# Public Key Certification & Secure File Transfer

*Christoph L. Schuba and Sulabha S. Sheth*

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
{schuba,sheth}@cs.purdue.edu

**Abstract**

This document describes secure file transfer between user agents. Our approach guarantees confidentiality and integrity of transferred files, originator authentication, and non-repudiation. We achieve these goals through the usage of DES, MD5-RSA, ANSI X9.17 and a distributed scheme for the validation of public component certificates (similar to X.509.)

## 1 Introduction

Our goal is to build a secure file transfer protocol between user agents that guarantees the following features:

- confidentiality of the file contents,
- integrity of the file contents,
- originator authentication, and
- non-repudiation.

Integrity, originator authentication, and non-repudiation are achieved by applying a digital fingerprint, or message digest to the transferred file. We are using MD5[1]-RSA[2] for the digital signature. Confidentiality is achieved by DES[3] encryption of the file with a different private key for each message that is generated according to ANSI X9.17[4].

The security of the RSA asymmetric public key[5] cryptosystem depends on the validity of the public keys. That means that the correct binding between an entity and its public key must be established undoubtably. The correct binding can be established through the usage of public key certificates. Each entity that possesses a public key/private component pair also has a certificate, a tupel containing the entities identifier (`subject`), the entities public key (`public_key`), and a signature over the previous fields (`signature`) signed by the issuer with its private key.

[1] Message Digest Algorithm
[2] Rivest, Shamir, and Adleman scheme
[3] Data Encryption Standard
[4] ANSI session key generation
[5] We use the terms *public key* and *private component* for the keys of asymmetric cryptosystems, the term *private key* for symmetric cryptosystems.

Issuers, or certification authorities are organized in a hierarchical fashion like in PEM[6] which is based on X.509[7]. The public key of the root of this certification graph must be known to everybody by some out-of-band mechanism. User agents are the leaves of the certification graph.

A certificate is validated by verifying the signature applied by the issuer of the certificate. This use of certificates transforms the problem of acquiring the public key associated with a user into one of acquiring the public key of the issuer of the user's certificate. The recursion terminates when the issuer is the root, whose public key is well known.

# 2 Algorithms

This section gives definitions of the algorithms used in our approach. We present the Message Digest 5 algorithm (MD5), the RSA scheme, the Data Encryption Standard (DES), X9.17 session key generation, and the usage of public key certificates like in the ISO authentication framework X.509.

## 2.1 The MD5 Algorithm

The algorithm takes as input a message of arbitrary length and produces as output a 128-bit fingerprint or message digest of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest.

Suppose the input is a $b$-bit message $M$ ($b \in I\!N$): $M = m_0 m_1 m_2 ... m_{b-1}$. The algorithm is defined in five steps:

1. **Append Padding Bits**

   The message is padded so that length$(M) \equiv 448 \pmod{512}$.

   Padding: $1\{0\}*$ (i.e. at least 1 bit and at most 512 bits are appended.)

2. **Append Length**

   A 64-bit representation of $b$ is appended to the result of the previous step. If $b > 2^{64}$, then only the low-order 64-bits of $b$ are used.

3. **Initialize MD Buffer**

   A four-word buffer (A,B,C,D) is used to compute the message digest. These 32-bit registers are initialized with certain values.

4. **Process Message in 16-Word Blocks**

   Each 16-word block of the input is run through 4 rounds of application of transformation functions. The four-word buffer serves as temporary memory.

[6] Privacy Enhanced Mail
[7] ISO authentication framework

5. **Output**

   The message digest produced as output is A, B, C, D.

MD5 was developed by R. Rivest and is defined in [Riv92].

## 2.2 The RSA Scheme

The RSA is an exponentiation cipher based on a public key system. It is based on a public $n$ which is the product of two large secret primes $n = pq$. A $d$ is chosen such that it is relatively prime to the Euler Phi function of $n$ , $\phi(n)$. It is chosen such that it lies in the interval $[\max(p,q)+1, n-1]$. The encryption exponent $e$ is calculated by calculating inverse of $d$ and $\phi(n)$.

The *reduced set of residues* modulo $n$ is the subset of residues $\{0, ...., n-1\}$ relatively prime to $n$. The *Euler Totient Function* $\phi(n)$ is the number of elements in the reduced set of residues modulo $n$.

The Euler Totient Function $\phi(n)$ is determined by first creating the reduced set of residues and then determining the relative primality of each element of this set with $n$. For a prime number $p$ $\phi(p) = p-1$. Thus, for RSA, if $n = pq$, then $\phi(n) = \phi(p)\,\phi(q)$.

- To encrypt M we use the following function: $C = M^e \bmod n$
- To decrypt C, the encrypted message, we use the following function: $M = C^d \bmod n$

In this way, the encryption exponent $e$ and $n$ are made public. One cannot calculate $d$ without the knowledge of $p$ and $q$. Therefore the enciphering transformation is made public and the deciphering transformation is kept secret. The security of the system depends on the difficulty and speed with which $n$ can be factored into its factors $p$ and $q$. The security of the system also depends on using carefully selected primes $p$ and $q$. If $n$ is 200 digits then $p$ and $q$ should be large primes of approximately 100 digits. Rivest, Shamir and Adleman suggest using 100-digit numbers for $p$ and $q$; then $n$ is 200 digits, and factoring would take several billion years at the rate of one step per microsecond.

In hardware, at its fastest RSA is about 1000 times slower than DES. The fastest VLSI hardware implementation for RSA with a 512-bit moduli has a throughput of 64 kbps.

In software, DES is about 100 times faster than RSA. These numbers may change slightly as technology advances but RSA will never approach the speed of symmetric algorithms.

The RSA scheme is originally defined in [RSA78] and nicely described in [Den82, §2.7.2].

## 2.3 Prime Number Generation

For many data encryption schemes, a number of keys is required. To make the scheme less susceptible to breaking it is often suggested that huge prime numbers be used. Prime number generation plays a pivotal role in Data Security & Cryptography and many years and PhD thesis have been spent on generating prime numbers efficiently and moreover verifying that the number that is generated is truly prime. Current methods are quite efficient and with the computation time being

decreased while the interested reader peruses this document, there are now algorithms that can generate 100 digit prime numbers in a matter of seconds at the same time verifying that the number is prime with a high degree of confidence.

We followed the algorithm in [Den82, §2.7.2] for the generation of prime numbers. It is a simple formula :

$P_{i+1} = 2k \cdot P_i + 1$

That means the previously generated prime $P_i$ is multiplied with an even random number $2k$. The random number $2k$ should have less digits than $P_i$. Since the number thus obtained is an even number, 1 is further added to make it odd. There is a pretty good chance that this number is a prime number. Then we carry out the *sieve operation*, that is we keep adding $2P_i$ to the number determined above for an array of about $5 \cdot \ln(10^{current\_number\_of\_digits})$ numbers and then cancel out all the numbers which have a small prime factor. The small prime factor can be any prime number which is less than say 1000. The numbers which are not cancelled out are then checked for primality using the ballistic primality test which was handed out in class and is a very fast and efficient algorithm. We have implemented the algorithm and it works fine giving excellent results. As soon as a prime number is found, the search is terminated and we repeat the process with the newly obtained prime until the required number of digits is obtained.

Experience shows that close to the termination of the algorithm it becomes increasingly difficult to determine a prime number of the desired number of digits.

## 2.4 The Data Encryption Standard

Unlike in RSA which is an asymmetric cryptosystem, in the symmetric DES system the same key is used for both enciphering and deciphering. It enciphers 64-bit blocks with a 56-bit key. Refer to figures 1 and 2. The algorithm may be summarized in 5 steps :

1. **Initial Permutation**

   The input block is first transposed under an initial permutation. The IP table is a public table. It is read from left-to-right, top-to-bottom.

2. **The 16 iterations**

   After the initial permutation 16 iterations (or rounds) are performed on the permuted block. This involves a combination of substitutions and transpositions. The block is identified as two blocks of 32 bits each, $L_i$ and $R_i$. Then $L_i = R_{i-1}$ and $R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$. Where $K_i$ is a series of 48-bit keys generated from K.

3. **The function $f$ and the *S-boxes***

   Initially, the first block say, $R_{i-1}$ is expanded to 48-bits using the bit-selection table. The bit-selection table is used in essentially the same way as the initial permutation table except that a few bits are chosen more that once as we have to expand from 32-bits to 48-bits. After expanding to 48 bits, the block is broken up into eight 6-bit blocks after calculating the exclusive OR of the expanded blocks and the key. Next, each 6-bit block is fed into the

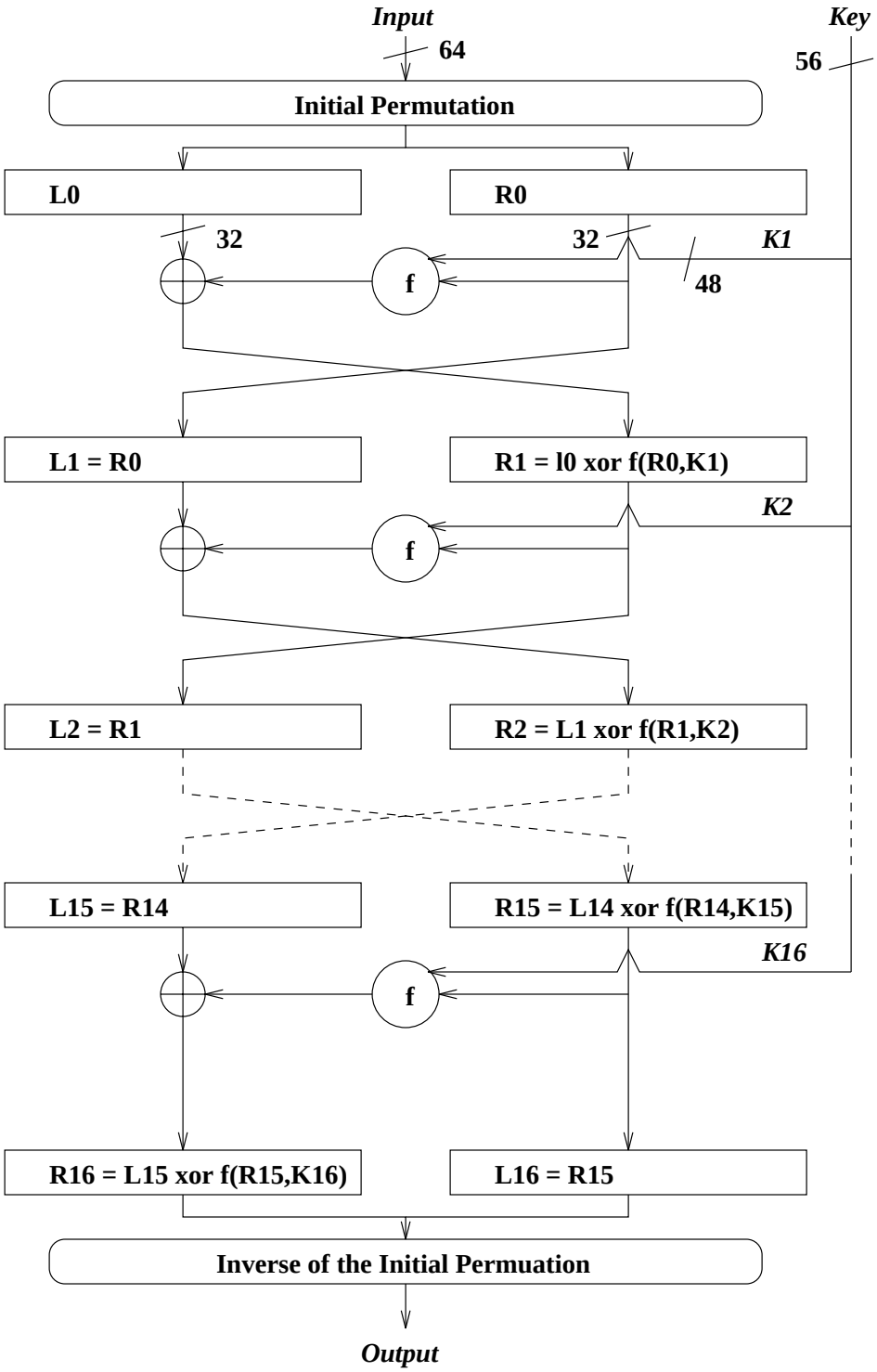


Figure 1: DES Enciphering Algorithm

selection fuctions which return 4-bits. These bits are concatenated together to give 32 bits.

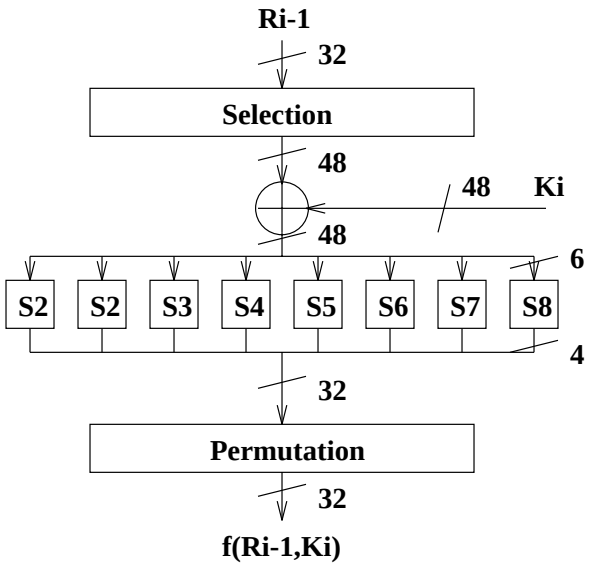


Figure 2: Calculation of $f(R_{i-1}, k_i)$

A point to be noted is that in the last (16th) round, an interchange does not take place.

That is :

$R_{16} = L_{15} \oplus f(R_{15}, K_{16})$ and it remains on the left side. $L_{16} = R_{15}$ and it remains on the right side.

This is so because decryption takes place with the same algorithm with the keys being fed in, in the reverse order, see 5.

4. **The key calculation**

Each of the 16 iterations mentioned above uses a different 48-bit key derived from the initial 56-bit key K. K is input as a 64-bit block, with 8 parity bits in positions 8,16,...,64. The parity bits are discarded using the permutation. It is then split into two halves of 28-bits each, say $C_i$ and $D_i$. The blocks $C_i$ and $D_i$ are each successively shifted left to derive each key $K_i$.

5. **Deciphering**

Deciphering is performed using the same algorithm except that $K_{16}$ is used in the first iteration, $K_{15}$ is the second and so on. This is so because the final permutation is the inverse of the initial permutation. Note that initial and final permutation do not enhance the security of the DES crytosystem, however to adhere to the standard the permutations cannot be omitted.

The DES is originally defined in [NBS77] and nicely described in [Den82, §2.6.2].

### 2.5 X9.17 Key Generation

The ANSI standard X9.17 specifies a method of key generation, which is suitable for generating session keys within a system (see [Sch94, §7.2.1].)

Let $E_k(X)$ be DES encryption of $X$ with key $k$. The key $k$ is a key reserved for secret key generation. $V_0$ is a secret 64-bit seed. $T$ is a timestamp. To generate the random key $R_i$, we calculate:

$$R_i = E_k(E_k(T_i) \oplus V_i)$$

To generate $V_{i+1}$, we calculate:

$$V_{i+1} = E_k(E_k(T_i) \oplus R_i)$$

To turn the $R_i$ into a DES key, we simply adjust every eighth bit for parity and interpret it as such.

### 2.6 Public Key Certificates

The process of validating public keys received from remote entities is described in section 3.5.

Our approach to validate public keys is based on [CCI88] and nicely described in [Sch94, §17.6], [Ken93a], and [Ken93b].

## 3 Application

This section motivates the usage of the algorithms described in the previous section and shows how they are used for signing, verifying signatures, encrypting and decrypting data, and validating public key certificates.

### 3.1 Signing

Two of the problems with file transfer are that the contents of the file might be altered or forged without detection. The usage of digital signatures ensures that such transgressions are detected. Figure 3 depicts the process.

To start with, the file being signed is run through a one-way hash function. This function, also called a cryptographic hash function or message digest, in our case MD5, takes a file of arbitrary size and produces a has value. With a good one-way hash function, it is computationally infeasible to modify the message so that it produces the same hash value.

Once the hash value has been determined, the user agent creates the signature by encrypting the hash value with its RSA private component and sends the signature with the message. If a one-way hash was not used, the signature would have to be as large as the message.


Figure 3: Signing a file

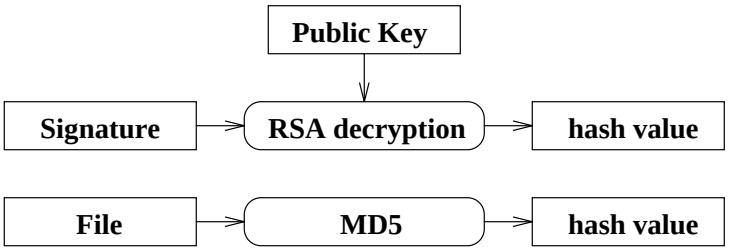


Figure 4: Verifying a Signature

### 3.2 Verifying

In order for the signature to be useful, it must be possible to verify it. The verification process is depicted in Figure 4. To verify the signature on a file that a user agent receives, the same one-way hash function that was used when the file was signed is applied to the file to recompute the hash value. The user agent then takes the RSA public key of the originator and decrypts the signature he had received. If the calculated hash value and the decrypted signature match, then the file must have come from the originator, and it cannot have been altered.

### 3.3 Encrypting

It is very useful to know that a file was genuine, but there is still at least one more problem with file transfer. Anyone having access to any of the computers, networks, or communication lines on which a file is stored or travels can potentially read the message. Using encryption makes sending private files that are supposed to stay private possible. Encrypting a file is depicted in Figure 5.

We use the DES algorithm to encrypt the messages mainly for performance reasons. DES is a much faster algorithm than RSA, and additionally available in hardware, therefore useful to sign large files, while the RSA scheme with its asymmetric nature is useful for signing files and communicating keys. Since DES is a symmetric encryption algorithm, the problem of the key distribution between the two peer user agents (sender/receiver) must be solved.

The user agent encrypts the file using a random DES key called the data encryption key (DEK). That key is encrypted with the recipient's RSA public key. Since only the receiving user agent has access to its RSA private component, only the recipient can decrypt the encrypted DEK and decrypt the message.

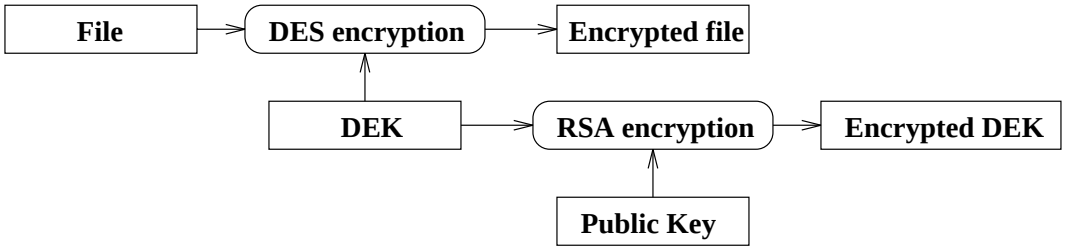


Figure 5: Encrypting a File

### 3.4 Decrypting

Upon receiving a confidential file, the receiving user agent finds a copy of the DEK that was encrypted with its public key. The decryption process is depicted in Figure 6.

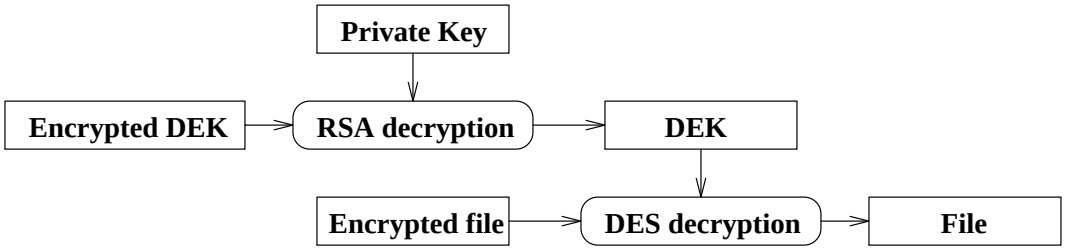


Figure 6: Decrypting a File

The user agent decrypts the file by using its RSA private component to decrypt the DEK and using DES to decrypt the file using the DEK .

### 3.5 Certificate Validating

It is important for the user agents to have others' RSA public keys. It is vital that user agents can trust that they really have the public keys of the user agents that they think they do. To ensure that the certificates are placed in a verifiable certification hierarchy.

Certificates are signed to ensure that they are not altered and to identify who signed them. User agents' certificates are signed by their certification authority, certification authorities certificates by their certification authority, and so on up to the root of the hierarchically organized certification graph.

A user agent that wants to validate another user agent's certificate, does so by verifying it with the issuer's RSA public key. However the issuer's RSA public key must now be validated. This process is clearly recursive. The final verification of the certificate issued by the root of the certification graph is done by the well known RSA public key of the root that must have been received through some out of band mechanism.

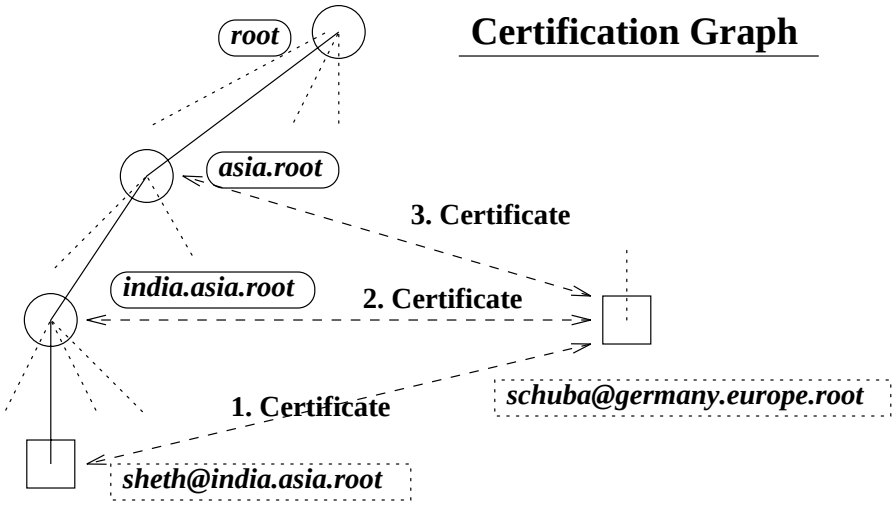


Figure 7: Validating Certificates

Figure 7 depicts an example of this recursive validation process. User agent *schuba@germany.europe.root* wants to obtain validation for user agent *sheth@india.asia.root*'s public key. User agent *schuba* checks 3 certificates (1-3) until he can validate the third one with the well known public key of the certification authority *root*.

1. **Certificate for** *sheth@india.asia.root*

   (`subject` = *sheth@india.asia.root*, `public_key` = $\mathrm{K}_{pub}^{sheth@india.asia.root}$,
   `signature` = $\mathrm{Signature}_{\mathrm{K}_{priv}^{india.asia.root}}$)

   must be validated with $\mathrm{K}_{pub}^{india.asia.root}$.

2. **Certificate for** *india.asia.root*

   (`subject` = *india.asia.root*, `public_key` = $\mathrm{K}_{pub}^{india.asia.root}$,
   `signature` = $\mathrm{Signature}_{\mathrm{K}_{priv}^{asia.root}}$)

   must be validated with $\mathrm{K}_{pub}^{asia.root}$.

3. **Certificate for** *asia.root*

   (`subject` = *asia.root*, `public_key` = $\mathrm{K}_{pub}^{asia.root}$,
   `signature` = $\mathrm{Signature}_{\mathrm{K}_{priv}^{root}}$)

   must be validated with $\mathrm{K}_{pub}^{root}$, which is well known by out-of-band mechanisms (perhaps an advertisement in the New York Times.)

# 4 Implementation

## 4.1 Terminology

We use the OSI[8] terminology for primitives and types of service elements.

Figure 8 depicts *unconfirmed service*, which does not require an explicit end-to-end confirmation to be issued upon the completion of the procedure. Figure 9 depicts *confirmed service*, which does require such a confirmation.

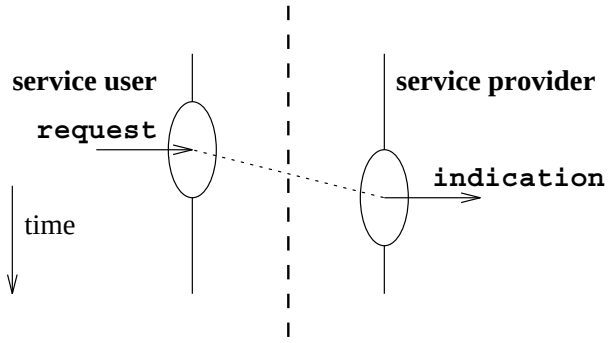


Figure 8: Unconfirmed Service

The *request* primitive is issued by the service user to invoke or initiate the use of the service. The notification of the service provider is performed by the *indication* primitive. Similarly the reply of the provider is the *response* primitive that the service user receives in form of the *confirmation* primitive.

## 4.2 Network Protocol

In this section we define the protocol data units[9] that are transmitted over the network. We will use six different packet types of which four are used for a confirmed service, and two are used for an unconfirmed service. Table 1 shows an overview over the packets with their names, service type, and associated parameter list. Our data dictionary is defined in section 4.5.

## 4.3 Functionality Certification Authority

This section describes the functionality of the certification authority daemon.

1. Setup:
    - system logging

[8] Open Systems Interconnection
[9] PDU, also known as packet or datagram

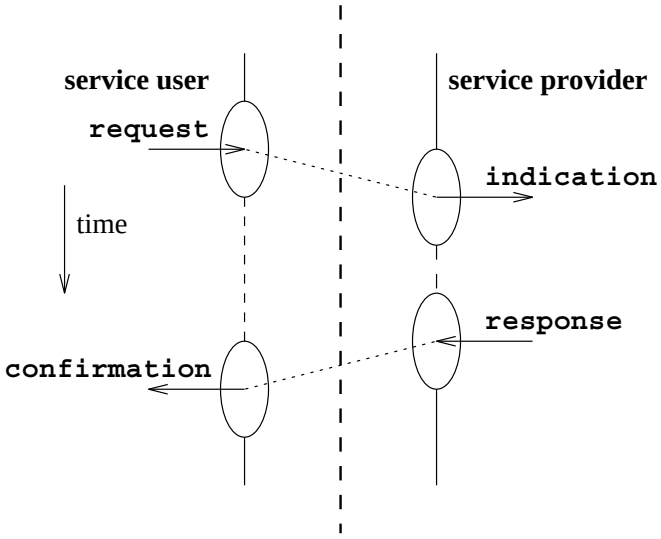


Figure 9: Confirmed Service

Table 1: Protocol packets

| packet name | service type | code | parameters |
|---|---|---|---|
| register.request | confirmed | 0x11 | (subject, rsa_n, rsa_public, ip, port) |
| register.response | | 0x21 | (signature) |
| lookup.request | | 0x13 | (subject) |
| lookup.response | | 0x23 | (ip, port) |
| certificate.request | | 0x14 | () |
| certificate.response | | 0x24 | (subject, rsa_n, rsa_public, signature) |
| transfer.request | | 0x15 | (filename, filesize, from, sig_digest, sig_dek) |
| transfer.response | | 0x25 | () |
| deregister.request | unconfirmed | 0x12 | (subject) |
| data.request | | 0x16 | (data) |

- list data structure for user agent management

2. Initialization:

   - Determine and Save: subject, issuer, rsa_n, rsa_public, rsa_private in .<subject>.key, or
   - Restore: from previously saved .<subject>.key

3. Registration:

- if `subject` = *root*:
  No registration necessary.
  Create: NewYorkTimes and save `rsa_n`, `rsa_public`.
- if `subject` $\neq$ *root*:
  Register with `issuer`:
  `register.request/register.confirmation`.

4. Service on well known port:
   - Registration service to ca's and ua's:
     `register.indication/register.response`.
   - Deregistration service to ca's and ua's:
     `deregister.indication`
   - Lookup service to ca's:
     `lookup.indication/lookup.response`.
   - Certificate service to ca's:
     `certificate.indication/certificate.response`.

### 4.4 Functionality User Agents

This section describes the functionality of the user agent client.

1. Setup:
   - system logging
2. Initialization:
   - Determine and Save: `subject`, `issuer`, `rsa_n`, `rsa_public`, `rsa_private` in .<subject>.key, or
   - Restore: from previously saved .<subject>.key
3. Registration:
   - Register with `issuer`:
     `register.request/register.confirmation`.
4. Select input from standard input and well known port:
   - Standard input:
     - **send <filename> <subject>**
       Send a file to the given subject:
       `lookup.request/lookup.confirmation`
       `{certificate.request/certificate.confirmation}+`
       `transfer.request/transfer.confirmation`
       `{data.request}+`

- **list**
  Show the list of files in the inbox.
- **whoami**
  Display information about the subject.
- **help**
  Display this list of commands and their short explanation.
- **cls**
  Clear the screen.
- **quit**
  Gracefully terminate the session.

- Advertised port:
  - Certificate validation service:
    `certificate.indication/certificate.response`
  - Accept file transfer request, handle file transfer:
    `transfer.indication/transfer.response`
    {`data.indication`}+
    {`certificate.request/certificate.confirmation`}+
    Decryption of files and validation of message digests. The connection will be closed when the file transfer is complete (i.e. when `size` bytes were transmitted.)

## 4.5 Data Dictionary

### 4.5.1 Data Types

- `ip`: `unsigned long` - Internet protocol address
- `quad`: `char [DOTTEDQUADSIZE]` (= 16)
- `subject`: `char [SUBJECTSIZE]` (= 63)
- `issuer`: `char [SUBJECTSIZE]`
- `filename`: `char [FILENAMESIZE]` (= 128)
- `des_key`: `char[DES_KEY_SIZE]` (= 8)
- `digest`: `char [MD5DIGESTSIZE]` (= 16)
- `signature`: `char [MPCDIM]` (= 200)
- `rsa_n`: `char[MPCDIM]`
- `rsa_public`: `char[MPCDIM]`
- `rsa_private`: `char[MPCDIM]`

Table 2: File Naming Conventions

| filename | description |
|---|---|
| .<subject>.key | where <subject> stands for a complete subject name, contains subject, issuer, and all key components. (note this is a hidden file!) |
| NewYorkTimes | contains the well known public key of the subject *root* |
| etc/hosts | contains the subject (host portion) to IP address mapping |
| log/<subject> | where <subject> stands for a complete subject name, contains the log file. Log entries are appended. |
| inbox/<ua>.dir | where <ua> stands for a complete ua name, contains the number of files and a directory entry for each of these files with filename, size, originator and encrypted DES key. The according files have been received and still have to be converted. |
| inbox/<ua>.<filename> | where <ua> stands for a complete ua name and where <filename> is a filename to distinguish several files that were received before converted. |

### 4.5.2 File Names

Table 2 contains an overview over the filename conventions used in this implementation.

### 4.5.3 File Formats

The file formats are described in Extended Backus-Naur form. The terminals `ip`, `subject`, `issuer`, `des_key`, `signature`, `rsa_n`, `rsa_public`, `rsa_private` were defined in section 4.5.1; `int`, `long`, `string` (= `char*`) are the according C types. Comment lines are valid in etc/hosts. They start with a hash (`#`) as the first character of the line and end with NL (new-line) or EOF (end-of-file).

- .<subject>.key

```
file ::= subject\n issuer\n rsa_n\n rsa_public\n rsa_private\n.
```

- NewYorkTimes

```
file ::= rsa_n\n rsa_public\n.
```

- etc/hosts

```
lines    ::= lines line | epsilon.
line     ::= quad hostname\n.
hostname ::= char*.
```

- log/<subject>

```
lines ::= lines line | epsilon.
line  ::= date time anything_that_makes_sense\n.
date  ::= dd.mmm.yy.
time  ::= hh:mm:ss.
```

- inbox/<ua>.dir

```
files    ::= files file | epsilon.
file     ::= filename filesize from dek_encr.
filename ::= string.
filesize ::= long\n.
from     ::= subject\n.
dek_encr ::= signature\n.
```

- inbox/<ua>.<filename>

```
contents ::= string.
```

# 5 Example

## 5.1 Configuration

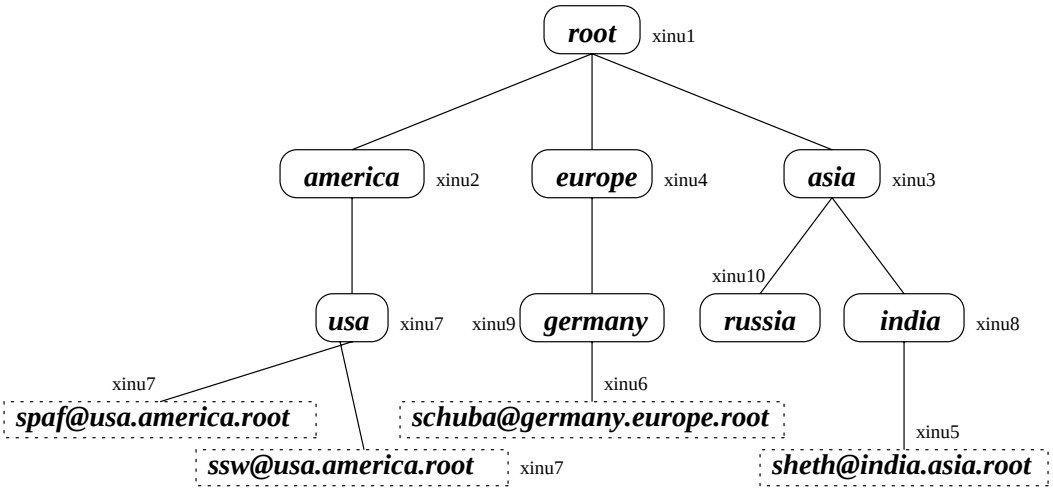


Figure 10: Configuration of test environment

```
#
# local etc/hosts database.
#
# format: data    : ^IPaddr hostname$
#         comments: ^#.*$
#
# root node
#
128.10.3.101   root
#
# first level nodes
#
128.10.3.102   america.root
128.10.3.103   asia.root
128.10.3.104   europe.root
#
# second level nodes
#
128.10.3.105   russia.asia.root
128.10.3.107   usa.america.root
128.10.3.108   india.asia.root
128.10.3.109   germany.europe.root
# EOF
```

Figure 10 depicts the configuration of our test environment. The previous file etc/hosts defines the hostname to IP address mapping for our certification authority graph.

Each certification authority node in the graph is labeled with a simple name without dots. The full certification authority name of any node in the graph is the sequence of labels on the path from that node to the root. Names are always read from the node towards the root (*up* the graph) and with dots separating the names in the path. User agent names are built the same way with the leading user name and an @ as separator. A typical user agent name would therefore be `schuba@germany.europe.root`.

Our test environment executes each certification authority on a different machine. Each of them listens on a well known service port. Note that this poses the restriction that at any given time a maximum number of one authority daemon can run on one particular host. This feature also holds for all well known UNIX services. Because we did not want to restrict the user agents to such a limited number, user agents bind to system supplied ports and register with their particular authority according to their name. In our implementation this registration is done in band for the ease of demonstration. However, in reality the registration procedure must happen out-of-band. Each certification authority maintains a list of currently registered users.

In our example we execute users `spaf` and `ssw` on the same physical host. The following two paragraphs contain the real runtime logs of a session. The test run includes:

- start up of all certification authorities.

- registration of all user agents.
- sending a file from user agent `schuba` to user agent `sheth`.
- retrieving of the file by user agent `ssw`.
- taking down a node and restarting it to demonstrate our fault tolerant approach.

The time stamps provide help in tracing what exactly happened. The last paragraph in this section contains the configuration files and the at runtime generated files for key storage.

### 5.2 Logs of Certification Authorities

The test data for our project that is not online is appended to this document.

### 5.3 Logs of User Agents

See previous section.

### 5.4 Configuration Files and Runtime Generated Files

See previous section.

## 6 Tests

This section describes our testing efforts during the project development and after the implementation of each module.

### 6.1 Design Principles

We would like to stress that we consider our bottom up technique of development as part of the testing process. We split the project up into modules like data structure handling, name binding, DES, MD5, RSA, MPLIB, system logging, communication subsystem, and others. The code for each module resides in a different directory with it's own makefile. It was therefore easy to incorporate main programs that test only certain modules. These main programs were modified while the project grew to test each newly implemented feature and in some cases the interaction between several modules, like in the case of the list and element handling.

The testing of the collection of modules in interaction with each other is very runtime dependent. It is difficult to document it accordingly. We therefore consider as one example for the testing the printouts of the previous section 5 that demonstrate many notions of the functionality of our system.

One of the primary design criteria for distributed systems is the amount of fault tolerance. We build into our system the ability to restart nodes that are crashed with previously calculated keys to make the rest of the system unaware of the temporary loss of functionality. It is clear that services that rely on a node that is down cannot be provided, but once the node is up again, except for a few exceptions, further authentication validations can be done without any noticeable difference.

## 6.2 Prime Generation

We implemented a prime number generator and also a prime testing routine based on the proababilistic approach but needed a means of verifying that first, our routines functioned as desired and second, the primes that we generated were most likely prime. Therefore, we thought of utilizing an existing math package. We used the *maple* number theory package, developed at the University of Waterloo.

We wrote the following shell script to generate a number of primes with our routine in sequential order and then store these numbers in a format that could be interpreted by *maple*. *Maple* then used its `isprime()` funtion to verify each of the previously generated primes as probably prime.

```
#!/usr/local/bin/tcsh
#
# Generate a bunch of prime numbers and write in maple program like format
#

echo > aaahhh
set i = 1
while ($i != 200)
  echo p:= >> aaahhh
  mpprimetest >> aaahhh
  echo ':' >> aaahhh
  echo 'isprime (p);' >> aaahhh

  set i = `expr $i + 1`
end
```

The results that we obtained were superb! All the prime numbers that were generated by our program were accepted as being most likely prime. *Maple* uses [Knu81, §4.5.4, Algorithm P] for primality testing.

Therefore we established a high level of trust in the functionality of our prime generation as well as our prime testing routine.

## 6.3 DES, MD5, MPLIB

The modules DES, MD5 and MPLIB were largely obtained from different sources on the Internet. DES was found online on the Department of Computer Science at Purdue University. The MD5 implementation was taken out of the defining RFC and ran after a bug that was present in the standard was fixed. The module MPLIB was obtained from Dr. S. Wagstaff, Jr. and modified by us to have a more modular structure. We separated header file and the rest of the library of functions and linked all object codes into a UNIX library that can now easily be bound to any program requiring multi precision integer arithmetic routines. We augmented this library by functions for efficient prime generation and primality testing, as further described in the previous paragraph 6.2, calculating inverses in modular arithmetic, and further useful functions.

## References


[CCI88] CCITT. *Recommendation X-509 The Directory Authentication Framework.* CCITT, 1988.

[Den82] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley Publishing Company, Inc., 1982.

[Ken93a] Stephen T. Kent. Internet Privacy Enhanced Mail. *Communications of the ACM*, 36(8):48–59, May 1993.

[Ken93b] Stephen T. Kent. *RFC-1422 Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management.* Network Working Group, February 1993.

[Knu81] Donald E. Knuth. *The Art of Computer Programming.* , volume 2. Addison-Wesley Publishing Company, Inc., second edition, 1981.

[NBS77] NBS. Data Encryption Standard. National Bureau of Standards, Washington D.C., Jan. 1977. FIPS PUB 46.

[Riv92] Ronald L. Rivest. *RFC-1321 The MD5 Message-Digest Algorithm.* Network Working Group, April 1992.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2):120–6, February 1978.

[Sch94] Bruce Schneier. *Applied Cryptography.* John Wiley & Sons, Inc., 1994.

[Tru93] Trusted Information Systems, Incorporated. *TIS/PEM User's Guide*, 6.0.1 edition, June 1993.