Encrypted Shared Data Spaces

Giovanni Russello[1], Changyu Dong[1], Naranker Dulay[1],
Michel Chaudron[2], and Maarten van Steen[3]

[1] Imperial College London
[2] Eindhoven University of Technology
[3] Vrije Universiteit Amsterdam

Abstract. The deployment of Share Data Spaces in open, possibly hostile, environments arises the need of protecting the confidentiality of the data space content. Existing approaches focus on access control mechanisms that protect the data space from untrusted agents. The basic assumption is that the hosts (and their administrators) where the data space is deployed have to be trusted. Encryption schemes can be used to protect the data space content from malicious hosts. However, these schemes do not allow searching on encrypted data. In this paper we present a novel encryption scheme that allows tuple matching on completely encrypted tuples. Since the data space does not need to decrypt tuples to perform the search, tuple confidentiality can be guaranteed even when the data space is deployed on malicious hosts (or an adversary gains access to the host). Our scheme does not require authorised agents to share keys for inserting and retrieving tuples. Each authorised agent can encrypt, decrypt, and search encrypted tuples without having to know other agents' keys. This is beneficial inasmuch as it simplifies the task of key management. An implementation of an encrypted data space based on this scheme is described and some preliminary performance results are given.

1 Introduction

Coordination through Shared Data Spaces (SDS) also called generative communication, forms an attractive model for developing distributed and component-oriented systems as it supports referential and temporal decoupling of processes [9]. Referential decoupling means that components exchange data without the need to know each other. Temporal decoupling means that those components do not even have to be online at the same time. This way, components can be connected to or disconnected from the data space at any time, making it easier to combine or replace. The SDS model was introduced by the coordination language Linda [8]. Storage in Linda takes place in a so-called *tuple space*. In a tuple space, data is stored as persistent objects, called *tuples*.

The early implementations of SDSes were *closed* systems, in the sense that they were realised by compiling application and SDS code altogether. Once the system was deployed and executed, it was not possible to add or remove application components. In such systems, security was not a issue and the original Linda model was conceived without addressing security concerns.

D. Lea and G. Zavattaro (Eds.): COORDINATION 2008, LNCS 5052, pp. 264–279, 2008.

In contrast, *open* systems were introduced where the SDS is not bound to an application but is an autonomous process with its own resources. The main advantage of open SDS systems is that persistent data storage can be offered to applications. In this way, applications could dynamically join and leave the computational environment. This is clearly a feature suited to distributed applications. With its small API and the decoupling of communication in space and time, the SDS model provides an effective coordination layer for distributed applications.

Distributed applications are deployed in environments that range from small ad-hoc networks of portable devices (such as Body Area Networks) up to wide area networks (such as the Internet). In such scenarios applications are faced with many security threats that the original Linda model does not address. For instance, denial of service attacks could be performed by malicious agents inserting a large number of tuples into the data space. Still, a malicious agent can remove any tuples from the space interfering with the other agents that are using the space. This can be even more serious when the tuples contain sensitive information.

Such security deficiencies pose a limitation on the usability of the SDS model for real-world applications. Early work presented by Wood in [20] discusses the introduction of access control mechanisms, such as ACLs and capabilities, that could be used for controlling access to the SDS content. Several other approaches have proposed access control mechanisms that employ secret information associated with SDSes and their content. In [12], an agent must know the password associated with the space in order to get access to it. In [11,19] the secret information is represented by locks that are associated with tuple instances. To get access to a tuple, an agent must provide the specific lock associated with the tuple.

Although access control mechanisms are necessary for allowing authorised operations on the data space, they are not always adequate to protect data confidentiality. The common assumption of these approaches is that the host where the data space is deployed is managed by a trustworthy entity that (1) correctly enforces access control mechanisms and (2) is *oblivious* of the data that is stored in the data space. However, such an assumption does not always hold when sharing data over a wide area network such the Internet.

A solution to enforce the confidentiality of tuples against malicious hosts is to encrypt the tuple content as proposed in [2]. However, because the ciphertexts are not meaningful, it is not possible to perform search operations. A trusted data space can temporarily decrypt the data, perform the search and return the results to the agent. Alternatively, if the data space doesn't have access to the decryption keys, the encrypted data can be returned to the agent that decrypts the data locally. The first solution cannot protect data confidentiality from malicious hosts, while the second one is potentially very inefficient in communications. Moreover, issues related to key management (i.e., key distribution, key revocation, etc.) are not addressed.

In this paper we propose an efficient approach for guaranteing that tuple confidentiality is protected against malicious hosts. We developed a novel encryption scheme that allows the execution of search operations on encrypted tuples,

without having the data space decrypt the data. Another important property of our encryption scheme is that it does not require agents to share secret keys. Each authorised agent can encrypt, decrypt, and search encrypted tuples without having to know other agents' keys. This greatly simplifies the task of key management. In particular, it avoids re-encrypting the tuple content when a key needs to be revoked. To the best of our knowledge, this is the first approach that proposes such features for the shared data space model. Finally, we integrate the encryption scheme in a SDS implementation and carry out some preliminary performance analysis.

The paper is organized as follows. Section 2 introduces the original SDS model. Section 3 surveys recent developments aiming at providing security in the SDS model. In Section 4, we describe on our encryption scheme. In Section 5, we discuss and evaluate our encrypted SDS implementation. In this paper our main focus to guarantee data confidentiality in case a host is compromised. However, the SDS and its content can face other security threats when its host is compromised. In Section 6, we discuss some of those security threats. We conclude in Section 7 with some final thoughts and future research directions.

2 The Shared Data Space Model

The shared data space model was introduced by the coordination language Linda [8]. Linda provides three basic operations: out, in and rd. The out operation inserts a tuple into the tuple space. The in and rd operations respectively take (destructive) and read (non-destructive) a tuple from the tuple space, using a template for matching. The tuple returned must exactly match every value of the template. Templates may contain wildcards, which match any value. Whereas putting a tuple inside the tuple space is non-blocking (i.e. the process that puts the tuple returns immediately from the call to out), reading and taking from the tuple space is blocking: the call returns only when a matching tuple is found. In the original model two more operations were introduced: the inp and rdp. These operations are predicate versions of in and rd: they too try to return a matching tuple. However, if there is no such tuple they do not block but return a value indicating failure.

In Linda it is also possible to fork a process inside a tuple space through so-called *live tuples*. To insert a live tuple inside a tuple space the eval operation is used. eval is similar to an out and it is specific for live tuples. Once a live tuple is inserted in a tuple space it carries out the specified computation. Afterwards, a live tuple turns into an ordinary data tuple, and it can be used as such. In the implementation of a SDS presented later on in this paper the inp, rdp, and eval operations are not supported.

3 Related Work

This section provides a critical overview of exiting approaches providing security for shared data space.

Secure Lime, described in [12], introduces several security extension to Lime [14]. Since Lime's primary environment is a network of mobile low-resource hosts, the main concern of the developers was to introduce security enhancements with low overhead of the original Lime's model. Security extensions are implemented as two levels of access control: at tuple space level and single tuple level. At the tuple space level, it is possible to protect access to a tuple space by means of a password. An agent will be considered authorized to access a tuple space if it knows the password for the given tuple space. At the tuple level, agents can specify for each tuple that they insert passwords for granting both read and take accesses. Inter-host communication uses unsecured links. For avoiding eavesdropping of messages, each serialized tuple is encrypted using the respective password for accessing the tuple space. It should be noted that it is not a good practice to use a password as an encryption key.

SecOS [19] introduces the notion of *lock* for controlling access to a tuple. A lock is a labeled value that specifies the key that should be used to grant access to a given tuple. The simplest lock is represented by a symmetric key where the same label can be used for locking and unlocking a tuple. Also, asymmetric locks can be used. In this case, two different keys are necessary for locking and unlocking a tuple. A public key is used for locking a tuple and a private one is used for unlocking it. SecOS also provides finer grained access control at the level of single fields in a tuple. Each field in a tuple can be protected by a separate lock.

SecSpaces [11] provides a similar approach to that of SecOS. In SecSpaces labels are used as an access control mechanism to protect tuples and tuple fields. SecSpace provides two more extensions. The first extension concerns partitioning the tuple space. The partitioning of a tuple space avoids all agents having the same view on the data contained in a tuple space. Instead of a physical separation in different tuple spaces, in SecSpaces the tuple space partitioning is achieved through the introduction of a partition field in the tuples. A template can match a tuple in a given partition only if the correct actual value is given in the partition field. A template with a wildcard value in the partition field is considered not valid. This means that a process has to know the name of the partition for accessing the content. The second extension regards the distinction between consumers that can only execute read operations and consumers that can only execute take operations. This extension is provided via specified fields in the tuples, called *control fields*. To be an authorized read consumer, the process has to provide in the template issued by the read operation the exact value on the read control field of a tuple.

Linda with multicapabilities [18] is an approach where the capability concept is applied to the Linda model. Capabilities are the means by which agents can access to tuples and SDS. In particular, a multicapability is a special capability that refers to a group of tuples. A multicapability consists of three parts: u, a unique identifier which is the reference to a collection of tuples; t, a template that matches the tuples that the multicapability refers to; p, a set of permitted operations on the matching tuples. To be able to exchange tuples, tow or

more agents have to share the same multicapability that refers to the same set of tuples. In case a multicapability has to be revoked, the authors adopt the common solution of introducing *indirect multicapability objects*. A multicapability now refers to the indirection object, which in turn refers to the intended tuple set. The deletion of the indirection object has the effect of removing the multicapability.

In all the approaches presented above, tuples are stored in the data space as plaintext. Indeed, then basic assumption of these approaches is that the data space host is trusted. However, if an adversary gets access to the host where the data space is deployed, tuples could still be retrieved and accessed. The only exception to this is KLAIM [2]. KLAIM provides privacy by means of encryption. In the framework proposed, a key can be used for encrypting the data value contained in a field. The model does not provide any access restrictions to the tuple space. This means that encrypted tuples can be retrieved by agents that do not have the right key for decrypting the content. If a tuple is withdrawn from the tuple space by an agent that cannot access it, it is up to that agent to reintroduce the tuple back to the space. The tuple space API is extended with two operations that execute the decryption process before returning the tuple to the application: ink and readk. If the decryption fails, then the ink operation inserts the tuple back into the space. It should be made clear that the key used for encrypting the data is not shared between the entities and the data space. The ink and readk operations perform the decryption locally to the node where the entity is deployed. This has a negative impact on the communication utilisation.

Although KLAIM is the only approach that encrypts the data when it is stored in the space, it does not support encrypted search. Therefore it is necessary to have in the tuples cleartext fields. Assuming that there is a secure channel between the agent and the data space, an attacker can still gain some information on the matched tuple if it has access to the data space host. However, if the data space supports encrypted search then an attacker can not gather any information about the tuple content by just looking at the ciphertext. Another common drawback of the above approaches is that agents are required to share a secret (either a key or a password). The revocation of the secret in the event that it gets compromised requires the re-distribution of a new secret and the creation and/or modification of the data space to be protected by the new secret. The same needs to be done in case that access privileges have to be removed to an agent.

To protect the confidentiality of shared data spaces from both unauthorised clients and from the SDS host(s) we introduce a novel encryption scheme that supports encrypted searches over encrypted shared data spaces. Furthermore, the scheme does not require agents to share secret keys.

4 Multi-agent Searchable Encryption Scheme

This section presents our encryption scheme for a multi-agent searchable encrypted data space. The aim of this section is to describe the required

cryptographic details of the scheme and its properties. For a more detailed description refer to [4].

4.1 Cryptographic Preliminaries

Our multi-user searchable encryption scheme employs *RSA public-key encryption* [15] and *Discrete Logarithms*. RSA involves two asymmetric keys. The key pair is generated as follows: First choose two random large prime p and q such that $|p| \approx |q|$. Then compute $n = pq$ and $\phi(n) = (p-1)(q-1)$. Find a random integer $e < \phi(n)$ and $gcd(e, \phi(n)) = 1$. Compute d such that $ed \equiv 1 \ mod \ \phi(n)$. (n, e) is the public key and d is the private key. To encrypt, compute $c = m^e \ mod \ n$. To decrypt, compute $m = c^d \ mod \ n$. In the rest of the paper, we assume all arithmetic to be $mod \ n$ unless stated otherwise. Discrete Logarithms in finite fields are one-way functions. Namely, given a prime p, a generator g of the multiplicative group Z_p^* and $g^x \bmod p$, it is hard to find x. Discrete Logarithms have been used in constructing public-key encryption schemes [5], digital signature schemes and zero-knowledge proof protocols.

Both RSA and Discrete Logarithms use *Modular exponentiation* as basic operations and the exponents can be split multiplicatively. In RSA, for example we can find e_1, e_2 such that $e_1 e_2 \equiv e \ mod \ \phi(n)$. The two shares of e can be given to two parties, then the two parties can collaboratively encrypt a message. Given a message m, one party encrypts it as $m^{e_1} \bmod n$ and the other party re-encrypts it as $(m^{e_1})^{e_2} \equiv m^{e_1 e_2} \equiv m^e \bmod n$. The decryption key can also be split in the same way.

This idea is used in *proxy cryptography* and was first introduced in [3]. In a proxy encryption scheme, a ciphertext encrypted by one key can be transformed by a proxy function into the corresponding ciphertext for another key without revealing any information about the keys and the plaintext. There are many applications of proxy encryption, e.g. secure email lists [17], access control systems [18] and attribute based publishing of data [19]. A comprehensive study on proxy cryptography can be found in [13].

The encryption schema that we use in our system combines the property of proxy cryptography where each authorised agent has a unique key with the capability of performing tuple matching on encrypted data.

4.2 Architecture

The system has the following components:

- Client: a client is any agent interacting with the data space.
- Encrypted Shared Data Space(eSDS): is used for storing and retrieving tuples, performing encrypted searching operations, authenticating valid clients, and safely storing encryption and decryption keys. The eSDS is also capable of storing and retrieving tuples in plaintext or partially encrypted. The basic assumption is that we trust the eSDS to perform these operations correctly. Although conceptually we refer to the eSDS as a single component, it could be physically distributed across several hosts.

– Key Management Server (KMS): The KMS is a fully trusted server which is responsible for all the key-related operations, e.g. key generation, distribution, and revocation. Although requiring a trusted KMS seems at odds with using a less trusted node where the data space is running, we will show that the KMS is lightweight, it requires less resources and management. Securing the KMS is also much easier. Because of this, the KMS can be offline most of the time.

4.3 System Setup

To initialise the encryption system, the KMS runs the setup algorithm to generate public and secret parameters which will be used for the whole lifetime of the system. The algorithm is described as follows:

The algorithm first takes a security parameter k and runs the key generation algorithm using standard RSA which generates $(p, q, n, \phi(n), e, d)$. It then generates $\{p', q', g, x, h, a, g^a h^a\}$ satisfying the following constraints: p' and q' are two large prime numbers such that q' divides $p' - 1$; g is a generator of $G_{q'}$, the unique order-q' subgroup of $Z^*_{p'}$; and $h \equiv g^x \ mod \ p'$ where x is chosen uniformly randomly from $Z_{q'}$. a is also a random number from $Z_{q'}$.

The parameters needed for encryption/decryption are $n, p', q', g, h, g^a h^a$ and need to be published system-wide. The key material is represented by the parameters $p, q, \phi(n), e, d, x, a$ and must be kept secretly. In particular, the (e, d, a) are called "*Master Keys*" for the system.

4.4 Client Key Generation and Revocation

When a new client is enrolled into the system, the KMS must generate a unique key set for the client. The key set is derived from the key material using the following algorithm:

For a client i, the KMS generates $e_{i1}, e_{i2}, d_{i1}, d_{i2}, a_{i1}, a_{i2}$ such that $e_{i1} e_{i2} \equiv e \ mod \ \phi(n)$, $d_{i1} d_{i2} \equiv d \ mod \ \phi(n)$ and $a_{i1} a_{i2} \equiv a \ mod \ q'$. Key generation can be efficiently done in the following way. Let us consider the generation of the e_{i1}, e_{i2} pair. The KMS randomly chooses $e_{i1} < \phi(n)$, where $gcd(e_{i1}, \phi(n)) = 1$. Since $e_{i1} x \equiv 1 \ mod \ \phi(n)$ has always a solution, then $e_{i2} \equiv ex \ mod \ \phi(n)$ always satisfies $e_{i1} e_{i2} \equiv e \ mod \ \phi(n)$. The KMS then sends (e_{i1}, d_{i1}, a_{i1}) to client i and (e_{i2}, d_{i2}, a_{i2}) to the eSDS through secure channels.

In our system it is possible to authenticate a client and establish a secure channel between the client and the eSDS using the corresponding key pairs. Because $e_{i1} d_{i1} e_{i2} d_{i2} \equiv ed \equiv 1 \bmod \phi(n)$, $k_1 = e_{i1} d_{i1}$ and $k_2 = e_{i2} d_{i2}$ form another RSA key pair. This key pair can be used for public key mutual authentication and for establishing a secure channel, e.g. SSL.

When a client's access privilege is revoked, the KMS sends an instruction to the eSDS to request the removal of the client's corresponding keys. After the keys have been removed, the client cannot access the data unless the KMS generates new keys for it.

4.5 Tuple Encryption

In our system, tuple encryption is performed in two steps. A tuple is first encrypted by the client using its own private key. The encrypted tuple is then sent to the eSDS, where the tuple is re-encrypted using the node's key that correspond to that client. Client side encryption prevents the eSDS (and its hosting site) from knowing the data in the tuple whereas the eSDS side encryption makes it possible for other authorised clients in the system to retrieve the tuple in clear text. The encryption process for client i is shown in Fig. 1. For a tuple $t = \langle d_1; ...; d_n \rangle$, we denote the value of a field at position x by d_x.

On the client side, a tuple is first encrypted using a semantically secure symmetric encryption algorithm E [10]. For each tuple, client i randomly picks a key K from the key space of E. Each value of the tuple's fields d_x is encrypted under the key K which generates a ciphertext $c_{x1} = E_K(d_x)$. The symmetric key K is then encrypted by algorithm $CEnc$ which is identical to the RSA-OAEP (Optimal Asymmetric Encryption Padding) encryption algorithm [1] and uses

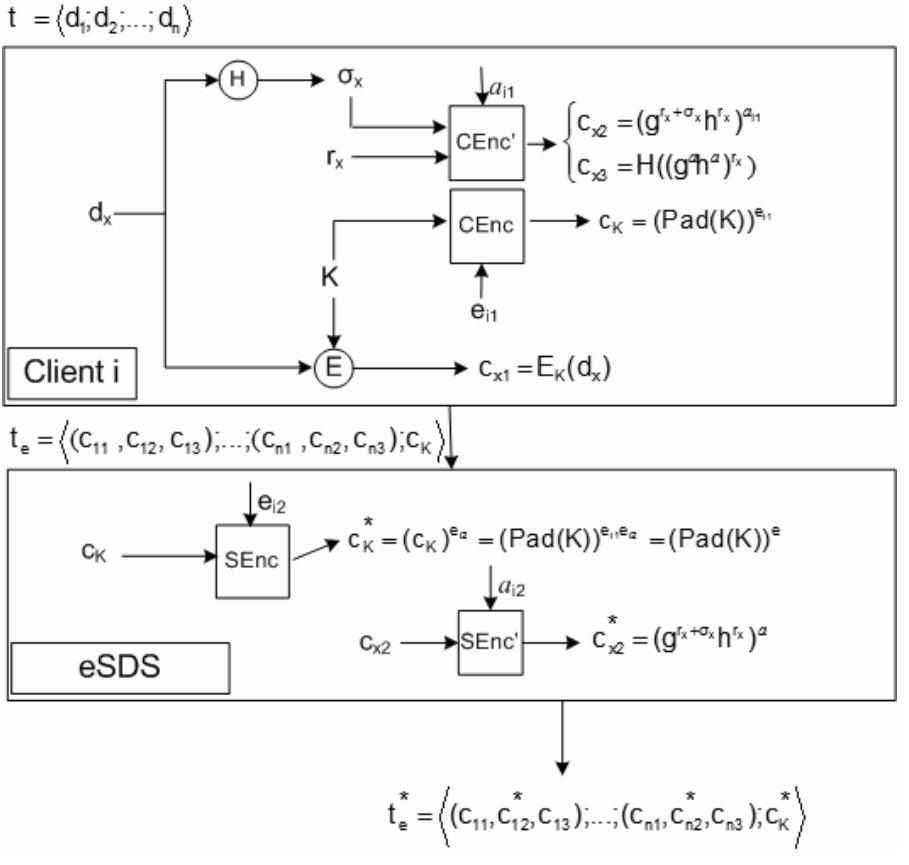

Fig. 1. Encryption of a tuple on client i and data space

e_{i1} as the encryption key. RSA-OAEP enhances RSA by using a probabilistic padding scheme and has been proved to be IND-CCA2 (Indistinguishable Adaptive Chosen Ciphertext Attack) secure [7]. The ciphertexts of the symmetric keys is $c_K = (Pad(K))^{e_{i1}}$.

During the search for a matching tuple, the data space content is kept encrypted. The matching is done using an opportunely modified values of the tuple field, called *keywords*. Keywords are computed as follows by the client using the algorithm $CEnc'$ and sent together with the tuple to the eSDS. For each value d_x of a tuple field , the client i computes $\sigma_x = H(d_x)$ using a hash function H. The client also picks a random number $r_x \in Z_{q'}$ and computes $c_{x2} = (g^{r_x+\sigma_x} h^{r_x})^{a_{i1}} \ mod\ p', c_{x3} = H((g^a h^a)^{r_x})$, where $g, h, g^a h^a, p'$ are public parameters in the system and a_{i1} is the client's keyword encryption key. The client then sends the encrypted tuple $t_e = \langle (c_{11}, c_{12}, c_{13}); ...; (c_{n1}, c_{n2}, c_{n3}); c_K \rangle$ to the eSDS.

After receiving the encrypted tuple, the eSDS retrieves e_{i2} and a_{i2}, the corresponding encryption keys for the client i. It re-encrypts the symmetric key by computing $c_K^* = c_K^{e_{i2}}$ using the $SEnc$ algorithm. The eSDS processes the keywords information that is contained in the tuple using the $SEnc'$ algorithm. For each filed x, the eSDS computes $c_{x2}^* = c_{x2}^{a_{i2}} = (g^{r_x+\sigma_x} h^{r_x})^{a_{i1} a_{i2}} = (g^{r_x+\sigma_x} h^{r_x})^a \ mod\ p'$. The final encrypted tuple stored is $t_e^* = \langle (c_{11}, c_{12}^*, c_{13}); ...; (c_{n1}, c_{n2}^*, c_{n3}); c_K^*) \rangle$.

4.6 Encrypted Search

The searching of a tuple in the data space is done by means of a template. A template may contains wildcard fields, that in our system are represented as null values. When a client j wants to retrieve a tuple matching the template $temp = \langle z_1, ..., z_n \rangle$, j first computes the hash value of all actual fields in the template. Since a wildcard field matches any actual values in a tuple, it is not necessary that our encrypted search algorithm processes wildcard fields of a template. For each non-null field x the client j generates $\sigma_x^* = H(z_x)$. Then j encrypts σ_x^* as $Q_x = g^{-\sigma_x^* a_{j1}}$. At this point, the encrypted template is $temp_e = \langle Q_1; ...; Q_n \rangle$. j sends $temp_e$ to the eSDS.

The eSDS computes for each field of the received template $Q'_x = Q_x^{a_{j2}} \ mod\ p' = g^{-\sigma_x^* a} \ mod\ p'$. During the search, for each encrypted tuple, the data space computes the following two values for each x-th non-null field in the template:

$$y_{x1} = c_{x2}^* Q'_x = (g^{r_x+\sigma_x} h^{r_x})^a g^{-\sigma_x^* a} = (g^{a r_x + a\sigma_x} h^{a r_x}) g^{-a\sigma_x^*} \ mod\ p'$$
$$y_{x2} = H(y_1)$$

We can see that if $d_x = z_x$ then $a\sigma_x - a\sigma_x^* = 0$, and therefore $y_{x1} = (g^{a r_x} h^{a r_x}) = (g^a h^a)^{r_x} \ mod\ p'$. From this follows that the value in the x-th field of the template matches the value of the x-th filed in a tuple if and only if $y_{x2} = c_{x3}$ (because $y_{x2} = H((g^a h^a)^{r_x}) = c_{x3}$).

4.7 Tuple Decryption

When a matching tuple is found, the eSDS computes the following before sending the tuple to the client j. For each field x in the matching tuple $t_e^* = \langle (c_{11}, c_{12}^*, c_{13}) ; ...; (c_{n1}, c_{n2}^*, c_{n3}); c_K^* \rangle$ the eSDS computes $c'_K = (c_K^*)^{d_{j2}}$ and sends to j the following tuple $t'_e = \langle c_{11}; ...; c_{n1}; c'_K \rangle$. The client j retrieves the key for encrypting the data items by computing $(c'_K)^{d_{j1}} = (c_K^*)^d = (K)^{ed} = K$. The client j can decrypt the value of each field by computing $d_x = E_K^{-1}(c_{x1})$.

5 Implementation and Performance

In this section, we discuss the implementation and performance of the eSDS based on the encryption scheme. The prototype is an extension of our implementation of a distributed SDS, called GSpace [16].

Figure 2 provides an overview of the modules that are part of our architecture. Clients and the eSDS are different processes that reside in different hosts. A client C_i communicates with the eSDS by means of a proxy, called **eSDSProxy**. The eSDSProxy takes care of hiding from the client all the details for the communication with the eSDS and deals with the cryptographic operations. To connect to an eSDS, a client creates a new **eSDSProxy** as follows:

```
eSDSProxy p = new eSDSProxy ("SpaceName");
```

The argument is used by the proxy to load in its **KeyStore** (KS) the appropriate key pair for tuple encryption and decryption and for establishing a secure connection with the eSDS. The proxy performs tuple encryption and decryption using the **Proxy Encryption Module (PEM)**.

Tuples and templates are subclasses of the `Tuple` class. A tuple can be defined in such a way that when it is stored in the eSDS it can contain both cleartext and encrypted fields. A field in a tuple will be stored encrypted only when its type is one of the following: `eInt`, `eChar`, `eDouble`, and `eString`. These are classes that we define to represent the encrypted form of the corresponding Java classes. Therefore, if a tuple is defined as follows:

```
MyTuple(eString name, eInt age, Integer weight)
```

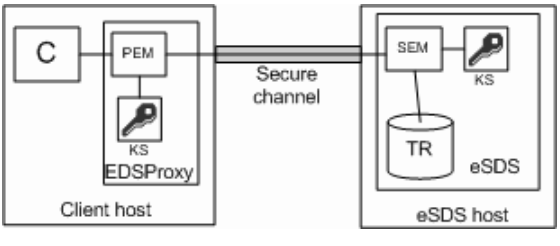

Fig. 2. Overview of the architecture of the our eSDS prototype

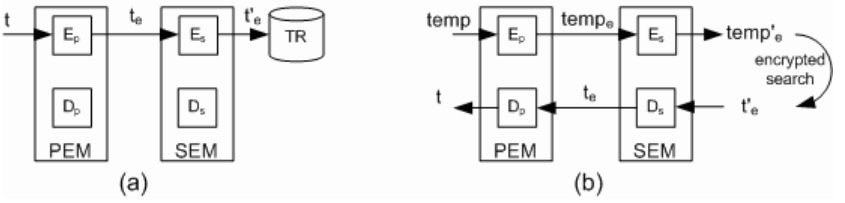

Fig. 3. Encryption steps executed for storing and retrieving a tuple using our scheme

when such a tuple is stored in the eSDS, only the first two fields will be encrypted. Field `weight` will be stored in cleartext.

A client (by means of its proxy) establishes a secure communication with the eSDS. The eSDS authenticates the client and the corresponding key is loaded into the KS of the eSDS. The eSDS performs tuple encryption, decryption and encrypted search by means of its **Space Encryption Module (SEM)**. Tuples are stored in the **Tuple Repository (TR)**.

In the implementation, a `put` operation is used to insert a tuple in the space. `read` and `take` operations are used for retrieving tuples; the former returns a copy of a matching tuple whether the latter destructively removes the matching tuple. When these operations are executed, tuples and templates are transformed according to our encryption scheme. Figure 3 shows the cryptographic operations executed in the PEM and SEM on tuples and templates for a `put` and a `read` (or `take`) operation.

Let us assume that a tuple `t` has to be stored encrypted (i.e., all of its fields must be encrypted). Figure 3-(a) shows the steps executed for a `put` operation. The fields of tuple t are encrypted in the PEM using the submodule E_p [1]. The encrypted tuple t_e is sent to the eSDS where it is re-encrypted by SEM's submodule E_s[2]. The tuple $t_e^{'}$ is stored in the TR.

Figure 3-(b) shows the case of a `read` operation. For a `read` operation a template *temp* is used for finding a matching tuple. The non-null fields in the template are encrypted by the submodule E_p that produces the encrypted template $temp_e$. $temp_e$ is sent to the eSDS where it is re-encrypted in $temp_e^{'}$. This is used for performing the encrypted search. When an encrypted tuple $t_e^{'}$ matches the template $temp_e^{'}$, the tuple must be decrypted before it is returned to the client. First, $t_e^{'}$ is decrypted in the SEM using the D_s submodule and it is transformed in t_e. t_e is returned to the client's proxy that decrypts it using D_p, returning the tuple in cleartext t to the client.

5.1 Evaluation

The eSDS prototype is implemented in Java using the packages provided in the standard Java 1.5 distribution. We chose AES as the symmetric cipher which

[1] This submodule implements the algorithms $CEnc$ and $CEnc'$ that we described in Section 4.

[2] This submodule implements the algorithms $SEnc$ and $SEnc'$ that we described in Section 4.

encrypts the actual data and SHA-1 as the hash function. For the RSA-based proxy encryption scheme, we used 1024-bit keys. For the keyword encryption scheme, q' was 160-bit and p' was 1024-bit. The tests were executed on a Intel Pentium IV 3.2 GHz (dual core) with 1 GB of RAM.

The first evaluation consisted of measuring the execution time for the encryption and decryption submodules. In particular, we measured the execution time for:

- Client Encryption: consists in the execution of E_p, that is encrypting tuple fields using the symmetric cipher, encrypting the symmetric key and encrypting the keywords.
- eSDS Encryption: consists in the execution of E_s, that is the re-encryption of the symmetric key and the keywords using the eSDS keys.
- eSDS Decryption: pre-decryption of the symmetric key by executing D_s.
- Client Decryption: decryption of the symmetric key and the tuple fields by executing D_p.

Table 1 provides the results of our test for the execution of the encryption and decryption operations. The time is given in milliseconds for a single execution of each operation calculated on the average time for 10,000 executions. The tuple and template used for the experiments consisted in a single field of type `eString` with 4 chars.

We also measured the time for finding a matching tuple using our encrypted search. In the data space, 10000 encrypted tuple were stored and only one was a match for the template used in the search. We ensured that the matching tuple was the last tuple to be evaluated (worst case scenario). Tuples and template consisted of a single `eString` filed with 4 chars. Under these conditions, the time required for finding the matching tuple is around 600 milliseconds. Basically, each matching test takes around 0.06 milliseconds.

Given the results of this performance analysis, we can say that the use of our scheme is well suited for cases where a large number of tuples need to be searched. The search is performed entirely within the data space and the result that is returned is a tuple matching the given template. In contrast, when executing the same experiment using an approach as in KLAIM, executing cycles and bandwidth would be wasted. In fact, the result that is given back to a client is a partial match to the given template (only the fields not encrypted are used for the matching). The client has to decrypt the tuple and if the values of the encrypted fields are not the intended ones then the client has to re-encrypt the tuple and send it back to the space.

Table 1. Performance of Encryption and Decryption Operations

Execution Step	Execution Time (ms)
Client Encryption	53
eSDS Encryption	37
eSDS Decryption	37
Client Decryption	37

6 Host Attack

In this section, we discuss some of the attacks that can be performed by malicious hosts.

Existing research focused on protecting the data space from attacks performed by malicious clients. The assumption is that hosts where the data space is deployed are fully trusted while clients are not to be trusted. As discussed in [6], existing approaches protect the data space against malicious clients that:

1. remove and/or forge tuples from a data space to disrupt the collaboration between genuine clients, and
2. insert into a data space a large number of tuples to consume all resources.

Because hosts are fully trusted, there are no mechanisms in place that can guarantee the confidentiality of data stored in the data space against the hosts other than encrypting non-searchable data.

In our attack model, we assume that a host is *honest-but-curious*. We trust the host to correctly authenticate the clients and to perform the operations as requested by the clients. The confidentiality of the data is protected from the host while supporting search on the protected data. However, in deploying the data space on untrusted hosts other concerns need to be addressed. In the following, we list some of these concerns and the threats to which the data space is exposed to. Our aim is not to provide a concrete solution to each of them, but to highlight possible future research directions that aim to protect the data space from untrusted hosts.

Integrity. An attacker that has access to the data space hosts could threaten the integrity of the data space in several ways. For one, the attacker could alter the authorisation process allowing unauthorised clients to access the tuples (even if the clients are not able to decrypt them) or it could deny access to authorised clients. An attacker can alter the semantics of the data space operations. For instance, a client can be blocked in executing a retrieving operation while the matching tuple is in the space; the attacker can re-send back to a client a tuple that was the result of a previous operation (replay attack); additionally, the attacker can discard tuples inserted by legitimate clients modifying in this way the results of retrieval operations. Although no mechanisms could prevent the attacker form performing such attacks, methods developed for database systems could help in detecting and mitigating some of those attacks. For example, methods based on cryptographic techniques and hash functions would allow a client to determine whether the returned result corresponds to the real content of the database. These methods could be extended to include the notion of time with the encrypted representation of the actual content of the data space. In this way, a client would be able to detect whether the blocking for a removal operation was caused maliciously by a host or just because the tuple was not present at the time the request was made. To make sure that tuples inserted by genuine clients are not discarded by malicious hosts some global encrypted indexing could be used. Finally, the integrity of tuples can also be compromised. For instance, an attacker can change or reorder tuple fields (reordering attack).

Availability. Clients that try to connect to the SDS hosts may experience some disruptions. For instance, the data space host is not reachable or it requires a long time for replying. These disruptions may be caused maliciously by the attacker. In order to mitigate such attacks, mechanisms that ensure accountability are required. Accountability is the property that allows the participants of a system to determine and expose misbehavior. In this way, clients can determine whether hosts are behaving correctly. Accountable mechanisms have been proposed for network storage as in [21].

Traffic Analysis. By monitoring the timing and frequency of the communication between hosts and clients, an attacker can gather useful information. By monitoring the execution time of encryption and decryption operations on tuples an attacker can gather enough information to efficiently recover the client key. For instance, in [17] Song shows that it is possible to use such an attack to recover a password exchanged in the SSH protocol 50 times faster than using a brute force attack. The attacker can also built a statistical attack by comparing the templates with the matching tuples.

Collusion. One of the major concerns in proxy encryption schemes comes from a collusion attack. If a client colludes with an attacker that has access to all the EDS side keys, then it is possible to recover the master keys by combining their keys. Collusion-resistant proxy encryption schemes is an open problem. However, we can lower the risk of collusion to an acceptable level by implementing other mechanisms. For example, we can limit the access to the keys by using tamper-proof devices. We can also split the master keys into multiple shares and introduce additional servers, making collusion more difficult. Monitoring and auditing to detect collusion can also help to mitigate the risk.

7 Conclusions and Future Work

In this paper, we presented a novel encryption scheme that ensure tuples confidentiality even in the case that the data space is deployed on untrusted hosts.

The scheme supports encrypted search for a matching tuple over the encrypted data space. In this way, the data space never has access to tuples in cleartext protecting the confidentiality of the its content from nosey hosts. Moreover, the scheme does not require the clients to share secret keys. Each client has its own key that can be used for retrieving tuples encrypted by other clients' keys. This greatly reduces the burden of key management. for instance, when a key of a client is revoked it is not necessary to invalidate all the other clients' keys and re-encrypt the entire data space content.

We provided an implementation of an encrypted SDS using the presented scheme and performed some preliminary performance analysis.

Finally, we discussed a wider class of security threats that arise when data spaces are deployed in untrusted hosts. This threat analysis can be seen as a starting point for some future work. We are currently looking at Private

Information Retrieval (PIR) schemes that would allow a user to retrieve tuples from a data space without revealing to its host which items were searched.

As concluding thought, we would like to point out that although this scheme has been presented in the context of the SDS model, it could be applicable to any other systems where the confidentiality of data shared among several entities must be protected, i.e. databases, publish subscribe systems, email servers, etc.

References

1. Bellare, M., Rogaway, P.: Optimal Asymmetric Encryption. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 92–111. Springer, Heidelberg (1995)
2. Bettini, L., De Nicola, R.: A Java Middleware for Guaranteeing Privacy of Distributed Tuple Spaces. In: Guelfi, N., Astesiano, E., Reggio, G. (eds.) FIDJI 2002. LNCS, vol. 2604, pp. 175–184. Springer, Heidelberg (2003)
3. Blaze, M., Bleumer, G., Strauss, M.: Divertible protocols and atomic proxy cryptography. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 127–144. Springer, Heidelberg (1998)
4. `http://www.doc.ic.ac.uk/~cd04/papers/noshare.pdf`
5. Elgamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE Transactions on Information Theory 31(4), 469–472 (1985)
6. Focardi, R., Lucchi, R., Zavattaro, G.: Secure shared data-space Coordination Languages: a Process Algebraic survey. Science of Computer Programming 63(1), 3–15 (2006)
7. Fujisaki, E., Okamoto, T., Pointcheval, D., Stern, J.: Rsa-oaep is secure under the rsa assumption. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 260–274. Springer, Heidelberg (2001)
8. Gelernter, D.: Generative Communication in Linda. ACM Trans. Prog. Lang. Syst. 7(1), 80–112 (1985)
9. Gelernter, D., Carriero, N.: Coordination Languages and their Significance. Commun. ACM 35(2), 96–107 (1992)
10. Goldreich, O.: Foundations of Cryptography. Basic Applications, vol. II. Cambridge University Press, Cambridge (2004)
11. Gorrieri, R., Lucchi, R., Zavattaro, G.: Supporting Secure Coordination in SecSpaces. In: Fundamenta Informaticae, IOS Press, Amsterdam (2005)
12. Handorean, R., Roman, G.C.: Secure Sharing of Tuple Space in Ad Hoc Settings. In: Focardi, R., Zavattaro, G. (eds.) Electronic Notes in Theoretical Computer Science, Elsevier, Amsterdam (2003)
13. Ivan, A.A., Dodis, Y.: Proxy cryptography revisited. In: NDSS, The Internet Society (2003)
14. Picco, G.P., Murphy, A.L., Roman, G.-C.: Lime: Linda Meets Mobility. In: Garlan, D., Kramer, J. (eds.) Proc. 21st Int'l Conf. on Software Engineering (ICSE 1999), Los Angeles (USA), pp. 368–377. ACM Press, New York (1999)
15. Rivest, R.L., Shamir, A., Adleman, L.M.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM 21(2), 120–126 (1978)
16. Russello, G.: Separation and Adaptation of Concerns in a Shared Data Space. Ph.D. Thesis, Department of Computer Science, Eindhoven University of Technology (June 2006)
17. Song, D.X., Wagner, D., Tian, X.: Timing Analysis of Keystrokes and Timing Attacks on SSH. In: Proc. of 10th USENIX Security Symposium (2001)

18. Udizir, N., Wood, A., Jacob, J.: "Coordination with Multicapabilities. In: Jacquet, J.-M., Picco, G.P. (eds.) COORDINATION 2005. LNCS, vol. 3454, pp. 79–93. Springer, Heidelberg (2005)
19. Vitek, J., Bryce, C., Oriol, M.: Coordinating Processes with Secure Spaces. In: Proc. of Conf. on Coordination Models and Languages, Science of Computer Programming, vol. 46, pp. 163–193 (2003)
20. Wood, A.: Coordination with attributes. In: Ciancarini, P., Wolf, A.L. (eds.) COORDINATION 1999. LNCS, vol. 1594, p. 21. Springer, Heidelberg (1999)
21. Yumerefendi, A.R., Chase, J.S.: Strong accountability for network storage. ACM Trans. on Storage 3(3) (October 2007)