

## Design and Implementation of Three Cryptographic Systems:

Diffie-Hellman Key Exchange  
Diffie-Hellman Digital Signature  
RSA Encryption/Decryption

Lewis Baumstark  
Murat Guler  
Peter Sassone

ECE 8823  
April 25, 2002

## Diffie-Hellman Key Exchange

Diffie-Hellman is not an encryption or signature technique, but rather a method for two users (Alice and Bob) to jointly construct a shared key without an intermediary (Oscar) obtaining that key or information that would let him construct the key easily.

### Algorithm

The procedure is fairly straightforward.

1. Either Alice or Bob chooses a large prime number  $p$  and a generator  $g$  for that prime, which are shared publicly with the other user and consequently, with anyone observing (Oscar).
2. Alice and Bob independently and privately choose random numbers ( $a$  and  $b$  respectively) which are between 1 and  $p-1$ .
3. Both users separately compute  $g$  raised to the power of their random number ( $g^a$  or  $g^b$ ) and mod by  $p$ .
4. Both users now exchange those values over the public network. Alice now has  $g^b \bmod p$  and Bob now has  $g^a \bmod p$ . Oscar, of course, has seen both.
5. Alice and Bob now raise the value they just received to the power of their random number. So Alice computes  $(g^b)^a \bmod p$  and Bob computes  $(g^a)^b \bmod p$ . Because of the commutability of exponentiation, both will end up with the same number.
6. We call this result  $k$ , the shared key.

Oscar sees both  $g^a$  and  $g^b$ , yet has no polynomial-time way of determining the exponents  $a$  and  $b$  from either of these numbers, or a way of constructing  $(g^a)^b \bmod p$ . Thus Alice and Bob now share a secure key which can be used for secure communication or any other use they see fit.

The man-in-the-middle attack, however, can break this system. If Oscar intercepts the  $g^a$  and  $g^b$  transmissions and replaces them with his own value, Alice and Bob will agree on different keys. Oscar can then intercept any messages sent between them using those keys and decrypt them. He could also possibly modify the message, re-encrypt and send the message along. This exploitation is present because simple DH key-exchange does not include authentication. Diffie-Hellman digital signatures, addressed in the next section, address this issue.

### Implementation

This project presents C++ classes which implement the above algorithm.

Each user is represented by a thread of execution. The exchange of values is done via synchronization primitives on shared variables. Inspection of the code shows that the threads do not 'cheat' and look at the other thread's values.

Typical output looks like:

```

Finding prime p.....
Finding generator g...
Common prime p chosen as 4158841839
A generator, g, for the prime is 2907825353
Client with ID 1026 is online
Client1026 chose 'a' as 3395547715
Client with ID 2051 is online
Client1026 calculated  $g^a = 269270990$ 
Client2051 chose 'a' as 2801823619
Client2051 calculated  $g^a = 844586073$ 
Client2051 got back  $g^b$  as 269270990
Client2051 produced key  $(g^b)^a = 3789380176$ 
Client2051 finished
Client1026 got back  $g^b$  as 844586073
Client1026 produced key  $(g^b)^a = 3789380176$ 
Client1026 finished
Done

```

The algorithm works correctly if both threads (1026 and 2051) agree on the produced key (in this case, 3789380176). As one can see, this example uses 32-bit precision. Arbitrarily high precision is also supported, though it is much slower.

Prime numbers are found using the Miller-Rabin testing algorithm. Generators are found by only using primes  $p$  of the form  $p = 2q + 1$  where  $q$  is another prime, and picking random  $g$  values until both  $g^2$  and  $g^q$  are not equal to 1 mod  $p$ . Each period in the “Finding prime  $p$ ” line above represents the number of primes that were created until one of the form  $2q + 1$  was created. Each period in the “Finding generator  $g$ ” line represents the number of random  $g$  values that were picked until a generator was found.

### Test Cases

Table I.

Test	P	G	a	b	$K_{hand}$	$K_{prog}$
1	59	47	41	29	<b>58</b>	58
2	23	17	13	7	<b>14</b>	14
3	<b>47</b>	<b>29</b>	<b>43</b>	<b>29</b>	20	20

As shown, the routine produces the correct  $k$  value for each test case.

## Diffie-Hellman Digital Signature

### Prime Number Generation

The prime number generator is responsible for generating the prime numbers with the given PRIME\_NUMBER\_SIZE, which happens to be half the amount of PRECISION, which at the moment is defined in the li\_math.h file to be 128. The generator subroutine first obtains a random large integer, then tests it 100 times using the Miller-Rabin Primality test. Any large integer which passes this extensive testing is output as a prime number.

### Diffie-Hellman Signature Protocol

By itself, Diffie-Hellman Key-Exchange protocol is vulnerable to man-in-the-middle attacks. To prevent against that, we have to employ a signature strategy, where each user authenticates itself securely, and verifies each other's identity. One way to do it is to use the Simplified version of the Station-to-Station Protocol [1].

The Station-to-Station protocol starts with each user U and V having acquired a prime value p, and a generator alpha for that prime's  $Z_p^*$ . When the algorithm begins, each user picks up a unique number a that should be between 0 and p-2 inclusive. Then user U computes:

$$(\alpha^{aU}) \bmod p$$

and sends it to the user V. V on the other hand, computes:

$$(\alpha^{aV}) \bmod p.$$

After that he computes his private key K:

$$K = (\alpha^{aU})^{aV} \bmod p.$$

V also computes his signature. In order to do that, he uses both  $(\alpha^{aU})$  and  $(\alpha^{aV})$ .

$$yV = \text{sigV}(\alpha^{aV}, \alpha^{aU}).$$

Where sigV is the signing algorithm. In addition to this, V calculates his certificate C(V), which consists of:

$$C(V) = (\text{ID}(V), \text{verV}, \text{sigTA}(\text{ID}(V), \text{verV}))$$

ID is the unique id number that each user has. VerV is the verification algorithm that user U uses to verify the signature sigV. SigTA(ID(V), verV) is the number obtained from a central Trusted Authority (in our case, the Verisign global object), which is the

signed format of ID and verV. To this certificate C(V), V adds its own  $(\alpha^aV)$  and  $yV$ , then sends the whole thing to U.

U, upon receipt of the certificate structure from V, verifies the authenticity of C(V) by using the SigTA part of the certificate. After that, U verifies  $yV$  using verV that was transmitted with the certificate. At this point, being certain of the authenticity of data transmitted from V, U computes its own K value:

$$K = (\alpha^aV)^{aU} \text{ mod } p.$$

After this, U computes its own  $yU$ :

$$yU = \text{sigU}(\alpha^aU, \alpha^aV).$$

U also computes C(U) in the same way described above, and then transmits it along with  $yU$  to V. Upon receipt, V verifies C(U), and then verifies  $yU$  using verU. This way, V has also verified that it has been communicating with U, and U has been receiving the correct messages from V. Upon completion of the both algorithms, the value of K retained by both U and V should be exactly same. No other party has any idea what K could be, and no other party could have interfered with the transmission, because of the authentication schemes used.

The signing algorithms sigV and sigU are functions which take in two values and output one encrypted value. In our case, we decided to use the RSA algorithm for signature scheme, since we already had it developed. However RSA signature algorithm takes in only one value as input. Therefore, we had to combine the values  $(\alpha^aU)$  and  $(\alpha^aV)$  in a suitable fashion.

The verification algorithms verV and verU transmitted with the certificates consist of the public key b and the modulo base n used by the RSA encryption engines of the users V and U. Therefore in our case, a certificate is actually like:  
 $C(U) = (ID(U), bU, nU, \text{sigTA}(ID(U), bU, nU)).$

### Test Cases

The following table verifies the operation of the Diffie-Hellman Signature scheme, with hand calculated values:

Tables II and III, for U:

Test	prime p	alpha	n	a	b	id
1	227	223	143	103	7	1001
2	37	17	143	103	7	1001
3	59	47	143	103	7	1001

<b>Test</b>	<b>aU</b>	<b>(alpha)^aU</b>	<b>yU</b>	<b>K</b>
1	39	153	135	40
2	3	29	101	10
3	131	37	78	49

Tables IV and V, for V:

<b>Test</b>	<b>prime p</b>	<b>alpha</b>	<b>n</b>	<b>A</b>	<b>b</b>	<b>id</b>
1	227	223	55	37	13	1007
2	37	17	55	37	13	1007
3	59	47	55	37	13	1007

<b>Test</b>	<b>AV</b>	<b>(alpha)^aV</b>	<b>yV</b>	<b>K</b>
1	90	76	4	40
2	8	33	17	10
3	104	41	23	49

---

## References

1. Stinson, Douglas R., *Cryptography: Theory and Practice*, CRC Press, Boca Raton, 1995, p. 272.

## RSA Encryption and Decryption

In this project, for encryption and decryption of text messages and for Diffie-Helman digital signing and verification, the RSA cryptographic system is used. An RSA system can be defined as  $(K, e_K(x), d_K(y))$  with key  $K$  defined as:

$K = (n, p, q, a, b)$  such that

$n = p * q$ ,  $p$  and  $q$  different primes, and

$ab = 1 \pmod{\Phi(n)}$ .

The encryption and decryption functions  $e_K(x)$  and  $d_K(y)$ , respectively, are defined as:

$e_K(x) = x^b \pmod{n}$

$d_K(y) = y^a \pmod{n}$ .

This being a public-key encryption system, the public key is given as  $(n, b)$  while  $p, q$ , and  $a$  are kept secret.

This project presents a C++ class, `RSA_system`, that implements the creation of an RSA system (i.e., generating values for  $p, q, n, a$ , and  $b$ ) and that provides a decryption method for returning the public key values  $n$  and  $b$ , and for performing decryption while hiding the values of  $p, q, n$ , and  $a$ . The set-up (implemented in the `RSA_system` constructor) is as follows:

1. Generation of two random primes  $p$  and  $q$ . The Miller-Rabin primality test was implemented (as described in section ? of this report) to insure that  $p$  and  $q$  are prime.
2.  $n$  calculated as  $p * q$ .
3.  $\Phi(n) = (p-1)(q-1)$  calculated.
4. A random number  $b$ , such that  $1 < b < \Phi(n)$ , is chosen.
5.  $a = b^{-1} \pmod{\Phi(n)}$  is computed.

At this point, a user may request  $n$  and  $b$  via the `RSA_system::b_key()` and `RSA_system::n_key()` methods, respectively. A decrypted message may be obtained by calling the `RSA_system::decrypt(longint y)` method.

For encryption, the function `encrypt(longint n, longint b, longint x)` returns the encryption of  $x$  using public key  $(n, b)$ . It is not part of `RSA_system` class in order to remain general for any instance of an RSA system.

## Test Cases

Table VI gives 3 encryption test cases for a sample RSA system, comparing the results of the program with those obtained by hand calculations. Table VII gives the decryptions for the encryption cases above, both for the program and hand-calculated results.

Table VI. Encryption Test-Cases

Test	x	b	n	$y_{\text{program}}$	$y_{\text{hand-calc}}$
1	100	13	391	87	87
2	71	13	391	165	165
3	223	13	391	49	49

Table VII. Decryption Test-Cases

Test	y	a	n	$x_{\text{program}}$	$x_{\text{hand-calc}}$
1	87	325	391	100	100
2	165	325	391	71	71
3	49	325	391	223	223

As shown, the encryption and decryption routines perform correctly for the given test cases. The following is a program output listing for a test case with a larger number:

```
$ ./rsatest.exe
Plaintest? (less than 2^128)
123456789012345678901234567890
RSA: Initializing MP integers...
RSA: Finding prime p...
932469648110791547
RSA: Finding prime q...
1033504800643134977
RSA: Calculating n = p*q...
963711857776517841314857615031639419
RSA: Calculating phi(n) = (p-1)*(q-1)...
963711857776517839348883166277712896
RSA: Finding b such that gcd(b,phi(n))=1 --
1418227916859036949
RSA: finding a = b^(-1) mod phi(n)...
463464869620517003929737175919345213
RSA: Finished building RSA, cleaning up.
Generated plainnum = 123456789012345678901234567890
Encryption: y = x^b mod n...
Ciphernum = 82362662845077651455620486062105588
RSA: Decrypting as x = y^a mod n...
Decrypted plainnum = 123456789012345678901234567890
```

As shown, the system works for larger numbers, up to 128 bits, as well. A simple re-configuration will allow even larger numbers, at the expense of slower performance.