Data Privacy in Tuple Space Based Mobile Agent Systems

Lorenzo Bettini

Dipartimento di Sistemi e Informatica, Università di Firenze
Via Lombroso 6/17, 50134 Firenze, Italy
bettini@dsi.unifi.it

Abstract

More recently, distributed variants of tuple spaces have been proposed to exploit the Linda model for programming distributed applications over wide area networks, possibly exploiting code mobility. However, the flexibility of the shared tuple space model opens possible security holes; it basically provides no access protection to the shared data. In this paper we investigate some possible scenarios where mobile agents can benefit from our cryptographic tuple space based framework, CryptoKLava, and sketch how to possibly implement such agents in order to keep the privacy of items collected by the mobile agent during its itinerary. The functionalities of the framework are general enough to be applied to other Java frameworks using multiple distributed tuples spaces possibly dealing with code mobility.

Keywords: Mobile agents, wide area networks, Linda, Java, mode mobility

1 Introduction

The Internet has been cursed by intrusions and attacks since the early days [23,30,22]. Such intrusions used to exploit bugs in existing software in order to gain access to the computer, replicate themselves, and spread to other computers. Thus worms and viruses can rely on the concept of mobile code. Indeed, the high flexibility of mobile agents, and mobile code in general, do not come at no cost. Downloading code from the network for local execution

1 This work has been partially supported by EU within the FET – Global Computing initiative project MIKADO IST-2001-32222. The funding bodies are not responsible for any use that might be made of the results presented here.

1571-0661 © 2005 Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2004.11.038
exposes to possible threats at a higher level with respect to an isolated context [26,8].

Distributed and mobile code systems raise new security issues mainly because they “violate a number of assumptions that underlie most existing computer security measures” [11]. Assumptions that can be safely accepted for isolated computers are destined to fall when the executing scenario scales to an open network. Interesting features of mobile agents, such as, e.g., autonomy, also have drawbacks, since the owner of a system may ignore that remote code is executing on his machine.

A successful approach to concurrent programming is the one relying on the Linda coordination model [17]. Processes communicate by reading and writing tuples in a shared memory called tuple space. Control of accesses is guaranteed by requiring that tuples selection be associative, by means of pattern matching. The communication model is asynchronous, anonymous, and generative, i.e., tuple’s life-time is independent of producer’s life time.

The Linda model has been adopted in many communication frameworks such as, e.g., JavaSpaces [2] and TSpaces [16], and for adding the tuple space communication model to existing programming languages. More recently, distributed variants of tuple spaces have been proposed to exploit the Linda model for programming distributed applications over wide area networks [12,4], possibly exploiting code mobility [13,27]. As shown in [15], where several messaging models for mobile agents are examined, the blackboard approach, of which the tuple space model is a variant, is one of the most favorable and flexible.

Sharing data over a wide area network such as the Internet, calls for very strong security mechanisms. Computers and data are exposed to eavesdropping and manipulations. Dealing with these issues is even more important in the context of code mobility, where code or agents can be moved over the different sites of a net. Malicious agents could seriously damage hosts and compromise their integrity, and may tamper and brainwash other agents. On the other hand, malicious hosts may extract sensible data from agents, change their execution or modify their text [36,28].

The flexibility of the shared tuple space model opens possible security holes; it basically provides no access protection to the shared data. Indeed there is no way to determine the issuer of an operation to the tuple space and there is no way to protect data: a process may (even not intentionally) retrieve/erase data that do not belong to it and shared data can be easily modified and corrupted. In spite of this, within the Linda based approaches, very little attention has been devoted to protection and access control.

In [6] we presented CRYPTOKLAVA, a Java middleware for building dis-
tributed and mobile code applications interacting through tuple spaces, by means of cryptography. In this middleware, classical Linda operations are extended for handling encrypted data. Primitives are also supplied for encrypting and decrypting tuple contents. The proposed extension, while targeted to our middleware for mobile agents interacting through distributed tuple spaces, KLAVA [7], is still general enough to be applied to other Java frameworks using multiple distributed tuples spaces possibly dealing with code mobility, such, e.g., [27,2,12]. Indeed, this extension represents a compromise between the flexibility and open nature of Linda and of mobile code, and the privacy of data in a distributed context.

The finer granularity provided by CRYPTOKLAVA allows mobile agents (that are not supposed to carry private keys with them when migrating) to collect data encrypted with public keys, while executing on remote sites, and decrypt them safely with private keys when back at the home site. In this paper we investigate some possible scenarios where mobile agents can benefit from our cryptographic tuple space based framework and sketch how to possibly implement such agents in order to keep the privacy of items collected by the mobile agent during its itinerary. All the software presented here is freely available at http://music.dsi.unifi.it. The paper is organized as follows: in Section 2 we describe the design issues of our framework and in Section 3 we briefly recall CRYPTOKLAVA’s main features. Section 4 presents the mobile agent scenarios implemented with CRYPTOKLAVA and Section 5 compares our approach to some related works.

2 Distributed Private Generative Communications

The Linda communication model [17] is based on the notion of tuple space that is a multiset of tuples. These are just sequences of items, called fields that are of two kinds: actual fields, i.e., values and identifiers, and formal fields, i.e., variables. Syntactically, a formal field is denoted with !ide, where ide is an identifier. Tuples can be inserted in a tuple space with the operation out and retrieved from a tuple space with the operations in and read (read does not withdraw the tuple from the tuple space). If no matching tuple is found, both in and read block the process that execute them, until a matching tuple becomes available. Pattern-matching is used to select tuples from the tuple space; two tuples match if they have the same number of fields and corresponding fields do match: a formal field matches any value of the same type, and two actual fields match only if they are identical (but two formals never match). For instance, if Val is an integer variable, then tuples (“foo”, “bar”, !Val) and (“foo”, “bar”, 300) do match. After matching, the
variable of a formal field gets the value of the matched field; in the previous example, after matching, Val will contain the integer value 300.

The middleware CRYPTOKLAVA we presented in [6] is based on KLAVA [7], a Java framework implementing KLAIM (Kernel Language for Agent Interaction and Mobility) [13] that provides features for programming distributed applications with mobile code and mobile agents, relying on communication via multiple distributed tuple spaces. KLAIM extends Linda by handling multiple distributed tuple spaces: tuple spaces are placed on nodes (or sites), which are part of a net. Each node contains a tuple space and a set of processes, and can be accessed through its locality. Thus, classical Linda operations are indexed with the locality of the node they have to be performed at. A reserved locality, self, can be used to access the current execution site. Moreover in KLAIM processes are first class data, in that they can be transmitted and exchanged among sites, so that mobile code and mobile agent applications can be easily programmed.

For guaranteeing privacy of data stored in tuple spaces we have extended KLAVA with some cryptographic primitives. In our view, this extension is a good tradeoff between the open nature of Linda (and of mobile code) and data privacy. In particular we aim at having this extension as smooth as possible, so that the original model is not perverted.

The basic idea is that a tuple may contain both clear text fields and encrypted fields. All the encrypted fields of a specific tuple are encrypted with a single key. This choice simplifies the overall design and does not harm usability of the system; it would be unusual that different fields of the same tuple are encrypted with different keys. Encrypted fields completely hide the encrypted contents that they embody: they even hide the type of the contents. This strengthens the secrecy of data (it is not even possible to know the type of sensible information).

In line with the open nature of the Linda model, our main intention is not to prohibit processes to retrieve data belonging to other processes, but to guarantee that these data be read and modified only by entitled processes. A shared tuple space is basically a shared communication channel: in such a channel information can be freely read and modified.

At the same time one of our aims is avoiding that wrong data be retrieved by mistake. Clear text fields of a tuple can be used as identifiers for filtering tuples (as in the Linda philosophy), but if a matching tuple contains encrypted fields, which a process is not able to decrypt, it is also sensible that the tuple is put back in the tuple space if it was withdrawn with an in. Moreover, in such cases, a process may want to try to retrieve another matching tuple, possibly until the right one is retrieved (i.e., a tuple for which it has the appropriate
decryption key), and to be blocked until one is available, in case no such tuple is found.

Within our framework it is possible to

- use tuple fields with encrypted data;
- encrypt tuple fields with specific keys;
- decrypt a tuple with encrypted fields;
- use variants of the operations in and read (ink and readk) to atomically retrieve a tuple and decrypt its contents.

The modified versions of the retrieving operations, ink and readk, are based on the following procedure:

(i) look for and possibly retrieve a matching tuple,
(ii) attempt a decryption of the encrypted fields of the retrieved tuple,
(iii) if the decryption fails:
   (a) if the operation was an ink then put the retrieved tuple back in the tuple space,
   (b) look for alternative matching tuples,
(iv) if all these attempts fail, then block until another matching tuple is available.

Thus the programmer is relieved from the burden of executing all these internal tasks, and when a readk or an ink operation succeeds it is guaranteed that the retrieved tuple has been correctly decrypted. Basically the original Linda pattern matching mechanism is not modified: encrypted fields are seen as ordinary fields that have type KCipher (as shown in Section 3). It can be seen as an extended pattern matching mechanism that, after the structural matching, also attempts to decrypt encrypted fields.

In case mobile code is used, the above approach may be unsafe. Indeed, symmetric and asymmetric key encryption techniques rely on the secrecy of the key (in asymmetric encryption the private key must be kept secret). Thus, a fundamental requirement is that mobile code and mobile agents must not carry private keys when migrating to a remote site ("Software agents have no hopes of keeping cryptographic keys secret in a realistic, efficient setting" [36]). This implies that the above introduced operations ink and readk cannot be used by a mobile agent executing on a remote site, because they would require carrying over a key for decryption.

For mobile agents it is then necessary to supply a finer grain retrieval mechanism. For this reason we introduced also operations for the explicit decryption of tuples: a tuple, containing encrypted fields, will be retrieved by
a mobile agent by means of standard \texttt{in} and \texttt{read} operations and no automatic decryption will be attempted. The actual decryption of the retrieved tuples can take place when the agent is executing at the home site, where the key for decryption is available and can be safely used. Typically a mobile agent system consists of stationary agents, that do not migrate, and mobile agents that visit other sites in the network, and, upon arrival at the home site, can communicate with the stationary agents.

Thus the basic idea is that mobile agents collect encrypted data at remote sites and communicate these data to the stationary agents, which can safely decrypt their contents. Obviously, if some data are retrieved by mistake, it is up to the agents to put it back on the site from where they were withdrawn. This restriction of the protocol for fetching tuples is necessary if one wants to avoid running the risk of leaking private keys. On the contrary, public keys can be safely transported and communicated. By using public keys mobile agents are able to encrypt the data collected along their itinerary.

Notice that there is no guarantee that a “wrong” tuple is put back: our framework addresses privacy, not security, i.e., even if data can be stolen, still it cannot be read. Should this be not acceptable, one should resort to a secure channel-based communication model, and give up the Linda shared tuple space model. Indeed the functionalities of our framework are similar to the one provided, e.g., by \textit{PGP} [37] that does not avoid e-mails be eavesdropped and stolen, but their contents are still private since they are unreadable for those that do not own the right decryption key.

An alternative approach could be that of physically removing an encrypted tuple, retrieved with an \texttt{in}, only when the home site of the agent that performed the \texttt{in}, notifies that the decryption has taken place successfully. Such a tuple would be restored if the decryption is acknowledged to have failed or after a specific timeout expired. However, this approach makes a tuple’s life time dependent on that of a mobile agent, which, by its own nature, is independent and autonomous: agents would be expected to accomplish their task within a specific amount of time. Moreover, inconsistencies could arise in case successful decryption acknowledgments arrive after the timeout has expired.

3 The middleware CryptoKlava

\textsc{Klava} [7] is deployed as an extensible Java package, \texttt{Klava}, that defines the classes and the run-time system for developing distributed and mobile code applications according to the programming model of \textsc{Klaim}. In \textsc{Klava} processes are instances of subclasses of class \texttt{KlavaProcess} and can use methods for accessing a tuple space of a node: \texttt{out(t,1)}, for inserting the tuple $t$
into the tuple space of the node at locality $l$, read$(t,l)$ and in$(t,l)$, for,
respectively, reading and withdrawing a tuple matching with $t$ from the tuple
space of the node at locality $l$. Moreover the method eval$(P,l)$ can be used
for spawning a KlavaProcess $P$ for remote execution on site $l$. Some wrapper
classes are supplied for tuple fields such as KString, KInteger, etc.

The extension of this package, CRYPTOKLAVA, provides the cryptography
features described in the previous section. We have used the Java Cryptog-
raphy Extension (JCE) [31], a set of packages that provide a framework and
implementations for encryption, key generation and key agreement, and Mes-
sage Authentication Code (MAC) algorithms. JCE defines a set of standard
API, so that different cryptography algorithms can be plugged into a system
or an application, without modifying the existing code. Keys and certificates
can be safely stored in a Keystore, an encrypted archive.

CRYPTOKLAVA is implemented as a subpackage of the package Klava,
namely Klava.crypto, so that it is self-contained and does not affect the
main package. In the rest of this section we will briefly describe the main
classes of the package Klava.crypto, implementing cryptographic features,
that are relevant to our context. We refer the interested reader to [5] for a
more detailed description.

The class KCipher is introduced in order to handle formal and actual fields
containing encrypted data. It can be seen as a wrapper for standard KLAVA
tuple fields. An actual encrypted tuple field can be created by firstly creating
a standard KLAVA tuple field (in the example a string) and then by passing
such field to an instance of class KCipher:

```java
KString s = new KString("foo");
KCipher ks = new KCipher(s);
```

Similarly the following code creates an encrypted string formal tuple field (In
KLAVA a formal field is created by instantiating an object from a KLAVA
class for tuple fields – such as KString, KInteger, etc. – through the default
constructor):

```java
KString s = new KString();
KCipher ks = new KCipher(s);
```

The class Tuplex extends the standard KLAVA class Tuple, in order to
contain fields of class KCipher, besides standard tuple fields; apart from pro-
viding methods for cryptographic primitives, it also serves as a first filter
during matching: it will avoid that ordinary tuples (containing only clear text
data) be matched with encrypted tuples. Once tuple fields are inserted into a
Tuplex object, the KCipher fields can be encrypted by means of the method
encode. For instance, the following code

```java
KString s = new KString("foo");
KCipher ks = new KCipher(s);
Tuplex t = new Tuplex(s, ks);
```
KString ps = new KString("clear");
KCipher ks = new KCipher(new KString("secret"));
Tuple t = new Tuple();
t.add(ps); t.add(ks);
t.encode();

creates a tuple where the first field is a clear text string, and the second is a field to be encrypted, and then actually encrypts the KCipher field by calling encode. Also encode can receive parameters specifying the key and the algorithm for the encryption; otherwise the default values are used. The method ensures that all encrypted fields within a tuple rely on the same key and algorithm.

As for the retrieval operation, this can be performed either with the new introduced operations, ink and readk, if they are executed on the local site

KString s = new KString();
KString sec = new KString();
KCipher ks = new KCipher(sec);
Tuple t = new Tuple();
t.add(s); t.add(ks);
ink(t, l);
Print("encrypted data is: " + sec);

or by first retrieving the tuple and then manually decoding encrypted fields:

... // as above
in(t, l);
...
t.decode();
Print("encrypted data is: " + sec);

Notice that in both cases references contained in an encrypted field (such as sec) are automatically updated during the decryption. The ink in the former example is performed at a remote site but this does not mean that the key travels in the net: as explained in the previous section, the matching mechanism is implicitly split into a retrieve phase (which takes place remotely) and a decryption phase (which takes place locally).

Finally, let us observe that, thanks to abstractions provided by the JCE, all the introduced operations are independent of the specific cryptography mechanism, so both symmetric and asymmetric encryption schemes can be employed.

In the next section we will present a programming example of use of these new cryptographic primitives; further examples can be found in [5].
4 A mobile agent exploiting encryption

This example relies on the well known scenario of a migrating agent visiting some sites and collecting information on behalf of the owner. During its itinerary, the agent is exposed to attacks from the sites themselves or from possible eavesdropping processes on other sites. The information collected by the agent could be read and possibly modified by these intruders. Since this information could be sensible data, it is important that it is not accessible by no one but the owner of the agent and the agent itself.

For this reason the agent will encrypt, with the public key of its owner, the data collected during its itinerary, so that, even if eavesdropped, these cannot be read by intruders. The agent can safely travel with the public key, and the collected data, once the agent has come back home, can be decrypted by the owner by means of his private key. Unfortunately this does not come at no cost: it is a well-known problem (see, e.g., [36]) that the agent is not able to act according to the information collected during its itinerary since it cannot decrypt data (it does not hold the private key):

```java
KString s1 = new KString();
KString fString1 = new KString(); // retrieve clear text data...
in(s1, fString1, self, 1000);
Tuplex txf1 = new Tuplex();
txf1.add(s1);
txf1.add(new KCipher(fString1));
txf1.encode(my_public_key); // ... encrypt it
collectedData.add(txf1);

if (! done ) {
    // ... migrate to the next site
} else {
    out(collectedData, owner);
}
```

Once the owner receives these data he can try to decrypt them, once they are safely stored in its local tuple space:

```java
// decrypt the collected data stored in the
// the local tuple space
...
while ( true ) {
    Tuplex txf2 = new Tuplex();
    KString s1 = new KString();
    txf2.add( s1 ) ;
    KString fString1 = new KString();
    KCipher k = new KCipher( fString1 ) ;
    txf2.add( k ) ;
    if (ink_nb( txf2, self, 100 )) {
        out( "decoded", s1, fString1, self );
    } else {
```
In the previous code, \texttt{ink\_nb} is the non-blocking version of \texttt{ink} (the process is not blocked if no matching tuple is available).

Let us now consider a slightly different scenario: the sites visited by mobile agents want to be sure that information destined to specific entities cannot be read by others. Even in this case asymmetric encryption helps in solving this problem: the site encrypts data for a person \textit{A} with the public key of \textit{A}. This way, even if a mobile agent is able to retrieve data that does not belong to its owner, these data will be useless since they cannot be decrypted. The example we show here implements this scenario: the mobile agent retrieves data according to a specific identifier (represented here by the string \texttt{s1}). The data related to this identifier are encrypted. Once the itinerary of the agent is over, the agent sends all the collected data to its owner.

```java
Tuplex txf1 = new Tuplex();
KString s1 = new KString("item1");
txf1.add(s1);
KString fString1 = new KString();
KCipher k = new KCipher(fString1);
txf1.add(k);
read(txf1, self);
collectedData.add(txf1);

if (!done) {
    // ... migrate to the next site
} else {
    out(collectedData, owner);
}
```

Notice that the previous agent, instead of reading data through the tuple

\(("item1", !k)\)

could also read with the tuple

\((!s, !k)\)

This way, it would be able to retrieve also data that is not pertinent with
"item1" (possibly data that do not belong to its owner). However the agent owner will not be able to decrypt tuples that do not belong to him, since he does not own the keys for decrypting them.

Moreover the type of encrypted data, for the sake of security and privacy, is hidden, so the agent could also retrieve remotely encrypted tuples of the form \(("item1", !k)\) where k does not contain a string. In this case, at the
owner site, such tuples would not be retrieved, since, during an `ink` or `readk`, once an encrypted field is decoded, its type is used for the actual matching.

5 Conclusions and Related Work

Since tuple space operations can be used both by local processes and by mobile agents, the extended operations address both the privacy of hosts and of mobile agents. We did not deal with key distribution explicitly that can be seen as an orthogonal problem. Digital signatures can be smoothly integrated in our framework and the pattern matching extended accordingly. We are currently working on the addition of the presented cryptography primitives to the kernel language KLAIM and to the programming language based on KLAVA, X-KLAIM [5].

The work that is closer to ours is [9], which introduces the Secure Object Space (SecOS) model. This model is intended to extend Linda with fine-grained access control semantics. In SecOS all tuple fields are locked with a key, and each field must be locked with a different key. The basic idea is that a process, upon retrieving a tuple, can see only the fields for which he owns the corresponding key. The structure of a tuple does not influence pattern matching: due to an introduced subsumption rule, a template can match also a bigger tuple, and fields can be reordered during the matching. [10] proposes a similar, but richer framework, SecSpaces, where also resource access control and tuple space partitioning facilities are provided (orthogonal and complementary to our approach).

All these features tend to alter the original Linda model and indeed the SecOS calculus is based on the asynchronous $\pi$-calculus [1,21], exploiting its channel-based communication mechanism. Our principal aim, instead, is to provide an extension of the Linda communication model that can be smoothly integrated into the existing features, without significantly changing the original model. As for communications among sites we rely on the KLAIM model, which can be seen an implementation of distributed tuple spaces.

Moreover, neither SecOS nor SecSpaces handle code mobility, which is one of our main concerns. Mobility imposes additional restrictions on the underlying model, e.g., requiring that agents do not carry private keys during migrations, and calls for alternatives such as explicit encryption and decryption mechanisms and a two-stage pattern matching.

The problem of cryptography in a mobile scenario is investigated in [29]. There, techniques for achieving “non-interactive evaluation with encrypted functions” are presented to avoid carrying clear text keys. With encrypted functions, an agent is able to securely perform a cryptographic primitive,
even in an untrusted execution environment. The approach makes use of homomorphic encryption schemes and function composition techniques. At the moment, however, no general scheme for achieving this has been found, and thus no effective implementation of these techniques is available.

An alternative approach to security is the one based on resource access control. This is completely orthogonal to our approach and can be seen as complementary to it. Here, we would like to mention access control mechanisms based on granting or denying access to sensible resources according to specific policies (e.g., based on the source of the request, digital signatures, etc.). Type system based solutions are already available for access control in a distributed and mobile scenario [14,19]. Java [3] relies on security policies (possibly exploiting digital signatures) for implementing access control, especially for downloaded code [18]. Proof-Carrying Code [24] imposes mobile code to also provide a proof that an host can easily verify in order to check whether the code adheres to the local security policy. However, the difficulty in specifying and efficiently encoding a proof makes the approach not implementable in practice, at the moment.

Basically the above mentioned solutions deal with the issue of protecting a host from mobile agents, and not with the problem of protecting a mobile agent from a malicious host and from other agents: When a mobile agent is executing on a remote host, it is almost completely defenseless, and indeed the problem of protecting an agent against a malicious host is even more complicated than the previous issues [20,25,28,32,33,34,35,36]. However, due to the inherent nature of the problem, the provided solutions either limit the flexibility of mobile agents or allow only an a posteriori analysis of the execution of agents.

Acknowledgments.

I would like to thank Rocco De Nicola, the co-author of CRYPTOKLAVA, for the helpful discussions on the mobile agent scenarios.

References


