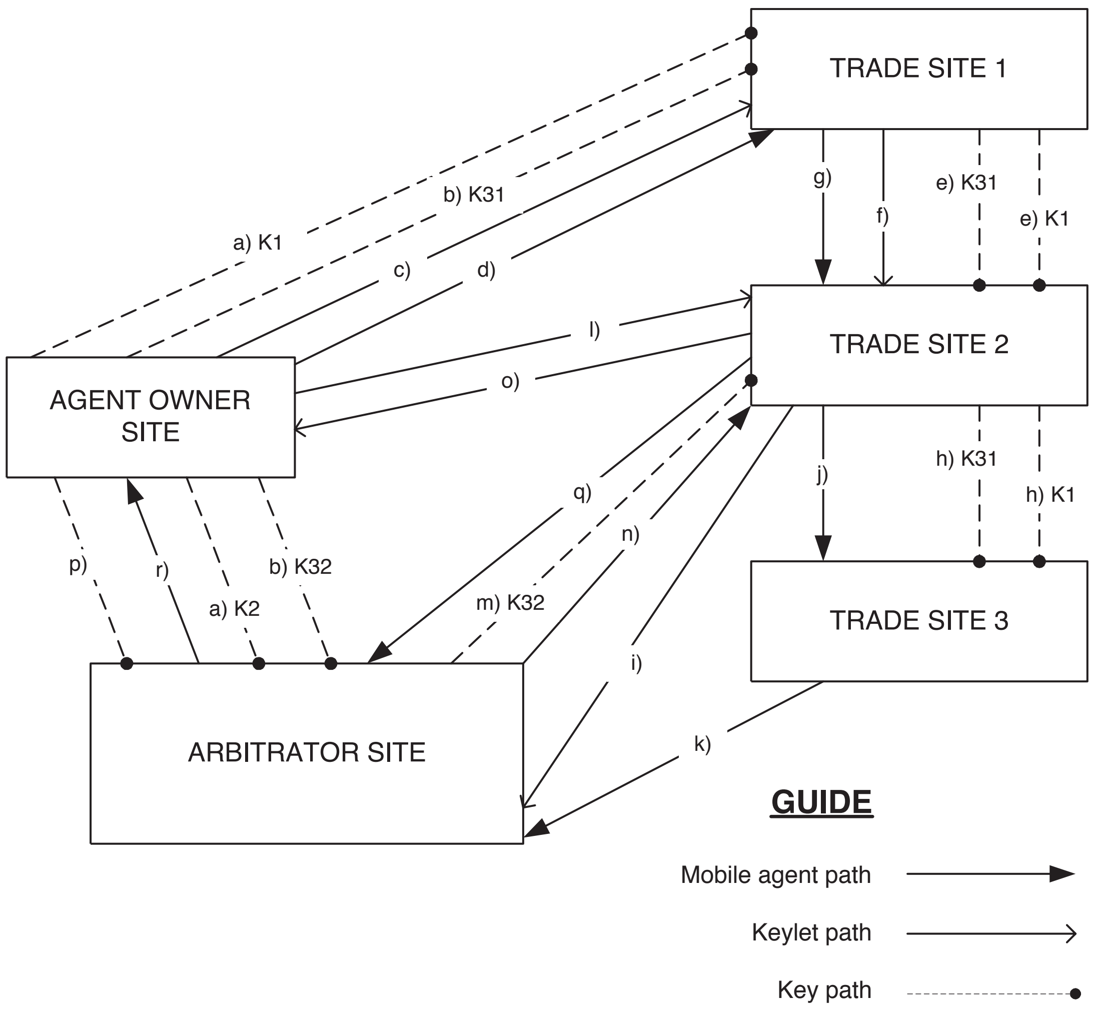

**Figure 6: Keylet Scenario**

in component D of the agent. It proceeds to accomplish this by using the public portion of K4 just before the agent dispatches itself to trade site 3. Trade site 2 also sends a request to the agent owner to permit visibility of this information to the arbitrator site in the event that it is selected as the "best value" site by the arbitrator site. This results in the creation of keylet Q by the agent owner site, whose first operation is to dispatch itself to trade site 2 ($l$), where it then blocks on the Wait for K32.

Upon execution of component B on the arbitrator site, it happens (by chance) that trade site 2 is selected as the "best value" site. Keylet P, which was blocking on the Send operation, now continues by propagating K32 to trade site 2 ($m$) before terminating . The mobile agent proceeds by migrating to trade site 2 as well ($n$). On successful reception of K32, trade site 2 can now merge it with its other key fragment (K31) to form a complete key to decode and then execute component C of the mobile agent to conclude the purchase. Keylet Q, which is also on this site and blocking on the Wait for K32, can now proceed by migrating back to the agent owner site ($o$) and propagating the private portion of K4 to the arbitrator site ($p$). Once component C is complete, trade site 2 specifically dispatches the agent back to the arbitrator side ($q$), who is now able to access component D with the use of K4. The tour of the agent finally terminates with a return to the agent owner site ($r$).

The fact that K3 is fragmented into K31 and K32 from the start ensures that none of the trade sites is able to gain access to component C from the start. In addition, although the arbitrator site may be able to decide which trade site is finally able to execute component C (by having the keylet to propagate K32 to that trade site), it is itself unable to access component C as well. In this way, we preserve the security property of exposing components only to platforms that need to execute them, regardless of the amount of trust we may accord them.

The risk of possible collusion between the arbitrator site and a trade site could be minimized by duplicating the functionality of the arbitrator sites and allowing them to be administered independently. The key for Component C could be fragmented and distributed to all these sites, in effect making it necessary for a 'unanimous' decision to be reached by all sites before a trade site could gain access to C. This is similar in spirit to the idea of co-operating agents, which was first proposed by Roth [11] as another alternative to safeguarding mobile code integrity.

## 5. DISCUSSION AND RELATED WORK

In the scenario described, the distribution of keys could have also been achieved equally as well by having the agent owner initially appending them to the codelets of the mobile agent (thus eliminating the need for keylets). However this would impose an additional requirement on the sites receiving the codelets to be actively involved in the further propagation of keys; an activity which necessitates a communication overhead with the agent owner site. This overhead is likely to be more expensive than the generation and deployment of the keylet and may also result in a communications bottleneck developing at the agent owner site in the event that a large number of trade sites are involved. Another possible alternative could involve embedding key propagation functionality into the mobile agent itself. We eschew this approach in favour of the use of keylets because it is important to maintain a distinction between general functionality (original agent code) and security functionality (key propagation); the agent code designer need not (and should not) concern himself about the latter.

In addition, it may be useful to permit the mobile agent to dynamically alter its route while traversing the network. In the scenario, the agent was supplied with a predefined itinerary of trade sites and the keylets were created to ensure that these trade sites obtained the required keys to decrypt the necessary components. To provide flexibility in the application of the mobile agent, we could permit the agent to deviate and visit an additional site at some point in its itinerary before resuming its predefined route. To accomodate this route diversion properly, the correct key fragments must be propagated to the additional site to be visited. Since key propagation functionality is limited only to keylets in our system, we can achieve this in two ways : allow the mobile agent to spawn a new keylet which then propagates the necessary keys to the additional site, or establish a communication channel between the mobile agent and the exisiting keylet in order to convey the agent's intention. Both options give rise to new security considerations. For the first possibility, there is now the issue of the authenticity of the keylets to be executed. In the original scenario, all keylets were generated only by the agent owner and hence a signature with the agent owner's private key would suffice to gurantee keylet code integrity. If keylet code can be generated via a mobile agent, it can no longer be signed as it is not possible for the mobile agent to be carrying the owner's private key for obvious security reasons. The inability to authenticate keylet code before executing it could result, for example, in a hostile agent generating a bogus keylet that could migrate to a new site and propagate fragments from there to other sites that would not by right be able to access them. For the second possiblity, a keylet has to ascertain correctly the identity of a mobile agent making a request as well as ensure that the request is authentic (i.e. not originating from a man in the middle attack from a malicious site). In both instances, no immediate solution is obvious. It may be possible to tackle some of these security considerations for either one or both of these possiblities; this remains to be investigated in future work and is beyond the scope for consideration in this paper.

There are two main advantages that are achieved through the use of keylets and code components which we explain here.

### 5.1 Selective revelation of functionality and/or data

Code components encrypted with different keys allow code to be revealed only to entities that posses the correct keys. By carefully partitioning code into well defined components, we can minimize the amount of information a hostile party could extrapolate about other components from any single component (assuming it does not have keys to the other components). In addition, it becomes easier to construct an audit trail leading back to a site making a malicious making a code alteration. For example, if at some point during the agent's tour, it is discovered (using checking mechanisms such as those detailed in [15] and [6]) that a particular code component has been modified in a subtle way, the search for the culprit could be immediately narrowed down to the sites in possession of the key corresponding to that particular component. This can easily be ascertained by examining the keylets that control the propagation of that key. In our example scenario, if the agent owner were to detect a modification in Component C of the agent upon its return to the original site, it would be reasonably safe to assume the perpetrator was trade site 2.

### 5.2 Two level approach to expressing the functionality of the mobile agent

By making key propagation the sole duty of the keylet, we make a distinction between the functionality of key propagation and and the general functionality of a mobile agent. In order to understand how the mobile agent functions in the system, it is no longer sufficient to observe its code and site itinerary (the first level). For example, even if we knew the composition of all the components of the mobile agent as well as its predefined site itinerary, we would not be able to make an accurate prediction on its behaviour without knowledge of the component(s) to be executed at each site. This knowledge would only be available by studying the keylet (the second level). Since the generation of the keylet and the exact partitioning of code into components is known only to the agent owner, even the producer of the mobile agent template employed by the user would find it difficult to predict the behaviour, and hence launch an attack on an agent in the system.

As mentioned earlier, keylets and code components are not a new code security technique per se, rather they provide a framework in which existing techniques can be used more efficiently as part of a trust model. As such, we did not consider the possibility of checking to ensure the execution environment on a site does not attempt to subvert the correct flow of execution of a code component. There have been methods suggested for performing such checks, these include techniques such as state appraisal [5], execution tracing [15] and reference states [6]. Any of these methods could be employed in the scenario towards this end, and we note that the use of small, self-contained components would most likely increase the effectiveness of these methods that were originally proposed for checking large pieces of code with complex functionality. Also, if incorrect execution of code is detected, there is greater flexibility in dealing with the perpetrator. The agent owner could decide to stop propagating any keys to the site in question, or may propagate certain keys while withholding others. This would represent the change in the level of trust the owner has in that site.

### 5.3 Related work

The idea of keylets was first introduced in [14]; this paper extends on it by providing a simple formalisation for possible keylet operations. Our work is very much motivated by the idea of trust management [2], where authorizations to perform specific tasks are associated with keys which are propagated in a system. [12] presents a convincing argument that a proper trust model is necessary in order to build security into mobile agent system. There is a lot of active research in the area of protecting mobile agents from hostile platforms (a recent example includes [6]); our work is distinguished from the rest by including a specific mechanism (propagation of keys) through which we can express the flow of trust in a system and construct an appropriate trust model to describe such flows. The use of keylets also represents a potentially new area of application for mobile code. Keylets could, for example, be used to distribute keys/certificates between large certificate repositories in a public key infrastructure system [1] in order to permit easier processing, storage and retrieval of these certificates.

## 6. CONCLUSION

We have presented the concept of keylets, mobile code that direct the distribution of keys in a system. We show how the use of keylets combined with encryption of partitioned code components permit a new approach to mobile agent code security. A formalisation for keylets is presented and a scenario involving the use of them to implement this new approach to mobile agent code security is developed and described in detail.

As an extension to our formalization in this paper, static analysis could be performed on a keylet specification in order to determine whether certain properties can be maintained. This could eventually lead towards the possibility of automatic generation of a keylet that fulfills certain criteria. The security of a keylet is another area that deserves further attention. It is likely that the mechanisms used to address this would differ from conventional security mechanisms for mobile code as the functionality of the keylet is limited and more well defined in comparison with generic mobile code. The issue of incorporating some form of trust management based on the propagation of keys will also be studied further as this represents an important avenue through which we can model the flow of trust in a mobile agent system.

## 7. ACKNOWLEDGMENTS

We are grateful to the anonymous referees for their constructive and detailed comments. This research is funded in part by EPSRC and DERA project "Magnitude" reference GR/N35816.

## 8. REFERENCES

[1] C. Adams and S. Lyold. *Understanding Public Key Infrastructure : Concepts, Standards and Deployment Considerations*. Macmillan Technical Publishing, 1999.

[2] M. Blaze *et al.* The Role of Trust Management in Distributed Systems Security. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS. Springer-Verlag, 1999.

[3] D. M. Chess. Security Issues in Mobile Code Systems. In *Mobile Agents and Security*, number 1419 in LNCS. Springer-Verlag, 1998.

[4] D. L. Tennenhouse *et al.* A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1), January 1997.

[5] W. Farmer *et al.* Security for Mobile Agents: Authentication and State Appraisal. In *Proceedings of the 4th European Symposium on Research in Computer Security (ESORICS '96)*, September 1996.

[6] F. Hohl. A Framework to Protect Mobile Agents by Using Reference States . In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, 2000.

[7] W. Jansen. Countermeasures for Mobile Agent Security. In *Computer Communications, Special Issue on Advances in Research and Application of Network Security*, November 2000.

[8] P. Mell and M. McLarnon. Mobile Agent Attack Resistant Distributed Hierarchical Intrusion Detection Systems. In *Second International Workshop on Recent Advances in Intrusion Detection*, September 1999.

[9] R. Oppliger. Security issues related to mobile code and agent-based systems. *Computer Communications*, 22(12), July 1999.

[10] P. V. Rangan. An axiomatic basis of trust in distributed systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1988.

[11] V. Roth. Mutual protection of co-operating agents. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS. Springer-Verlag, 1999.

[12] K. Schelderup and J. Olnes. Mobile Agent Security - Issues and Directions. In *Intelligence in Services and Networks. Proceedings of the 6th International Conference on Intelligence and Services in Networks*, Barcelona, Spain, April 1999.

[13] V. Swarup and Javier Thayer Fabrega. Trust : Benefits, Models and Mechanisms. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS. Springer-Verlag, 1999.

[14] H. K. Tan. Keylets and Mobile Agent Security. In *Symposium on Software Mobility and Adaptive Behaviour, AISB 2001*, March 2001.

[15] G. Vigna. Protecting Mobile Agents Through Tracing. In *Proceedings of the 3rd ECOOP Workshop on Mobile Object Systems*, Jyvlskyl, Finland, June 1997.