

Providing Data Confidentiality against Malicious Hosts in Shared Data Spaces[☆]

Giovanni Russello^a, Changyu Dong^a, Naranker Dulay^a, Michel Chaudron^b,
Maarten van Steen^c

^a*Imperial College London, UK*

^b*Leiden Universiteit, The Netherlands*

^c*Vrije Universiteit Amsterdam, The Netherlands*

Abstract

This paper focuses on the protection of the confidentiality of the data space content when Shared Data Spaces are deployed in open, possibly hostile, environments. In previous approaches, the data space content is protected against access from unauthorised application components by means of access control mechanisms. The basic assumption is that the hosts (and their administrators) where the data space is deployed have to be trusted. When such an assumption does not hold, then encryption schemes can be used to protect the data space content from malicious hosts. However, such schemes do not support searching on encrypted data. As a consequence, performing retrieval operations is very expensive in terms of resource consumption. Moreover, in these schemes applications have to share secret keys requiring a very complex key management. 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 components to share keys for inserting and retrieving tuples. Each authorised component can encrypt, decrypt, and search encrypted tuples without having to know other components' 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

With its referential and temporal decoupling of processes, the Shared Data Space (SDS) model provides an attractive paradigm for developing distributed applications. The model was introduced by Gerlenter in the coordination language Linda [11] and its properties further analysed with Carriero in [12]. The

[☆]A preliminary version of this paper appeared in Coordination'08.

referential decoupling property means that application components exchange data without the need to know each other's references. The temporal decoupling property means that application components do not need to be synchronise their execution to communicate. This enables a loosely-coupled model of building applications since components can be connected to or disconnected from the data space at any time, making it easier to compose and/or replace them. The unit of data that is exchanged through the SDS is called *tuple* and it is a ordered sequence of typed fields.

Early implementations of SDSs were *closed* systems. These systems were realised by compiling together the code of an application and of the SDS system. Once the system was deployed and executed, it was not possible to add or remove application components. As a consequence, the original Linda model was conceived without addressing security concerns. However, the SDS model becomes an effective coordination layer for distributed applications with the introduction of *open* SDS systems. In open SDS systems, the SDS is an autonomous process with its own resources that are not bound to applications. In this way, persistent data storage is offered to all applications allowing components to dynamically join and leave the computational environment taking full advantages of its decoupling of communication in space and time properties.

The SDS model has been successfully used for building distributed applications deployed in environments that range from wide area networks [20] to small ad-hoc sensor networks [9, 4]. In light of the fact that the original Linda model does not address security, in such open scenarios SDS models are vulnerable to security attacks. This security deficiency poses a limitation on the usability of the SDS model for real-world applications since it is very simple for malicious components to mount security attacks. For instance, a denial of service attack could be performed by a malicious component that could insert a large number of tuples into the data space or remove any tuples from the space, interfering with the other components that are using the space. This can lead to even more serious consequences when tuples stored in the SDS contain sensitive information.

In order to mitigate the effects of attacks against a SDS, several approaches have been proposed in the literature. Most of the approaches [25, 15, 14, 24] focus on providing access control mechanisms for allowing only authorised components to access the data space. The common assumption of these approaches is that the hosts where the data space is deployed are managed by trusted entities that will not try to compromise the data space and its content. As such, (1) the hosts correctly enforce the access control mechanisms and (2) they are *oblivious* of the data that is stored in the data space. However, it is becoming increasingly popular for its cost-effectiveness to outsource to third-party organisations the management of servers where the SDS system could be deployed. In such a scenario, the assumption above does not always hold.

For instance, if the hosts where the data space is deployed are managed by a malicious administrator, then there are several security attacks that the administrator could mount. The administrator could compromise the integrity of the data space by either altering the semantics of the operations or the matching

algorithm for finding a tuple. Data confidentiality can also be violated since the administrator can easily gain access to the data, ultimately making the access control mechanisms not effective.

The examples above are just some of the security threats that that may be considered when a SDS host is compromised. One of the main contributions of this paper is to provide a wide review of other security attacks and for each sketch possible solutions. Our main goal is to trigger other research efforts in this direction. For this paper however, we focus on providing confidentiality for the data space content even in the presence of compromised hosts. Our other main contribution is to present a novel encryption scheme that supports encrypted tuple search. Our scheme guarantees tuple confidentiality because the retrieval operation are performed without having the data space to decrypt the data. Therefore, the host cannot gain any meaningful information on the data being searched. Moreover, the scheme does not require the application components to share secret keys, allowing for a more flexible key (user) management. In particular, the scheme avoids having to re-encrypt the tuple space when a key needs to be revoked. To the best of our knowledge, this is the first approach that implements such features for the SDS model.

The rest of this paper is organised as follows. Section 2 outlines the original SDS model. In Section 3, we review different types of attack that an SDS deployment can face if an SDS host is compromised. In Section 4, we present an application scenario to illustrate the application domain and security threats that our approach is targeting. In Section 5, we discuss the architecture of our approach and its implementation. Section 6 describes our new encryption scheme that supports encrypted searches over encrypted tuples without a shared key for clients. We formally prove the security properties of the scheme in Section 7. Section 8 provides details on key management by revisiting the application scenario. An evaluation of the prototype is then presented in Section 9. Section 10 compares our approach to other related approaches aimed at providing security for the SDS model. We conclude in Section 11 with some final thoughts and future research directions.

2. The Shared Data Space Model Explained

The shared data space model was introduced in the coordination language Linda [11]. 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 parameter of the template. Templates may contain wildcards, which match any value. 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 omitted.

3. The Host Attack Model

Existing research on SDS security has focused on protecting the data space from attacks performed by malicious application components. The assumption is that SDS hosts are fully trusted while application components are not. As discussed in [8], existing approaches protect the data space against malicious components that:

1. remove and/or forge tuples from a data space to disrupt the collaboration between genuine components, 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 protect the data stored from attacks performed by malicious host administrators. In the following, we discuss several attacks that can be performed by such an adversary and for each highlight possible solutions.

3.1. Integrity

An attacker that has access to the data space hosts can threaten the integrity of the data space in several ways. The attacker could alter the authorisation process allowing unauthorised users to access the tuples (even if the users are not able to decrypt them) or it could deny access to authorised users. An attacker can alter the semantics of the data space operations. For instance, a user can be blocked in executing a retrieving operation while the matching tuple is in the space; the attacker can re-send to a user a tuple that was the result of a previous operation (replay attack); additionally, the attacker can discard tuples inserted by legitimate users modifying in this way the results of retrieval operations. Although no mechanisms can prevent the attacker from performing such attacks, methods developed for database systems can help in detecting and mitigating some of those attacks. For example, methods based on cryptographic techniques and hash functions would allow a user to determine whether a returned result corresponds to the real content of the database. Such 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 user 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 users are not discarded by malicious hosts global encrypted indexing [17] can be used. Finally, the integrity of tuples can also be compromised. For instance, an attacker can change or reorder tuple fields (reordering attack).

3.2. Availability

Users 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. 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 misbehaviour. In this way, users can determine whether hosts are behaving correctly. Accountable mechanisms have been proposed for network storage as in [26].

3.3. Traffic Analysis

By monitoring the timing and frequency of the communication between hosts and users, an attacker can gather useful information. By monitoring the execution time of encryption and decryption operations on tuples an attacker might be able to gather enough information to efficiently recover the user key. For instance, in [22] 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.

3.4. Confidentiality

Tuple confidentiality can be violated when a privileged user or an adversary who becomes a privileged user has access to the host where the data space is stored. Even if the space is protected by means of access control mechanisms, a privileged user can still access the content of the tuples. To address this problem, Bettini and De Nicola proposed in [2] an encryption scheme that could protect the tuple confidentiality from attacks from the hosts. However, because the tuples are encrypted they are not meaningful therefore it is not possible to perform search operations. The encrypted tuple can be returned to the user that decrypts the data locally. This results in an approach that is very inefficient in terms of bandwidth. Moreover, issues related to key management (e.g., key distribution, key revocation) are not addressed.

In this paper, we concentrate on confidentiality attacks performed by an attacker that has access to the hosts where the space is deployed. In the following, we present a case study to better illustrate the application domain and security threats that our approach focused on.

4. An Example of an Application Scenario

As an example of an application scenario, in this section we discuss the case of a messaging application realised via the SDS model. In our scenario, the SDS

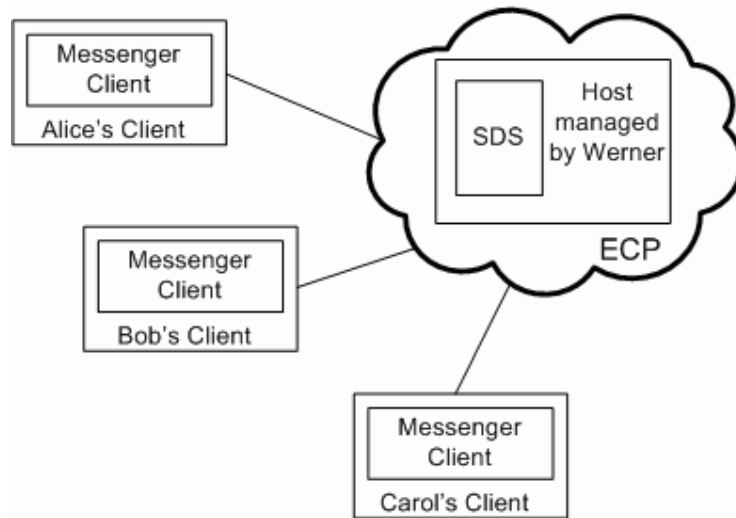


Figure 1: Overview of the case study realised through the SDS model.

is deployed on a computing platform managed by a third party company. This type of service is increasingly becoming popular among companies like Amazon and Google that are expert in managing large numbers of servers scattered across the world. By paying a fee that is proportional to the actual usage of resources, it is possible to expect the same level of availability and reliability that every day millions of users experience with the companies' main portals (Amazon's book store and Google's search engine).

Alice is a software engineer that wants to share messages with a group of close friends and relatives (i.e., her boyfriend Bob and her sister Carol). Alice creates a messaging application based on the SDS model, with application components providing a graphical interface for inserting and retrieving messages to and from the space. Since Alice does not want to set up a server to deploy the space, she decides to use the Exciting Computing Platform (ECP) service provided by the company Amtron. The overview of the case study is depicted in Figure 1.

Werner is a curious system administrator working for Amtron that manages the ECP hosts. If Alice uses a SDS implementation that provides access control mechanisms for guaranteeing data confidentiality then for Werner it is not too difficult to access the actual content of the space (since the tuples are stored in cleartext). Alice could use KLAIM [2] as her SDS implementation to be deployed on the ECP. However, the KLAIM encryption scheme is not efficient in terms of communication since it does not support encrypted searches. Moreover, since the scheme is based on all users sharing a secret key, key management becomes a burden if the key needs to be revoked (i.e., the key is compromised or a user has to be removed from the group).

The following sections discuss our implementation of an Encrypted Shared Data Space (eSDS) that is based on a novel encrypting scheme that supports

encrypted searches and does not require its users to share encryption keys. After the details of our eSDS implementation and encryption scheme have been provided and its security properties formally proved, we will return to our case study discussing an implementation of the messaging application based on the eSDS.

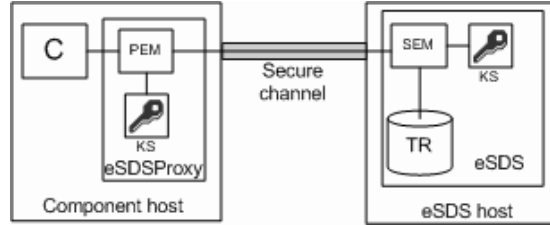


Figure 2: Overview of the architecture of the eSDS.

5. Encrypted Shared Data Space

In this section, we discuss the implementation of the eSDS. The prototype is an extension of our implementation of a distributed SDS, called GSpace [21].

Figure 2 provides an overview of the modules that are part of our architecture. Components and the eSDS are different processes that reside in different hosts. A component C_i communicates with the eSDS by means of a proxy, called **eSDSProxy**. The eSDSProxy takes care of hiding from the application component all the details for establishing a secure channel with the eSDS and deals with the cryptographic operations. The eSDSProxy and the eSDS are provided with a **KeyStore** (KS) for storing the appropriate keys used for encryption and decryption of tuples.

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. For example, a tuple defined as follows:

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

is stored in the eSDS, only the first two fields will be encrypted while the field **weight** will be stored in cleartext. The eSDS provides a **Tuple Repository (TR)** where tuples can be stored and retrieved.

The idea of our encryption scheme is based on *proxy cryptography* [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 [16].

In our implementation, the **Proxy Encryption Module (PEM)** and the **Space Encryption Module (SEM)** provide the implementation details of our encryption scheme (more details on the encryption scheme will be provided in Section 6). 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 transformations executed in the PEM and SEM on tuples and templates for a `put` and a `read` operation (the case of a `take` operation is similar to that of a `read`).

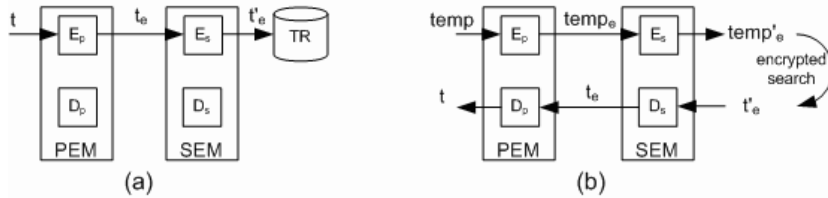


Figure 3: Encryption steps executed for storing and retrieving a tuple using our scheme.

Figure 3-(a) shows the steps executed for a `put` operation on a tuple t . First, tuple t is transformed 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 $temp'_e$. Once in this form, the template can be used for performing the encrypted search. If an encrypted tuple t'_e matches the template $temp'_e$, the tuple must be decrypted before it is returned to the client. The decryption is also a two-step transformation. First, t'_e is transformed in t_e by the SEM using the D_s submodule. t_e is returned to the client's proxy that transforms it using D_p , returning the tuple in cleartext t to the client.

In the following, we describe our encryption scheme that guarantees the confidentiality of the tuples from untrusted hosts while supporting search on encrypted data.

¹This submodule implements the algorithms $CEnc$ and $CEnc'$ that we will describe later in Section 6

²This submodule implements the algorithms $SEnc$ and $SEnc'$ that we will describe in Section 6

6. 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.

6.1. Cryptographic Preliminaries

Our multi-user searchable encryption scheme employs *RSA public-key encryption* [19] and *Discrete Logarithms*. RSA involves two asymmetric keys. The key pair is generated as follows: First choose two random large primes 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 $ed \equiv 1 \pmod{\phi(n)}$. (n, e) is the public key and d is the private key. To encrypt, compute $c = m^e \pmod{n}$. To decrypt, compute $m = c^d \pmod{n}$. In the rest of the paper, we assume all arithmetic to be \pmod{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 $g^x \pmod{p}$, it is hard to find x . Discrete Logarithms have been used in constructing public-key encryption schemes [6], 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 $e_1 e_2 \equiv e \pmod{\phi(n)}$. 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 $m^{e_1} \pmod{n}$ and the other party re-encrypts it as $(m^{e_1})^{e_2} \equiv m^{e_1 e_2} \equiv m^e \pmod{n}$. The decryption key can also be split in the same way.

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.

6.2. Architecture

The system has the following components:

- **Client:** a client is any agent interacting with the data space (i.e., application components).
- **Encrypted Shared Data Space (eSDS):** this 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 tuple fields in plaintext or 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.

6.3. 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 $h \equiv g^x \pmod{p'}$ 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.

6.4. 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 $e_{i1}e_{i2} \equiv e \pmod{\phi(n)}$, $d_{i1}d_{i2} \equiv d \pmod{\phi(n)}$ and $a_{i1}a_{i2} \equiv a \pmod{q'}$. 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 $e_{i1}x \equiv 1 \pmod{\phi(n)}$ always has a solution, then $e_{i2} \equiv ex \pmod{\phi(n)}$ always satisfies $e_{i1}e_{i2} \equiv e \pmod{\phi(n)}$. 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 $e_{i1}d_{i1}e_{i2}d_{i2} \equiv ed \equiv 1 \pmod{\phi(n)}$, $k_1 = e_{i1}d_{i1}$ 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.

6.5. 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. 4. For a tuple $t = \langle d_1; \dots; d_n \rangle$, we denote the value of a field at position x by d_x .

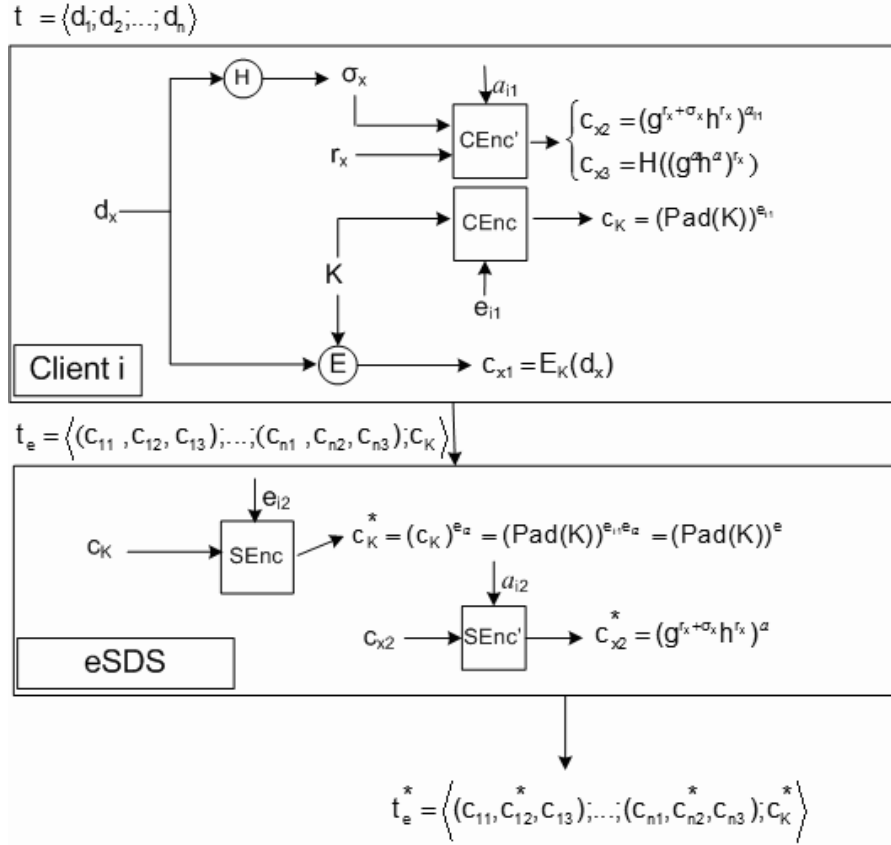


Figure 4: Encryption of a tuple on client i and data space.

On the client side, a tuple is first encrypted using a semantically secure symmetric encryption algorithm E [13]. 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 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 [10]. The ciphertexts of the symmetric key 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 appropriately 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 $c_{x2} = (g^{r_x + \sigma_x} h^{r_x})^{a_{i1}} \bmod p'$, $c_{x3} = H((g^a h^a)^{r_x})$, 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 $t_e = \langle (c_{11}, c_{12}, c_{13}); \dots; (c_{n1}, c_{n2}, c_{n3}); c_K \rangle$ 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 field x , the eSDS computes $c_{x2}^* = c_{x2}^{a_{i2}} = (g^{r_x + \sigma_x} h^{r_x})^{a_{i1} a_{i2}} = (g^{r_x + \sigma_x} h^{r_x})^a \bmod p'$. The final encrypted tuple stored is $t_e^* = \langle (c_{11}, c_{12}^*, c_{13}); \dots; (c_{n1}, c_{n2}^*, c_{n3}); c_K^* \rangle$.

6.6. 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 $temp = \langle z_1, \dots, z_n \rangle$, 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 $temp_e = \langle Q_1; \dots; Q_n \rangle$. j sends $temp_e$ to the eSDS.

The eSDS computes for each field of the received template $Q'_x = Q_x^{a_{j2}} \bmod p' = g^{-\sigma_x^* a} \bmod p'$. 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:

$$\begin{aligned} y_{x1} &= c_{x2}^* Q'_x = (g^{r_x + \sigma_x} h^{r_x})^a g^{-\sigma_x^* a} = (g^{ar_x + a\sigma_x} h^{ar_x}) g^{-a\sigma_x^*} \bmod p' \\ y_{x2} &= H(y_{x1}) \end{aligned}$$

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

6.7. 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 $t_e^* = \langle (c_{11}, c_{12}^*, c_{13}); \dots; (c_{n1}, c_{n2}^*, c_{n3}); c_K^* \rangle$ the eSDS computes $c'_K = (c_K^*)^{d_{j2}}$ and sends to j the following tuple $t'_e = \langle c_{11}; \dots; c_{n1}; c'_K \rangle$. 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 $d_x = E_K^{-1}(c_{x1})$.

7. Security Analysis

7.1. Attack Model

We focus the scope of our scheme on protecting data confidentiality, therefore we will not consider attacks on data integrity and availability which can be handled by other mechanisms. For the scheme, we assume that the KMS and the authorised users are fully trusted. We also assume they can properly protect their secrets, for example, the key pairs and the parameters for generating keys. The server is modelled as “honest-but-curious”, i.e. we trust it to correctly execute the instructions from the clients, but do not want it to access the plain data. An adversary Adv is an attacker (or a software agent) that gains privileged access to the data storage: either an outsider or a untrustworthy employee in the data centre. The adversary can also intercept the communications between clients and the server, but it is computationally bounded. In addition, the adversary is restricted to only perform passive attacks, i.e. attacks are based upon observed data. This restriction is reasonable because: (1) in most cases Adv is physically isolated from the users; (2) most communications between the clients and the server are one-round and initialised by the client, i.e. query-reply. The goal of the adversary is to gather direct or indirect information about the stored data.

7.2. Formal Security Proof

We now give the formal notions of security and proof of security for our Multi-agent Searchable Encryption Scheme.

We first prove that the RSA proxy encryption scheme is semantically secure. Semantic security means that the ciphertexts are indistinguishable to the adversary, therefore the adversary learns nothing by looking at the ciphertext. Loosely speaking, the proxy encryption scheme is semantically secure if by knowing the public parameter n , all the key pairs on the server side, ciphertexts encrypted under an authorised user’s encryption key and any information can be derived from above, e.g. intermediate ciphertexts calculated using the server side keys, but without knowing any key pairs in the authorised user key pair set \mathcal{K}_u , no PPT adversary can distinguish the corresponding plaintext.

Lemma 1. *Let $IGen$ be the master key generation algorithm which is identical to the key generation algorithm in the standard RSA; $UGen$ be the algorithm for generating the key pairs for the users and the proxy; \mathcal{K}_u be the set of user-side key pairs; \mathcal{K}_p be the set of server-side key pairs. The proxy encryption scheme \mathcal{E} is semantically secure against any PPT attacker (i.e. $Succ_{A,\mathcal{E}}$ is negligible) where*

$$Succ_{\mathcal{A},\varepsilon} = Pr \left[b' = b \left| \begin{array}{l} m_0, m_1 \in \{0, 1\}^l, \\ b \stackrel{R}{\leftarrow} \{0, 1\}, \\ (p, q, n, \phi(n), e, d) \leftarrow IGen(1^k), \\ (\mathcal{K}_u, \mathcal{K}_p) \leftarrow UGen(\phi(n), e, d), \\ b' \leftarrow \mathcal{A}(\mathcal{K}_p, n, m_b^\varepsilon), \varepsilon \in \mathcal{K}_u \end{array} \right. \right] - \frac{1}{2}$$

PROOF. We will show that if a PPT attacker Adv can break the proxy encryption scheme, i.e. $Succ_{\mathcal{A},\varepsilon}$ is not negligible, then there is an attacker \mathcal{B} who can break RSA-OAEP, which is semantically secure.

The goal of \mathcal{B} is to distinguish ciphertexts encrypted by RSA-OAEP where the corresponding RSA key pair is (e, d) . Given m_0, m_1 and a ciphertext c_b where $b \stackrel{R}{\leftarrow} \{0, 1\}$, \mathcal{B} can pick x pairs of random primes $\frac{n}{2} < (e_{\mathcal{B}}, d_{\mathcal{B}})_i < n - 2^{161}$. The primes are relatively prime to $\phi(n)$ because $\frac{\phi(n)}{2} < (e_{\mathcal{B}}, d_{\mathcal{B}})_i < \phi(n)$. \mathcal{B} then sends $m_0, m_1, c_b, n, (e_{\mathcal{B}}, d_{\mathcal{B}})_i, i = 1, \dots, x$ to Adv .

Adv can compute $c_{b_1} = c^{e_{\mathcal{B}_1}}, c_{b_2} = c^{d_{\mathcal{B}_1}}$. Next we will show that $c_b, c_{b_1}, c_{b_2}, n, (e_{\mathcal{B}}, d_{\mathcal{B}})_i, i = 1, \dots, x$ can correctly simulate adv 's knowledge in the proxy encryption scheme. First we will show that c_b, c_{b_1}, c_{b_2} are valid ciphertexts for the proxy encryption scheme. The ciphertexts are valid if there exists a d' such that $c_{b_2}^{d'} = m_b$, i.e. $ee_{\mathcal{B}_1}d_{\mathcal{B}_1}d' \equiv 1 \pmod{\phi(n)}$. Because $e_{\mathcal{B}_1}, d_{\mathcal{B}_1}$ are relatively prime to $\phi(n)$, we can always find y such that $e_{\mathcal{B}_1}d_{\mathcal{B}_1}y \equiv 1 \pmod{\phi(n)}$. Therefore there always exists $d' \equiv dy \pmod{\phi(n)}$ such that $ee_{\mathcal{B}_1}d_{\mathcal{B}_1}d' \equiv ee_{\mathcal{B}_1}d_{\mathcal{B}_1}dy \equiv (ed)(e_{\mathcal{B}_1}d_{\mathcal{B}_1}y) \equiv 1 \pmod{\phi(n)}$. We also need to show that $(e_{\mathcal{B}}, d_{\mathcal{B}})_i, i = 1, \dots, x$ are valid server side key pairs, this can be easily proved using the similar method as above therefore is omitted.

Now with the message from \mathcal{B} , Adv can distinguish m_b with probability $Succ_{\mathcal{A},\varepsilon}$ and returns the result to \mathcal{B} . This means \mathcal{B} can distinguish ciphertext encrypted under RSA-OAEP with non-negligible probability $Succ_{\mathcal{A},\varepsilon}$, which is impossible because RSA-OAEP has been proved to be semantically secure.

We then prove that the keyword encryption is semantically secure.

Lemma 2. *Let the keyword encryption $\mathcal{KE} = (Pub_para, Sec_para, \mathcal{K}_u, \mathcal{K}_p, Enc)$ where Pub_para is the public parameter set, Sec_para is the secret parameter set, $\mathcal{K}_u, \mathcal{K}_p$ are the user and proxy key sets respectively, Enc, Dec are the encryption/decryption algorithms. It is semantically secure against any PPT attacker (i.e. $Succ_{\mathcal{A},\mathcal{KE}}$ is negligible) where*

$$Succ_{\mathcal{A},\mathcal{KE}} = Pr \left[b' = b \left| \begin{array}{l} m_0, m_1 \in \{0, 1\}^l, \\ b \stackrel{R}{\leftarrow} \{0, 1\}, \\ b' \leftarrow \mathcal{A}(Pub_para, \mathcal{K}_p, Enc_k(m_b)), k \in \mathcal{K}_u \end{array} \right. \right] - \frac{1}{2}$$

PROOF. The ciphertext of a keyword m_b in the form of $c_{m_b} = ((g^{r_{m_b} + \sigma_{m_b}} h^{r_{m_b}})^{a_{i1}}, H((g^a h^a)^{r_{m_b}}))$. It's easy to see that if r_{m_b} is selected uniformly randomly from $Z_{q'}$, then $g^{r_{m_b} + \sigma_{m_b}} h^{r_{m_b}}$ is distributed uniformly in $G_{q'}$. We will show that if $Succ_{\mathcal{A},\mathcal{KE}}$ is non-negligible, then there is an attacker \mathcal{B} who can win the following game with a non-negligible probability $Succ_{\mathcal{B},\mathcal{C}}$, which contradicts the fact that r is random.

$$Succ_{\mathcal{B},\mathcal{C}} = Pr \left[b' = b \mid \begin{array}{l} m_0, m_1 \in \{0,1\}^l, \\ b \xleftarrow{R} \{0,1\}, r \xleftarrow{R} Z_{q'}, \sigma_{m_b} = H(m_b) \\ b' \leftarrow \mathcal{A}(p', q', g, h, H, g^{r+\sigma_{m_b}} h^r) \end{array} \right] - \frac{1}{2}$$

\mathcal{B} first sends m_0, m_1 to the encryption oracle and receives $g^{r+\sigma_{m_b}} h^r$. Then it chooses a random number $a \in Z_{q'}$ and generates n pairs of (a_{i1}, a_{i2}) such that $a_{i1}a_{i2} \equiv a \pmod{p'}$. It also computes $\sigma_{m_0} = H(m_0)$ and $\theta = g^{r+\sigma_{m_b}} h^r g^{-\sigma_{m_0}}$, it is clear that $Pr[\theta = g^r h^r] = \frac{1}{2}$. Then \mathcal{B} sends $(m_0, m_1, p', q', g, h, g^a h^a, (g^{r+\sigma_{m_b}} h^r)^{a_{i1}}, (g^{r+\sigma_{m_b}} h^r)^{a_{i2}}, H(\theta^a), a_{i2}, \dots, a_{n2})$ to \mathcal{A} . If $\theta = g^r h^r$, then \mathcal{A} can output $b' = b$ with probability $Succ_{\mathcal{A},\mathcal{K}\mathcal{E}}$. Therefore the probability of \mathcal{B} winning the game is $Succ_{\mathcal{B},\mathcal{C}} = Succ_{\mathcal{A},\mathcal{K}\mathcal{E}}/2$, which is non-negligible.

The semantically secure definition for searchable encryption is tricky because searching leaks information inevitably. As long as the searching algorithm is correct, it always returns the same result set for the same query. Although the queries and the result sets are encrypted, the adversary can still build up search patterns. Therefore the security definition for searchable encryption should be modified to reflect the intuition that nothing should be leaked beyond the outcome and the pattern of a sequence of searches. Here we adapt the definition from [5] and prove our scheme is non-adaptive semantically secure. Informally, non-adaptive semantic security means that given two non-adaptively generated query histories with the same length and outcome, no PPT adversary can distinguish one from another with non-negligible probability. Non-adaptive means the adversary cannot choose queries based on the prior queries and results. This is acceptable because in our setting, only the authorised user can generate queries.

We first introduce some notions to be used in the definition. Δ is the set of all possible data items, i.e. documents. $\mathcal{D} = \{D_1, \dots, D_n\}$ denotes an arbitrary subset of Δ , i.e. $\mathcal{D} \in \mathcal{P}(\Delta)$, and each D_i is a document. $\mathcal{W} = \{w_1, \dots, w_d\}$ is a dictionary which contains all the possible words can be used in the queries. Each document in \mathcal{D} is associated with a local unique identifier $id(D_i)$, and a set of keywords $kw(D_i)$ which is a subset of \mathcal{W} . The result set of a search query w on a document set is denoted by $rs(w)$, which is the set of document identifiers of all the documents in \mathcal{D} that contain the keyword, i.e. $\{id(D) \mid D \in \mathcal{D} \wedge w \in kw(D)\}$. A *history* is defined in terms of a sequence of queries made on a document set.

Definition 1 (History). A history $H_q \in \mathcal{P}(\Delta) \times \mathcal{W}^q$ is an interaction between a client and a server over q queries on a document set \mathcal{D} , i.e. $H_q = (\mathcal{D}, w_1, \dots, w_q)$.

During the interaction, the adversary cannot directly see the history because the documents, keywords and queries are encrypted. What the adversary can see is a *view*, i.e. the encrypted version of the history. Let E be the symmetric key encryption scheme, \mathcal{E} be the proxy encryption scheme and $\mathcal{K}\mathcal{E}$ be the keyword encryption scheme, Q_i be an encrypted query, the view of the adversary is then defined as:

Definition 2 (View). Given a document set \mathcal{D} with n documents and a history over q queries $H_q = (\mathcal{D}, w_1, \dots, w_q)$, an adversary's view of H_q is defined as:

$V(H_q) = (id(D_1), \dots, id(D_n), E_{k_1}(D_1), \dots, E_{k_n}(D_n), \mathcal{E}(k_1), \dots, \mathcal{E}(k_n), \mathcal{KE}(kw(D_1)), \dots, \mathcal{KE}(kw(D_n)), Q_1, \dots, Q_q)$.

As we have stated above, searching leaks information. The maximum information we have to leak is captured by a *trace*. In our settings, a trace contains information from three sources: the encrypted file stored on the server, e.g. the id, length and number of keywords of each document, the result set and the query pattern.

Definition 3 (Trace). Given a document set \mathcal{D} with n documents and a history over q queries H_q , the trace of H_q is defined as:

$Tr(H_q) = (id(D_1), \dots, id(D_n), |D_1|, \dots, |D_n|, |kw(D_1)|, \dots, |kw(D_n)|, rs(w_1), \dots, rs(w_q), \Pi_q)$. Π_q is the search pattern over the history which is a symmetric binary matrix where $\Pi_q[i, j] = 1$ if $w_i = w_j$, and $\Pi_q[i, j] = 0$ otherwise, for $1 \leq i, j \leq q$.

The security definition is then based on the idea that the scheme is secure if no more information is leaked beyond what the adversary can get from the traces. This intuition is formalised by defining a game where the adversary has to distinguish two histories, possibly on two different document sets, which have the same trace. Since the traces are identical, the adversary cannot distinguish the two histories by the traces, i.e. the knowledge he already has. He must extract additional knowledge from what he can see during the interactions, i.e. the views. The negligible probability of the adversary successfully distinguishing the two histories implies that he cannot get extra knowledge and in consequence the scheme is secure.

Definition 4 (Non-Adaptive Semantic Security). Our searchable data encryption is Non-Adaptive Semantically Secure if for all $q \in \mathbb{N}$, for all (H_0, H_1) which are histories over q queries and $Tr(H_0) = Tr(H_1)$, and any PPT adversary \mathcal{A} , $Succ_{\mathcal{A}}$ is negligible:

$$Succ_{\mathcal{A}} = Pr \left[b' = b \mid \begin{array}{l} Pub_para, Sec_para, \mathcal{K}_u, \mathcal{K}_p \leftarrow SETUP(1^k), \\ H_0, H_1 \in \mathcal{P}(\Delta) \times \mathcal{W}^q, \\ b \xleftarrow{R} \{0, 1\}, \\ b' \leftarrow \mathcal{A}(Pub_para, \mathcal{K}_p, V(H_b)) \end{array} \right] - \frac{1}{2}$$

Theorem 1. *The enhanced construction is non-adaptive semantically secure.*

PROOF. Let's examine each part of the view.

Document identifiers $id(D_1), \dots, id(D_n)$: Because $Tr(H_0) = Tr(H_1)$, this part of the view must be identical for the two histories. So the adversary cannot distinguish the two histories by the document identifiers.

Encrypted documents $E_{k_1}(D_1), \dots, E_{k_n}(D_n)$: The adversary cannot distinguish because E is semantically secure.

Encrypted symmetric keys $\mathcal{E}(k_1), \dots, \mathcal{E}(k_n)$: \mathcal{E} is based on RSA-OAEP which is IND-CCA2 secure. Therefore is also indistinguishable.

Encrypted keywords $\mathcal{KE}(kw(D_1)), \dots, \mathcal{KE}(kw(D_n))$: We have proved they are indistinguishable to the adversary in lemma 2.

Encrypted queries Q_1, \dots, Q_q : Because $Tr(H_0) = Tr(H_1)$, we don't need to consider the query pattern and can reduce the problem to distinguish any two sequences of distinct queries: $(Q_{01}, \dots, Q_{0m}), (Q_{11}, \dots, Q_{1m}), m \leq q$. For each $Q_{ij}, i \in 0, 1, 1 \leq j \leq m$, it is a pseudorandom number $g^{a_1 H(w_{ij})} \bmod p'$. Therefore the queries are not distinguishable as long as the discrete logarithm problem is hard.

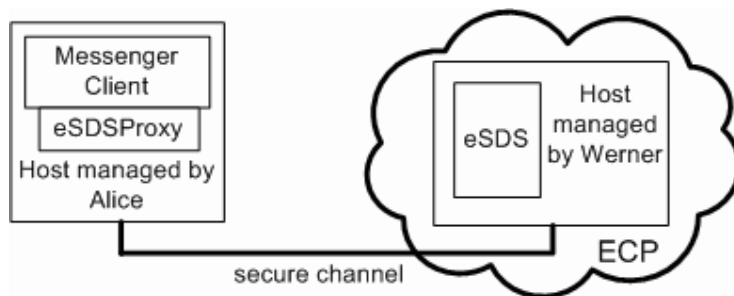


Figure 5: Deployment of the Messenger application component and eSDS.

8. Key Management

In this section, we use the application scenario presented early in Section 4 to provide more details on how key management is achieved through the eSDS. Firstly, we describe the actual implementation and deployment of the application realised through the eSDS. Afterwards, we concentrate on the key initial setup and management.

Figure 5 provides an overview of the actual deployment of the messaging application and eSDS. The application component `MessengerClient` is deployed on the user's host (in this case Alice's host). The eSDS is deployed on the ECP on a host managed by Werner. The application component provides a graphical interface for composing, sending and retrieving messages. The component is securely connected to the eSDS via the `eSDSProxy` that establishes a secure channel with a kernel deployed on the hosts managed by Amtron.

Messages are represented by means of tuples that are instances of the type defined as follows:

```
TupleMessage(eString sender,
             eString receiver,
             eString message,
             String date);
```

The tuple contains four fields representing the sender, receiver, message content, and the date when the message was sent, respectively. When the tuple is

inserted in the space the sender, receiver and the content of the message will be encrypted while the date is stored in cleartext (for Alice decided that the confidentiality of the date in the message is not crucial for her application).

The sending of a message is executed by inserting a tuple in the space by means of a `put` operation. A user can send a message to multiple receivers. In this case, for each receiver a new instance of a tuple is created and inserted according to the following code excerpt:

```
while(!receiver_list.isEmpty())
    put(new TupleMessage(MY_ID,
                        receiver_list.next(),
                        msg,
                        current_date));
```

The loop iterates through the receiver values inside the `receiver_list`, inserting a new instance of a tuple for each receiver. The value `MY_ID` represents the id of the user running the application and in this case sending the message. `msg` represents the content of the message and `current_date` provides in the string form the date and time when the message is sent.

To retrieve the messages for a user, the client executes in a separate thread an infinite loop shown in the code excerpt below:

```
tmp = new TupleMessage(null, MY_ID, null, null);
while(true){
    message = take(tmp);
    deliver(message);
}
```

Inside the loop, a `take` operation is performed with the template `tmp`. The template specifies only the receiver id while the other fields are set to the wildcard value `null`. If no messages are available for the user specified in the receiver field, the template specified in `take` operation will not match with any tuple and the `take` blocks. Once a matching tuple is found, then the `take` returns such a tuple (`message`) to the user application. It is worth noticing here that the matching operation is executed on typed fields as for the Linda model. In fact, the template is just an instance of a tuple type where each field has a specific type associated with it. Such an approach is for instance used in JavaSpaces [7] where all the tuples are subclasses of the `Entry` class. In the example presented here, the template `tmp` will match only tuples where the first three fields have type `eString` and the last one has type `String`.

8.1. Initial Setup

Figure 6-(a) shows the steps that Alice performs to deploy her application. The steps are described in detail in the following:

1. Alice creates an executable image of the eSDS that is deployed on the ECP. Alice initialises the eSDS for creating a space instance called `WonderSpace`.

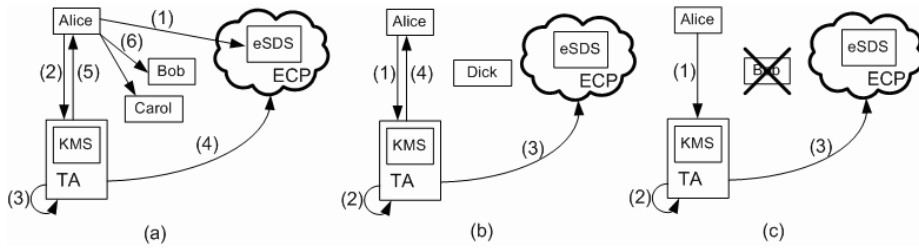


Figure 6: (a) Initial setup. (b) Adding of a new user. (c) Revocation of a user.

Alice associates with her space instance a certificate of a trusted authority TA that manages the KMS.

2. Alice is a registered user of TA and once she has authenticated herself Alice can request the KMS managed by the TA to generate the keys for her space. The request contains the information about the users' identity (i.e., Alice as owner of the space; Bob and Carol as normal users); the name of the space instance (in this case *WonderSpace*) and the location of where the instance is deployed (for example, ECP can provide a fixed IP address to identify the host where the image of the eSDS is executing).
3. The KMS executes the setup algorithm as described in Section 6. For each user specified in Alice's request, the KMS generates two key pairs, the client key pair (CKP) and the space key pair (SKP). The KMS stores the information about the users and pair keys for the *WonderSpace* in its local table. The information in the table can be updated by Alice. For instance, Alice can request to create a new key pair for a new user or can revoke the key of an existing user (more details on how a user can be added or an existing user removed will be provided in the following).
4. The KMS contacts the eSDS on the ECP and sends the user ids and SKPs regarding Alice's *WonderSpace* encrypted with the private key of the TA. The eSDS uses the public key in the certificate provided by Alice for the decryption and stores the information in its key store.
5. The KMS sends back to Alice the CKPs. Alice stores her CKP into the key store of the space proxy deployed on her device. Once the CKP is stored, Alice can connect to the space and deliver the first message for the other users.
6. Furthermore, it is Alice's task to securely distribute and install in the key store of the proxies the CKPs to the respective users. When this step is completed, then Bob and Carol will be able to retrieve the message that Alice sent to them and (possibly) write back.

8.2. User Management

User management is performed by the owner of the space. Adding and removing users requires the involvement of the KMS and the eSDS to update their information about the users and key pairs. In the following, we describe the actions required by Alice for adding and removing users from *WonderSpace*.

Figure 6-(b) depicts the case in which Alice wants to add Dick, Carol's boyfriend, as a user of **WonderSpace**. Each step is described in the following:

1. Alice authenticates to the TA and informs the KMS to add a new user for her space **WonderSpace**.
2. The KMS generates a new set of keys for the new user and updates its local table with the new information.
3. The KMS contacts the eSDS and sends the new information (Dick's id and SKP).
4. The KMS sends back to Alice the new CKP for Dick. Once Alice has stored Dick's CKP to the key store of Dick's application, Dick will be able to send new messages to the other and retrieve messages that have been sent to him even before he was enrolled in the system by the KMS.

Let us assume that Alice and Bob split up. Alice decides to revoke Bob access to **WonderSpace**. To achieve this, Alice has to perform the steps depicted in Figure 6-(c) and described as follows:

1. Alice authenticates with the TA and sends the request for the revocation of the key pairs for user Bob from **WonderSpace**.
2. The KMS removes the entry for user Bob from its table.
3. Moreover, the KMS sends a request to the eSDS to remove the entry for user Bob from its key store.

It is important to note here that in order to exclude Bob from her space, Alice does not need to revoke all the other keys or to re-encrypt the messages in the space. Once Bob's SKP has been removed from the eSDS key store, it is impossible for Bob to decrypt messages inserted in the space even if Werner would allow him to bypass the authentication mechanism of the eSDS. In fact, with just Bob's CKP it is impossible to decrypt the messages in the space.

9. 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 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.

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

Table 1: Performance of Encryption and Decryption Operations

- 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 [2], 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.

10. Related Work

This section provides a review of existing approaches to shared data space (SDS) security.

Secure Lime, described in [15], introduces several security extensions to Lime [18]. 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. 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. To counter 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 [24] introduces the notion of a *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 [14] 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 multcapabilities [23] is an approach where the capability concept is applied to the Linda model. Capabilities are the means by which agents can access tuples and the SDS. In particular, a multcapability is a special capability that refers to a group of tuples. A multcapability 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 multcapability refers to; p , a set of permitted operations on the matching tuples. To be able to exchange tuples, two or more agents have to share the same multcapability that refers to the same set of tuples. In case a multcapability has to be revoked, the authors adopt the common solution of introducing *indirect multcapability 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, the 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 can still be retrieved. The only exception to this is KLAIM [2]. KLAIM provides confidentiality 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 costs.

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 cannot 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 the case that access privileges have to be removed to an agent.

11. Conclusions and Future Work

In this paper, we have presented a novel encryption scheme that ensures tuple confidentiality even in the case that the data space is deployed on an untrusted hosts. The scheme supports encrypted search for matching tuples over the encrypted data space and 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 - when the 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. The security properties of the scheme were proved. Although the scheme has been presented in the context of the SDS model, it is applicable to any other system where

the confidentiality of data shared among several entities must be protected, i.e. databases, publish subscribe systems, email servers, etc.

The paper also discussed many of the additional security threats that can arise when data spaces are deployed on untrusted hosts and suggested possible solutions for them. An implementation of the encrypted SDS scheme was described and preliminary performance results presented. A case study of a messaging application based on our implementation of an encrypted SDS was also described.

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 the SDS host which items were searched.

References

- [1] M. Bellare, P. Rogaway. “Optimal asymmetric encryption.” *In EUROCRYPT*, 92–111, 1994.
- [2] L. Bettini and R. De Nicola. “A Java Middleware for Guaranteeing Privacy of Distributed Tuple Spaces.” *In Proc. of FIDJI’02, Int. Workshop on scientific engineering of distributed Java applications*, pp. 175–184, LNCS 2604, N. Guelfi, E. Astesiano, G. Reggio, Eds., Springer, 2003.
- [3] M. Blaze, G. Bleumer, M. Strauss. “Divertible protocols and atomic proxy cryptography.” *In EUROCRYPT*, 127–144, 1998.
- [4] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. “Programming Wireless Sensor Networks with the TeenyLIME Middleware.” *In Proceedings of the 8th ACM/IFIP/USENIX International Middleware Conference (Middleware 2007)*, Newport Beach (CA, USA), November 26–30, 2007.
- [5] R. Curtmola, J.A. Garay, S. Kamara, R. Ostrovsky. “Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions.” *In Proceeding of the 13th ACM Conference on Computer and Communications Security*, 79–88, Alexandria, VA, USA, October 2006.
- [6] T. Elgamal. “A public key cryptosystem and a signature scheme based on discrete logarithms.” *IEEE Transactions on Information Theory*, 31(4), 469–472, 1985.
- [7] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces, Principles, Patterns and Practice*. Addison-Wesley, Reading, MA, 1999.
- [8] R. Focardi, R. Lucchi, and G. Zavattaro. “Secure shared data-space Coordination Languages: a Process Algebraic survey.” *Science of Computer Programming*, 63(1): 3–15, Elsevier, 2006.
- [9] C. Fok, G. Roman, C. Lu. “Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications.” *In Proceedings*

of the 24th International Conference on Distributed Computing Systems (ICDCS'05), pp. 653–662, Columbus, Ohio, June 6-10, 2005.

- [10] E. Fujisaki, T. Okamoto, D. Pointcheval, J. Stern. “Rsa-oeap is secure under the rsa assumption.” In *Kilian, J., ed.: CRYPTO*, 260–274, Volume 2139 of Lecture Notes in Computer Science, Springer, 2001.
- [11] D. Gelernter. “Generative Communication in Linda.” *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, 1985.
- [12] D. Gelernter and N. Carriero. “Coordination Languages and their Significance.” *Commun. ACM*, 35(2):96–107, Feb. 1992.
- [13] O. Goldreich. “Foundations of Cryptography: Volume II Basic Applications.” Cambridge University Press, 2004.
- [14] R. Gorrieri, R. Lucchi, G. Zavattaro. “Supporting Secure Coordination in SecSpaces.” In *Fundamenta Informaticae*, IOS Press, 2005.
- [15] R. Handorean and G. C. Roman. “Secure Sharing of Tuple Space in Ad Hoc Settings.” In *Electronic Notes in Theoretical Computer Science*, Riccardo Focardi and Gianluigi Zavattaro Eds., Elsevier, 2003.
- [16] A.A.Ivan, Y. Dodis. “Proxy cryptography revisited.” In: *NDSS, The Internet Society*, 2003.
- [17] H. Pang, A. Jain, K. Ramamritham, and K. Tan. “Verifying Completeness of Relational Query Results in Data Publishing.” In *Proceeding of the 2005 ACM SIGMOD International Conference on Management of Data*, 407–418, Baltimore, MD, 2005.
- [18] G. P. Picco, A. L. Murphy, and G.-C. Roman. “Lime: Linda Meets Mobility.” In *Proc. 21st Int’l Conf. on Software Engineering (ICSE’99)*, ACM Press, pp. 368-377, Los Angeles (USA), D. Garlan and J. Kramer, eds., May 1999.
- [19] R.L. Rivest, A. Shamir, L.M. Adleman. “A method for obtaining digital signatures and public-key cryptosystems.” *Commun. ACM*, 21(2), 120–126, 1978.
- [20] A. Rowstron and S. Wray. “Run-Time System for WCL.” *Internet Programming Languages*, eds. H. Bal, B. Belkhouche and L. Cardelli, pages 78–96, Springer-Verlag, LNCS 1968, 1999.
- [21] G. Russello. “Separation and Adaptation of Concerns in a Shared Data Space.” Ph.D. Thesis, Department of Computer Science, Eindhoven University of Technology, June 2006.
- [22] D. X. Song, D. Wagner, and X. Tian. “Timing Analysis of Keystrokes and Timing Attacks on SSH.” In *Proc. of 10th USENIX Security Symposium*, 2001.

- [23] N. Udizir, A. Wood, and J. Jacob. “Coordination with Multicapabilities.” *Proc. 7th Int’l Conf. on Coordination Models and Languages (Coordination 2005)*, 3454:79–93, Jean-Marie Jacquet and Gian Pietro Picco, editors, Springer-Verlag, Berlin, 2005.
- [24] J. Vitek, C. Bryce and M. Oriol. “Coordinating Processes with Secure Spaces.” In *Proc. of Conf. on Coordination Models and Languages*, Science of Computer Programming, 46, pp. 163–193, 2003.
- [25] A. Wood. “Coordination with attributes”. In *Proc. of Coordination Languages and Models*, LNCS, vol. 1594, pp. 21-36, Springer-Verlag, Berlin, Heidelberg, 1999.
- [26] A. R. Yumerefendi and J. S. Chase. “Strong accountability for network storage.” In *ACM Trans. on Storage*, 3(3), October 2007.