JINI DISTRIBUTED KEY EXCHANGE AND FILE TRANSFER SERVICE WITH DIGITAL SIGNATURES

By

Kevin Ligozio

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Department of Computer Science

Rochester Institute of Technology

11/12/2004

Approved by _____

Stanislaw Radziszowski, Chairman

_____

Alan Kaminsky, Reader

_____

Hans-Peter Bischof, Observer

ABSTRACT

Rochester Institute of Technology
Department of Computer Science
Master's Thesis Defense

Title: Jini Distributed Key Exchange and File Transfer Service with Digital Signature
Author: Kevin Ligozio
Date: November 12, 2004, 10:00 am
Place: 70-3000

This thesis compares security algorithms for the Jini Networking Technology. The first method is to add security at the infrastructure level, and is the method chosen by the Jini Community in Jini version 2.0. The second method is the original research portion of this thesis, and explores adding security in the application layer. In order to explore the application layer security, the thesis includes four components. First, we investigate and implement some cryptographic algorithms. Specifically, we use a Java Cryptography Extension (JCE) Provider containing the Diffie-Hellman Key Agreement algorithm and the RSA Digital Signature Algorithm. Second, we perform a novel experiment by creating two Jini services, the JINI Key Agreement Service (JKAS) and the JINI Secure File Transfer Service (JSFTS), in order to analyze Jini application layer security. These services use both the provider developed for this thesis, and also the Cryptix JCE provider that provides the Rijndael encryption and MD5 message digests. Combined with the JCE providers the services perform authenticated key agreement and secure file transfers, both with digital signatures. Third, we create a client application with CLI interface to experiment with the services. Finally, we create a second application that acts as a certificate generator for use with a pseudo-certificate authority.

Committee: Stanislaw Radziszowski, Chairman
Alan Kaminsky, Reader
Hans-Peter Bischof, Observer

This Abstract: http://www.cs.rit.edu/~kml4510/thesis/
Thesis: http://www.cs.rit.edu/~kml4510/thesis/thesis.pdf
Proposal: http://www.cs.rit.edu/~kml4510/thesis/proposal.pdf

TABLE OF CONTENTS

Chapter 1

THESIS OVERVIEW

There are many different ways to add security to a distributed system, and many distributed network technologies. This thesis compares two ways to add security to Jini, a distributed networking technology invented at Sun Microsystems. The first method is to add security at the infrastructure level, and is the method chosen by the Jini Community in Jini version 2.0. The second method is the original research portion of this thesis, and explores adding security in the application layer.

This thesis uses Jini, the Java Cryptographic Extension (JCE), and standard cryptographic methods, all of which are overviewed in the early chapters of this thesis. It also provides an overview of the infrastructure level security added to Jini 2.0, in order that it may be compared with the application level security used by this thesis. The thesis then compares the two methods of adding security. Finally, the thesis details the software development effort for this thesis, namely the kmlJCE provider, the Cryptographic Service Application and the Certificate Generation Utility.

Chapter 2

JINI TECHNOLOGY OVERVIEW

Jini Network Technology is a set of core components built on top of Java to provide a distributed network infrastructure [Edwards, 61]. This chapter provides an overview of Jini for those readers not yet familiar with this technology. This section applies to Jini version 1.1, as that is the version used in most of this thesis.

Jini – The Big Picture

What does it mean to provide a "distributed network infrastructure"? In the case of Jini, it means that it provides a defined set of protocols for basic inter-process interactions, and some utility applications to run with. The following sections will summarize the key Jini protocols [Jini Technology Core Platform Specification: Contents], and the utility applications provided in the Jini toolkit.

Services

A central concept in Jini is the idea of a *service*. According to [Jini Technology Core Platform Specification: AR.2] a service is "an entity that can be used by a person, a program, or another service". They can be thought of as a piece of code that provides a published service in a Jini community[1]. An application may have zero or more services running within them. An application with no Jini services in a Jini community would be a pure client application. That is, an application that is only interested in using services provided elsewhere in the system.

Services Have Persistent State

Services must persist certain state information so that if the service is terminated, and re-run, the service can be re-identified by the Jini community and continue where it left off. The information that is persisted is [Jini Technology Core Platform Specification: DJ – Discovery and Join, DJ.3.1]:

[1] A Jini *community* is a set of Jini services that have joined together. All of the services in a community can find and use each other [Stinson, 63].

- A Service ID: An identifier that uniquely represents a service [Edwards, 286].

- A set of attributes. The attributes describe the service's lookup entry. That is, this describes what services the service provides.

- A set of groups to participate in. (Groups are described in a later section.) If blank, the default group is used.

- A set of specific lookup services. This may be blank if the service wishes to participate with all lookup services.

Proxies

A *proxy* is a Serializable Java component with an advertised interface. It is similar in use and concept to an RMI proxy in that it is used by remote clients to (1) communicate with the back-end service and/or (2) provides services without needing to forward requests to the back-end service. In Jini, however, proxies must be registered with the Jini Lookup Services as belonging to a particular service. (Jini Lookup Services are described in a later section.) The Jini Lookup Service then forwards these proxies to interested clients so that the service and client do not need to locate one another independently.

The Jini Lookup Service (JLUS)

The JLUS acts as a central repository of services available in a Jini community. The Jini toolkit provides a simple implementation of a JLUS as a standalone application. Alternatively, one could a JLUS as part of another application. Interactions between services and the JLUS are governed by the Discovery and Join Protocols, which are described in a later section. The big picture is that the services locate the JLUS's in the Jini community and register *proxies* for each of the services they provide. Then, these services or other applications (clients) use the JLUS to look for other services that they are interested in. For example, all of the printers in an office may have services that register with the JLUS's. When a host on that network wants to print, it will search the JLUS's for services in the "Printers" group. Once found, the JLUS will return all of the proxies for services in that group to the client. The client can then use those proxies to query the printer for information (for example, "where are you located?"), or simply to print on it.

Distributed Leasing

Jini is designed to be self-healing. This means that the system can recover from services exiting the system in unexpected ways (e.g., crashing, killed, etc). Self-healing is accomplished by way of leasing.

The basic interaction is that whenever services locate one another, they are given a *lease* at some pre-configured time interval. Both entities know the time interval, and it is the responsibility of the service consumer to renew the lease within that interval. If not renewed, then the service providing the lease removes its reference to the consumer service.

As an example, consider the JLUS. When a service registers with the JLUS, it takes out a lease. If it doesn't renew that lease in time, the JLUS will remove its reference to that service. If the JLUS didn't do this, then in the earlier printers example applications might get references to printers removed years before!

Distributed Events

Distributed events are a common element in distributed systems, and do not require a lengthy explanation. Jini uses distributed events to allow entities in the Jini system to receive asynchronous notification of a change in state from other entities in the system. In the case of the JLUS, for example, services can register to find when services enter and leave groups. It's notable that leases govern Jini events in order that the system is self-healing, so that unnecessary events are not sent (at least not outside of the lease duration).

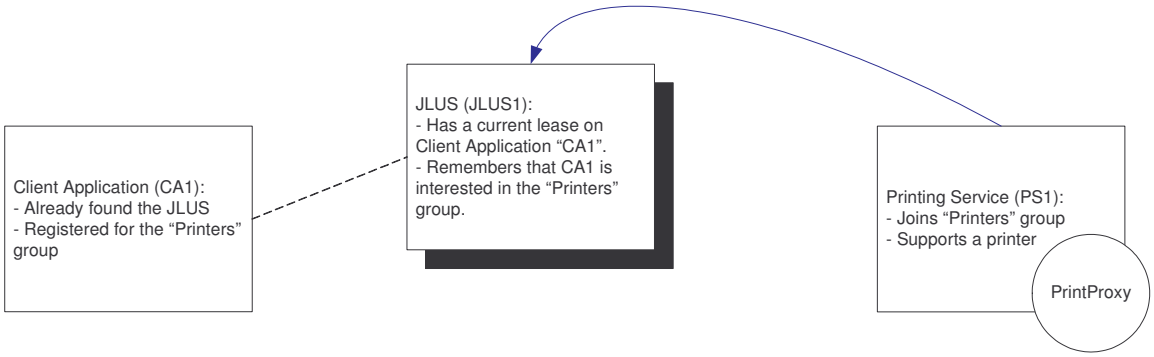


Figure 1: Stage 1 Event Example - Printing Service PS1 Discovers and Joins JLUS1

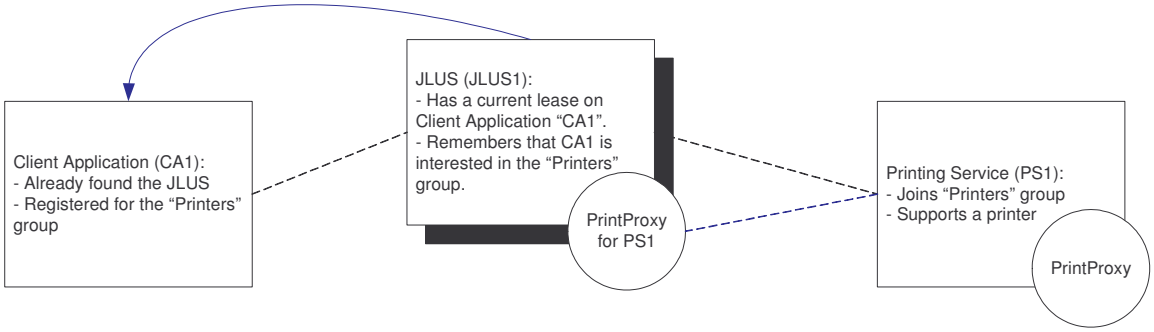


Figure 2: Stage 2 Event Example - JLUS1 Sends a Remote Event to CA1 because a new service has joined the "Printers" group.

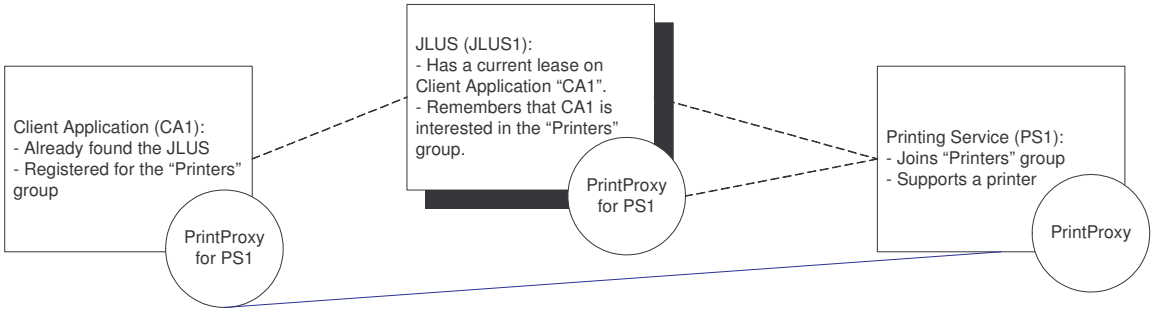


Figure 3: Stage 3 Event Example - CA1 now has an RMI connection, via PrintProxy for PS1, to PS1. The PrintProxy interface would need to have been previously known to CA1.

The Key Jini Protocols

The first thing that services do in a Jini environment is to obtain references to any Lookup Services. The process of finding lookup services is defined by the *discovery protocol*.

The Discovery Protocol

The discovery protocol defines the way in which services find Lookup Services. The protocol makes use of multicast and unicast network communication. The basic mechanism is that when the Jini service starts, it multicasts its presence, and Jini Lookup Services respond. Once discovered, the Jini services register with the lookup services.

The first concept in discovery is that of a *group*. A group in Jini is a logical string name identifier that acts as a name. For example, a Jini service written for a printer may join the "Printers" group. Or, if the system wishes to be more selective, it may join the "Engineering Printers" group, so that the marketing folks know not to use that one. The group concept is important because it is a way for

services to be more selective about which other services they discover. All Jini Lookup Services have a default group of empty string, which is referred to as the "public" group.

If we imagine a system where we can have one or more Jini Lookup Services (JLUS), and one or more other services, we can see that there are two basic scenarios to consider in order that the other services find references to the lookup services. Either a service starts *before* a JLUS, or it starts *after* a JLUS.

The *Multicast Request Protocol* is used when a service first becomes active and needs to find the nearby JLUS's. When an application starts, it uses multicast to announce its presence to the Jini community[2]. [Edwards, 77]. In this way, services can find JLUS's that started before they did.

For the second case, we use the *Multicast Announcement Protocol.* This protocol is used by JLUS's to announce their presence to services that are already running at the time they start. Again, multicast on a well-known port is used.

The last discovery protocol is not used in this thesis. It is the *Unicast Discovery Protocol.* According to [Edwards, 77] this protocol is used to talk directly with lookup services that may not be part of the local network subnet. Using this protocol, two Jini communities can be made to see each other. In Jini language, we say that the two communities are *federated.*

Once discovered, the JLUS returns a proxy to the discovering service that implements the well-known ServiceRegistrar interface. This interface allows the discovering service to *Join* the Jini community [Edwards, 77].

The Join Protocol

Once services have obtained references to the Jini Lookup Service, they need to establish communication with the Lookup Service and other services of interest in the Jini community. The Join Protocol governs this process. Essentially, the protocol gives a sequence of steps that a well-behaved service will follow in order to become a participating member of a Jini community.

In brief, the result of the Join protocol will be that the JLUS's discovered during the Discovery Protocol will have the service proxies registered, and that the service will have a lease with each JLUS. More detailed information on the Join Protocol can be found [Jini Technology Core Platform Specification: DJ – Discovery and Join, DJ.3.2] and [Edwards, 279-288].

[2] Typically, a "community" is defined by a subnet because of the use of multicast.

Jini - Design Tenets

The core components of Jini were built with several tenets in mind. The following is a summary of these principles as given by [Edwards, 62]:

Simplicity

Jini is designed to be simple, both in concept and implementation. During the course of this thesis, I found this to be generally true, though there is a startup cost in running Jini. To run a basic Jini environment, one must run a main web server and Jini lookup service. Typically, one would also run the Java rmid service as well. Next, for each machine in the network running Jini services, one must run an additional web server. Once all of this is running, it's fairly simple to add services to the system, but the initial setup is not without cost to system resources.

Reliability

Jini, in some ways, operates like a name server. The Jini designers, however, built additional behavior into the Jini services to increase reliability. Through the use of multicasting, Jini is able to react to services entering and exiting the system in an expected way. Further, through the use of leasing, Jini is able to respond to services exiting the system in unexpected ways (e.g., a crash where the service does not follow the service exit protocol).

Scalability

A Jini "community" is essentially those services that are reachable by multicast to one another. This is both by design because communities should be small for performance reasons, and by technological limitations because Jini makes heavy use of multicast. Jini scales when a system administrator tweaks the system to allow Jini services from one subnet to see Jini services on another subnet (e.g., by tunneling all multicast requests from one subnet to another). When this is done, the Jini communities are "federated".

Device Agnosticism

This essentially means that Jini runs anywhere, or rather anywhere that a JVM is running. This thesis does not make use of this feature of Jini, as it was not ported. However, there is no reason that the services written could not be ported to a PDA, or some other non-host device.

The preceding design principles demonstrate some of Jini's potential as a network technology. If Jini itself does not become ubiquitous, then in time some other similar technologies likely will, and the research done for this thesis may apply.

Chapter 3

JINI 2.0 SECURITY OVERVIEW

Infrastructure Level Jini Security

During the time that this thesis was written, the Jini Developers Community addressed security in Jini version 2.0. While this thesis explores applying security to the application layer in a Jini specific application, the Jini Developers Community applied security to the infrastructure level. This chapter provides an overview of how this was done so that it can be compared to application level security in a later chapter.

The following sections summarize the three steps added to Jini in order to gain security. The main source for this chapter is [Sommers].

Step 1 – Trusting the Service Proxy

Jini Security assumes that the service is trusted, and centers on validating trust in the service proxy. Since all communication goes through the service proxy, if the service proxy can be trusted, then it is thought the system can be trusted. The following diagram shows the basic movement of data and objects in the initial phase of contact between a service and client. The service proxy is sent from the lookup service to the client, and the client uses this service proxy to make local calls that the proxy can handle itself, or to make remote calls that communicate with the back-end service via RMI.

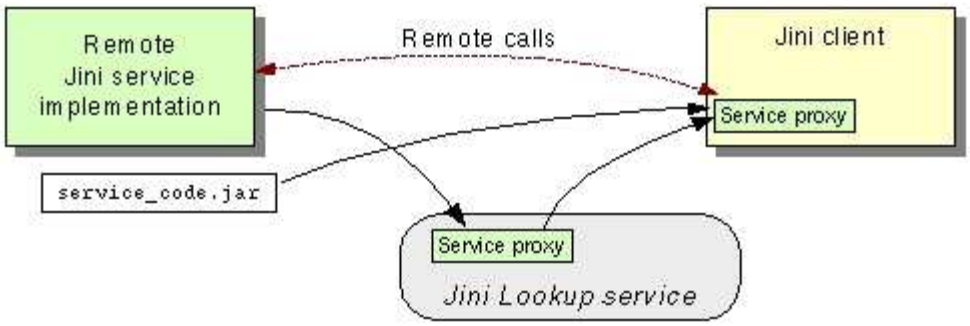


Figure 4: This figure is copied directly from [Sommers] with caption "Code and object mobility in a Jini client and service. " It shows the basic movement of the proxy object and remote procedure calls in a Jini environment.

The first problem in verifying the service proxy via the service itself is that we can only communicate through the un-trusted proxy. The way around this problem is to use a *bootstrap* proxy. A bootstrap proxy is obtained via the un-trusted proxy from the service, and is lightweight proxy that is capable of communicating with the back-end service. The client should be able to trust the bootstrap proxy by examining it to see that all of the code it runs is local and known. For example, it may consist entirely of common classes that the client can verify against a baseline to make sure they have not been tampered with.

Now that the client has a trusted bootstrap proxy, it asks the back-end service for a collection of *trust verifiers* that can validate that the service proxy is what the service proxy expects it to be. Trust verifiers are implemented by the service, and can use any mechanism to insure that the service trusts the proxy. For example, the service may compute a message digest hash on the proxy byte code, and pass that in the trust verifier. The trust verifier would then compute the hash on the service proxy that the client has, and compare. If the hash is the same, it's reasonable to assume it hasn't been tampered with. Now the service trusts the client's service proxy, and returns that result to the client. The following diagram, copied from [Sommers] shows this process in diagrammatic form. We have not yet discussed steps 2 and 5 in this diagram – constraints are discussed in the next section.

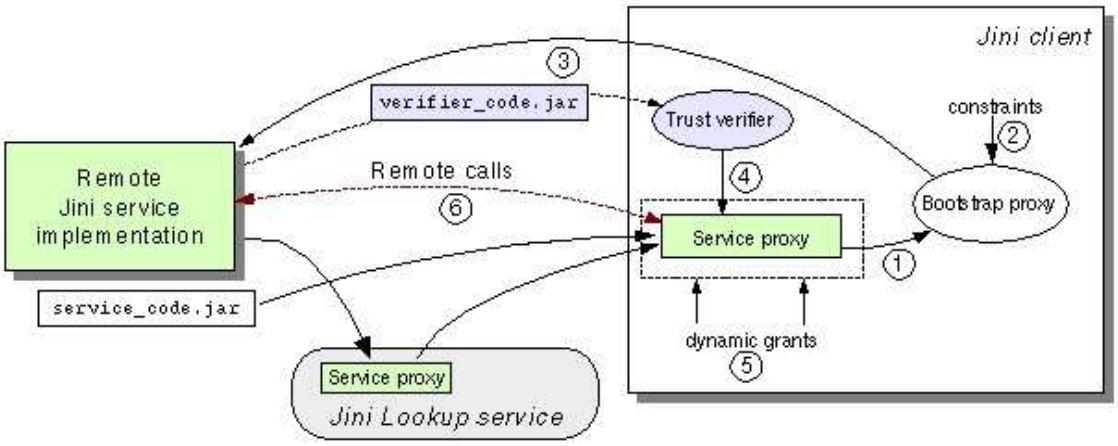


Figure 5: This image copied directly from [Sommers] with caption "Establish proxy trust.". It shows the Jini 2.0 Security Model's algorithm for a client to gain trust in a service proxy.

The following diagram shows the verification step, in this case the trust verifier is using the equivalence method that canonically compares the server's known service proxy with the one the client has.

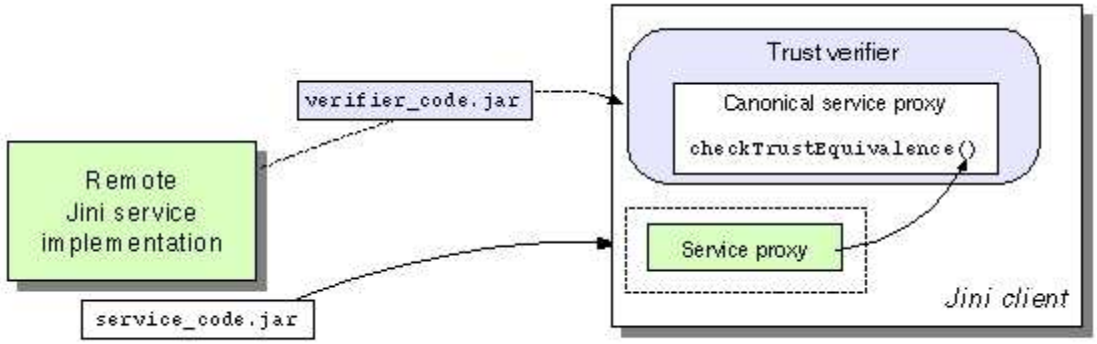


Figure 6: This image copied directly from [Sommers] with caption "Determine two service proxies' trust equivalence.". It shows the service verifying the service proxy using built in methods.

Step 2 – Ensure the Proxy Meets Your Needs

Now that we've learned to trust the service proxy, we need to make sure that the proxy provides any guarantees that we require on the level of service it provides. For example, we may require that the

service proxy give some guarantee that it will encrypt sensitive information before sending to the service. Or, we may require that the service have certain properties, such as that they are from the same company or group. Similarly, the *server* may want the same guarantees about the *client*.

The client and server accomplish this stage of security by using c*onstraints*. There are some system-defined constraints, such as requirements on the principal (that is, characteristics of the client or server's properties, such as being from the same company). Other system-defined constraints involve whether client, server or both require authentication, whether the client requires that data and class integrity is checked, and whether the client will allow the server to delegate for it (that is, the server can authenticate as the client when it communicates with a third service in the system).

Constraints can also be customized, as long as both server and client understand what they are. If either party does not understand what the constraint means, it fails and we cannot continue.

Step 3 – Grant Privileges

Once we have a trusted proxy, and once both server and client have verified that the other party can meet the constraint guarantees they require, the client must grant privileges to the downloaded service proxy. Privileges enable the client to restrict the level of access that the service proxy has. For example, while it is granted network connectivity, it may be denied operating system access. This stage follows the time-tested approach to security that the best policy is to only allow access that is absolutely necessary. As a real life example, most of us have keys to our offices, but very few are given the master key to the building! That doesn't necessarily mean we're un-trusted, it simply makes good sense to limit us to where we *need* to be.

Chapter 4

CRYPTOGRAPHY IN THIS THESIS

This chapter provides a brief overview of the cryptographic concepts used in this thesis. More detail is provided in the Appendices, but for a rigorous understanding of these methods a cryptographic text such as [Stinson] or [Schneier] should be referenced.

Cryptographic Algorithms

Digital Signatures

Digital Signatures provide a mechanism to mathematically verify the identity of other parties in a cryptographic protocol in a cryptographic system. For brevity, we will shorten "parties in a cryptographic protocol" to simply "party". Typically, there is a public key and private key for each party in the system (usually separate from their encryption key) whereby only a message encoded with the private key of a party can be decoded using their public key. When used in conjunction with a certificate service, which is described later in this chapter, digital signatures provide a reliable way to verify party identities. This thesis uses the RSA Digital Signature algorithm. See Appendix A for more information on digital signatures.

Key Agreement

Key Agreement algorithms provide a way in which two parties can securely agree on a secret key. The parties generate private and public components for the key exchange. The parties start with public and private key components, and using these components and a well know exchange algorithm they can arrive at the same encryption key. Key Agreement does not address the problem of trusting that the other party is in fact who they say they are – it is only intended to securely generate a common secret encryption key. This thesis uses the Diffie-Hellman Key Agreement algorithm. See Appendix A for more information on key agreement algorithms.

Certificate Services

We have reviewed the RSA digital signature and the Diffie-Hellman key agreement algorithms, both of which have a common vulnerability. In each case, an intruder can masquerade as another party in the

system, and destroy the system's security. The reason, then, that we require a certificate service is to have a centralized authority where the identity of a sender may be confirmed to prevent an attack on the security of our system. See Appendix A for more information on certificates.

Encryption

The purpose of encryption is well known, namely to alter data with a secret key such that it is unintelligible to any party without the decryption key. This thesis uses the new Advanced Encryption Standard, Rijndael. It's notable that I did not develop the Rijndael encryption code for this thesis. It is done via the Cryptix Foundation's JCE Provider. See chapter A for more information on Rijndael.

Message Digests

A message digest algorithm takes data of variable input length and creates a fixed length hash of the message. It is used with digital signatures in order that the data that must be signed is shorter than the encryption key. This thesis uses the MD5 Message Digest Algorithm and is done via the Cryptix Foundation's JCE Provider.

Java and Cryptography

Java lends itself well to cryptographic applications. Specific to this thesis, the math package, Java SDK and Java Cryptographic Extension provide a solid foundation from which to begin.

The most important aspect of Java in developing cryptographic applications is the BigInteger class in the math package. It allows the representation of very large numbers, and even has the built-in capability to generate probable-prime numbers. This saves a great deal of development time and effort.

This thesis uses Java SDK v1.2.1. This Java version has Digital Signature related classes and interfaces (e.g., SignatureSpi) in the java.security package. The Java Cryptographic Extension (JCE) contains the classes and interfaces needed for encryption (CipherSpi) and Key Agreement (KeyAgreementSpi). In version 1.4.x, Sun moved the JCE into the SDK itself, and it is no longer an optional package.

Java Cryptographic Extension (JCE)

Technology Overview

The Java Cryptographic Extension is an extension to the Java Core standard. It provides a framework and – in some cases – implementation, for cryptographic methods. The JCE defines Service Provider Interfaces (SPI), which define the application interfaces implemented by each engine. These include

SPI's for encryption (ciphers), message authentication code algorithms, key generation and key exchange [Java Cryptography Extension (JCE)].

The JCE is an optional package for J2SE 1.2.x and 1.3.x, and this thesis uses J2SE 1.2.1. In J2SE 1.4.x, Sun moved JCE into the core standard [Java Cryptography Extension (JCE)].

JCE Providers are a signed jar file and contain an implementation of JCE SPI's. They may contain any number and combination of cryptographic algorithm implementations. That is, provider writers may choose which SPI's to implement, and how to implement them, as long as the interface is followed.

The kmlJCE Provider

This provider was written for this thesis, and includes digital signature and key agreement functionality. Specifically:

- Diffie-Hellman Key Agreement Engine
    - KeyAgreement SPI – provides the functionality of the key agreement protocol.
    - AlgorithmParameterGenerator SPI – generates the Diffie-Hellman parameters (prime p and primitive element g).
    - KeyPairGenerator SPI – generates the public and private variables for key exchange (i.e., X and x in the equation $X = g^x \bmod p$).
    - KeyFactory SPI – bi-directional translation of opaque keys to key specifications. Used after transmission of opaque key material to rebuild key specification objects.
- RSA Digital Signatures
    - Signature SPI – provides the entire functionality of digital signatures: parameter generation, content update (i.e., setting the content to be signed), signature and verification.

The sunJCE Provider

During development, the application also loaded the sunJCE provider, which contains a superset of the functionality in the kmlJCE. In this way the application and Jini segments could be developed before the cryptographic portion. The sunJCE provider is no longer used – it has been entirely replaced by the

kmlJCE provider. For more information on the sunJCE provider, see [Java Cryptography Extension (JCE) 1.2.2].

The Cryptix Provider

The Cryptix Foundation Limited authored the Cryptix JCE Provider, which contains a great deal of cryptographic implementation. This thesis uses the Cryptix provider for the following:

- Rijndael Encryption – Cryptix provides the complete Rijndael encryption implementation for this thesis.

- Message Digest (MD5) – Cryptix provides the MD5 hash implementation used in this thesis for RSA digital signatures.

The Cryptix JCE provider is covered under a general use license agreement found on the Cryptix website at [http://www.cryptix.org/LICENSE.TXT]. The Cryptix JCE provider is downloadable under that agreement from [http://www.cryptix.org/].

Chapter 5

THIS THESIS AND JINI SECURITY

Application Level Jini Security

This thesis focuses on adding security in the application layer by creating two Jini cryptographic services. One of these services provides a secure file transfer service, and the other provides a key agreement service to generate a key for the file transfer service. Both services use Java Cryptographic Extension (JCE) providers to apply cryptography to the system.

The Security Algorithm

Stage 1 – Establish an Identity

The first step in adding the application level security in this thesis is to establish an identity for all parties. Having an identity here is defined by having two things: First, we must have a certificate registered with the certificate service. Second, we must have the corresponding private signature key and keep it secret.

The certificate must be registered with the certificate service in order that other parties can verify our identity. Certificates are covered in more detail in appendix A. The important components of it for this discussion are the RSA verifying exponent (public key) of the party, the modulus, the friendly name of the party, and a signed hash of that data. The hash is signed with the master certificate authority's private signature key.

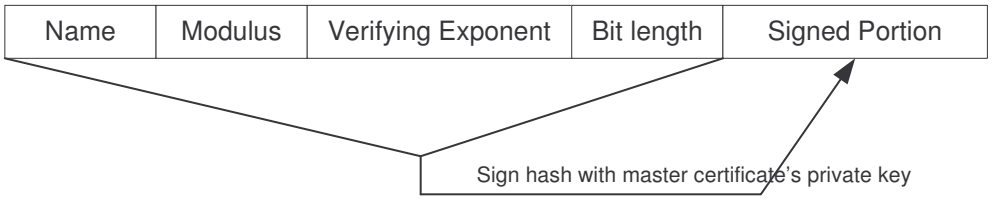


Figure 7: Certificate format used in this thesis

Having the secret signature key allows us to digitally sign a hash of any data. It must be kept a secret, obviously, so that others cannot sign their data and make it appear to have come from us. Having this secret signature key, and the corresponding verifying exponent available in the certificate registered with the certificate service, supplies us with a strong cryptographic identity. Naturally, the strength of this identity is dependent on the trust parties have in the certificate service itself.

Stage 2 – Exchange Keys and Verify the Identify of Your Partner

The second step in the application level security in this thesis is to verify the identity of the remote partner. Assume when we start that we've already received a proxy for the remote service in the usual Jini 1.2 way (that is, without the added security in Jini 2.0 discussed earlier). Note that since we haven't used the new Jini security model, we cannot trust the proxy. As we will see, however, we do not need to establish trust in the proxy[3]. All communication between the services may be open – the cryptographic algorithms used already assume that the network is insecure, so we simply consider the proxy to be part of the insecure network medium.

Our current level of trust in the other parties in this exchange is shown in Figure 8. In this example, we are *Bob* and we want to communicate with *Alice*. Notice that we cannot trust that the remote service that claims to be Alice really is Alice. Notice also that all of the communication links, the proxies, and even the lookup service are considered un-trusted. We trust only the master certificate service's public key[4] and ourselves.

[3] In this case, a corrupted proxy could do one of two things: (1) send the message to the wrong service, or (2) send it to the right service, but attempt to also send some result to a third service. In he first case, if the proxy doesn't deliver the message to the correct service, that service will not be able to sign the response to the key agreement with the correct signature, and we will abort the key agreement. In the second case, even when the proxy does deliver the message to the correct service, it will not be given any information that isn't public anyways, so it couldn't send any damaging information to a third party.

[4] Note the distinction between trusting the certificate service itself, and trusting it's public verifying exponent. We cannot trust everything from the certificate service because the communication link is insecure. Trusting the certificate service really means trusting in its public verifying exponent, so that we can validate the certificates that come from it.

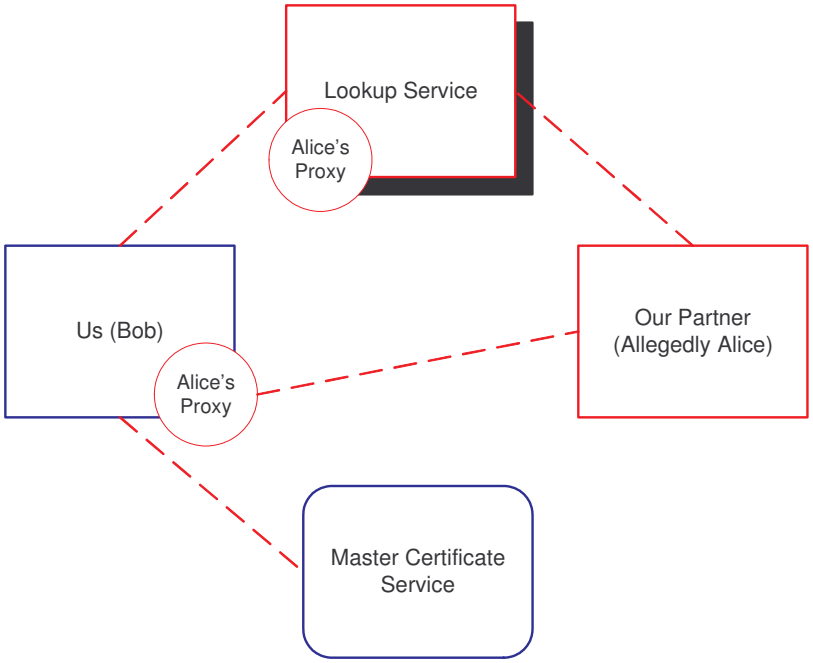


Figure 8: This figure illustrates the trust Bob has in the parties in this system. Currently Bob can only trust himself and the master certificate service. Blue represents trust, and red represents the unknown.

We need to gain trust in our partner before we can exchange an encrypted file with her. In this thesis, identity verification is done during all data exchange, namely during key agreement and file transfer, and identity is verified using RSA Digital Signatures. A hash of whatever data is being sent is signed by the originator, and verified by the destination using the certificate service. Note that in the case of key agreement, this happens twice because Bob must verify Alice's identity and vice-versa.

In order to thwart replay attacks, it is also necessary for both sizes to generate a random nonce during the signature phase. This nonce is sent in plaintext, and is also hashed and signed. Without this nonce, a third party could intercept and then replay the transmission for some nefarious purpose (for example, to confuse either party into thinking that the other is trying to re-establish a key). Both sides maintain a list of all prior nonces and assume a replay attack if a nonce is re-used [Ferguson and Schneier, 270].

The sequence of actions to complete stage 2 is detailed in Figure 9. This sequence includes both the Diffie-Hellman Key Agreement to arrive at the common key, k, and the RSA Digital Signature verification using the public exponents registered with the certificate service.

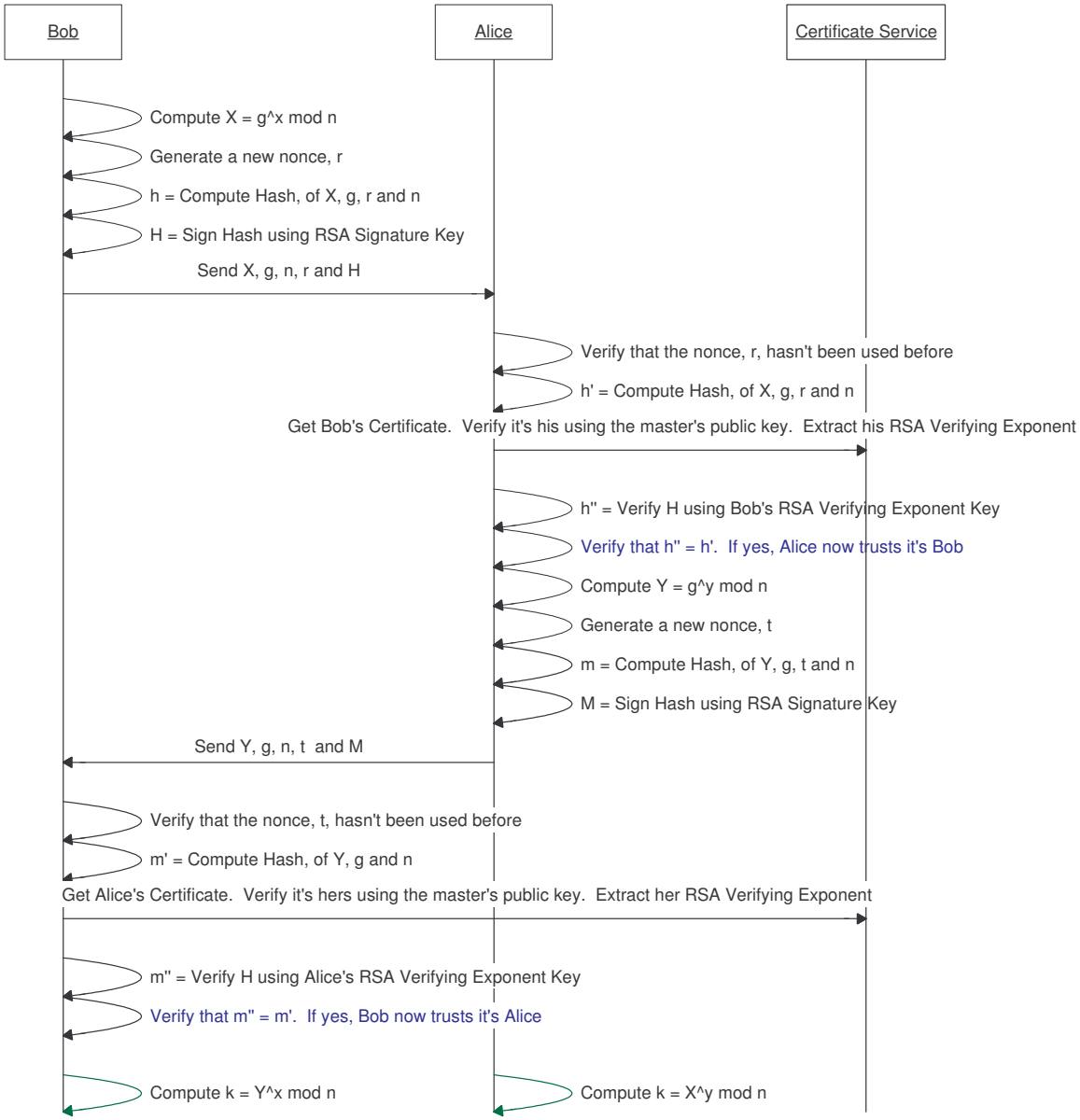


Figure 9: Application Level Security sequence diagram for stage 2, exchanging keys and verifying identity.

After this exchange, Bob and Alice "trust" each other's identity. There is a limitation to this trust in that they will re-verify each other's identity whenever they exchange data. The following diagram illustrates the trust state of the system after stage 2.

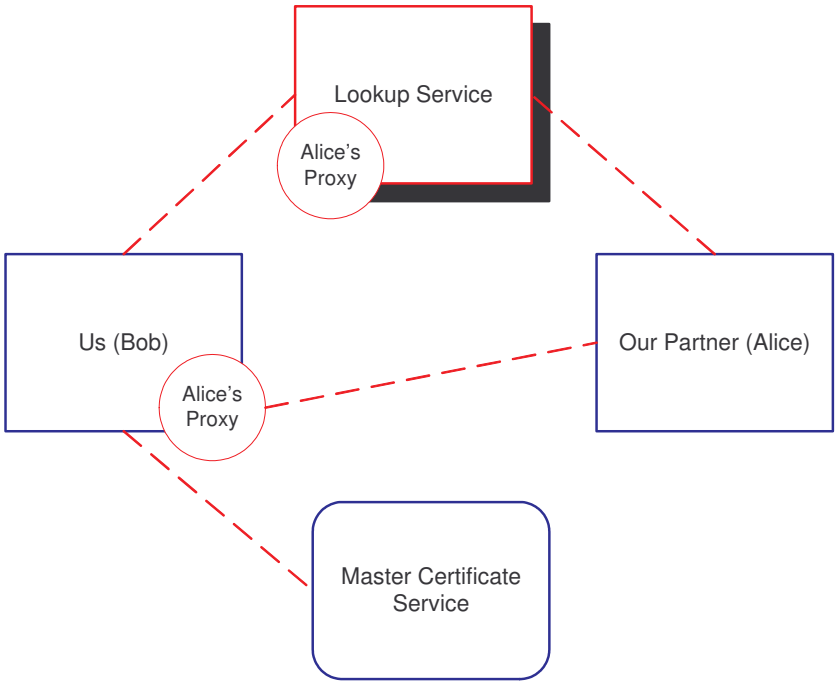


Figure 10: This figure illustrates the trust Bob has in the parties in this system. Currently Bob can only trust himself and the master certificate service. Blue represents trust, and red represents the unknown.

Note that when we say that Bob and Alice trust each other, what we really mean is that they trust that they have successfully and securely agreed upon an encryption key. Because the link between them is still insecure, though, they need to re-verify each other's identity on every communication.

Stage 3 – Transmit a Signed and Encrypted File

The third and final stage of application level security illustrated in this thesis is to sign and then encrypt a file, and send it to the remote partner. Recall that in stage 2, a common encryption key, k, was established using Diffie-Hellman Key Agreement. Figure 11 illustrates the steps performed during stage 3:

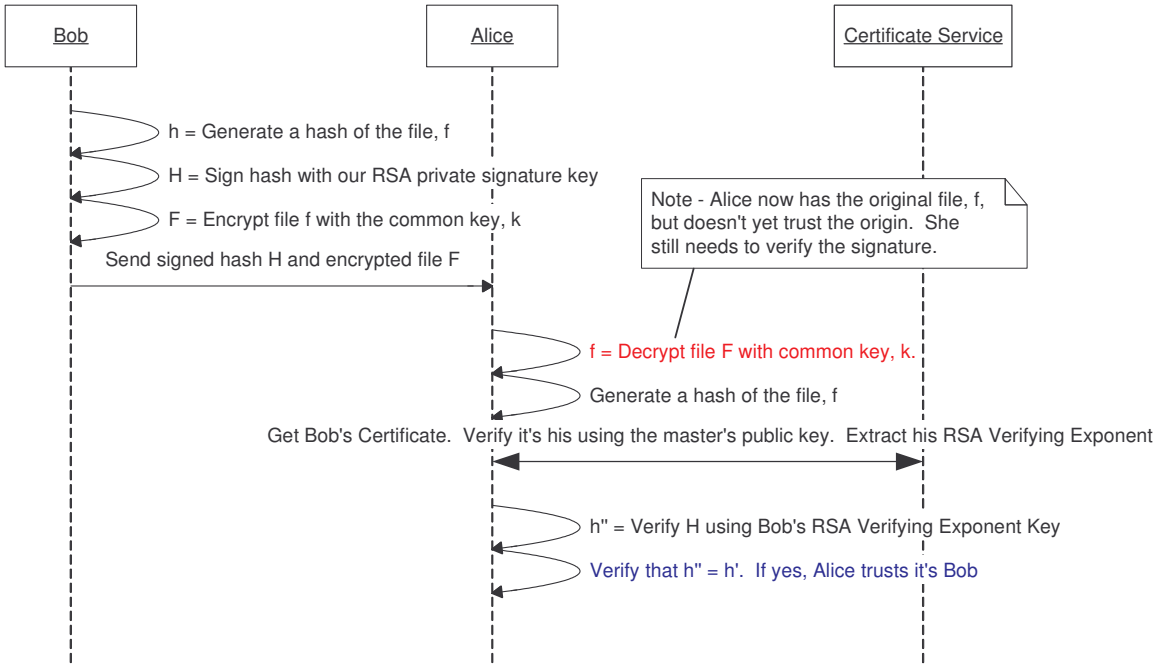


Figure 11: Application Level Security sequence diagram for stage 3, encryption and verifying identity.

Note that a nonce is not required in the file transfer phase. This is because the private key, *k*, is only used once and then discarded. Therefore a replay attack is not possible, and a nonce list is unnecessary.

The important thing to note during the encryption and transmission phase is that the receiver needs to re-validate the identity of the sender using the signed message hash. It's notable that the sender does *not* need to worry about identity, because he was sure of the identity of his partner when the key was agreed upon. Therefore even if he sends the file to the wrong party, they still do not have the decryption key to obtain the original file.

The Security Algorithm Implementation

There are three primary components to the software developed to implement the security algorithm. The first is a JCE *Cryptographic Service Provider* (provider) that implements the Diffie-Hellman Key Agreement Algorithm, and the RSA Digital Signature Algorithm. The Advanced Encryption Standard Algorithm (Rijndael) and MD5 message digest implementation is by the Cryptix JCE provider. The provider has two "engines", each engine implementing one of the aforementioned algorithms. The second component of the thesis is the development of two JINI services, a JINI Key Agreement Service and a JINI Secure File Transfer Service. This constitutes the original research portion of this

thesis. To the best of the knowledge of those involved, this had not previously been attempted prior to the start of this thesis. The third is a CLI Application called "Cryptographic Service" that uses the services to transfer a file.

The JCE Provider

A summary of the JCE provider development and use for this thesis is provided in Chapter 4.

The Jini Services – JKAS and JSFTS

There are two Jini Services in this thesis that provide the services described in the security algorithm: the Jini Key Agreement Service (JKAS) and the Jini Secure File Transfer Service (JSFTS). Both are implemented in the package jkml.CryptoService, and their design are described in Appendix B. The general design is discussed in this section.

Each part that is willing to be the source and destination of key secure file transfers will register an instance of both JKAS and JSFTS. The services are designed to be used together – the output from the Key Agreement Service negotiation is a private key that can be used for encryption in the JSFTS.

The registration includes service attributes that tell which user the services are associated with. When user *Bob* registers his JKAS and JSFTS, he includes a service attribute with the user name "Bob". When user *Alice* decides to send a secure message to Bob, she will search her cache of proxies from the JINI Lookup Service (JLUS) for an instance of JKAS and JSFTS associated with "Bob". If she has instances cached, then she is ready to begin using the services. The JKAS and JSFTS services are already connected via RMI to Bob's backend JKAS and JSFTS services. Alice initializes Bob's JKAS proxy with a reference to her JKAS backend service, and calls a method in Bob's JKAS service proxy to start the key agreement session. When the session has completed, Bob and Alice's backend JKAS services store the secret key. Now that keys have been exchanged, Alice can follow the same procedure to send Bob a secure file using the JSFTS service and the key generated during the Key Agreement using the JKAS service.

The Application – Cryptographic Service

There main application provided for this thesis is the Crypto Service CLI. The CLI is demonstrated in Appendix C.

The Application Segment and the JCE Providers

When starting, the CLI registers two providers, the kmlJCE provider written for this thesis, and the Cryptix provider written found at http://www.cryptix.org/.

The Application Segment and the Jini Services

The CLI creates and registers the two Jini Services at startup. When registering, the CLI gives the name of the party (e.g., "Bob") and a proxy for communicating with it. The services are registered in the normal Jini way, and the CLI receives events describing other parties that have the same services registered.

Using the CLI

The CLI menu contains several menu options, though only 3 are really required. First, the "Show Users" option allows the user to list other parties in the community, along with whether they have registered both required services. If so, then they are a valid user for the second option. The second required menu option, "Transfer File with User", allows the user to actually transmit a file to one of the parties given by the Show Users option. It asks only for the user name, and for a file name. Finally, the third required option is "Quit"!

The other menu options, which can be viewed in Appendix D, are for manual transfer of a file. They are used for debugging and demonstration purposes only.

Bringing It All Together

This section ties together certificate generation, the security algorithm, JCE and Jini Services.

Certificate Generation

This thesis uses a pseudo-certificate service to verify the public key for parties in the system. It is called a "pseudo"-certificate service because it is not fully implemented. The master certificate key is hard-coded into the application, which is obviously not something that would be acceptable in practice. This shortcut was taken to simplify the implementation.

The Certificate Generation Utility was implemented for this thesis as a manual key authority. It generates a master certificate key using RSA, and uses that to create certificates for parties. The certificate generated is described in the Figure 12:

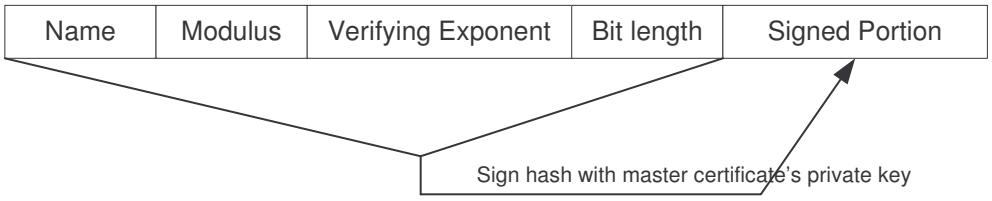


Figure 12: Certificate format generated by the Certificate Generation Utility and used by the Cryptographic Service application.

- Name – The name of the party. The party gives this parameter when the utility is run (e.g., Bob, Alice, etc.).

- Modulus – This is the $n=pq$ used in RSA digital signature, as generated by the utility.

- Verifying exponent – This is the public RSA key for the party given by Name, as generated by the utility.

- Bit Length – The length of the modulus.

- Signed Portion – This is a hash of the concatenation of Name, Modulus, Verifying Exponent, and Bit Length. It is signed by the *Certificate Authority's private key*.

The master certificate is hard-coded into the application. This contains the public verifying exponent of the master key, so that applications can use it to verify the signed portion of a party certificate.

Component Interactions During Key Agreement Phase

This section will detail how the main components in the system interact during the key agreement phase. We assume at the beginning of this scenario that both Bob and Alice have received each other's JKAS and JSFTS proxies from the JLUS.

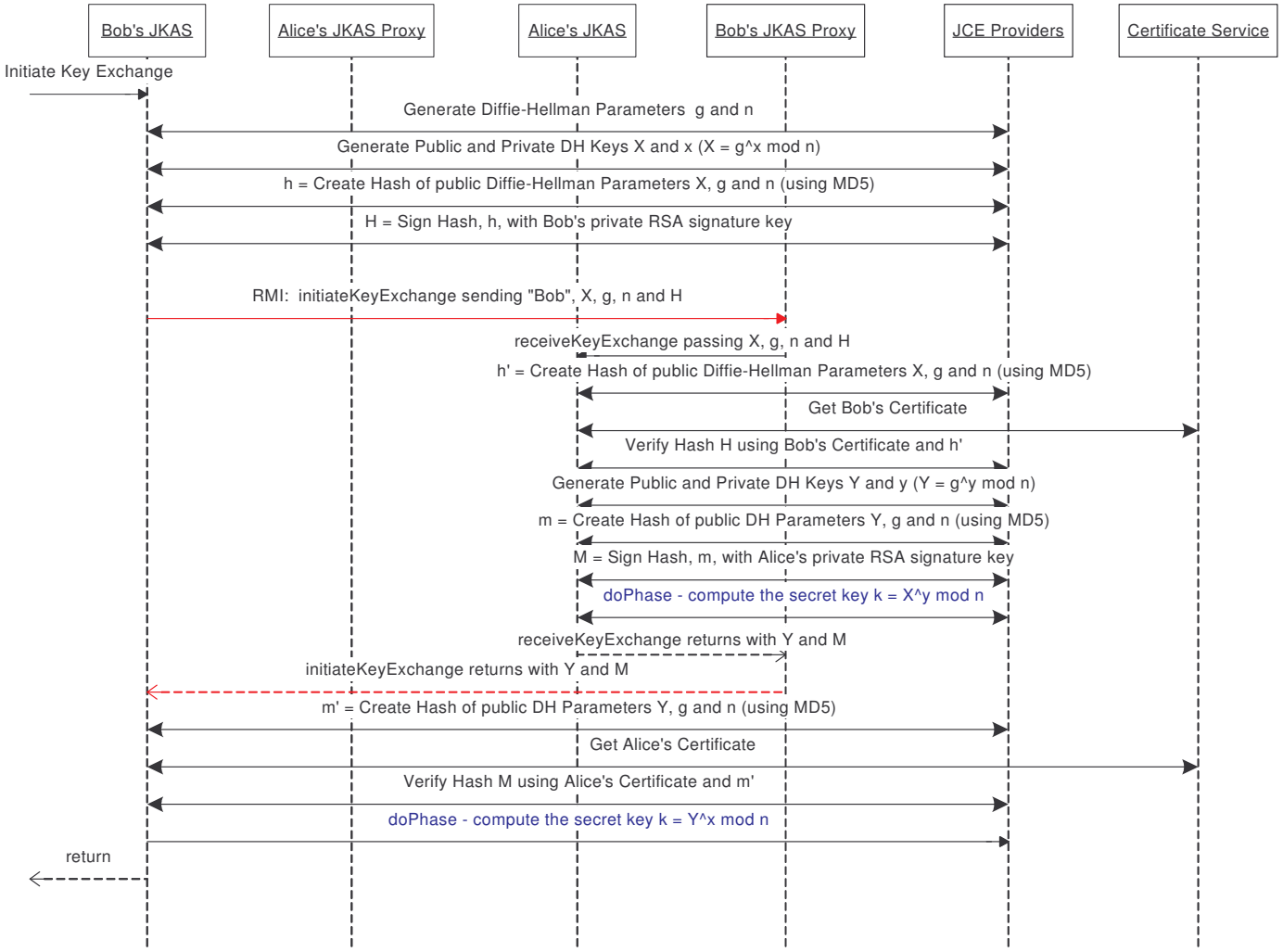


Figure 13: Sequence diagram showing the detailed interactions between the Jini Services, proxies, JCE providers and the pseudo-certificate service during the key agreement phase. Lines in red indicate remote procedure calls.

Component Interactions During the Encryption Phase

This section will detail how the main components in the system interact during the file encryption phase. We assume that two parties have already agreed upon a private key, *k*.

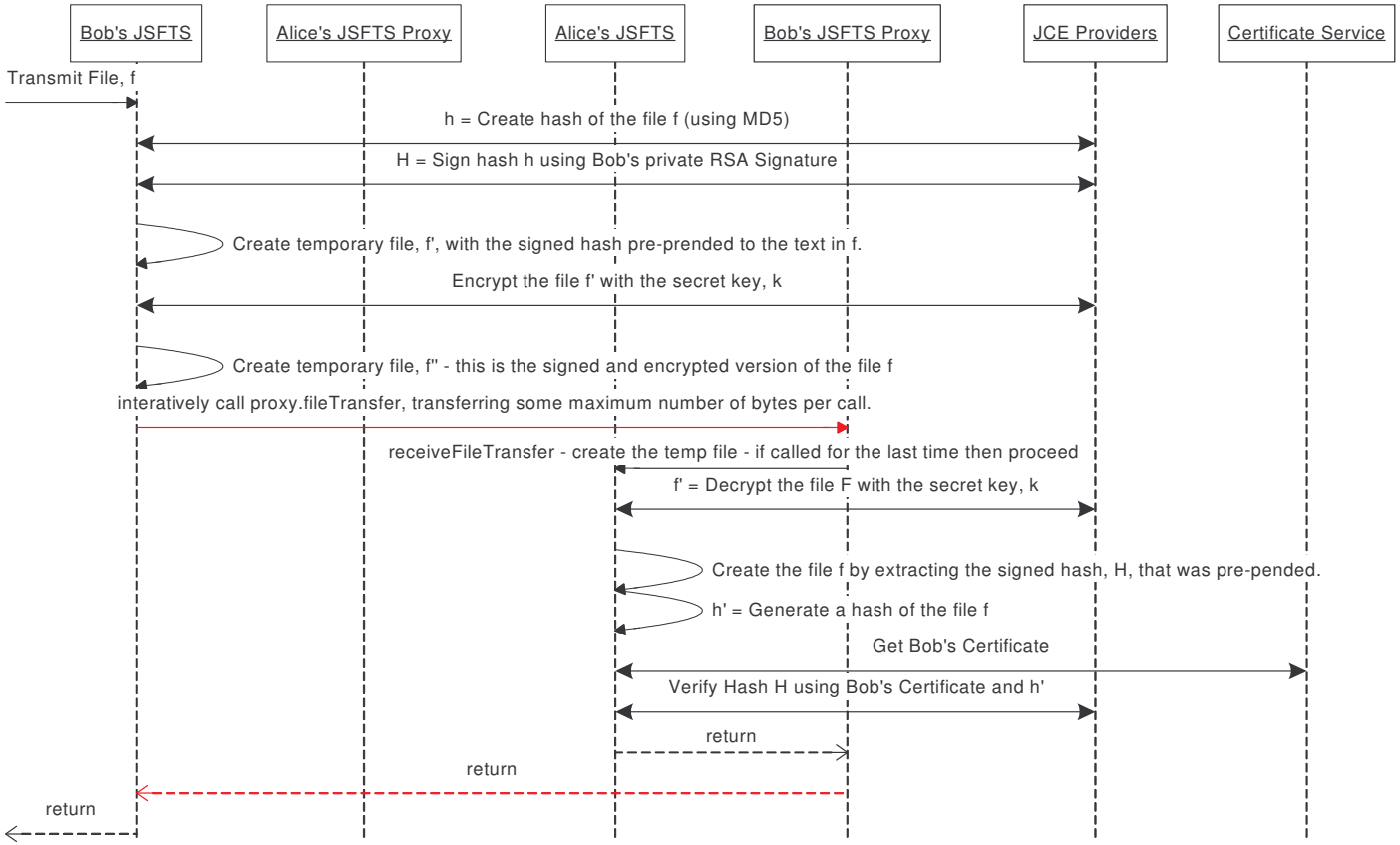


Figure 14: Sequence diagram showing the detailed interactions between the Jini Services, proxies, JCE providers and the pseudo-certificate service during the file transfer phase. Lines in red indicate remote procedure calls.

Chapter 6

JINI SECURITY COMPARISON

Jini 2.0 Security Versus My Jini Security Algorithm Security

This chapter compares and contrasts the approaches to Jini security discussed in this thesis. Each method has strengths and weaknesses, and it will be shown that combining the two gives a more complete solution than either provides independently.

Comparisons

Leaps of Faith

Both Jini Security and my security approach must *assume* trust in one or more remote services. Jini 2.0 Security assumes that all remote services are trustworthy, but that their proxies are not. My application security approach assumes that all services except for the certificate service are untrustworthy. Each approach has pros and cons.

Consider first an example of where the trust assumption in Jini 2.0 Security may be exploited, namely an intruder-in-the-middle attack. Consider a system where the lookup service has been compromised, as in the Figure 15:

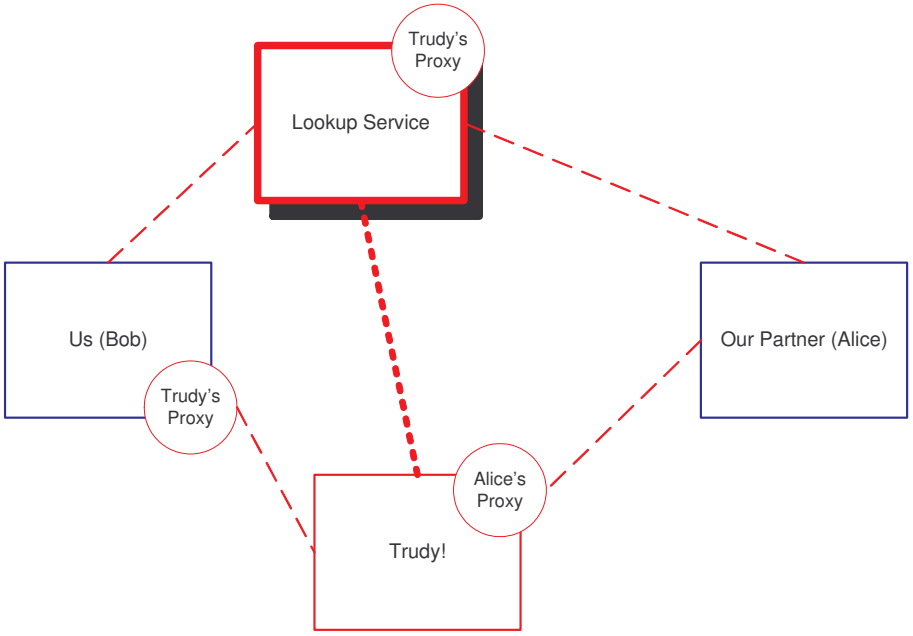


Figure 15: Intruder-in-the-middle attack with Jini 2.0 Security Infrastructure. Trudy controls the Lookup Service. She has replaced Alice's proxy with her own, but it will pretend to be Alice's proxy. I.e., the compromised lookup service will return Trudy's proxy when asked for Alice's proxy.

Because Bob trusts that the lookup service will return the correct proxy, he assumes that he'll actually get a bootstrap proxy from Alice. Instead, he'll get a bootstrap proxy from Trudy, who will send a verifier that confirms (falsely) that Bob has the correct proxy. Notice that by compromising the lookup service, Trudy may intrude on Bob's communications with all other services in the system! This is a worst-case scenario and some steps can be taken to guard against it, such as having multiple lookup services that are monitored for intrusion. Bob could even run his own JLUS within his own process. Even if one is compromised, the others may not be, and only *some* of the remote services in the system may be vulnerable.

Now consider the application level security added in this thesis. The attack used against Jini 2.0 Security would not work, because Bob wouldn't trust the lookup service in the first place. He would have expected the remote service to authenticate itself, and Trudy would have been unable to do so having only compromised the lookup service. However, Trudy can still wreak havoc in the system by compromising the certificate service! In fact, if the certificate service is compromised in this thesis' application level security the result is catastrophic – the entire system is compromised! There are steps that can be taken to guard against this, such as frequent master key updates and certificate changes

(though this only makes it harder to compromise the certificate service, it doesn't take care of the case where it has already been compromised unless new certificates are generated and delivered without using the old certificate key).

Implementation Effort

An important advantage to adding security at an infrastructure level is in having the bulk of the effort applied at a common layer. It is much easier for applications to use something built into the core framework than to have to either develop it from scratch or hook into application level services to do the work. Ease of use both decreases development time and increases reliability, since each individual developer does not need to write most of the security code. Finally, it's much easier to upgrade if problems are found in the security code, or if the security is upgraded. This assumes that good interfaces are developed where this type of change is simply handled in the infrastructure code. If done, however, it means that applications can rely on security experts to monitor the core security code and know that when they upgrade to newer versions of Jini, they are receiving security updates "for free". The Jini community was even careful in their design to provide an easy upgrade path for old Jini services to include security.

Putting Jini security in the infrastructure level also means it is much more likely than not to be included in Jini books and tutorials going forward. This will lead to an increased level of understanding of security problems and the Jini solution, and will mean that many developers will take the necessary steps to use the security framework. By making it easy to incorporate, the standard will be much more likely to become ubiquitous in Jini applications.

Conversely, even generic application level security will be harder to use than infrastructure level security. Assume that the services created for this thesis are entirely generic and pluggable[5]. Old applications would still need to modify code to plug into the security, and new applications would need to design with the services in mind. For example, every time data are sent over an insecure channel, the application would first need to make explicit calls to encrypt it. On the other side, the application would need to explicitly decrypt the data.

[5] Though creating generic services were a design goal (i.e., not a requirement), these services are really not easily pluggable.

Public Algorithm Review

Bob Scheifler developed the Java 2.0 security algorithm with input from the Jini Community. Having a public discourse concerning the security algorithm likely increases its security, simply because there are more people with a greater breadth and depth of experience considering and discussing it. There is no reason that an application service developer could not open his security algorithm to public review, but it would be – for many reasons – more difficult to have large numbers of people scrutinize it. Not the least of these reasons would be the implementation effort required for these people to use the security.

Data Security

Jini 2.0 security provides a mechanism for performing authenticated remote method calls. It does not, however, automatically encrypt the data that is being transmitted. Therefore even when the service and proxy really are trustworthy, there is still the opportunity for an intruder compromise the data when it is passing through the network. It's notable that while encryption isn't built into Jini 2.0 security, the framework for specifying that it should be used is included (via constraints). There's no reason, then, that it couldn't be added at the service application layer as was done in this thesis. Client applications would then add a constraint that encryption is required and have the advantages of encryption without additional effort[6].

My security algorithm assures that all data passing through the network is secure. That is, all data that is passing through the network is either encrypted, or public data in the key agreement phase.

Conclusion

Neither Jini 2.0 security nor the security algorithm developed here comprises a complete solution. By combining features of both, however, we could come closer to a general and complete solution for Jini security.

[6] A drawback of the Jini 2.0 implementation is that while the client can specify that encryption is required, it cannot specify the level of encryption required. A service application could claim encryption between the service and proxy, while only implementing something as insecure as the Caesar Cipher.

Appendix A

CRYPTOGRAPHIC BACKGROUND

A number of cryptographic techniques are used in this thesis. This chapter provides an overview of these methods.

Digital Signatures

Overview

Digital Signatures provide a mechanism to mathematically verify the identity of other parties in a cryptographic protocol in a cryptographic system. For brevity, we will shorten "parties in a cryptographic protocol" to simply "party". Typically, there is a public key and private key for each party in the system (usually separate from their encryption key) whereby only a message encoded with the private key of a party can be decoded using their public key. When used in conjunction with a certificate service, which is described later in this chapter, digital signatures provide a reliable way to verify party identities.

RSA Digital Signatures

Ron Rivest, Adi Shamir, and Leanard Adleman invented the most famous Digital Signature method, RSA. The RSA algorithm is very easy to understand, and implement [Schneier, 466].

Mathematical Basis – The Factoring Problem

Most cryptographic methods are based upon difficult to solve mathematical problems. Decrypting an RSA encryption without the appropriate key is thought to be equivalent to solving a very difficult factoring problem. An RSA key comprises two very large prime numbers multiplied together, and therefore has two very difficult to find factors.

The RSA algorithm was invented in 1977 [Boneh, 1], and since then it has been heavily analyzed for weaknesses. Common attacks can be found in [Stinson, 140], [Boneh] and [Killeen].

Parameter Generation

There are three parameters in RSA that must be carefully chosen. First, two distinct large prime numbers, call them p and q. The most secure system will have two prime numbers of equal length. [Schneier, 467]. From p and q, we compute the product n. That is, $n = pq$.

Next, we need to randomly choose an encryption key, e. This key should also be large, but must be within the bounds $1 < e < n\text{-}1$. Additionally, e and $(p\text{-}1)(q\text{-}1)$ must be relatively prime [Schneier, 467]. This insures that e will have an inverse in $Z_{(p-1)(q-1)}$, a necessity in computing the decryption key [Stinson, 116].

Computing the Decryption Key

We know that our encryption key has an inverse in $Z_{(p-1)(q-1)}$, and we call that inverse d. This can also be written as: $ed = 1 \bmod (p\text{-}1)(q\text{-}1) \rightarrow d = e^{-1} \bmod (p\text{-}1)(q\text{-}1)$ [Schneier, 467]. Computing the inverse can be accomplished using the Extended Euclidean Algorithm, which is described in detail in [Stinson, 117-118], with an algorithm given on [Stinson, 119].

Encrypting and Decrypting a Message

Encrypting a message using RSA is mathematically very simple. Assume we have Alice's private message m, Bob's public product of primes n, Bob's public encryption key e, and Bob's private decryption key d.

Alice divides m into pieces of size smaller than n. That is, $m = m_0 + m_1 + \ldots + m_w$, where the plus sign indicates concatenation, and $\forall i, m_i < n$.

Now, for each m_i, Alice compute the cipher $c_i = m_i^e \bmod n$.

Alice sends each cipher, c_i, to Bob.

Bob uses his private decryption key, d, to find $m_i = c_i^d \bmod n$.

Bob concatenates all m_i's together to find the original message, m.

[Schneier, 467]

The mathematics is quite elegant. The only question remaining is how one efficiently computes $m_i^e \bmod n$, and $c_i^d \bmod n$. Assuming a binary representation, one would commonly use the *Square and*

Multiply method of modular exponentiation. This is described in detail in [Stinson, 127-128]. The important property of the Square and Multiply algorithm is its performance. Given bit length l, the Square and Multiple algorithm can compute a modular exponentiation in $O(l^3)$ time, which is polynomial time and thus tractable [Stinson, 128].

RSA and Digital Signatures

Using RSA to sign and verify messages is very straightforward. Recall that if a message originator wishes to *encrypt* a message for a particular target, he must use the target's *public* key. The target then may use her *private* key to *decrypt* the message. Digital Signatures are accomplished by doing the opposite. That is, the originator first *signs* the message using his *private* key. Now anyone can *verify* the signature using the originator's *public* key.

It's notable that the message can be encrypted *after* it has been signed, in the same way that is described in the previous section. It's important that the signature be done first, or an intruder could simply use the signer's public key to verify the message, and then re-sign it using their private signature key, thus breaking the security of the system.

Functional notation lends itself well to understanding how RSA can be used for digital signatures and encryption. Let the function **S(m)** be the process of applying a party's **private** key to a message, m. Further let the function **V(m)** be the process of applying a party's **public** key to a message, m. We will denote parties with subscripts on the function. That is, $\mathbf{S_{alice}(m)}$ shall denote the application of Alice's private key to message m.

The following example demonstrates Bob sending a signed message to Alice, and Alice verifying the signature, using the described functional notation.

2. Bob signs the message using his private key. Because we are using this for signatures, we call the private key the **signing exponent**. $\mathbf{S_{bob}(m) = m'}$. He sends the signed message to Alice.

3. Alice receives a message, allegedly from Bob. She retrieves his public RSA key. Since we are using this for signatures, Bob's private key in this instance is called the **verifying exponent**. Alice applies Bob's public key to the signed message, m'. $\mathbf{V_{bob}(m')=m}$. We know from the previous section that this operation will retrieve the original message. That is, $\mathbf{V_{bob}(S_{bob}(m)) = m}$.

Now we will examine a signed and encrypted message using RSA.

1. Bob signs the message using his private key. Because we are using this for signatures, we call the private key the **signing exponent**. $\mathbf{D_{bob}(m) = m'}$.

2. Bob now encrypts the message using Alice's public key. $\mathbf{E_{alice}(m') = m''}$. Or, equivalently, $\mathbf{E_{alice}(D_{bob}(m)) = m''}$. Bob sends the signed and encrypted message to Alice.

3. Alice receives a message, allegedly from Bob. She first needs to decrypt the message. Alice applies her private key to get the signed message. $\mathbf{D_{alice}(m'') = m'}$.

4. Alice now needs to verify the message is from Bob. She applies his public key to verify the signature. $\mathbf{E_{bob}(m') = m}$.

The entire operation is described by the following equality:

$$m = E_{bob}(D_{alice}(E_{alice}(D_{bob}(m))))$$

Key Agreement

Diffie-Hellman Key Agreement

Mathematical Basis – The Discrete Logarithm Problem

The Diffie-Hellman Key Agreement protocol is based upon the difficulty of solving the Discrete Logarithm Problem in a finite field. [Schneier, 513].

During the key exchange phase, one party uses a large prime p, a primitive root mod p, g, and a randomly generated large number x to compute

$$X = g^x \pmod{p}.$$

This x is very easy to compute, and is called exponentiation in a finite field. The corresponding logarithm problem in a finite field would be having X, g, and p, and computing x.

Conceptually, imagine a very large numbers x, and prime p. Recall that we also constructed g such that it is primitive mod p, meaning that there is only one combination of g to the power of another number that will yield X modulo p. There is no known general algorithm for computing this in a reasonable time for $p > 2^{150}$ [Stinson, 162].

Parameter Generation

The Diffie-Hellman Key Agreement method requires only two parameters. First, we need a very large prime number, p. The security of the system is improved if we choose a p such that $(p-1)/2$ is also prime [Schneier, 514]. Next, we select the smallest parameter g, such that g is a primitive root modulo p. The actual size of g does not matter, but there's no good reason not to choose the smallest g, and that is computationally easier to deal with [Schneier, 514]. These two numbers provide the basis for the key pair generation phase of Diffie-Hellman Key Agreement.

In practice, we generate a large probable-prime number, q, and then compute $2q + 1$. If $2q + 1$ is *also* a probable-prime, then we say that $2q + 1 = p$. If $2q + 1$ is not a probable-prime, then we re-generate q and try again. This method of constructing p is important in being able to find the primitive root, g. Finding a primitive root means testing whether a number is a generator mod p. Since we know the factors of $p-1$ this is easy to do. We simply test whether the following two statements are true:

$$g^{(p-1)/q} \bmod p \neq 1, \text{ and}$$

$$g^{(p-1)/2} \bmod p \neq 1$$

We are testing with q and 2 because these are the factors of $p-1$. If neither is congruent to 1, then we've found a primitive root. Since the size of g doesn't matter, we simply start at 1 and start counting up until a number passes the above test, and that number is called g.

Key Pair Generation

The Diffie-Hellman Key Agreement algorithm may have multiple parties. For simplicity, consider the two-party case. The only additional parameter needed for each party is a random integer x and y, such that x and y are unique random integers between 1 and $p-2$ [Stinson, 264]. Uniqueness is guaranteed by the size of p. Mathematically, it is extremely unlikely that two parties could generate the same random number. Programmatically, a strong random number generation method is an important consideration.

Key Exchange

We now have p, g, and random exponents x and y. Note that while p and g are publicly known, the random exponents x and y are kept secret by their owners. To illustrate the key exchange sequence, we'll give the owners of x and y the names Bob and Alice, respectively.

1. Alice uses her random exponent, x, to generate $X = g^x \bmod p$. Alice sends X to Bob over a public channel.

2. Bob uses his random exponent, y, to generate $Y = g^y \bmod p$. Bob sends Y to Alice over a public channel.

3. Alice computes the secret key $k = Y^x \bmod p$.

4. Bob computes the secret key $k' = X^y \bmod p$.

Now we will show that , $k = k'$.

$$k = Y^x \bmod p = \boldsymbol{g^{yx} \bmod p}, \text{ by the construction of } Y.$$

$$k' = X^y \bmod p = \boldsymbol{g^{xy} \bmod p}, \text{ by the construction of X.}$$

By the commutativity of multiplication, $k = k'$, and we've shown that both Alice and Bob have the same private keys. [Schneier, 513-515] [Stinson, 270-272].

Key Exchange with Multiple Parties

It is trivial to expand the use of Diffie-Hellman to multiple parties. The standard method is to define a sequence of delivery between the parties. For each additional person added, we need also add another round of computation. To illustrate, consider a three party system, where we have Alice, Bob, and Chris, who have secret random exponents x, y and z, respectively. In a multi-party situation, each party always sends their information to the next party, but receives information from a different party. In the example with Alice, Bob, and Chris, Alice would send her $X=g^x \bmod p$ to Bob, who would in turn send his $Y=g^y \bmod p$ to Chris. Chris would then send his $Z=g^z \bmod p$ back to Alice. Note that each party does not yet have enough information to compute the same private key, and recall that earlier it was mentioned that for each additional party we require an additional round of computation. In this case, with three parties, we need only one more round. Currently:

1. Alice has X, x, and Z.

2. Bob has Y, y, and X.

3. Chris has Z, z, and Y.

So, for the second round,

4. Alice computes $Z' = Z^x \bmod p$.

5. Bob computes $X' = X^y \bmod p$.

6. Chris computes $Y' = Y^z \bmod p$.

Now each party has used a number generated with all other party's secret key. So:

7. Alice computes $k = Y'^x \bmod p = Y^{zx} \bmod p = \boldsymbol{g^{yzx} \bmod p}$.

8. Bob computes $k = Z'^y \bmod p = Z^{xy} \bmod p = \boldsymbol{g^{zxy} \bmod p}$.

9. Chris computes $k = X'^z \bmod p = X^{yz} \bmod p = \boldsymbol{g^{xyz} \bmod p}$.

[Schneier, 514]. It is trivial to carry the example forward into more than three parties.

Certificate Services

Overview – Why do we need a Certificate Service?

We have reviewed both a Digital Signature and Key Exchange algorithm, both of which have a common vulnerability. In each case, an intruder can masquerade as another party in the system, and destroy the system's security. Details on how this is accomplished are included later in this section. The reason, then, that we require a certificate service is to have a centralized authority where the identity of a sender may be confirmed to prevent an attack on the security of our system.

RSA and the Intruder-in-the-Middle Attack

The RSA Digital Signature mechanism provides a way to verify the authenticity of a message. However, if used alone without a centralized authority to verify the source of the verifying key, it is vulnerable to the intruder-in-the-middle attack. The way to accomplish this is very simple. If the intruder is able to convince a party in the system that their verifying key is really someone else's, they've compromised the system. The only ways, then, to prevent this from happening are to exchange the verifying keys over a secure medium (such as delivery by a courier), or to have a centralized trusted authority where the key is stored.

Diffie-Hellman and the Intruder-in-the-Middle Attack

The Diffie-Hellman Key Exchange algorithm is vulnerable to the infamous intruder-in-the-middle attack. This is characterized by having a third party with the ability to intercept the public communications between the parties attempting to establish secret keys. This is slightly different than the multi-party case, in that the three parties will not share the same private key, *k*. Instead, there will be two private keys generated, and while intruder will know both of them, the victims will actually only exchange a key with the intruder. The method:

1. Alice computes $X = g^x \bmod p$. She thinks she sends it to Bob, but Trudy the Intruder intercepts it and prevents it from getting to Bob.

2. Trudy computes $Y = g^y \bmod p$. She pretends to be Bob, and returns it to Alice. She then follows the standard procedure of establishing the private key, *k*.

3. Trudy now generates another private random exponent, w. She computes $W = g^w \bmod p$, and sends it to Bob.

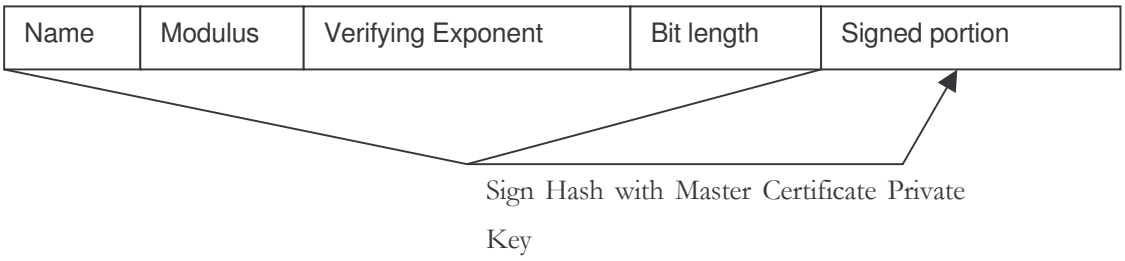
4. Bob, believing the message from Trudy to have really come from Alice, uses his private key y to compute $Y = g^y \bmod p$, and sends it back to Trudy. Believing to have sent it to Alice, Bob and Trudy establish a second private key, *m*.

Now any message sent from Alice to Bob is intercepted and understood by Trudy, who can then substitute or forward to the original message, and so forth. The system is compromised. The intruder-in-the-middle attack illustrates the need for a Certificate Service, so that a party can authenticate that they are communicating with the person whom they wish, instead of someone masquerading as that person.

An Example Certificate Service

Certificate Services may be implemented in many ways, but the overall service that they provide is the same. This example will serve to illustrate this service.

First, a centralized, trusted authority is required. This trusted authority has a master certificate that is known and trusted to all parties in a system. A *certificate* must have enough information to verify a message hash from that party. In the case of RSA, a certificate may look like the following:



The Signed portion of the Certificate are the components Name, Modulus, Verifying Exponent, and Bit Length signed using the *certificate authority's key*. (Recall from the RSA Digital Signature section in Chapter 1 that verifying exponent refers to the party's public RSA key.) The certificate gives another party the means to verify that the parameters that they are receiving are genuine.

Advanced Encryption Standard (Rijndael)

Rijndael is a block cipher designed by Joan Daemen and Vincent Rijmen. It was proposed and has been accepted that the Rijndael block cipher be the new Advanced Encryption Standard (AES)[7].

The Rijndael cipher uses five types of transformations. These transformations are applied in rounds, and the number of rounds required (excluding an extra round at the end of encipherment) varies from nine to thirteen according to key length[8].

The Transformations

The Round Key Addition (AddRoundKey)

This transformation is a bitwise XOR of the current state of the data being transformed. The round key is the same size as the block length. A bitwise XOR is its own inverse, so to reverse this transformation one simply reapplies it.

It will be shown in the next section that the AddRoundKey transformation will be applied one more time than the number of rounds. So, for a nine round cipher (that is, both key and block length are

[7] Rijndael and AES differ only in the range of supported values for the block length and cipher key length. In Rijndael block length can be independently specified to any multiple of 32 bits with a minimum of 128 bits and maximum of 256 bits, and it also supports 160 bits and 224 bits. In AES the block length is fixed at 128 bits, and the key length can be 128, 192 or 256 bits.

[8] This thesis uses 128-bit key lengths and block sizes, which uses 9 rounds excluding an extra round at the end of encipherment [http://home.ecn.ab.ca/~jsavard/crypto/co040401.htm].

128-bits) we will need to apply this transformation ten times. Therefore we need to have 128*10 = 1280 bits of round key material. This key material is obtained by expanding the cipher key using a key expansion algorithm. The key expansion algorithm for Rijndael can be found in [Daemen and Rijmen, 14]. References for this section were [Daemen and Rijmen, 13-16] and [Savard].

The ByteSub Transformation

The ByteSub transformation replaces each byte with a substitute byte. The mapping from bytes to their substitutes is defined in a substitution table (a.k.a., S-box). The S-box is an invertible matrix, so that the transformation can be reversed during decipherment. Details can be found [Daemen and Rijmen, 11] and [Savard].

The ShiftRow Transformation

The ShiftRow transformation cyclically shifts the rows in a matrix over different offsets [Daemen and Rijmen, 11]. The following illustrates the bit shifting with 128-bit block length.

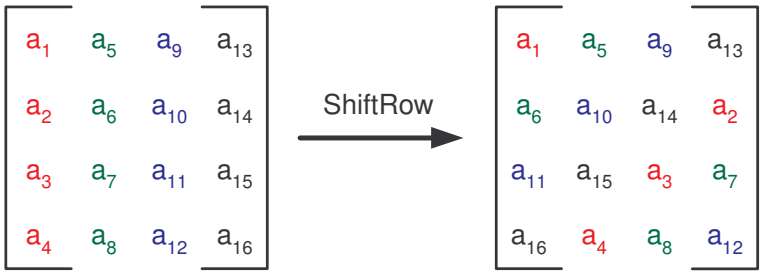


Figure 16: The ShiftRow Transformation for a 128-bit block length [Savard]. Colors are used to help see how the columns shifted.

The MixColumn Transformation

The MixColumn Transformation defines the current state of the data being transformed as a polynomial in GF(2^8), and multiplies it modulo $x^4 + 1$ by a fixed polynomial

$$c(x) = \text{'03'}x^3 + \text{'01'}x^2 + \text{'01'}\,x + \text{'02'}.$$

This polynomial is designed have greatest common divisor of 1 with the modulus $x^4 + 1$, so it is invertible for the reverse transformation.

According to [Savard], the above multiplication is equivalent to multiplying by the following matrix.

02	03	01	01
01	02	03	01
01	01	02	03
03	01	01	02

Figure 17: Matrix used for multiplication during MixColumn transformation.

Applying the Transformations

Rijndael specifies that transformations be applied in the following order.

1. AddRoundKey - Done once at the beginning
2. *The Rounds - This is the step that iterates*
   a. ByteSub
   b. ShiftRow
   c. MixColumn
   d. AddRoundKey
3. Extra Final Round
   a. ByteSub
   b. ShiftRow
   c. AddRoundKey

Because all of the transformations have inverses, this can be reversed to decipher ciphertext back into plaintext.

Prime Number Generation

All of the cryptographic techniques that we've discussed involve very large, random prime numbers. Hence an efficient way to compute these numbers is required. The brute force method of determining primality is unacceptably inefficient in practice. In order to quickly find large, random prime numbers, we must turn to probabilistic testing methods. Probabilistic testing methods typically choose a random large odd number, and then testing for primality using one of several known methods [Schneier, 258]. This project uses the Java math library's BigInteger to test for primality. According to Sun's web site, BigInteger uses the Miller-Rabin and Lucas-Lehmer tests in its probabilistic primality testing [McCluskey].

Key Sizes and Modulus Lengths

Key length choice is an important consideration. For the security algorithm in this thesis, three key sizes need be chosen: RSA Modulus, Diffie-Hellman Modulus, and Rijndael key size. The Rijndael key size is discussed in the Rijndael section (128-bit keys are used in this thesis).

RSA Key size refers to the bit length of the modulus, *n*. RSA Laboratories recommends a key size of 2048 bits as a lower bound for commercial software applications. This thesis uses 2096 bit keys. [RSA Laboratories, 4.1.2.1].

The Diffie-Hellman modulus size should be based on the size of the key one wants to generate with it. According to [Schneier, 166], for 128-bit keys a minimum of a 2304 bit modulus is recommended. This thesis abides by that recommendation and uses a 2304 bit modulus.

Appendix B

SOFTWARE DESIGN DESCRIPTION

kmlJCE Provider

jkml.crypto.utils Package

The Cryptographic Utilities portion of the kmlJCE Provider is a collection of classes designed to support cryptographic algorithms. It currently contains classes that encapsulate Diffie-Hellman and RSA Prime numbers, and associated operations, and a Certificate Entry. The following are the classes in the Cryptographic Utilities package.

class CertificateEntry

This class encapsulates a Certificate. The Certificate contains an RSA Public Key and a Signed Hash of that Public Key.

class DHPrime

DHPrime uses BigInteger to find prime numbers. It also implements the functionality to find primitive roots.

class RSAPrime

RSAPrime uses BigInteger to find large primes p and q in the RSA algorithm. It computes n = (p-1)(q-1), and chooses an encryption key, e, such that e and (p-1)(q-1) are relatively prime. Finally, it computes the decryption key for e.

class InvalidBitLengthException

Extends java.lang.Exception, and is used by DHPrime and RSAPrime when an invalid bit length is passed to one of the class methods.

class InvalidStateException

Extends java.lang.Exception, and is used when an invalid state is reached in an object.

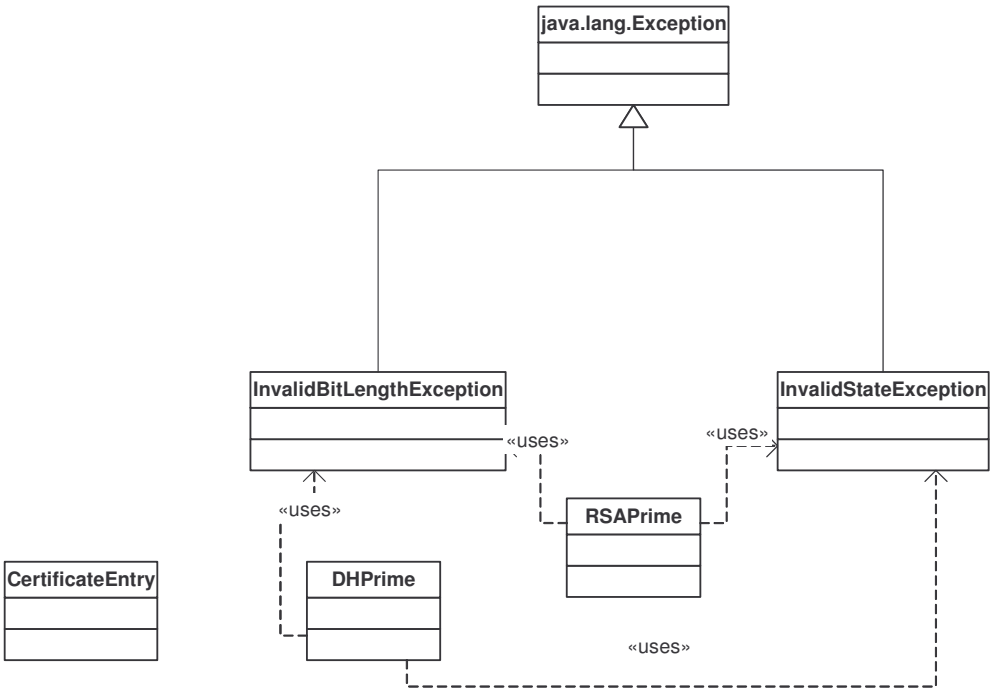


Figure 18: Class Diagram for the Cryptograhpic Utilities Pacakge showing uses and inheritance relationships.

jkml.crypto Package

The jkml.crypto Package is the JCE Service Provider Interface (SPI) implementation written for this thesis. All classes that have names pre-pended with "DH" are the Diffie-Hellman Key Agreement classes, and directly implement JCE SPI's. All classes that have names pre-pended with "MyDH" are support classes for Diffie-Hellman Key Agreement. All classes that have names pre-pended with "RSA" are the RSA Digital Signature classes, and directly implement JCE SPI's. All classes that have names pre-pended with "MyRSA" are support classes for RSA Digital Signatures. Finally, MyProvider is the provider "main" classes.

class MyProvider

The Provider "main" class, this class extends java.security.Provider. It identifies the names and aliases of all of the SPI engines implemented by the provider. For example, the AlgorithmParameterGeneratorSpi for the Diffie-Hellman Key Agreement is implemented in the class

jkmlx.crypto.DHAlgorithmParameterGenerator. Further, it can be referenced in queries to the provider simply as "DH" when retrieving an AlgorithmParameterGenerator. This is tied together in MyProvider with the following:

```
/**
 * Algorithm parameter generation engines
 */
put("AlgorithmParameterGenerator.DiffieHellman",
    "jkmlx.crypto.DHAlgorithmParameterGenerator");
put("Alg.Alias.AlgorithmParameterGenerator.DH",
    "DiffieHellman");
```

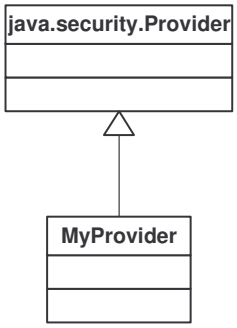


Figure 19: Class diagram showing MyProvider inheritence

class DHAlgorithmParameterGenerator

Extends and provides the implementation for java.security.AlgorithmParameterGeneratorSpi. This class uses the jkmlx.crypto.utils.DHPrime to generate the parameters. The DHAlgorithmParameterGenerator and its primary usage relationships is highlighted in a class diagram in the DHAlgorithmParameters section.

class DHAlgorithmParameters

Extends and provides the implementation for java.security.AlgorithmParametersSpi. This class is as a placeholder. While required to meet the JCE interface, it only encapsulates the javax.crypto.spec.DHParameterSpec.

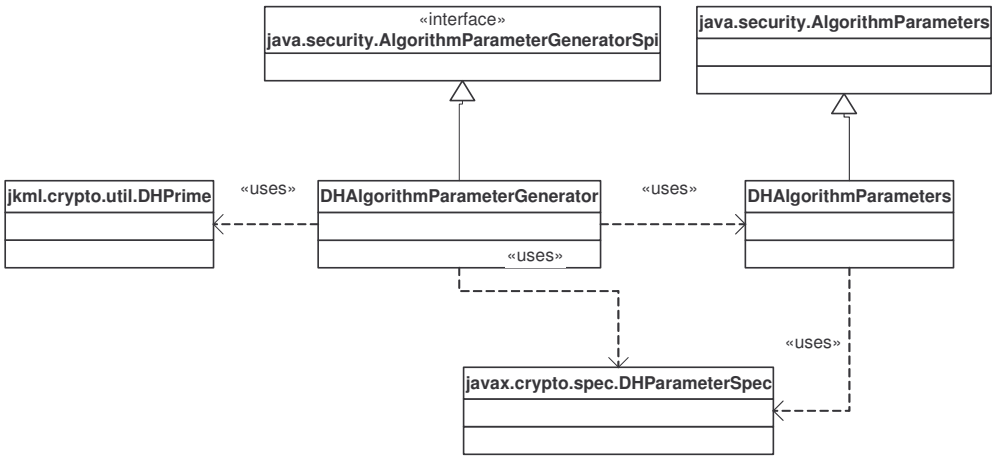


Figure 20: Class diagram showing inheritance and usage relationships of the DHAlgorithmParameterGenerator and DHAlgorithmParameter classes.

class DHKeyAgreement

Extends and provides the implementation for javax.crypto.KeyAgreementSpi. This class carries out the Diffie-Hellman Key Agreement. See the class diagram at the end of the class MyDHPublicKey section.

class DHKeyFactory

Extends and provides the implementation for java.security.KeyFactorySpi. This class bi-directionally translates between key encodings and key specifications (i.e., usable objects). See the class diagram at the end of the class MyDHPublicKey section.

class DHKeyPairGenerator

Extends and provides the implementation for java.security.KeyPairGeneratorSpi. This class generates the Diffie-Hellman parameters. Specifically, given the following equation:

$$X = g^x \bmod p$$

This class is initialized with the primitive root, *g* and the prime, *p*. It generates the variable x, and then computes X. See the class diagram at the end of the class MyDHPublicKey section.

class MyDHEncoding

Extends java.security.spec.EncodedKeySpec. This class encodes and decodes the Diffie-Hellman Key Agreement parameters. Specifically, it encodes the following, in order:

- The size of the prime, *p*.
- The size of the primitive root, *g*.
- The size of the public key, *X*.
- The originally requested size of *p*, called *l*, this should be the same as the first encoded parameter, but is included for ease in decoding.
- The prime, *p*.
- The primitive root, *g*.
- The public key, *X*.

See the class diagram at the end of the class MyDHPublicKey section.

class MyDHPrivateKey

Implements the interface javax.crypto.interfaces.DHPrivateKey. This class encapsulates a Diffie-Hellman Key Agreement private key. That is, the lower-case *x* in the expression

$$X = g^x \bmod p.$$

See the class diagram at the end of the class MyDHPublicKey section.

class MyDHPublicKey

Implements the interface javax.crypto.interfaces.DHPublicKey. This class encapsulates a Diffie-Hellman Key Agreement public key. That is, the upper-case *x* in the expression

$$\mathbf{X} = g^x \bmod p.$$

The following class diagram shows the inheritance and important usage relationships for the classes MyDHPublicKey, MyDHPrivateKey, MyDHEncoding, DHKeyPairGenerator, DHKeyFactory and DHKeyAgreement.

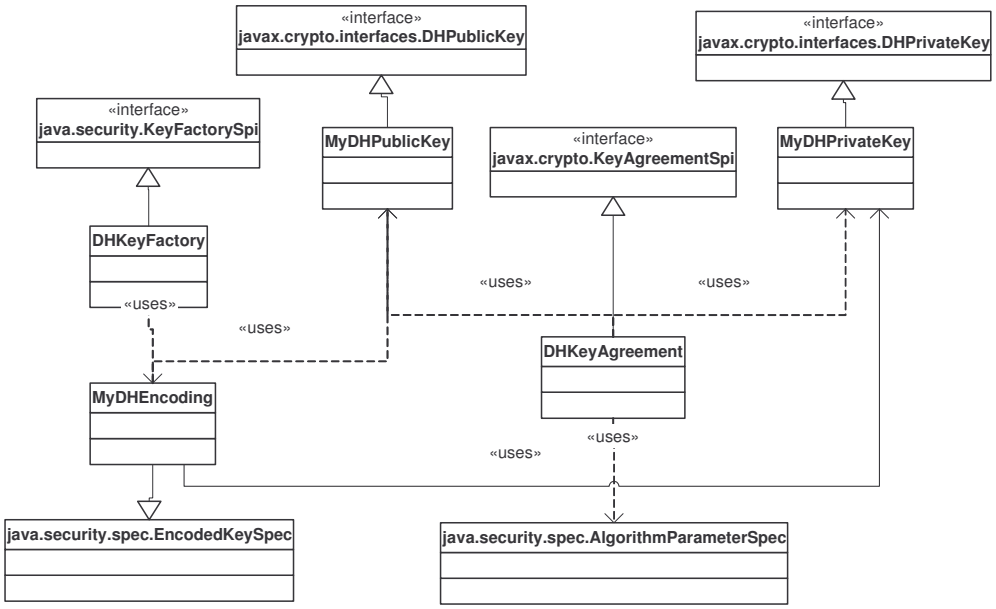


Figure 21: Class diagram showing inheritance and important usage relationships for key classes used in the Diffie-Hellman engine.

class RSASignature

RSASignature is the implementation the SignatureSpi.

class MyRSADataEncoding

This class encodes the public key parameters for transfer.

class MyRSAPrivateKey

MyRSAPrivateKey encapsulates the secret RSA key.

class MyRSAPublicKey

MyRSAPrivateKey encapsulates the public RSA key.

class MyRSASignedDataBlock

This class encodes the public key for transfer.

Cryptographic Service Application

jkml.util Package

The jkml.util package is intended to be a collection of re-usable classes developed during this thesis. Most are general utilities (e.g., a logging class), encapsulate common Jini services (e.g., ServiceBase) or provide common utilities between the Cryptographic Service and Certificate Generation applications (e.g., Utils).

class Utils

This class provides a place for common code, especially those routines common between the Certificate Generation utility and the Cryptographic Service Application.

class ClientBase

Class ClientBase provides interaction with JINI related functionality. It uses the JoinManager to register with and obtain leases from local lookup services and obtain.

class ClientBaseClient

Class ClientBaseClient provides a callback class for the ClientBase. This class is the destination for ClientProxy's.

class ClientProxy

Class ClientProxy provides a base class for proxies.

class Logger

Class Logger provides a logging facility.

class Partner

Class Partner provides a base class to store proxies retrieved from the same remote server. A remote server for which we retrieve one or more proxies shall be termed a "Partner". A Partner is identified by the common name field, returned from ClientProxy::getName(). If that function returns the same name for different proxies, it is assumed that the proxies are from the same Partner.

class ProxyContainer

Class ProxyContainer provides a container for a variable number of Objects (that should be Proxy's).

class ServiceBase

Class ServiceBase provides interaction with JINI related functionality. It uses the JoinManager to register with and obtain leases from local lookup services and obtain.

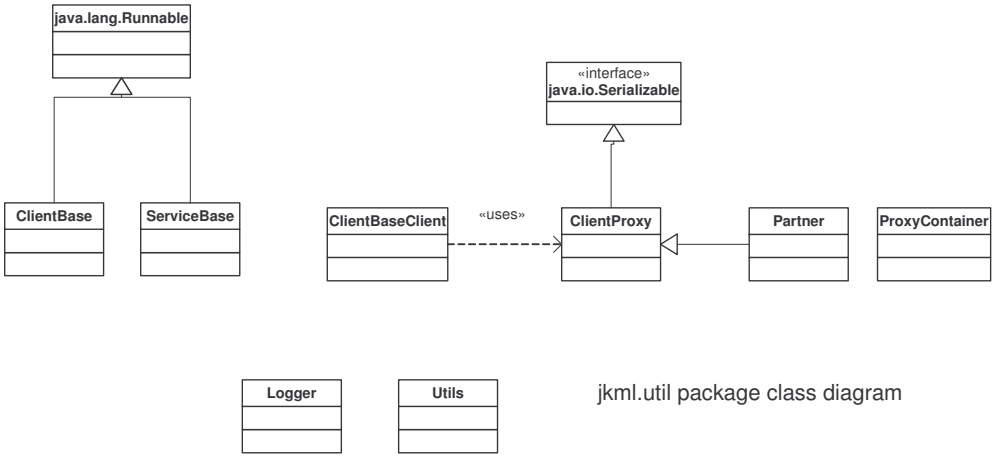


Figure 22: Class diagram for the jkml.util package

jkml.CryptoService Package

The jkml.CryptoService is the main application for this thesis. It contains the Command Line Interface (CLI) code, and creates, registers and starts all of the services. It also uses the JCE Providers to exchange keys, sign and encrypt files.

class CryptoServiceMain

Class CryptoServiceMain provides the main entry point for the service. It constructs the primary objects, initializes the services, and starts the CryptoClient (CLI).

class CryptoClient

This class provides the Command Line Interface (CLI).

interface CryptoRemoteAttendant

This is a remote interface (i.e., it extends java.rmi.Remote). Classes that receive key exchange and file transfer requests implement this interface.

class CryptoAttendant

Implements CryptoRemoteAttendant interface, and extends java.rmi.server.UnicastRemoteObject. This class receives remote requests for Key Exchanges and File Transfers and forwards the request to the appropriate Engine routines. The application's proxy keeps a reference to an instance of this class.

class CryptoPartner

Class CryptoPartner stores the session information for Cryptographic exchanges for a particular Partner.

interface JKASReceiveInterface

The JKASReceiveInterface is the back-end interface for key agreement.

interface JKASInitiateInterface

This interface is the Jini Key Agreement Service initiation interface. It is the local interface for the JKAS service, used by an application to initiate a key exchange.

class JKASProxy

This class implements the JKASInitiateInterface, and is one of the two remote proxies registered in the Jini environment. The proxy maintains a reference to the CryptoRemoteAttendant in order to handle requests (that is, forward requests back to the parent service).

interface JSFTSReceiveInterface

The JSFTSReceiveInterface is the back-end interface for file transfer.

interface JSFTSInitiateInterface

This interface is the Jini Secure File Transfer Service initiation interface. It is the local interface for the JSFTS service, used by an application to initiate a key file transfers.

class JSFTSProxy

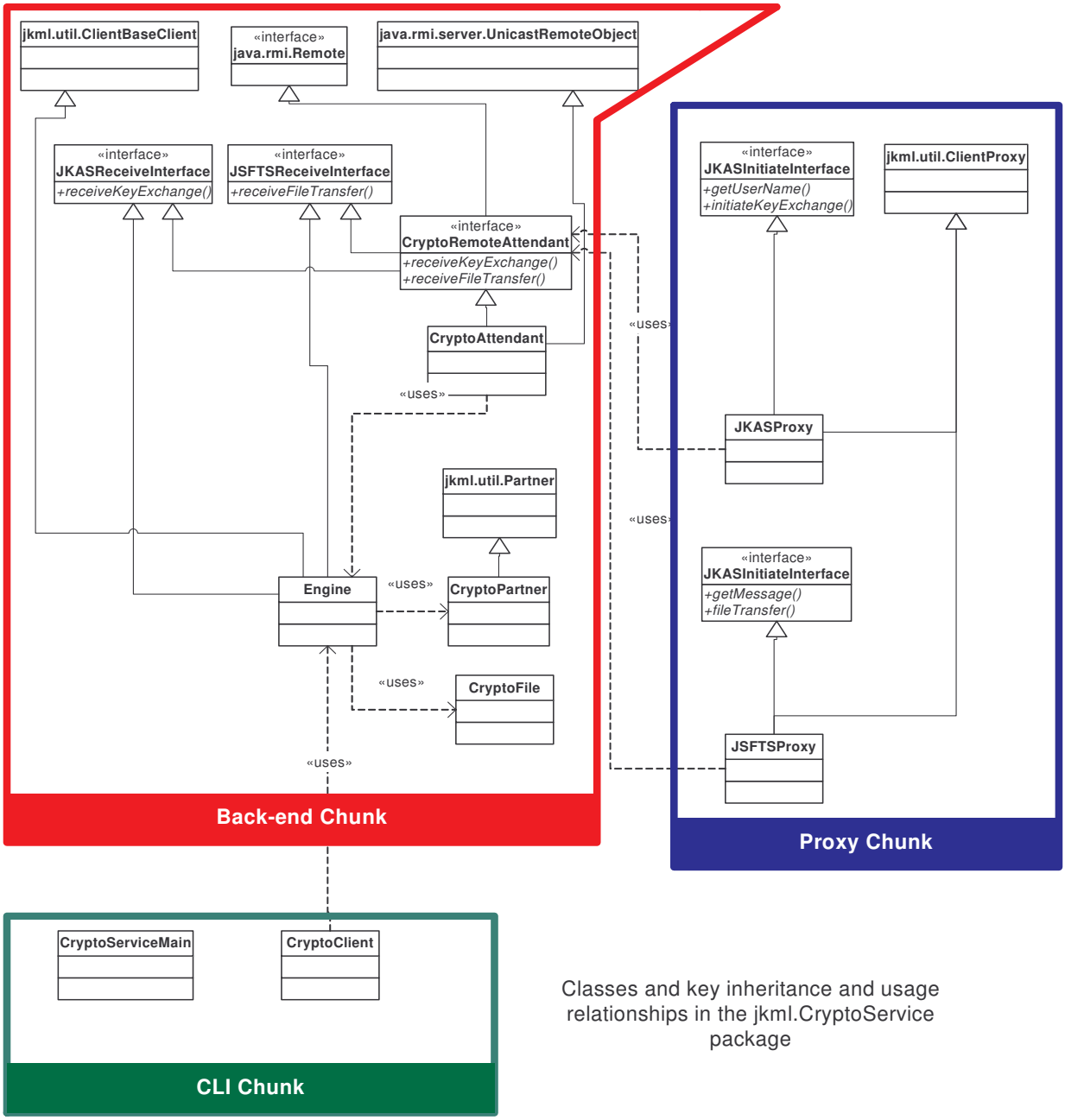
This class implements the JSFTSInitiateInterface, and is one of the two remote proxies registered in the Jini environment. The proxy maintains a reference to the CryptoRemoteAttendant in order to handle requests (that is, forward requests back to the parent service).

class Engine

Processes front-end requests from the CryptoClient (CLI), and processes back-end requests from the CryptoAttendant. This class uses CryptoFile for signatures and encryption, and the JCE Provider directly for Key Agreement.

class CryptoFile

Represents a class at some stage in the signature and encryption process. This file uses the JCE Provider to sign and encrypt a file.



Classes and key inheritance and usage relationships in the jkml.CryptoService package

Figure 23: Class Diagram for the jkml.CryptoService Package

Certificate Generator

jkml.CertificateGenerator

This package comprises only one class, the CertGen class, which implements the entire functionality for this utility.

class CertGen

Class CertGen is the certificate generation utility.

- Generates master certificate public and private keys.
- Generates user keys given a command line parameter (user name).
- Creates certificates with a signed hash using the master private key.

It is the only class in the certificate generation utility, and behaves differently given different command line parameters, as is described in Appendix B.

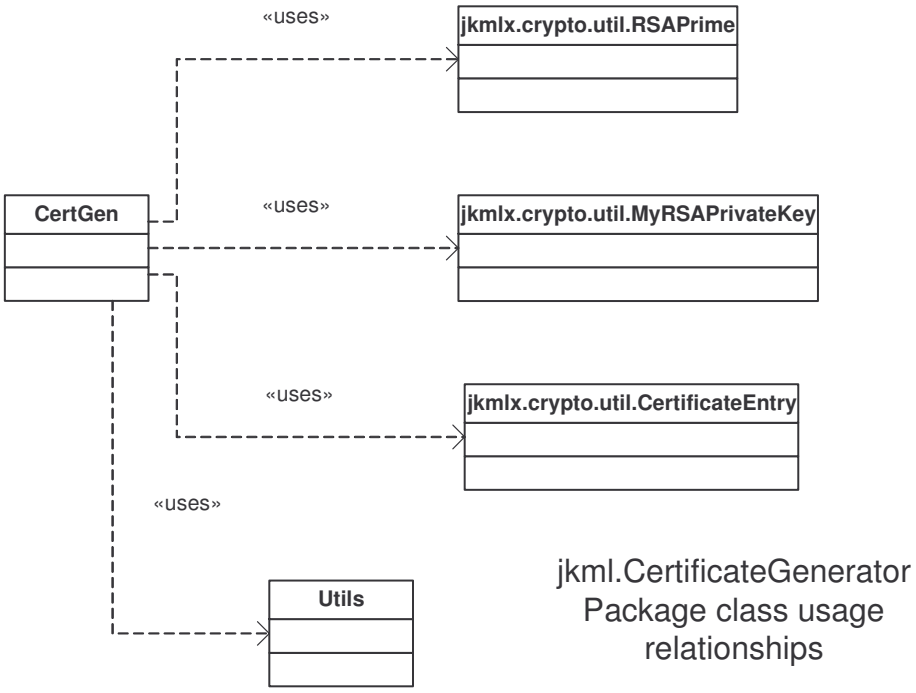


Figure 24: Class Usage Relationships for the jkml.CertGen Package

Appendix C

SOFTWARE USER'S MANUAL

Certificate Generator

The CertGen application has two modes of operation. It will (1) create the master public and private key; (2) create a public and private certificate for a user.

Creating the Master Certificate

If the master certificate has not yet been created, then simply running the application for the first time – without command line parameters – will generate it. Two files will be generated with names defined in jkml.crypto.util.Utils. The first is the private signature exponent of the master certificate. This is required to remain in the same directory as the CertGen utility as it is needed to create user certificates. The second file created is the public key "paste" file. The key in this file needs to be cut-and-pasted into the jkml.crypto.util.Utils file.

Creating a User Certificate

The master certificate must already be created. If it is, then calling CertGen *name,* where *name* is the user's name in the system (e.g., Bob), creates two files. The first file is *name*'s public certificate, and the second file is *name*'s private signature key. The certificate needs to go into the CryptoService folder itself, while the private key goes into a subfolder in CryptoService named *name*.

Crypto Service

The CryptoService application requires one command line parameter – the name of the user. E.g., CryptoService *Bob.* The application automatically finds all of the other users in the system via the JLUS. There are three supported operations, and several debug options. The debug options simply allow the user to walk step-by-step through the encryption process, and print out the keys obtained so that it can be easily verified in demonstrations that the keys are identical on both sides. The supported operations are as follows:

1. Show Users - allows the user to list other parties in the community, along with whether they have registered both required services. If so, then they are a valid user for the second option.

2. Transfer File with User - allows the user to actually transmit a file to one of the parties given by the Show Users option. It asks only for the user name, and for a file name.

3. Quit – Exits the application

Appendix D

CRYPTO SERVICE SAMPLE PROGRAM OUTPUT

The Scenario

For this scenario there will be two users, Bob and Alice, running on two different hosts, NewYork and Iowa, respectively. Bob will initiative the file transfer.

The Output

Figure 1 – The Main Menu

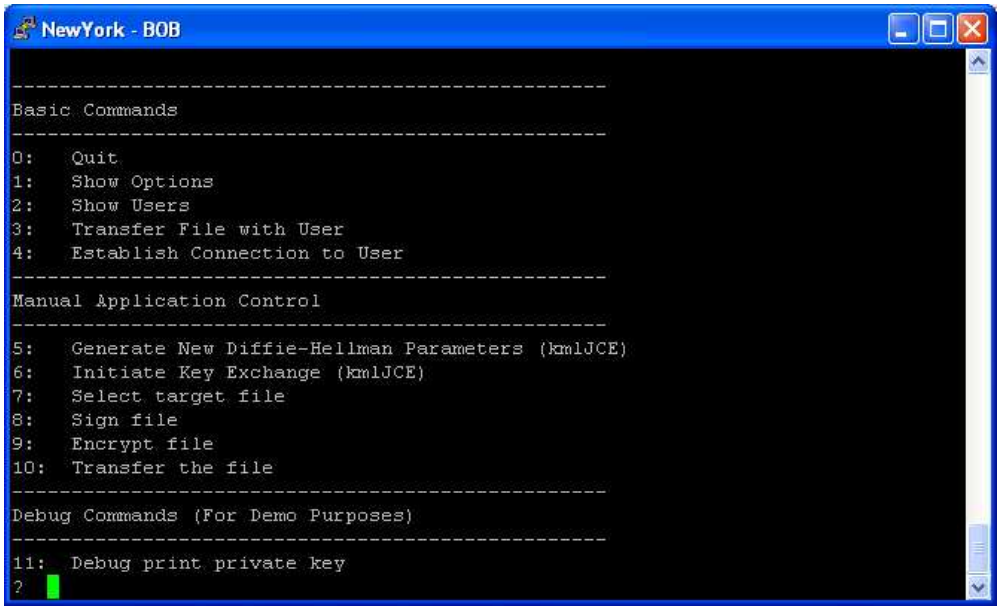


Figure 2 – Bob List's the Users in the System – menu option 2. Notice that Alice is a registered user, and that she supports both required services – JKAS for Key Agreement and JSFTS for File Transfer.

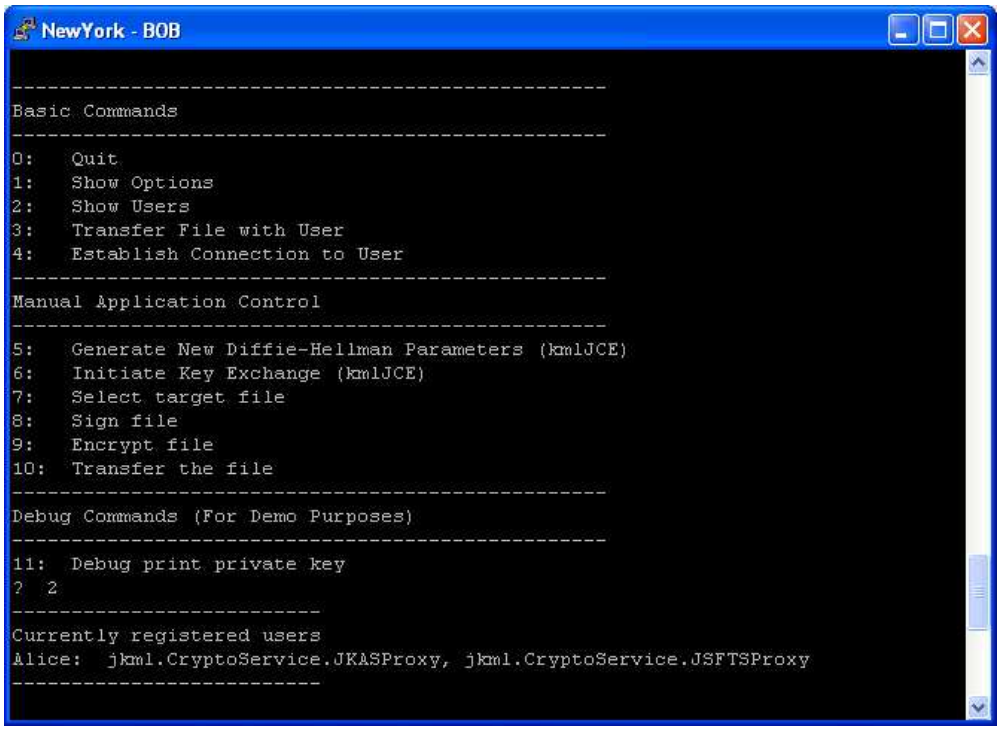


Figure 3 – Bob Selects to send a file to a user. He's asked for two pieces of information – the name of the other user (Alice) and the name of the file to transfer (test_bob).

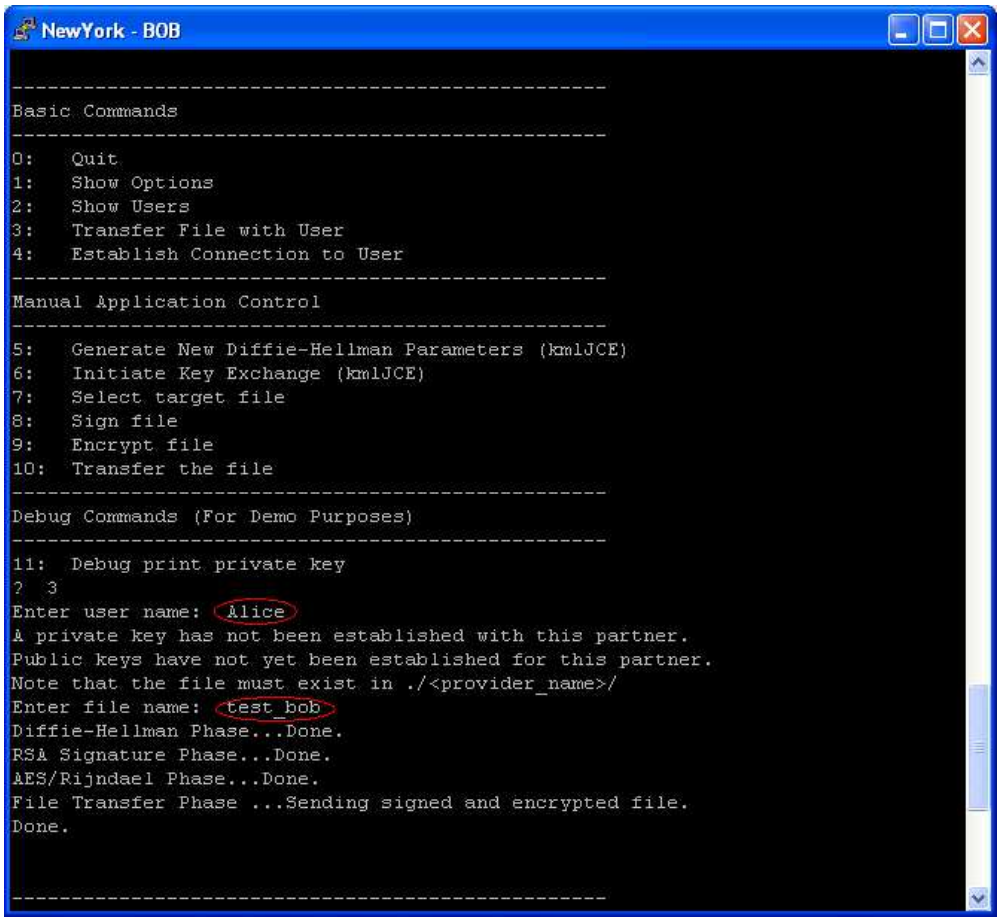
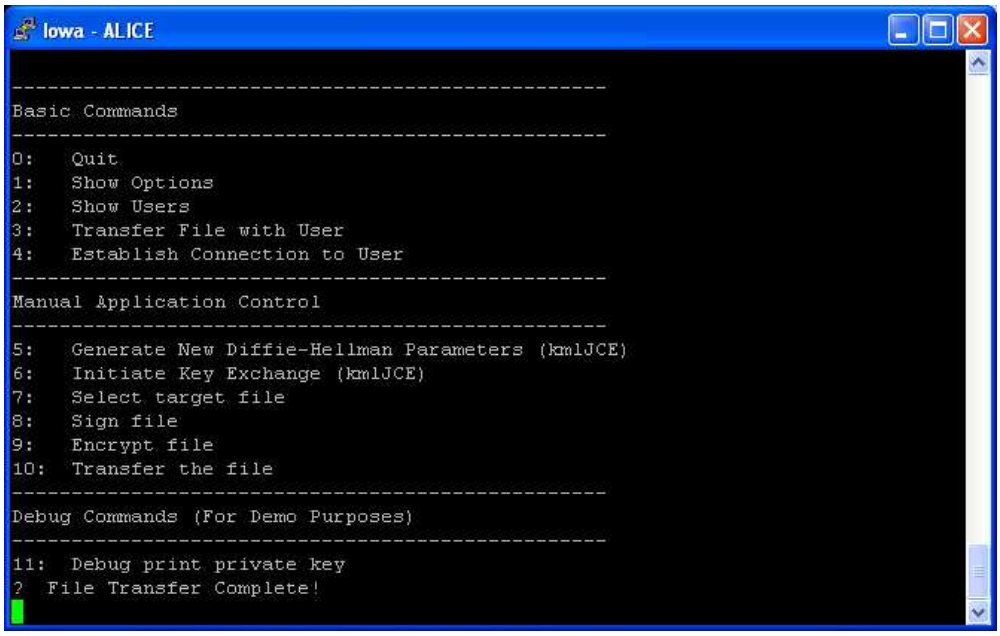


Figure 4 – Alice receives the file automatically. She is only informed when the file transfer is complete. She can then go and look in her local folder for the new file.



REFERENCES

Stinson, Douglas R. *Cryptography: Theory and Practice.* New York, NY: CRC Press, 1995.

Schneier, Bruce. *Applied Cryptography,* 2nd ed. New York, NY: John Wiley & Sons, Inc., 1996.

Edwards, W. Keith. *Core Jini*, 2d ed. Upper Saddle River, NJ: Prentice-Hall, Inc., 2001.

Knudsen, Jonathan. *Java Cryptography*. Sebastopol, CA: O'Reilly & Associates, Inc., 1998.

Niven, Ivan, et al. *An Introduction to The Theory of Numbers*, 5th ed. New York, NY: John Wiley & Sons, Inc., 1991.

Ferguson, Niels and Schneier, Bruce. *Practical Cryptography.* New York, NY: John Wiley & Sons, Inc., 2003.

Daemen, Joan and Rijmen, Vincent. *The Rijndael Block Cipher: AES Proposal,* version 2. Computer Security Division: Computer Security Resource Center (CSRC). National Institute of Standards and Technology. Published 3/9/1999. Last Accessed 10/26/2004. http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael-ammended.pdf.

Savard, John. *The Advanced Encryption Standard (Rijndael).* A Cryptographic Compendium. Publish date unknown. Last Accessed 10/26/2004. http://home.ecn.ab.ca/~jsavard/crypto/co040401.htm.

Sommers, Frank. *Jini Starter Kit 2.0 tightens Jini's security framework: Introducting Jini's new security features.* JavaWorld. Published 5/9/2003. Last Accessed 10/26/2004. http://www.javaworld.com/javaworld/jw-05-2003/jw-0509-jiniology_p.html.

Boneh, Dan. *Twenty Years of Attacks on the RSA Cryptosystem.* Publish date unknown. Last Accessed 11/11/2004. http://crypto.stanford.edu/~dabo/papers/RSA-survey.pdf.

Killeen, Ronan. *Possible Attacks on RSA.* RSA: Hacking and Cracking. Published 5/4/2001. Last Accessed 11/11/2004. http://members.tripod.com/irish_ronan/rsa/attacks.html.

Sun Microsystems. *Jini Technology Core Platform Specification: Contents.* Sun Microsystems. Copyright 1994-2004. Last Accessed 11/11/2004. http://wwws.sun.com/software/jini/specs/jini1.1html/coreTOC.html.

Sun Microsystems. *Jini Technology Core Platform Specification: AR.2 System Overview.* Sun Microsystems. Copyright 1994-2004. Last Accessed 11/11/2004. http://wwws.sun.com/software/jini/specs/jini1.1html/jini-spec.html#1001135.

Sun Microsystems. *Jini Technology Core Platform Specification: DJ – Discovery and Join.* Sun Microsystems. Copyright 1994-2004. Last Accessed 11/11/2004. http://wwws.sun.com/software/jini/specs/jini1.1html/discovery-spec.html.

Sun Microsystems. *Java Cryptography Extension (JCE).* Sun Microsystems. Copyright 1994-2004. Last Accessed 11/11/2004. http://java.sun.com/products/jce/.

Sun Microsystems. *Java Cryptography Extension (JCE) 1.2.2*. Sun Microsystems. Copyright 1994-2004. Last Accessed 11/11/2004. http://java.sun.com/products/jce/index-122.html.

McCluskey, Glen. *Generating Prime Numbers and When Not to Overload Members*. Sun Microsystems. Copyright 1994-2004. Last Accessed 11/11/2004. http://java.sun.com/developer/JDCTechTips/2002/tt0806.html.

RSA Labaratories. *RSA Security – 4.1.2.1 What key size should be used?*. RSA Labaratories. Copyright 2—4. Last Accessed 11/14/2004. http://www.rsasecurity.com/rsalabs/node.asp?id=2264.