# Local Key and Certificate Storage in JDK 1.3 *

Oskari Pirttikoski
Helsinki University of Technology
Department of Computer Science
FIN-02015 HUT, Espoo, Finland
oskari.pirttikoski@hut.fi

Yki Kortesniemi
Helsinki University of Technology
Department of Computer Science
FIN-02015 HUT, Espoo, Finland
yki.kortesniemi@hut.fi

## Abstract

*Cryptographic algorithms have already become a basic technique used in many application areas, like confidentiality and content protection. However, the big question of key management still remains unanswered. As a part of it, to efficiently use public key cryptography, we require a good local storage system to guarantee the confidentiality and availability of keys and certificates.*
*In this paper, we take a look at local key and certificate storage in JDK 1.3. We discuss the requirements for such systems, evaluate the standard JDK solution,* `KeyStore`*, and finally propose an improved solution that does not suffer from the many shortcomings of* `KeyStore`.

## 1. Introduction

Open, unprotected networks and the emergence of digital content have brought problems such as confidentiality and content protection to the lives of more and more people. As we no longer can rely on the physical protection of the communication lines or the distribution media, we have to find new solutions to guarantee the security where necessary.

The standard solution seems to be the use of cryptography. Cryptography used to be difficult and inefficient, known only to the experts, but today there exist numerous well-documented algorithms for solving many standard security problems. However, there still remains a big obstacle hindering widespread use of cryptography, namely key management. Sure, we can create keys, but how do we store and distribute them efficiently? Public key cryptography was supposed to solve many of these questions, but, in fact, it only transformed the question. Earlier, the question was: how do I confidentially transmit the key to the intended recipient? Now, the question goes: how do I know, to whom this key really belongs?

Public Key Infrastructure (PKI) and digital identity certificates are the current answer to the latter question, but they also have problems. They often imply a trust model not suitable to all applications, as a Trusted Third Party in the form of a Certification Authority (CA) is used to issue the certificates. The use of a CA is not always practical, because of the management overhead involved and the trust problems introduced. Furthermore, we can get the key directly from the person in question, so no CAs are required to assure us, that the key really came from that person.

Some PKIs also support authorisation certificates, which are used to bind authorisation to a public key. Authorisation certificates are not concerned with users name, but her rights and, hence, complement the identity certificates. With authorisation certificates, it is possible to implement access control systems that support anonymity, delegation and dynamic distributed security policy management.

We have used the JDK 1.3 to implement different applications that are based on authorisation certificates, and, in the process, also looked at the included key and certificate management solution, `KeyStore`. `KeyStore` relies on some good principles, but has also some severe limitations, which prevent its use in many cases.

In this paper, we take a look at the requirements a good local key and certificate storage system should

* This work was partly funded by the TeSSA research project at Helsinki University of Technology under a grant from Tekes.

fulfil, evaluate the `KeyStore` in the light of these requirements and, finally, present our solution that does not suffer from the limitations pointed out.

One important aspect to bear in mind is the intended platforms for this solution. As PDAs and other small devices become more commonplace, so does the need for security solutions that take into account their limitations. As the certificate based access control systems are in some ways particularly well suited for these small devices, it is important that the proposed storage solution can be adapted to their limitations, as well.

The improved storage solution that we are to present in this paper is designed and implemented medium and small storage needs in mind. It will be optimal for client side software. We do not suggest this solution to be used on a server used by thousands of users.

The rest of the paper is organised as follows: section 2 gives some background on public key cryptography and digital certificates. Section 3 discusses the requirements for a local key and certificate storage system. In section 4, we evaluate the existing solution, as well as discuss related work. Section 5 presents our design rational and section 6 goes over our implementation. Then, section 7 evaluates this solution, section 8 proposes future work, and, finally, section 9 presents our conclusions.

## 2. Cryptography and certificates

There are two basic types of cryptography: secret key and public key cryptography. Secret key cryptography is the older. It is based on the use of a single key known to all participant and to no-one else. With this key, it is possible to scramble and, later, unscramble information to guarantee its confidentiality. Secret key cryptography is fast, but distributing the keys and keeping them stored safely are big problems.

The more modern type of cryptography is based on the concept of a key pair - it contains a public key and a private key. The two keys are mathematically related but the private key can not be resolved from the public one without enormous computation. Private keys are the secret of the system and the security of the whole system lies in maintaining the confidentiality of the private keys. The public key, on the other hand, can be made public. Here, the big problem is still in the distribution, but this time it is not confidentiality, but authentication: how do we know, which public key belongs to which user.

The standard solution is to use a PKI and digital certificates. Formally, a digital certificate is a fixed form digital document cryptographically signed by the document issuer. The two main types of certificates are the identity and the authorisation certificate. In a PKI, a trusted CA issues identity certificates binding the user's name and her key - in many ways, it resembles a usual proof of identity issued by official government authority. The idea is that once this trusted CA has vouched for the key belonging to the named user, we can operate securely. X.509 [1] is the most popular format for identity certificates at the moment.

An authorisation certificate is more like a ticket that lets its holder to use some service. Here, the authorising ticket is usually issued by the more or less unofficial provider of the service itself, not some official CA. Then again, the certificate is meant to convince the issuer: later on, the users present their tickets to the issuer which then verifies the tickets and serves the justified users.

Currently, the most actively promoted authorisation certificate system is the Simple Public Key Infrastructure (SPKI) [2]. It is an initiative for a more flexible PKI freed from the requirement of an obligatory CA system. It is being prepared by the SPKI working group in the IETF.

### 2.1. SPKI authorisation certificates

In an SPKI authorisation certificate, there are five essential fields: issuer, subject, tag, delegation and validity. The certificate is signed by the issuer's private key and the subject i.e. the receiver of the authorisation can be a public key or a hash of it, an identity or a hash of some object. As no names have to be used, it is possible to operate anonymously.

The actual right granted is expressed in a tag-field and it is completely application dependent. The delegation field is used to express, whether the receiver can further grant the rights or not. This ability to support delegation is a big benefit, as it facilitates distributed management and other interesting applications. Finally, the validity field is used to limit the validity granted in the tag field.

Certificate chains are a result of delegating authorisation further. A receiver of SPKI authorisation can delegate the authorisation or a subset of it to another entity. Authorisation can be delegated by issuing a certificate in which the party that already gains the authorisation is the issuer and the subject is the to be authorised entity. A chain of certificates is a sequence of certificates where the subject of the previous certificate is the issuer in the next certificate. The chain originates from the first certificate issued by the owner of the resource which is followed by certificates delegating the authorisation further to the final subject.

When the final recipient in the chain wants to use the right, the whole chain has to be presented to the owner of the resource for verification. Now, creating a valid certificate chain from a multitude of certificates is equal to finding a path trough a directed graph. The chaining problem is not trivial and is discussed in [3].

Cryptography will allow many new applications if a user can create and sign new certificates on location e.g. in some bureau or shop. In particular, with small personal device such as a PDA, that are nowadays capable to perform the required signatures, these new kinds of applications become quite feasible. The benefits of using a PDA are due to the following two facts: the device does not have to reveal the key as it can perform encryption securely by itself and the user can trust her own personal device more than some terminal operated by the shop owner.

# 3. Criteria for a local repository

The need for an all-round local repository is evident if we intend to utilise cryptography on a more large scale in the future. This section tries to point out the general local repository functions that would be useful for applications using cryptographic keys and certificates. In particular we want to be able to use all kinds of certificates, not just identity certificates.

It should be noted that storing certificates locally is not the only way to have them available. Certificates can also be distributed over the Internet, for instance, by storing them in the DNS as presented in [4]. Distributed storage has great benefits such as providing fresh up to date certificates and ability to search trough numerous certificates available online. Downside is that we have to be online to use the certificates and even then the availability can not be guaranteed. In [4] the bandwidth consumption and response delays are considered not to bother the DNS certificate storage system in many applications.

## 3.1. Attached information

Keys and certificates have to be stored in such a way that they can be efficiently found among all the saved items. Traditionally, simple index or alias has been used to keep a repository organised. However that does not seem to be sufficient because one key or certificate can be used by several applications for several purposes. It is necessary to be able to attach additional describing information to the item to find and use it later. This is because the keys and certificates themselves can be everything but informational.

The attached information can be useful both for the user as well as for applications. The user might want to write down a description of the purpose of the key or certificate so that she knows how to apply the key or what rights the authorisation certificate grants and to whom. The user might also want to set an expiration date for the item so that applications know when the key or certificate should not be used anymore. An application, on the other hand, might want to attach data related to the usage of the key or certificate in this or some other application. Keeping all that in mind we find that the nature of attached information will vary considerably and therefore it should be possible to add new attachable types when necessary.

⇒ **Criterion 1** *Possibility to attach versatile information to stored items.*

## 3.2. Key management

In key management, the secure storage of private and symmetric key material is the most important function of a local repository. The private keys must be kept secret otherwise the whole PKI collapses and the public keys and certificates are of no use.

⇒ **Criterion 2** *Security of private key material.*

Another important function to the private key security is the ability to store public keys with or without a certificate. Public keys and certificates do not need the same security as private keys because they do not contain any secrets. A public key is usually accom-

panied by a certificate where some CA guarantees in various degree the trustworthiness of the key. Suppose one creates a new pair of keys and sends them to a CA to be certified. Until the CA replies there is certainly a need to store the key pair without a certificate. And furhermore, in some cases the reliability of a public key is not related to a certificate from a CA. Take PGP[5] key fingerprints for an example. There is no need for a CA certificate when you have received a hash of the key in person from your counterpart. Therefore, saving public keys without certificates is definitely a mandatory function in a repository.

⇒ **Criterion 3** *Possibility to store public keys without certificate.*

### 3.3. Certificate management

From the certificate management point of view, the most important feature is the ability to search for a desired certificate by various properties of the certificate. For instance, that is necessary to be able to make a chain of certificates. When chaining the certificates, the searched property can be the subject, the issuer or the validity of the certificate. If it would be possible to search by various properties the certificate chaining would be more efficient. The more efficient chaining is possible because we would have more means to limit the number of certificates in the search graph.

⇒ **Criterion 4** *Possibility to search the repository by varying properties.*

### 3.4. Other storable material

Even though key and certificate management is what the repository is mainly for, the repository could also store related information. By related we mean that the information is needed to be able to use the keys and certificates. One example of this related information could be revocation lists or other type of validity information. This material is somewhat dynamic by its nature and is usually downloaded over the network. Even though the validity material is bound to change sooner or later, it might be worth saving just to avoid unnecessary traffic. The benefits of storing all related material behind one interface include facilitated application development and better interoperability between applications. On the downside, it makes the repository more complicated and there is the trouble of finding the right material as separate applications might store similar type of material in the repository.

⇒ **Criterion 5** *Possibility to store the diversity of related material.*

### 3.5. The platform

A repository should naturally be optimised for a platform that is capable of encryption and decryption i.e. using the stored material, which includes practically all the standard Java Virtual Machines. The repository should also be as generally available as SUNs `KeyStore` is with the JDK API.

In small hand held devices, it is necessary to control the size of the repository. That is because the devices usually come with limited memory and storage capacity. The total storage need can also be reduced by using one repository for all the related material as mentioned earlier. The gain here is due to avoiding the overhead of several systems.

⇒ **Criterion 6** *Possibility to control the total size of repository.*

## 4. Related work

`KeyStore` [6] is the default JDK solution for storing cryptographic material locally. The material saved in `KeyStore` is divided into to two categories: *key entries* and *trusted certificate entries.* Key entries include a private key accompanied with its certified public key. Public keys without certificate can not be stored in the `KeyStore`. Trusted certificate entries, on the other hand, are to store identity certificates including public keys or other type of certificates such as authorisation certificates. The entry categorisation is clearly designed only with identity based PKIs in mind, as public keys supposed to be only within certificates. Furthermore, every entry is given a string alias which can later be used for retrieving the entry. The entries in the `KeyStore` can be browsed only by their aliases.

Compared to other interfaces and abstractions in Java Cryptography Extension (JCE), the interface of `KeyStore` class loses the generality and serves only one purpose, the identity certificate based infrastructure. Therefore, the problem with `KeyStore` is the constricted interface. And that problem can not be

solved or worked around by providing new implementations even though that is made possible by the provider architecture.

The KeyStore interface does not allow search by multiple properties which has been mentioned to be essential when creating a certificate chain. Therefore, KeyStore fails to satisfy criterion number 4. Neither does the KeyStore satisfy criterion number 1 as it does not allow one to attach information to stored items. Furthermore, KeyStore limits the set of storable information to be private keys and certificates so it is impossible to store a public key without a certificate or to store some other related material. Therefore, KeyStore also fails to satisfy criteria 3 and 5. Actually the only criterion it fulfils is number 2 as there is no support for controlling the size of a repository which is demanded as criterion number 6. One criterion out of six is not a very good score and obviously the KeyStore could be improved in many ways.

Of course, if the usage is solely based on X.509 certificates alone, KeyStore can be regarded to provide the minimum set of functions, but the result still lacks many features to make it truly usable.

To have some perspective, we looked at solutions developed on some other platform than Java and we found that the lack of such interfaces defined in JCE forces the solutions to be tailor made for the applications. The tailor made solutions are likely to increase the inertia in utilising cryptography in applications as even the most basic components have to be done again and again.

As an example, Distributed Computing Environment (DCE) stores symmetric keys in a keytab file [7]. This is certainly a tailor made solution as only symmetric keys are stored. However the interesting point here is the information saved with the keys. Every saved key is accompanied with a name of the key holder, a key version number and a timestamp. The version number and timestamp can be considered as examples of the attachable information that can be useful for applications using the key.

A more commonly known key management solution is the PGP [5] keyring. The keyring provides means for managing trust rather than just a way to store keys. No describing information can be added to the stored keys but the searching is somewhat more flexible than from KeyStore, as keys can be recovered with only a fragment of the user identifier string. All kinds of keys can be added to keyring without a certificate, as all saved keys are internally certified for their use in the keyring.

PGP keyring provides functionality way beyond what we have defined here for a repository. On the other hand, it is only able to store keys, which have to be keys from a fixed set of algorithms such as RSA and DSS depending on the version and implementation of PGP. Therefore, keyring is still a tailor made solution, even though PGP in general does not hinder the use cryptography, and can not be considered as general purpose repository.

## 5. Design choices

Once we came to the conclusion that a new repository solution was required, we were faced with several questions.

One of the first was, how to store the Java objects representing keys and certificates. The Java environment provides a general serialisation scheme which allows one to save an object and its state and then recover the object with the same state later. The saved data includes all the other objects that can be reached from the first object. Although Java serialisation scheme is available for most objects, it might not be as effective as possible. The user might know a much more efficient way to store the relevant information in the object and leave the irrelevant out. This is why we decided that the user should provide a class that contains the knowledge of how to encode and decode the object she wishes to save. This makes possible for the user to use serialisation or some more optimised encoding whichever she prefers.

Another choice of design was how to provide a convenient way to protect every private key with a separate passphrase. The choices were to encrypt the private key with another passphrase or to encrypt it with another key. If another key is to be used then the encryption should be done on a trusted portable device i.e. PDA which will store the key. Otherwise there is not much sense using another key because the problem of storing the key would remain the same. In the end, we decided to support both of these ways. Another passphrase can be applied or the user can get the private key bytes, encipher them with a portable

device and then store the encrypted key bytes in the repository.

Then there was the question of how the repository should be searched. Instead of integrating the searching functionality to the repository seamlessly we decided to provide it as a separate utility. In our opinion, this made the repository interface more rational. And as we thought more about the searching, it became evident that we should not fix the properties of the stored material that were searched for. After all, we were looking for a solution where the user could quite freely store various type of material and attach just as various type of information to the material. The application dependent tag field of SPKI certificate is a good example of a property that can vary greatly in structure. The varying structure makes it impossible to fix some method for searching SPKI certificates by tag. So it was not possible nor rational to enumerate all the searchable properties of all the potential material. Hence, we had to let the user define for what properties she wanted to search.

One a bit controversial feature that we came upon is whether the repository should remove automatically all the expired material or not. There is more than one angle to look at this problem. The benefit of automatic removal would be smaller repository with no cryptographic garbage around. However, on the downside we can see problems arising, if we have the liability to justify our action years after they took place. One example here could be access control to a database. Suppose the access is gained by presenting a valid authorisation certificate. If some online test is used to confirm the validity of the certificate then the online test results used making decisions should be stored. Otherwise it is later impossible to prove, why someone gained access and someone did not, if they both have valid certificates but the other did not pass the online test. If the database is popular, the repository will grow huge in size but that is the price to pay, if it is necessary to later justify the decisions made.

The scenario in last example is quite realistic as cryptographic signatures are no longer underrated in the court. Choosing between the two removing policies seemed not to be rational so we ended up with the following compromise: all the material is kept in the repository but the expiration date is easily available for every saved item. This way the expired material can be avoided when searching or examining the repository but is still available if necessary.

# 6. LocalRepository

`LocalRepository` is an interface that defines a key and certificate repository which we propose as an alternative for the Java `KeyStore`. We have also an implementation of the repository, which provides the same functionality as `KeyStore` and some more.

## 6.1. Implementation

`LocalRepository` is an open repository which demands the saved material to be wrapped in an implementation of a simple `LocalRepositoryEntry` interface but does not limit the type of material in any other sense. Furthermore, we have defined another straightforward interface `AttributeEntry` for the attachable extra information which we call the attributes. The repository can be searched with a separate search engine `LRSearchEngine` which accepts very versatile set of search criteria. The relation the repository has with the search engine and entries is illustrated in figure 1.

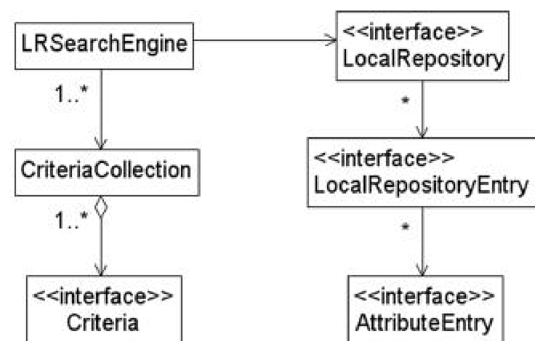


**Figure 1 The architecture of LocalRepository and LRSearchEngine**

## 6.2. Security

The repository as a whole is protected with one passphrase. Furthermore, we provide more finer grained security as we have made it possible to protect every private or symmetric key with a passphrase of its own or by encrypting the key bytes with a PDA like device. For those keys that another passphrase

does not seem to be enough, the encryption with trusted portable device should do.

### 6.3. LocalRepositoryEntry interface

The classes that implement `LocalRepository-Entry` interface include the key, certificate or some other item and the knowledge of how to save the item. The `LocaRepositoryEntry` inherits the `Entry` interface as depicted in figure 2. `Entry` defines two methods `getEncoded()` and `getInstance(byte[])`. The first is to encode the whole entry to bytes and the later is to retrieve the entry from the same bytes. To recover the saved key, certificate or some other object from the entry, a `getItem()` method is defined in the `LocaRepositoryEntry` interface. The `getItem()`is a method which has no counterpart for setting the item, as the item is supposed to be given to the constructor of the implementing class. Methods for attaching, retrieving and detaching the `AttributeEntry`'s are also defined in the `LocalRepositoryEntry` interface.

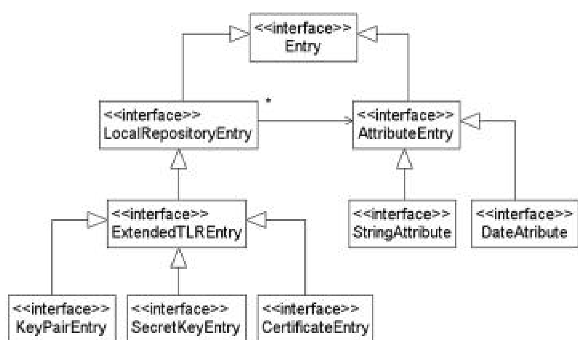


**Figure 2 The structure of repository entries and attributes**

### 6.4. AttributeEntry interface

Attributes are external information that will help both the user and applications to utilise the saved key, certificate or other material. The main purpose of a class that implements the `AttributeEntry` interface is to bring in the knowledge of encoding and decoding the attached information. That is why `AttributeEntry` inherits the same `getEncoded()` and `getInstance(byte[])`methods that were defined in the `Entry` interface. Furthermore, attributes have two properties: the name of the attribute and the value. We have defined the name to be a string. The name of an attribute is unique in the sense that several attributes can not be attached to one `LocalRepositoryEntry` with the same name. Therefore, the name can be used to retrieve the desired attributes unambiguously from an entry. The value of an attribute is of type `Object` because we do not want to limit the type of attribute in any way. In subclasses that implement the `AttributeEntry` interface the type of the value should be limited. One implementing class is `DateAttributeImpl` which encodes a `Date` object efficiently to bytes.

### 6.5. LRSearchEngine

Repository is a sort of a black box which includes useful material. To utilise the material one needs to search the box thoroughly. How thoroughly the search is to be done is left for the user to decide. The search engine we implemented provides only a way to browse trough all the entries in the repository.

The most important interface used searching the repository is the `Criteria` interface. It's implementation is provided by the user and it is used by the search engine to decide which entries are chosen from the repository. The only method `Criteria` interface defines is `satisfies(LocalRepositoryEntry)` which will return true or false depending on whether the given entry satisfies the criteria or not. So the rules by which a `Criteria` chooses the right entries are expressed in Java rather than as a set of keywords and special characters. This allows the rules to be very complex and dynamic. Furthermore, by implementing the `Criteria` interface the user is able to examine all the necessary properties in an entry. This is possible because all the properties are available to the `satisfies` method as the given repository entry includes the saved item itself and all the attached attributes. Hence, all the stored information can be used in the search and it can be examined in complex ways only limited by the Java language.

We have also implemented a way to define logical AND and OR relations between `Criteria` objects. More complex logical relations can be expressed with Java in a class implementing the `Criteria` interface. AND relations are presented as an array of

Criteria. This array of Criteria is put in a class called CriteriaCollection. OR relations are then presented as an array of CriteriaCollection.

# 7. Evaluation

In this section we are going to first evaluate the repository against the presented criteria and then discuss the performance in the light of the tests we ran on the repository.

## 7.1. Evaluation against the criteria

We found 6 criteria for a repository in section 3. Now it is time to check which of those criteria our LocalRepository fulfils.

**Criterion 1** *Possibility to attach versatile information to stored items.*
The criterion number 1 is met as our repository not only allows to attach information but also does not limit the type of attached material in any way.

**Criterion 2** *Security of private key material.*
This as already mentioned is the first and the most important criterion for a repository and is treated with the proper respect in our implementation. A private key can be protected with two passphrases or one passphrase and an external encryption. Passphrases might not be considered as the most secure way to protect secret material but encryption on a trusted device will provide a protection that will be more than enough for this application. Keeping all that in mind the LocalRepository can store the private key material without compromising its confidentiality.

**Criterion 3** *Possibility to store public keys without a certificate.*
This criterion is satisfied in our implementation as we provide possibility to store almost any type of data in our repository.

**Criterion 4** *Possibility to search the repository by varying properties.*
This criteria is not exactly fulfilled by the repository itself, but is dealt with the search engine we provide. The LRSearchEngine allows the user to define the searched properties with absolutely no limitations. This is possible when the user implements the Criteria interface.

**Criterion 5** *Possibility to store the diversity of related material.*
Our repository allows this because we defined a simple interface to implement for all items to be stored. Any arbitrary material can be stored just as well as the material that the repository was designed for.

**Criterion 6** *Possibility to control the total size of repository.*
The LocalRepository has yet no support for limiting or monitoring the size of the repository. However, as the repository could be used in such devices just as well, we feel that this is a imperfection in our implementation and we are interested in adding this functionality later on.

## 7.2. The performance of LocalRepository

To evaluate the performance of the LocalRepository implementation, the duration of storing operations and also the storage overhead were tested for. Where possible, the same tests were also performed on the KeyStore to get some perspective. The precision in timing was 10 ms and some operations lasted less than that.

The test setup was a 550 Mhz PC with Java 1.3 Runtime Environment including the Hot Spot virtual machine. The Java Cryptography Extension we used was implemented by Australian Business Access. The implementation of JCE probably effects the duration of saving and loading the repository because the repository is saved as cipher text. The KeyStore implementation used was the default JKS implementation that comes with JDK.

As table 1 shows, the add and remove operations were fast enough. Furthermore, a complicated search by certificate subject performed also well enough. The search engine of LocalRepository was used for this search and it only took 10 ms to find the right certificate from the multitude of 50 stored certificates. KeyStore does not provide this functionality so there is no result to compare.

The saving and loading operations were likely to take some time as they include encryption or decryption of the repository in addition to the file operations. That in mind, the results of LocalRepository are

excellent compared to those of `KeyStore`. The storing operation is executed over ten times faster in the `LocalRepository` than in the `KeyStore`.

**Table 1 Durations of basic repository and KeyStore operations**

| Operation | Repository | KeyStore |
|---|---|---|
| Adding one entry to the store | << 10 ms | << 10 ms |
| Removing one entry | << 10 ms | << 10 ms |
| Finding a certificate by subject among 50 certificates | 10 ms | N/A |
| Storing of 50 certificates on disc | 100 ms | 1180 ms |
| Reading of 50 certificates from the disc | 215 ms | 1150 ms |

The results of measurements on repository size and storage overhead are presented in table 2 and 3. As key pairs without certificates can not be stored in `KeyStore` the results concern only `LocalRepository`.

In table 2, the results indicate high overhead percentage when storing key pairs in the repository. This is partly due to the basic attributes that are added to an entry automatically. If two more attributes of 150 characters are added the overhead reaches 140 per cent which as high it is can still be considered reasonable and tolerated in the majority of applications.

**Table 2 The repository size and overhead for a repository of 50 1024 bit RSA key pairs**

| 50 RSA key pairs (1024 bit) | Total size | Size per item | Over head |
|---|---|---|---|
| Key pairs | 40 kB | 0,8 kB | N/A |
| Key pairs stored in the repository | 71 kB | 1,4 kB | 75 % |
| Key pairs stored in the repository with additional attributes | 94 kB | 1,9 kB | 140 % |

Table 3 presents the results of storing 50 SPKI certificates into both repository and `KeyStore`. The stored certificates include four public RSA keys which tend to determine the size of the certificate. The repository overhead percentage is considerably lower compared to table 2. This is due to the fact that the overhead in bytes stays pretty much the same even if the size of the stored item is increased. On the other hand, KeyStore reaches overhead of 0 per cent, which is remarkable and might be due to some more efficient coding than what is used with `LocalRepository`. Nevertheless, the results of `LocalRepository` are reasonable and bearable in most applications.

**Table 3 The overhead of a repository and a KeyStore of 50 SPKI certificates**

| 50 SPKI certificates | Total size | Size per item | Over-head |
|---|---|---|---|
| Certificates | 106 kB | 2,1 kB | N/A |
| Certificates stored in repository | 123 kB | 2,4 kB | 13 % |
| Certificates stored in `KeyStore` | 106 kB | 2,1 kB | 0 % |
| Certificates stored with additional attributes | 145 kB | 2,9 kB | 36 % |

The results presented here imply that `LocalRepository` can be used as an alternative for `KeyStore` without compromising performance. As overhead produced by `LocalRepository` is somewhat greater than that of `KeyStore`, but still resonable, the versatility and speed of operations compensate for the `LocalRepository`.

## 8. Future work

After the blueprints for a repository are set here, it is possible to continue the development with more advanced features.

The small portable devices with limited memory set new demands on the repository. These demands such as ability to control the size of the repository and possible problems with performance should be discussed more thoroughly.

The garbage collection in the repository is one non-trivial issue that should be considered. As discussed earlier, garbage collection is the art of balancing between the optimal repository size and filing all the important material for long periods of time. Using the attributes we presented in this paper it is possible to categorise material by its importance and thus perform garbage collection in much more rational way.

## 9. Conclusions

A flexible and secure local repository is a basic requirement for efficient application of cryptography. In this paper, we have glanced at services provided by a repository and discussed the requirements for a good repository.

We then evaluated the standard JDK 1.3 key and certificate repository, the `KeyStore`, against these requirements and found it to be limited in its usability in some application areas, because it makes some limiting assumptions. In particular, it assumes that all public keys are always stored in an identity certificate, and that a single alias is sufficient additional information to describe any key or certificate.

To overcome these limitations, we then proposed a new local repository, discussed our design rational and described our implementation. Finally we evaluated our solution against the criteria and found it capable of fulfilling most of them. We also measured the performance of our implementation and found it to come close with `KeyStore` in storage overhead and outdo it in the speed of operations.

With this new repository, we have been able to implement several applications using public keys and SPKI certificates and feel, that it is a significant improvement over the standard implementation.

For small portable devices, support for limiting the size of the repository is still required and in the future, we shall look into this matter further.

## 10. References


[1] R. Housley, W. Ford, W. Polk, D. Solo: *Internet X.509 Public Key Infrastructure Certificate and CRL Profile*, January 1999.

http://www.ietf.org/rfc/rfc2459.txt

[2] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, T. Ylönen: *SPKI Certificate Theory* IETF SPKI Working Group, September 1999.

http://www.ietf.org/rfc/rfc2693.txt

[3] Tuomas Aura: *Comparison of graph-search algorithms for authorization verification in delegation networks*, in the proceedings of 2nd Nordic Workshop on Secure Computer Systems NORDSEC'97, Espoo, Finland, November 1997.

http://www.tcs.hut.fi/Publications/papers/aura/aura-nordsec97.ps

[4] Tero Hasu, Yki Kortesniemi: *Implementing an SPKI Certificate Repository within the DNS*, Poster Paper Collection of the Theory and Practice in Public Key Cryptography (PKC 2000), 18-20 January 2000, Melbourne, Australia.

http://www.tcm.hut.fi/Research/TeSSA/Papers/Hasu-Kortesniemi/ImplementingAnSPKICertificateRepositoryWithinTheDNS.ps

[5] Philip Zimmermann: *PGP User's Guide*, Revised 11 October 94.

http://www.tml.hut.fi/Opinnot/Tik-110.350/Tehtavat/pgp/index.html

[6] Sun Microsystems: *Java Cryptography Architecture API Specification & Reference*, December 1999.

http://www.java.sun.com/j2se/1.3/docs/guide/security/CryptoSpec.html#KeyManagement

[7] IBM Transarc Lab: *Keytab file*, [Refered 7 August 2000]

http://www.transarc.com/Support/dce/admin_examples/security/keytab.html