Chapter 8 Practical Public-Key Security and Digital Signatures

8.1 Security with Public-Key Technology

8.1.1 Key Distribution Problems and Public-Key Encryption
8.1.2 Data Integrity, Digital Signature, and Non-repudiation

8.2 The Diffie-Hellman Key Exchange Scheme

8.2.1 The Diffie-Hellman (DH) Key Exchange
8.2.2 Using Arbitrary Precision Mathematics (APM) Package
8.2.3 The Discrete Logarithm Problem and Brute-Force Attack

8.3 The ElGamal Public-Key Algorithm and Digital Signatures

8.3.1 The ElGamal Public-Key Algorithm
8.3.2 ElGamal Encryption/Decryption with APM Package
8.3.3 Implementing Message Encryption/Decryption
8.3.4 Using ElGamal Scheme For Digital Signature and Data Integrity
8.3.5 Implementation of the Digital Signature Scheme

8.4 The RSA Scheme, Digital Signature and Hybrid Encryption

8.4.1 The RSA Public-Key Algorithm and Challenge
8.4.2 Generating RSA Public and Secret Keys
8.4.3 Message Encryption/Decryption with RSA Scheme
8.4.4 Sending and Receiving Secure Message with RSA Digital Signature
8.4.5 Building a Hybrid Encryption Scheme: RSA + AES

8.5 Elliptic Curves and Public-Key Encryption/Decryption

8.5.1 What are Elliptic Curves?
8.5.2 Elliptic Curve Cryptography (ECC)
8.5.3 Adding Two Points on an Elliptic Curve
8.5.4 Scalar Multiplication and Generating the Keys for ECC
8.5.5 Encryption/Decryption Using Elliptic Curves

8.1 Security with Public-Key Technology

8.1.1 Key Distribution Problems and Public-Key Encryption

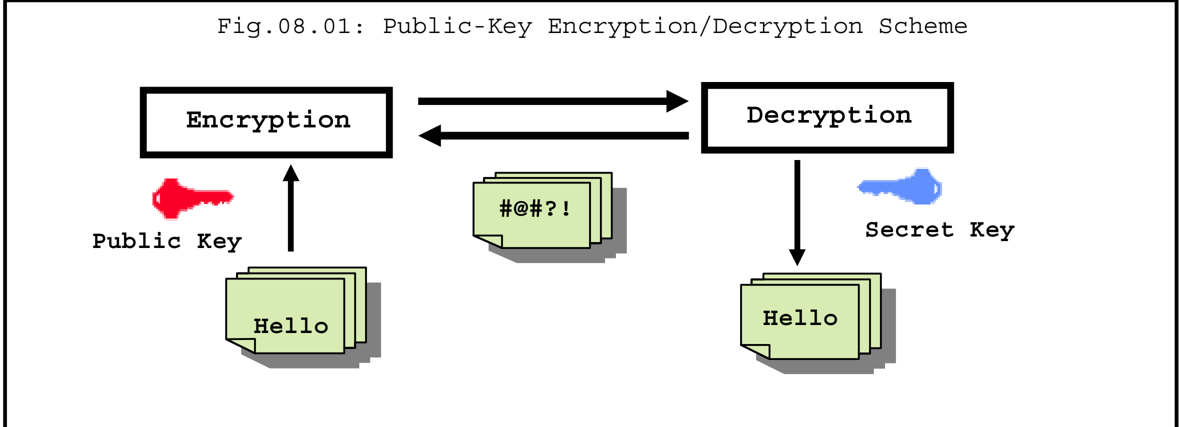
Conventional encryption methods such as symmetric-key only use one key for encryption and decryption. The sender encrypts the message or document with this key and sends it to the receiver. In order to decrypt this document, the receiver has to have the same key. The advantage of these types of algorithms is that they are fast, efficient and computationally safe.

If you want to share the encrypted message with someone, you may have to give up the secrecy of your password (or Key) so that your message can be properly decoded by the person you trust. This may create the so-called key distribution problems and compromise the data security. If the encrypted message only shared with one or two parties, exchange passwords or keys can be done in a strictly private manner and may not be a problem. The key distribution problem or key management would soon turn out to be a nightmare when more parties, say 40, are involved.

The major problem of symmetric-key encryption is that if somebody else does have the key, the entire security will be compromised. The use of the so-called “Public-Keys” can provide a solution to this problem. Instead of using one key, public-key is a concept where two keys are involved. One key is called public-key which can be spread by all means and can be obtained by anyone. Many people suggested that public-key server should be involved to distribute them. The other key is called the secret-key (or private-key) and should be kept secret by the owner permanently. Any good public-key encryption scheme should be computationally infeasible (or difficult) to derive the secret key from the public one or vice versa.

The public-key scheme operation is simple. When a message is encrypted with one key, the other key must be used to decrypt the message. An example is demonstrated in Fig.08.01.

Fig.08.01: Public-Key Encryption/Decryption Scheme



Suppose you want to send a sensitive message to Bob who has a public key called Bob_P. All you have to do is to obtain this key. Then apply this key to encrypt the message you want to send. The result is an encrypted message which can only be decrypted by using Bob’s secret key such as Bob_S. If the keys Bob_P and Bob_S are totally different and one cannot be obtained by the other, the key distribution problem is solved. It is because Bob doesn’t need to distribute his secret at all.

When use properly, this public-key structure can generate trust so that dedicated message can only be read by the intended recipient. In fact public-key technology (or infrastructure) not only solve the key-distribution problem, it creates an entire new chapter on security applications forming a solid foundation of modern cryptography.

8.1.2 Data Integrity, Digital Signature and Non-repudiation

Conventional encryption with one key is not easy to perform data integrity and verification. For example, how your partner can identity that the message was really sent by you. For a simple example, if you send a message to the bank, how does the bank manager determine the message was really sent by you? How does the bank know that the message sent by you hasn't been attacked by a hacker to transfer money to his/her account? Even you know how to protect the integrity of your message by using the hash-functions in chapter 3, a capable hacker can hack your message; change it, produce a new hash-value, and delivery them to your bank manager. Apart from the message confidentiality, public-key techniques also provide solutions to these security problems namely:

- Digital Signature - Proof of data and/or entity identification
- Non-repudiation - Protection against falsely denying
- Data Integrity - Make sure no alteration can be made

One of the great features of public-key technique is that it can be used to identify the message sender. This process is generally referred as digital signature.

For example, suppose the public-private key pairs of Bob and Sue are (Bob_P, Bob_S) and (Sue_P, Sue_S) respectively. When Sue wants to send Bob an important message, she can encrypt the message using her private key first and then with Bob's public key. The result is a double encrypted message ready to send to Bob. When Bob gets the message, he can use his own private key and then Sue's public key to decrypt it. When the message is successfully decrypted by Sue's public-key, the message must be sent by Sue herself. When Sue encrypts the message using her private-key, it is like to sign the message with her signature and therefore the process is called "Digital Signature". This process is demonstrated in Fig.08.01.

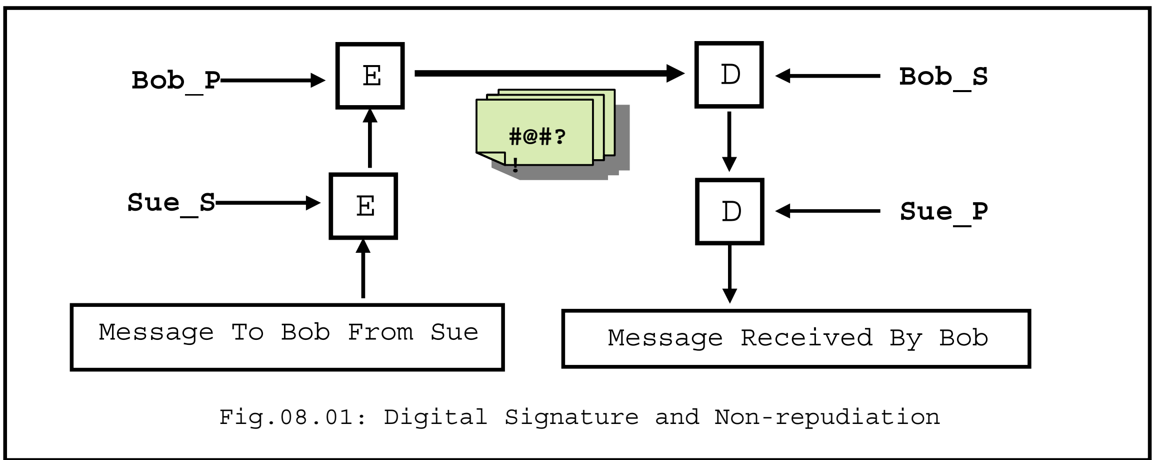


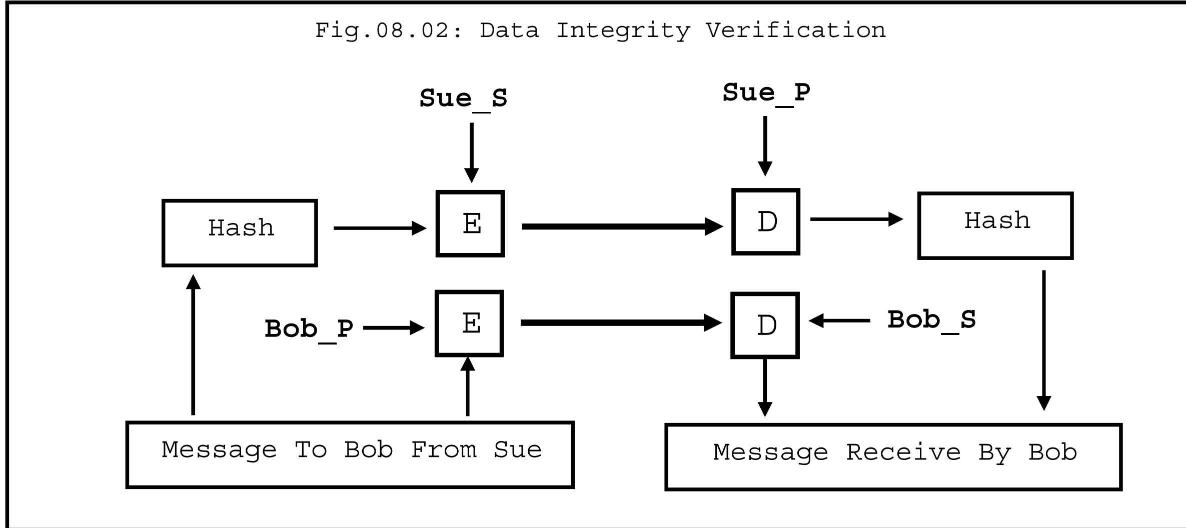
Fig.08.01: Digital Signature and Non-repudiation

In fact, Sue doesn't need to encrypt the entire message using her private-key. She can encrypt part of the message or some personal information to identify her with the private-key instead. This process can also be used for Non-repudiation. In other words, it will stop the fraudulent claim from Sue that she didn't send the message.

Another question is how to maintain the data integrity that you have sent. In other words, how can you verify the messages you send to the bank haven't been modified or replaced (cut and paste attack) by an intruder. Even if you generate the hash-value (or hash) of the document, a capable hacker can also replace yours with his generated hash. An un-protected hash is a dangerous move.

However, when combined with public-key encryption, data integrity is greatly enhanced. The data integrity verification process is illustrated in Fig.08.02.

Fig.08.02: Data Integrity Verification



From chapter 3, hash functions can produce a summary of a message called hash representing a signature only good for that particular message. When hash of a document is protected by public-key encryption, data integrity of the document is maintained.

Suppose Sue wants to send Bob an important message and protect the contents against alterations at the same time. She can

- Obtain the hast value of the message.
- Encrypt the hash with Sue's secret-key Sue_S.
- Encrypt the message using Bob's public-key Bob_P

The results are sent to Bob separately. When Bob got the encrypted message and hash, he can obtain the originals by decryption and perform data integrity test as follows:

- Using the key Sue_S to get the hash of the message
- Using the key Bob_P to obtain the message
- Using the hash to verify the integrity of the message

If the received message produces the same hash, the data integrity of the message is intact. In fact, the data integrity verification process is similar to verify the signature (secret-key) of the message.

Since the first version of public-key commercial product called Pretty Good Privacy (PGP) at the beginning of 90s, public-key encryption technologies are very popular. In fact, we have a standard for it. It is called the IETF - RFC2440 or known as Open Pretty Good Privacy (OpenPGP). Based on this standard specification, a freely available public-key security product called Gnu Privacy Guard (GnuPG) is developed for everyone wants to use it. Both PGP and GnuPG will be discussed in next chapter in details. In particularly, we will discuss how to use these two products to protect information communication over the WWW and/or Internet. To enhance our background on public-key security, let's consider some popular public-key encryption/decryption algorithms.

8.2 The Diffie-Hellman Key Exchange Scheme

8.2.1 The Diffie-Hellman (DH) Key Exchange

One of the public-key ideas is from the Diffie-Hellman (DH) key exchange created by Whitfield Diffie and Martin Hellman in the middle of 70s. The DH key exchange scheme would allow two people to share a secret key over an open or un-secure channel such as Internet or WWW. The scheme is based on the following elementary number theory.

Let p be a prime number, the set Zp={0,1,2,..,p-1} is a finite field. This set can be considered as the remainders of any integer divided by p. The non-zero elements Zp* of Zp form a group under multiplication modulo p, i.e.

$$Zp^* = \{1,2,..,p-1\}$$

This set has p-1 elements. An element g (1< g <p) is said to be the generator of Zp* if Zp* can be generated by the powers of g, i.e.

$$Zp^* = \{1,g^2,g^3,..,g^{p-2}\} \bmod p$$

For example, if p = 11, we have $Z_{11}^* = \{1,2,..,10\}$. Also, it is easy to see that 2 is a generator of Z_{11}^*

$$Z_{11}^* = \{1,2,2^2,...,2^9\} \bmod 11 = \{1,2,4,8,5,10,9,7,3,6\}$$

In other words, the number 2 generates the complete set of Zp*. Other numbers 3, 4, and 5 are not generators. Numbers 6 and 7 are again generators. Generators sometimes are called primitive roots of Zp^* and can be used to construct many security applications related to Zp*.

Given an integer a, the smallest integer n such that

$$1 = a^n \bmod p$$

is called the order of a(mod p). If the order of "a (mod p)" is p-1, then integer a is a generator of Zp^*. With this mathematical background, the DH key exchange can be described as follows:

The DH Key Exchange Scheme

Suppose Bob and Sue want to share a secret key K using the DH key exchange. The process are:

- Select a "large"' prime number p and a generator 1 < g < p-1 of Zp*.
- Bob picks a random secret integer x, 1 < x < p-1, compute the following result
  $X = g^x \bmod p$
- The secret key is x and publish the public key as (X, g, p), i.e.
  Bob_S = x, Bob_P = (X, g, p)
- Sue selects a secret number y, 1 < y < p-1 and compute the result of
  $Y = g^y \bmod p$

- The secret-key for Sue is y and the public-key is (Y, g, p)

```
Sue_S = y,    Sue_P = (Y, g, p)
```

- When Sue receives Bob's public-key Bob_P = (X, g, p), the shared secret-key can be compute by using Sue's secret-key y or

$$K = X^y \bmod p = (g^x)^y \bmod p$$

- When Bob receives Sue's public-key Sue_P = (Y, g, p), the shared secret-key can be compute by using Bob's secret-key x:

$$K = Y^x \bmod p = (g^y)^x \bmod p$$

In this case, the shared-key is K. In other words, key K is shared for communication by Bob and Sue over an in-secure environment (see Fig.08.03).

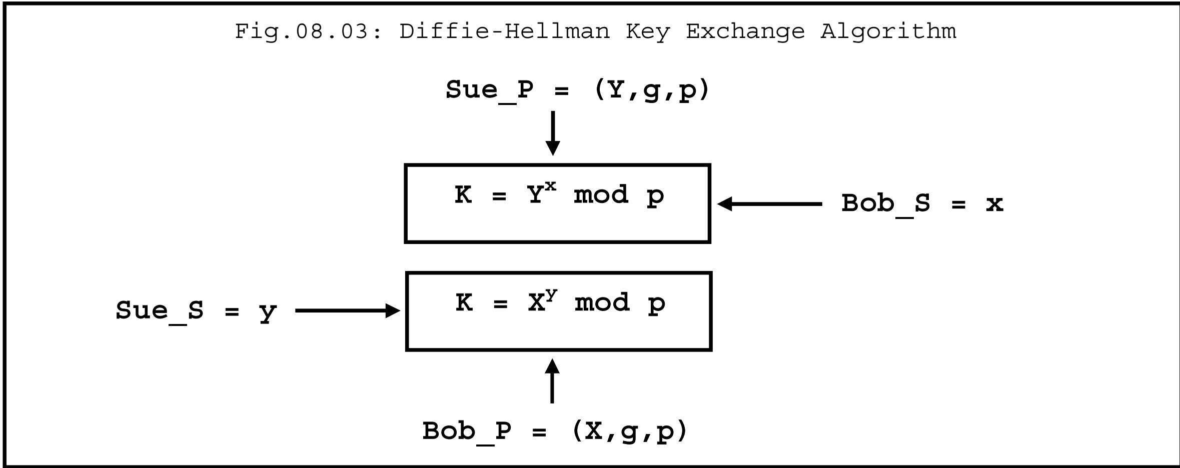


Fig.08.03: Diffie-Hellman Key Exchange Algorithm

Example: Consider a small prime p=73699 with generator g=2. Bob chooses x=137 for his secret-key. The public-key is computed by:

$g^x = 2^{137} \bmod 73699$

= 174224571863520493293247799005065324265472 mod 73699

= 31072 mod 73699
=> Bob_P=[X,g,p] =[31072,2,73699)

Sue picks y=193 as her secret-key and the public-key is:

$g^y = 2^{193} \bmod 73699$

= 12554203470773361527671578846415332832204710888928069025792 mod 73699

= 19048 mod 73699
=> Sue_P=[Y,g,p]=[19048,2,73699]

If Sue wants to send a secret message to Bob, she can compute the key K by:

$K= (31072)^{193} \bmod 73699$
$= 61344 \bmod 73699$

When Bob receives the message encrypted by K from Sue, he can compute the key K by:

```
K= (19048) ^137 mod 73699
 = 61344 mod 73699
```

The shared-key can be used to decrypt the message. One of the interesting features of this key exchange setting is that the algorithm is expandable. To see this, suppose Mary wants to join the shared-key scheme with Bob and Sue. All she needs to do is to pick her secret-key and calculate her public-key:

```
Mary_S=167 (say)
2^167 mod 73699 = 70188 mod 73699
Mary_P = [70188,2,73699]
```

The combination of all shared-key are shown in the Table 08.01.

Table 08.01: Combination of Shared-Keys of Bob, Sue, and Mary

The DH Key Exchange Scheme	Bob_S=(137) Bob_P=(31072,2,73699)	Sue_S=(193) Sue_P=(19048,2,73699)	Mary_S=167 Mary_P=(70188,2,73699)
Bob_S=(137) Bob_P=(31072,2,73699)	9314 mod 73699	61344 mod 73699	71352 mod 73699
Sue_S=(193) Sue_P=(19048,2,73699)	61344 mod 73699	66969 mod 73699	65763 mod 73699
Mary_S=167 Mary_P=(70188,2,73699)	71352 mod 73699	65763 mod 73699	59364 mod 73699

The 1st row and column are the DH keys of Bob, Sue, and Mary. The rest are the corresponding shared-key among them. From this example, we can see the shared-key between any two people are different. Also, the scheme is fully expandable. In other words, new members can be added easily.

The implementation of the algorithm may look straight forward and involving simple power modulo operation "$a^x \bmod p$". For a small x and prime p, the operation is trivial. For a big integer x and a large prime p, the operation can be troublesome. For example, consider the number of digits of the following numbers:

2^{137} has 42 digits 2^{193} has 59 digits 2^{167} has 51 digits

Not many computer systems can handle these big numbers directly. In a normal circumstance, the largest number in memory of a 32 bit computer is $2^{32}=4294967296$ (10 digits). Anything more than that needs additional treatment. Also, it is obvious that for a secure scheme, the prime p must be big enough to discourage the Brute-Force attackers.

In order to handle big numbers or the precision implementation problems, software packages such as Arbitrary Precision Mathematics (APM) are needed.

8.2.2 Using Arbitrary Precision Mathematics (APM) Package

In this section, we consider how to use an "Arbitrary Precision Mathematics" (APM) package to implement the DH scheme. The package is called "Big Integer Library". It is a script based library

(BigInt.js) freely available for everyone. A copy of BigInt.js can be downloaded from the site accompanying with this book or search engine with key-word BigInt.js.

From operational point of view, the Diffie-Hellman scheme has two parts. The first part is to generate and publish the public-key. The second part is to compute the shared-key for encryption/ decryption. To generate the DH public-key is simple, all you have to do is to

- Select a prime `p` and a generator `g`.
- Pick a random number `y` as your secret-key
- Compute and Publish the public-key (`Y,g,p`) where $Y = g^y \bmod p$

Since we are going to use an APM package, the calculation of a^y mod p is no longer an implementation problem. Consider the following page:

```
Example: ex08-01.htm – Generating The Diffie-Hellman Public-Key

 1: <head><title>The DH Key Exchange Scheme</title></head>
 2: <style>
 3:  .butSt{font-size:16pt;width:70px;height:35px;
 4:           font-weight:bold;background:#ddffdd;color:#ff0000}
 5:  .txtSt{font-size:14pt;width:240px;height:35px;font-weight:bold;
 6:         background:#dddddd;color:#ff0000}
 7: </style>
 8: <body style="font-family:arial;font-size:20pt;text-align:center;
 9:       background:#000088;color:#ffff00">
10:  Generating Diffie-Hellman Public-Key <br /> Y =(g) <sup>y</sup> mod p
11: <br /><img alt="pic" src="line1.gif" height="7" width="650" /><br />
12: <form action="">
13: <table style="font-size:16pt;width:560px" cellspacing="5" align="center">
14:  <tr><td>The Secret-Key: y</td><td >
15:           <input type="text" id="sec_y" size="32" maxlength="32"
16:            class="txtSt" value="137" /></td></tr>
17:  <tr><td>The Generator g: </td><td >
18:          <input type="text" id="gen_g" size="32" maxlength="32"
19:           class="txtSt" value="2" /></td></tr>
20:  <tr><td>The Prime p: </td><td>
21:          <input type="text" id="pri_p" size="32" maxlength="32"
22:           class="txtSt" value="73699" /></td></tr>
23:  <tr><td>The Public-Key (Y,a,p): </td><td>
24:          <input type="text" id="result" size="32"
25:           maxlength="32" readonly class="txtSt" value="" />
26:          <input type="button" class="butSt" value="OK"
27:           onclick="dh_key()" /></td></tr>
28: </table></form>
29: <script src="BigInt.js"></script>
30: <script>
31:  function dh_key()
32:  {
33:    var secret_y, public_Y,gen_g,prime_p;
34:    ty = document.getElementById("sec_y").value;
35:    tg = document.getElementById("gen_g").value;
36:    tp = document.getElementById("pri_p").value
37:    secret_y = str2bigInt(ty,10,0);
38:    gen_g    = str2bigInt(tg,10,0);
```

```
39:    prime_p  = str2bigInt(tp,10,0);
40:
41:    powMod(gen_g,secret_y,prime_p);
42:    llst = bigInt2str(gen_g,10)+","+tg+","+tp;
43:    document.getElementById("result").value = llst;
44:  }
45: </script>
46: </body>
```

This page contains 4 text boxes and one push button. The first text box in lines 15-16 is to get the user specified secret-key `y`. The next 2 boxes are for the generator `g` and prime `p`. When the OK button in line 26 is clicked, the function `dh_key()` is executed. This function computes the public-key (`Y,g,p`) and displays it in the 4th text box (see lines 24-25).

To use the APM library, the script file BigInt.js is included into the page at line 29. The values of secret-key `y`, generator `g`, and the prime `p` are captured by the statements in lines 34-36. Since these values are base 10 format and represented as strings, the function `str2bigInt()` are used to convert them into big numbers. Now, variables `secret_y`, `gen_g`, and `prime_p` are big numbers in APM format. Consider the function in line 41:

```
powMod(gen_g,secret_y,prime_p);
```

This function computes the value $(gen_g)^{secret_y} \bmod (prime_p)$ which is the component of the DH public-key. The result is stored back to the first variable `gen_g`. Variable `gen_g` is a big number and therefore the function `bigInt2str()` in line 42 is needed to convert it to string. When all the public-key components (`Y,g,p`) are composed to variable `llst` in line 42, this variable is displayed to the 4th text box by the statement in line 43. A screen shot of this example is shown in Fig.08.04.

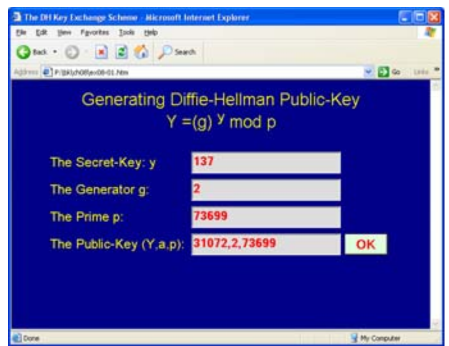


Fig.08.04: The DH Public-Key

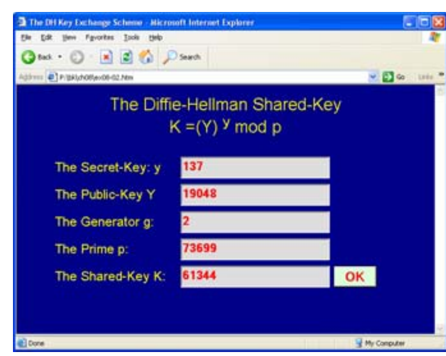


Fig.08.05: The Shared-Key

The second part of the DH scheme is to compute the shared-key `K`. This key is used to perform encryption/decryption. The value of `K` is determined by

- My Secret-key `y`
- Someone's Public-key (`Y,g,p`)
- Shared-key formula: $K = Y^x \bmod p$.

Again, with the APM package BigInt.js, the shared-key formula can be calculated easily. Consider the following example:

```
Example: ex08-02.htm - Compute The DH Shared-Key

 1: <head><title>compute The DH Key Shared-Key</title></head>
 2: <style>
 3:  .butSt{font-size:16pt;width:70px;height:35px;
 4:          font-weight:bold;background:#ddffdd;color:#ff0000}
 5:  .txtSt{font-size:14pt;width:240px;height:35px;font-weight:bold;
 6:         background:#dddddd;color:#ff0000}
 7: </style>
 8: <body style="font-family:arial;font-size:20pt;text-align:center;
 9:       background:#000088;color:#ffff00">
10:  The Diffie-Hellman Shared-Key <br /> K =(Y) <sup>y</sup> mod p
11: <br /><img alt="pic" src="line1.gif" height="7" width="650" /><br />
12: <form action="">
13: <table style="font-size:16pt;width:560px" cellspacing="5" align="center">
14:  <tr><td>The Secret-Key: y</td><td >
15:          <input type="text" id="sec_y" size="32" maxlength="32"
16:           class="txtSt" value="137" /></td></tr>
17:  <tr><td>The Public-Key Y</td><td>
18:         <input type="text" id="pub_Y" size="32" maxlength="32"
19:          class="txtSt" value="19048" /></td></tr>
20:  <tr><td>The Generator g: </td><td >
21:         <input type="text" id="gen_g" size="32" maxlength="32"
22:          class="txtSt" value="2" /></td></tr>
23:  <tr><td>The Prime p: </td><td>
24:         <input type="text" id="pri_p" size="32" maxlength="32"
25:          class="txtSt" value="73699" /></td></tr>
26:  <tr><td>The Shared-Key K: </td><td>
27:         <input type="text" id="result" size="32"
28:          maxlength="32" readonly class="txtSt" value="" />
29:         <input type="button" class="butSt" value="OK"
30:          onclick="sh_key()" /></td></tr>
31: </table></form>
32: <script src="BigInt.js"></script>
33: <script>
34:  function sh_key()
35:  {
36:    var secret_y, public_Y,gen_g,prime_p;
37:    secret_y = str2bigInt(document.getElementById("sec_y").value,10,0);
38:    public_Y = str2bigInt(document.getElementById("pub_Y").value,10,0);
39:    gen_g    = str2bigInt(document.getElementById("gen_g").value,10,0);
40:    prime_p  = str2bigInt(document.getElementById("pri_p").value,10,0);
41:    powMod(public_Y,secret_y,prime_p);
42:    llst = bigInt2str(public_Y,10);
43:    document.getElementById("result").value = llst;
44:  }
45: </script>
46: </body>
```

This page contains 5 text boxes and one `OK` button. The first 4 text boxes are used to get the secret-key `y`, public-key component `Y`, generator `g`, and the prime `p`. When the `OK` button at line29 is clicked, the function `sh_key()` is activated. This function computes the shared-key `K` and displays it into the 5th text box in line 28-30.

First, the function `sh_key()` in lines 34-44 captures the strings `y,Y,g,` and `p`. These strings are converted into big numbers `secret_y`, `public_Y`, `gen_g`, and `prime_p` respectively. The statement in line 41:

```
powMod(public_Y,secret_x,prime_p);
```

is equivalent to computes the value $Y^y \bmod p$ which is the shared-key `K`. Since this value is a big number, the function `bigInt2str()` is used to convert it to string `llst`. This `llst` is output by the statement in line 43. A screen shot of this example is shown in Fig.08.05.

Now, we know how to perform Diffie-Hellman key exchange. The next question is: how safe is this scheme? In other words, can someone get the secret-key `y` when all public-key components (`Y,g,p`) are available?

8.2.3 The Discrete Logarithm Problem and Brute-Force Attack

It is well-known that the security of Diffie-Hellman algorithm is equivalent to the so-called "Discrete Logarithm Problem". If you can solve this problem effectively, the DH scheme can be cracked.

The discrete log problem

Let `p` be a prime and `a` be an arbitrary integer such that `1<a<p-1`. Let the set `Zp={0,1,2,..,p-1}` be the finite field and `Zp*` be the non-zero elements of `Zp`. The following power modulo mapping

$$e_a(n) = a^n \bmod p = N \qquad (n \geq 0)$$

maps any non-negative integer `n` (i.e. $n \geq 0$) to an element `N` in `Zp*`. When this map is onto, the element `a` is a primitive root (or generator).

Given integer values (`n,a,p`), it is easy to compute the number `N`. However, given values (`N,a,p`), it is not that easy to compute the integer value `n`.

For floating point (or real) number situation, the mapping above may look like the exponential function: $y=a^x$. Given the ordered pair (`a,x`), it is easy to compute y. For values (`y,a`), x can be computed by the logarithm function, i.e. `y = x log a`. If you have a calculator, you can obtain the approximated value instantly. It is because there is a Taylor series approximation and converges rapidly inside your calculator circuitry. However, it doesn't make sense to have an approximation to an integer `n`. The discrete log problem can be described by the following statement:

Given an element N in Zp*, if $e_a(n)=N$ holds for some integer n then find n.

If such `n>0` exists, the smallest `n` is called the "discrete log of `N` to base `a mod p`". From the definition of generator, we know that the solution `n` (or the discrete log of `N`) exists when `a` is a generator of `Zp*`. However, there is no "fast" algorithm to compute the integer `n` from values (`N,a,p`).

In terms of DH scheme, the secret-key `y` is protected by the discrete logarithm problem even if the public-key (`Y,g,p`) are known. In fact, the security of a number of cryptographic schemes depends on the assumption that "Discrete Logarithm Problem" is hard to solve.

Next, we consider how to perform a simple brute-force attack to the Diffie-Hellman secret-key.

Brute-Force Attack on DH secret-key

To perform a brute-force attack on the DH scheme is simple. Given the public-key component [Y,g,p], the secret-key y can be obtained by the equation: $Y = g^y \bmod p$. Solving this equation may be hard, we can apply brute-force to search all y in 1<y<p-1 for a match to Y. The pseudo-code for the attack is:

```
ii=1
while(ii>0)  {
  if (ii != p-2) {
    tmp = g^ii mod p;
    if (tmp == Y) {
       ii=0; output "ii as the secret-key";
    } else ii++;
  } else {
     ii=0; output "No Solution Found";
  }
}
```

To convert this pseudo-code into an implementation is not difficult. Consider the example page below:

```
Example: ex08-03.htm - Brute-Force Attack On DH Key Exchange Scheme

 1: <head><title>Brute-Force Attack On DH Scheme</title></head>
 2: <style>
 3:  .butSt{font-size:16pt;width:70px;height:35px;
 4:          font-weight:bold;background:#ddffdd;color:#ff0000}
 5:  .txtSt{font-size:14pt;width:240px;height:35px;font-weight:bold;
 6:         background:#dddddd;color:#ff0000}
 7: </style>
 8: <body style="font-family:arial;font-size:20pt;text-align:center;
 9:       background:#000088;color:#ffff00">
10:  Brute-Force Attack On the <br /> Diffie-Hellman Scret-Key
11: <br /><img alt="pic" src="line1.gif" height="7" width="650" /><br />
12: <form action="">
13: <table style="font-size:16pt;width:560px" cellspacing="5" align="center">
14:  <tr><td>The Public-Key Y</td><td>
15:         <input type="text" id="pub_Y" size="32" maxlength="32"
16:          class="txtSt" value="19048" /></td></tr>
17:  <tr><td>The Generator g: </td><td >
18:         <input type="text" id="gen_g" size="32" maxlength="32"
19:          class="txtSt" value="2" /></td></tr>
20:  <tr><td>The Prime p: </td><td>
21:         <input type="text" id="pri_p" size="32" maxlength="32"
22:          class="txtSt" value="73699" /></td></tr>
23:  <tr><td>The secret-key y: </td><td>
24:         <input type="text" id="result" size="32"
25:          maxlength="32" readonly class="txtSt" value="" />
26:         <input type="button" class="butSt" value="OK"
27:          onclick="bt_skey()" /></td></tr>
28: </table></form>
29: <script src="BigInt.js"></script>
30: <script>
31:  function bt_skey()
32:  {
33:    var secret_y, public_Y,gen_g,prime_p, llst="";
34:    var ii=1,bii;
35:    tY = str2bigInt(document.getElementById("pub_Y").value,10,0);
36:    tg = str2bigInt(document.getElementById("gen_g").value,10,0);
37:    tp = str2bigInt(document.getElementById("pri_p").value,10,0);
38:
39:    p2 = dup(tp);  p1 = int2bigInt(2,1,1);  sub(p2,p1);
```

```
40:
41:    bii = int2bigInt(1,1,1);
42:    while (ii > 0) {
43:      copyInt(bii,ii);
44:      ttg  = dup(tg);  ttp  = dup(tp);
45:      if ( !equals(p2,bii) ) {
46:        powMod(ttg,bii,ttp);
47:        if (equals(ttg,tY)) {
48:            ii=0;
49:            llst = bigInt2str(bii,10);
50:        } else ii++;
51:      } else {
52:         ii=0;
53:         llst = "No solution Found";
54:      }
55:    }
56:    document.getElementById("result").value = llst;
57:  }
58: </script>
59: </body>
```

The XHTML code of this page contains 4 text boxes and one `OK` button. The first 3 boxes are used to obtain the DH public-key information `(Y,g,p)`. When the `OK` button is clicked, the function `bt_skey()` is run to search the secret-key using brute-force attack. Apart from the big integer manipulation, the entire process is the same as the pseudo-code above. First, the public-key values `(Y,g,p)` are captured and converted into big integers `(tY,tg,tp)` by lines 35-37. Since they are big integers, ordinary operations such as plus and/or minus will not work properly. For example, to minus the prime by 2, the statements in line 39 are used:

```
p2 = dup(tp);  p1 = int2bigInt(2,1,1);  sub(p2,p1);
```

The `dup()` function is to duplicate a copy of the prime `tp` to another big integer `p2`. The second function `int2bigInt(2,1,1)` converts number 2 into big integer p1 with minimum 1 bit and 1 array. The `sub()` function finally performs the `p2-p1` and store the result back into `p2`. Line 41 generates a big integer `bii` with value 1. The while-loop in line 42-55 performs the brute-force search for the secret-key.

The search process starts by making a copy of the generator `tg` and prime `tp` into variables `ttg` and `ttp` (see line 44). When the search is not exhausted, the power modulo "$g^{ii} \bmod p$" is calculated at line 46. The result is then compared with the public-key component `tY`. If a match is found, the secret-key is obtained by variable `bii`. In this case, the setting `ii=0` terminates the search and the value `bii` (or secret-key) is output by the statement in line 49. Otherwise, the search continues until the while-loop is exhausted. A screen shot of this example is given in Fig.08.06

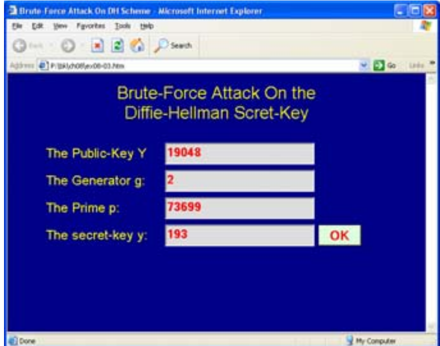


Fig.08.06: Brute-Force Attack on Diffie-Hellman Scheme

For a small prime p=73699, this brute-force search is effective. For example, the search completes less than 0.1 second on a low end portable PC. A small prime and secret-key value provide no security for the Diffie-Hellman scheme. It will be harder to find the secret-key y if, for example, the following public-key information [Y,g,p] is chosen:

- Prime p = 27419669081321110693270343633073797
- Generator g = 5;
- Public-key Y = 22522678373082478391129854703586869

The secret-key, in this case, is y=1000003. As we have mentioned in section 8.2.3, the security of the Diffie-Hellman scheme is equivalent to the discrete logarithm problem and therefore is called the discrete log based crypto-system. Another widely used discrete log crypto-system is ElGamal.

8.3 The ElGamal Public-Key Algorithm and Digital Signatures

8.3.1 The ElGamal Public-Key Algorithm

Unlike the Diffie-Hellman key exchange, the ElGamal algorithm is designed for public-key encryption/decrypiton based on discrete logarithms. It was created by Taher ElGamal and promoted by many big organization on the Internet. For example, the ElGamal algorithm is used in the free GNU Privacy Guard (GnuPG) software which is an implemenation of a standard called Open Pretty Good Privacy (PGP). Many other crypto systems use it too. It can be used for message encryption/decryption direct, and also for digital signatures. NSA's Digital Signature Algorithm is based on ElGamal.

Given a prime p and a generator g, the ElGamal encrypiton/decryption algorithm works as follows:

- Bob select his secret-key y, and compute the public-key component (Y,g,p) where $Y = g^y \bmod p$
- If Sue wants to send a message to Bob, she first converts the message into a number m
- Sue then generates a random integer k and compute the number pair (c,d) and send to Bob, where

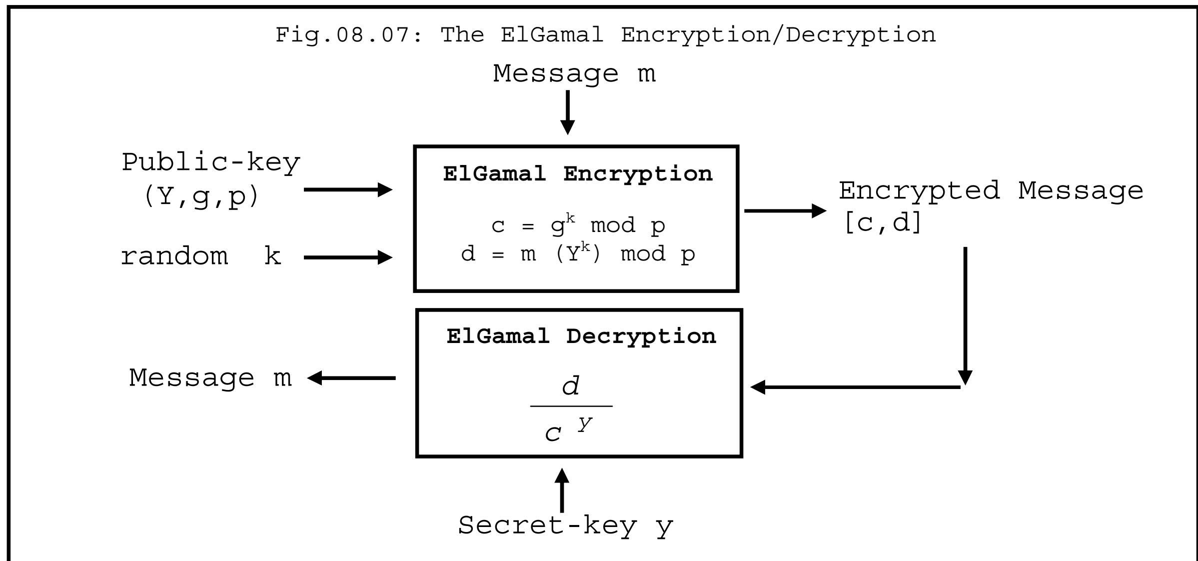
$$c = g^k \bmod p \qquad \text{and} \qquad d = m\,(Y^k) \bmod p$$

- Bob can then reconstruct the message m by using his secret-key y to calculate $\frac{d}{c^y}$

$$\text{where} \quad \frac{d}{c^y} = \frac{m(Y^k)}{(g^k)^y} = \frac{m(g^{yk})}{g^{ky}} = m$$

- In practice, the message m can be split into multiple m_i. In this case, the encrypted result is the number c together with a sequence of d_i, i.e. $\{c, d_1, d_2, .., d_n\}$

This encryption/decryption process is demonstrated in Fig 08.07.



To better understand the operation of ElGamal scheme, let's consider a step-by-step example.

An ElGamal Encryption/Decryption Example

Suppose Bob and Sue want to communicate messages using ElGamal encrytion scheme. Bob's secret and public key components are:

Bob_S = 520   Bob_P = (65079,5,170003)

Note that $65079=5^{520} \bmod 170003$. When Sue wants to send Bob a message m="Hello Bob", say, the first thing is to convert this message into an array of numbers. For simplicity, we use ASCII code as example below:

m ={Hello Bob} = {072 101 108 108 111 032 066 111 098}

Note that we have used decimal ASCII code and each code has 3 digits. As one single number, this message may be a bit too long. It can be grouped into 5 numbers. Each number represents 2 characters:

m = {072101,108108,111032,066111,098}

The next step is that Sue picks a random number k (e.g. k=429) and uses it to encrypt the message m. The first constant c is computed by $g^k \bmod p$.

$$c = g^k \bmod p = 5^{429} \bmod 170003 = 47049$$

For each element of the message array m[], the following encryption formula is used to compute the elements d_i as an array d[].

$$d_i = ((Y^k \bmod p) * m[i]) \bmod p \qquad \text{(for i=1 to 5)}$$
$$= \{26751,91891,127714,145578,151087\}$$

The values c and d_i are sent to Bob. When Bob receives the following encrypted message

{c,d[1],d[2],…,d[5]} = {47049,26751,91891,127714,145578,151087}

The original plaintext message can be obtained by Bob's secret-key `y` and the formula:

$$m[i] = \frac{d\ [i]}{c^{y}} \quad \text{mod} \quad p$$

For example, the element `m[1]` is obtained by:

$$m[1] = \frac{26751}{47079^{429}} \ \text{mod} \ 170003 = 72101$$

This number represents the first 2 characters of the plaintext `{072,101} = {He}`. When all plaintext characters are recovered, we have:

```
m[i] = {72101,108108,111032,66111,98}
     = {072,101,108,108,111,032,066,111,098}
     = {Hello Bob}
```

It is easy to see that the security of ElGamal algorithm depends on the discrete logarithm problem so that the security strength is similar to Diffie-Hellman. Now, let's see how to implement this ElGamal encryption/encryption scheme.

8.3.2 ElGamal Encryption/Decryption with APM

From the previous section we know that the ElGamal encryption algorithm depends on the following information:

- Plaintext message `m`
- Public-key information (`Y,g,p`)
- A selected encryption number `k`

Given a plaintext `m`, the encrypted message is a pair of numbers namely `[c,d]`. The formulae to compute `c` and `d` are:

$$c = g^{k} \bmod p \qquad \text{and} \qquad d = m * (Y^{k} \bmod p) \bmod p$$

These `[c,d]` values are sent to the recipient. With our arbitrary precision mathematics package BigInt.js, the implementation of these formula are quite straight-forward. Consider the page below:

```
Example: ex08-04.htm - The ElGamal Encryption Algorithm

 1: <head><title>The ElGamal Encryption</title></head>
 2: <style>
 3:  .butSt{font-size:16pt;width:70px;height:35px;
 4:           font-weight:bold;background:#ddffdd;color:#ff0000}
 5:  .txtSt{font-size:14pt;width:240px;height:35px;font-weight:bold;
 6:          background:#dddddd;color:#ff0000}
 7: </style>
 8: <body style="font-family:arial;font-size:20pt;text-align:center;
 9:       background:#000088;color:#ffff00">
10:  The ElGamal Encryption [c,d] <br /><span style="font-size:16pt"><br />
11:  c = g <sup>k</sup> mod p     &    
12:  d = m*(Y<sup>k</sup> mod p)  mod p </span>
13: <form action="">
14: <table style="font-size:16pt;width:580px" cellspacing="5" align="center">
```

```
15:  <tr><td>Plaintext Number m</td><td>
16:          <input type="text" id="tm" size="32" maxlength="32"
17:           class="txtSt" value="072101" /></td></tr>
18:  <tr><td>Encryption number k:</td><td >
19:           <input type="text" id="tk" size="32" maxlength="32"
20:            class="txtSt" value="429" /></td></tr>
21:  <tr><td>The Public-Key Y: </td><td >
22:          <input type="text" id="tY" size="32" maxlength="32"
23:           class="txtSt" value="65079" /></td></tr>
24:  <tr><td>The generator g: </td><td >
25:          <input type="text" id="tg" size="32" maxlength="32"
26:           class="txtSt" value="5" /></td></tr>
27:  <tr><td>The Prime p: </td><td>
28:          <input type="text" id="tp" size="32" maxlength="32"
29:           class="txtSt" value="170003" /></td></tr>
30:  <tr><td>Encrypted Numbers [c,d] </td><td>
31:          <input type="text" id="result" size="32"
32:           maxlength="32" readonly class="txtSt" value="" />
33:          <input type="button" class="butSt" value="OK"
34:           onclick="elgamal_en()" /></td></tr>
35: </table></form>
36: <script src="BigInt.js"></script>
37: <script>
38:  function elgamal_en()
39:  {
40:    var ttm,ttk,ttY,ttg,ttp;
41:    ttm = str2bigInt(document.getElementById("tm").value,10,0);
42:    ttk = str2bigInt(document.getElementById("tk").value,10,0);
43:    ttY = str2bigInt(document.getElementById("tY").value,10,0);
44:    ttg = str2bigInt(document.getElementById("tg").value,10,0);
45:    ttp = str2bigInt(document.getElementById("tp").value,10,0);
46:
47:    tmp01 = dup(ttg);
48:    powMod(tmp01,ttk,ttp);
49:    tmp02 = dup(ttY);
50:    powMod(tmp02,ttk,ttp);
51:    multMod(tmp02,ttm,ttp);
52:    llst = "[ "+bigInt2str(tmp01,10)+" , "+bigInt2str(tmp02,10)+" ]";
53:    document.getElementById("result").value = llst;
54:  }
55: </script>
56: </body>
```

For simplicity, only two numbers `[c,d]` are consider in this example. This page implements the ElGamal encryption algorithm. The XHTML code in lines 14-35 generates 6 text boxes and one `OK` button. The first two boxes are used to obtain the plaintext number m and encryption number `k`. The next three boxes get the public-key information (`Y,g,p`) of the recipient. When the `OK` button is clicked, the encryption function `elgamal_en()` is activated. The encrypted numbers `[c,d]` are displayed at the last text box. The details of the function `elgamal_en()` is given in lines 38-54.

First, the information `m`, `k`, `Y`, `g` and `p` are captured and converted into big integers by the statements in lines 41-45. The two statements in lines 47-48 compute the constant `c` stored in variable `tmp01`. Lines 50-51 calculate the constant d stored in `tmp02`. These two values are then converted to strings and output as `[c,d]` format (see line 52). A screen shot of this example is given in Fig.08.08.

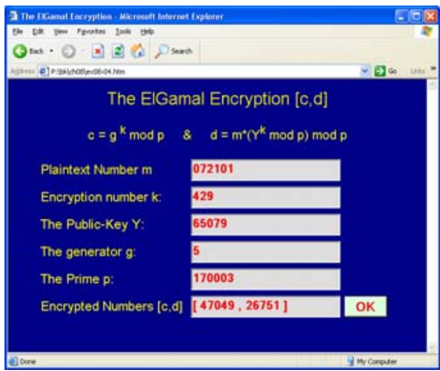


Fig.08.08: The ElGamal Encryption

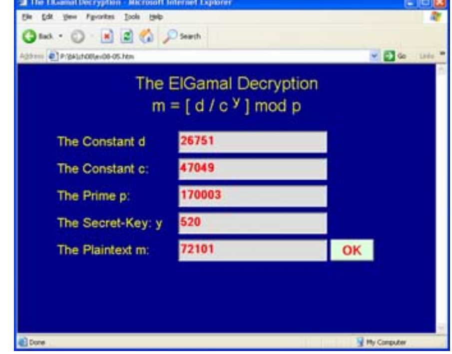


Fig.08.09: The ElGamal Decryption

Compare to encryption, the input information for decryption is relatively simpler. The decryption is depended on the following:

- The encrypted numbers `[c,d]`
- The secret-key for the recipient `y`

The plaintext number m can be computed by "$m = (d / c^{y}) \bmod p$." A simple implementation of this decryption scheme is given in ex08-05.htm.

```
Example: ex08-05.htm - The ElGamal Decryption Algorithm

 1: <head><title>The ElGamal Decryption</title></head>
 2: <style>
 3:  .butSt{font-size:16pt;width:70px;height:35px;
 4:           font-weight:bold;background:#ddffdd;color:#ff0000}
 5:  .txtSt{font-size:14pt;width:240px;height:35px;font-weight:bold;
 6:         background:#dddddd;color:#ff0000}
 7: </style>
 8: <body style="font-family:arial;font-size:20pt;text-align:center;
 9:       background:#000088;color:#ffff00">
10:  The ElGamal Decryption<br /> m = [ d / c <sup>y</sup> ] mod p
11:
12: <form action="">
13: <table style="font-size:16pt;width:560px" cellspacing="5" align="center">
14:  <tr><td>The Constant d</td><td>
15:          <input type="text" id="td" size="32" maxlength="32"
16:           class="txtSt" value="26751" /></td></tr>
17:  <tr><td>The Constant c: </td><td >
18:          <input type="text" id="tc" size="32" maxlength="32"
19:           class="txtSt" value="47049" /></td></tr>
20:  <tr><td>The Prime p: </td><td>
21:          <input type="text" id="tp" size="32" maxlength="32"
22:           class="txtSt" value="170003" /></td></tr>
23:  <tr><td>The Secret-Key: y</td><td >
24:           <input type="text" id="ty" size="32" maxlength="32"
25:            class="txtSt" value="520" /></td></tr>
26:  <tr><td>The Plaintext m: </td><td>
27:          <input type="text" id="result" size="32"
28:           maxlength="32" readonly class="txtSt" value="" />
29:          <input type="button" class="butSt" value="OK"
30:           onclick="elgamal_de()" /></td></tr>
31: </table></form>
```

```
32: <script src="BigInt.js"></script>
33: <script>
34:  function elgamal_de()
35:  {
36:    var secret_y, public_Y,gen_g,prime_p;
37:    ttd = str2bigInt(document.getElementById("td").value,10,0);
38:    ttc = str2bigInt(document.getElementById("tc").value,10,0);
39:    tty = str2bigInt(document.getElementById("ty").value,10,0);
40:    ttp = str2bigInt(document.getElementById("tp").value,10,0);
41:
42:    tmp01 = dup(ttc);
43:    powMod(tmp01,tty,ttp);
44:    inverseMod(tmp01,ttp);
45:    multMod(tmp01,ttd,ttp);
46:    llst = bigInt2str(tmp01,10);
47:    document.getElementById("result").value = llst;
48:  }
49: </script>
50: </body>
```

The XHTML code in lines 13-31 generates five text boxes and one `OK` button. The first two boxes would allow a user to the ElGamal encryption numbers `c` and `d`. The next two boxes are for the prime `p` and secret-key `y`. When the `OK` button is pressed, the plaintext number `m` will be computed and displayed in the last text box. The decryption engine is the function `elgamal_de()` defined in lines 34-48.

The decryption information `d,c,y`, and `p` are captured and converted to big integers by the statements in lines 37-40. Line 43 computes the value $c^y \bmod p$. The `inverseMod()` function is used to calculate $1/c^y \bmod p$. Finally, the multiplication modulo function `multMod()` in line 45 finishes the computation of d/c^y mod p which is the ElGamal decryption formula. The result is a big integer `tmp01` representing the plaintext number `m`. After converting into string in line 46, the result is output to the last text box by line 47. A screen shot of the decryption is shown in Fig.08.09.

If you enter the encryption numbers c and d obtained from Fig.08.07, you will have the plaintext `m=72101`. This number is, in fact, the two ASCII codes {`072,101`} representing the two characters {He} in the example discussed in section 8.2.4.

Now, we know how to implement ElGamal encryption/decryption algorithm with APM package. The next step is to extend the implementation for general message encryption.

8.3.3 Implementing Message Encryption/Decryption

In examples ex08-04.htm and ex08-05.htm, we see that ElGamal encryption works with big numbers. By grouping two ASCII codes together, a number representing two characters is generated for the encryption process. In fact, there are a number of methods to convert message characters into numbers for ElGamal encryption. Another method is to use the following mapping before the character grouping.

```
01 -->"a",  02 -->"b", .... ,  26 -->"z"
```

This method produces smaller plaintext number, but cannot handle other non-English characters or languages easily. In this section, a technique to convert message into a sequence of numbers is presented. This method is simple and particularly efficient for 32-bit computer and systems.

Given a message string `msgSt[]` as an array of characters. A big number `lrArr` can be easily constructed by grouping every 4 characters together with their corresponding ASCII codes. This process is illustrated by the while-loop below:

```
m=0;
while ( m < msgSt.length)  {
 lrArr= (msgSt.charCodeAt(m++) << 24)|(msgSt.charCodeAt(m++) << 16)|
        (msgSt.charCodeAt(m++) << 8) | msgSt.charCodeAt(m++);
 //-- Perform ElGamal Encryption on number lrArr here ---//
}
```

This while-loop group every four characters from the string `msgSt[]` to form a 32-bit integer `lrArr` which is particularly handy for 32-bit computer platforms. In order to perform ElGamal encryption, the public-key information `[Y,g,p]` and an random number `k` as described in section 8.3.2 are needed. Now, consider the encryption page below:

```
Example: ex08-06.htm - The ElGamal Message Encryption             (Part I)

 1: <head><title>ElGamal Message Encryption</title></head>
 2: <style>
 3:  .butSt{font-size:14pt;width:250px;height:30px;
 4:          font-weight:bold;background:#dddddd;color:#ff0000}
 5: </style>
 6: <body style="font-family:arial;font-size:22pt;text-align:center;
 7:       background:#000088;color:#ffff00">
 8:   Message Encryption Using<br />ElGamal Public-Key Scheme
 9: <form action="">
10: <table style="font-size:14pt;width:750px" align="center" cellspacing=10>
11:  <tr><td colspan="2">Enter The Input Message: </td></tr>
12:  <tr><td colspan="2"><textarea style="font-size:14pt;width:700px;
13:     height:150px;font-weight:bold;background:#dddddd" rows="5"
14:     cols="40" id="in_mesg">Meet Me At 2pm Tomorrow</textarea></td></tr>
15:  <tr><td >Key Info: [ k Y g p ] </td>
16:      <td><input type="text" id="key_v" size="100" maxlength="100"
17:     style="font-size:14pt;width:440px;height:35px;font-weight:bold;
18:     background:#dddddd;color:#ff0000"
19:     value="5218523 927268357353 5 2684702861777" /></td></tr>
20:  <tr><td style="height:40px;width:250px" valign=middle>
21:     The Output Message is: </td>
22:       <td><input size="20" type="button" class="butSt" style="width:180px"
23:       value="OK" onclick="encrypt_fun()" /> </td>
24:  <tr><td colspan="2"><textarea rows="5" cols="40" id="out_mesg" readonly
25:            style="font-size:14pt;font-weight:bold;width:700px;
26:            height:150px;background:#aaffaa"></textarea></td></tr>
27: </tbody></table>
28: </form>
```

This page fragment is the interface part of the example containing two text areas, one text box, and one `OK` button. The first text area in lines 12-14 allows a user to enter the plaintext message in character form. The default message for this page is "`Meet Me At 2pm Tomorrow`" as illustrated in line 14. The text box defined in line 16-19 is used to obtain the public-key information: `[k,Y,g,p]`. As a demonstration example, the default values are:

```
[k Y g p] = [ 5218523 927268357353 5 2684702861777 ]
```

Note that k is a random number selected by the user and `[Y,g,p]` is the ElGamal public-key of someone. Also, if you group four characters into one 32-bit integer, the prime `p` that you are using should be bigger than 32-bit.

When the `OK` button is clicked, the function `encrypt_fun()` is run (see line 22-23). This function will encrypt the plaintext and display the result in the second text area specified in lines 24-26. The function `encrypt_fun()` is defined in the second part of the page.

```
Example: Continuation of ex08-06.htm                              (Part II)

29: <script src="hexlib.js"></script>
30: <script src="BigInt.js"></script>
31: <script>
32:  function encrypt_fun()
33:  {
34:    var keySt="",message="",llst="", keyV;
35:    document.getElementById("out_mesg").value = llst;
36:    key = document.getElementById("key_v").value;
37:    message = document.getElementById("in_mesg").value;
38:
39:    llst = elgamal_en(key, message);
40:    document.getElementById("out_mesg").value = llst;
41:  }
42:
43:  function elgamal_en(keySt, msgSt)
44:  {
45:    var m=0,len = msgSt.length;
46:    var ii, ttc="", ttd="",ttk,ttY,ttg,ttp;
47:    var lrArr, keyV, tmp01, tmp02;
48:
49:    keyV = myParseSt(keySt);
50:    ttk = str2bigInt(keyV[0],10,5);
51:    ttY = str2bigInt(keyV[1],10,5);
52:    ttg = str2bigInt(keyV[2],10,5);
53:    ttp = str2bigInt(keyV[3],10,5);
54:    tmp01 = dup(ttg);
55:    powMod(tmp01,ttk,ttp);
56:    ttc = bigInt2str(tmp01,10);
57:
58:    while (m < len) {
59:      lrArr= (msgSt.charCodeAt(m++) << 24)|(msgSt.charCodeAt(m++) << 16)|
60:             (msgSt.charCodeAt(m++) << 8) | msgSt.charCodeAt(m++);
61:      ttm = int2bigInt(lrArr,10,5);
62:      tmp02 = dup(ttY);
63:      powMod(tmp02,ttk,ttp);
64:      multMod(tmp02,ttm,ttp);
65:      ttd += bigInt2str(tmp02,10) +" ";
66:    }
67:   return ttc+" "+ttd;
68:  }
69: </script>
70: </body>
```

This page contains two functions. The first function `encrypt_fun()` in lines 32-41 controls the entire encryption process. In lines 36-37, the public-key information `[k,Y,g,p]` and the plaintext message are captured and stored in variables key and message respectively. These values are used as input to call the function `elgamal_en()` for the encryption. The result is returned to variable `llst` and displayed by the statement in line 40.

The main encryption engine of this example is the function `elgamal_en()` in lines 43-68. This function takes two input strings namely `keySt` and `msgSt` representing the public-key and plaintext message information. It will use the keySt information to encrypt the plaintext `msgSt` producing the encrypted message {`c,d[1],d[2],..,d[n]`}

First, the information inside the `keySt` are parsed by the function `myParseSt()` in line 49 to obtain the individual strings `k`, `Y`, `g`, and `p`. These values are stored in array `keyV[]` with order. That is `k` is stored in the first element of array `keyV[]` (ie. `keyV[0]`). The statement in line 50 converts this `keyV[0]` string into a big integer called `ttk` where `ttk` is a base 10 number with minimum element size as 5 (see the instructions of the APM package BigInt.js). Similarly, the public-key values `[Y,g,p]` are obtained and converted to big integers `ttY`, `ttg`, and `ttp` respectively.

Lines 54-56 use the encryption formula "$c = g^k \bmod p$" to compute the constant `c` stored in variable `ttc`. The while-loop in lines 58-66 group every four characters from the plaintext `msgSt` into a number `lrArr`. This number is converted to a big integer variable called `ttm` and then uses it to compute the encrypted message d_i with the formula:

$$ttd = d[i] = ((Y^k \bmod p) * ttm_i) \bmod p$$

The encrypted message `[ttc,ttd]` is returned to the caller function by the statement in line 67. A screen shot of this example is given in Fig.08.10.

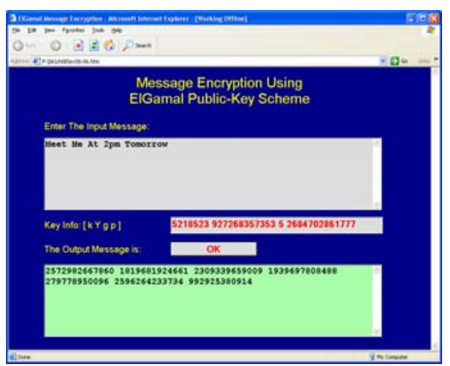


Fig.08.10: ElGamal Message Encryption

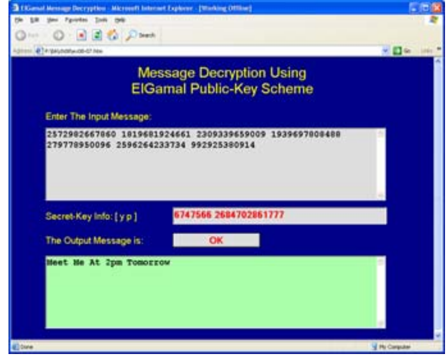


Fig.08.11: ElGamal Message Decryption

As you can see from Fig.08.10, the ElGamal encrypted results are big integers. The first one is `c` and the remaining elements are `d[i]`. These values are used in the decryption process to recover the plaintext.

To perform ElGamal decryption, let's make a copy of the part I of ex08-06.htm and call it the part I of ex08-07.htm. Inside this page fragment, modify the following lines:

```
8:   Message Decryption Using<br />ElGamal Public-Key Scheme
15:  <tr><td>Secret-Key Info: [ y p ] </td>
23:        value="OK" onclick="decrypt_fun()" /> </td>
```

Line 8 changes the title of the page so that the program is for decryption. For decryption, the secret-key of the user is required and is illustrated at line 15. In this case, the secret-key value `[y,p]` is expected. Finally, when the `OK` button is clicked, the statement in line 23 execute the function `decrypt_fun()` to perform the decryption. The interface structure of ex08-07.htm is the same as the ex08-06.htm. The decryption engine `decrypt_fun()` is defined in the second part of the page.

```
Example: ex08-07.htm - Continuation of ex08-07.htm                (Part II)

29: <script src="hexlib.js"></script>
30: <script src="BigInt.js"></script>
31: <script>
32:  function decrypt_fun()
33:  {
34:    var keySt="",message="",llst="", keyV;
35:    document.getElementById("out_mesg").value = llst
36:    key = document.getElementById("key_v").value
37:    message = document.getElementById("in_mesg").value
38:
39:    llst = elgamal_de(key, message);
40:    document.getElementById("out_mesg").value = llst;
41:  }
42:
43:  function elgamal_de(keySt, msgSt)
44:  {
45:   var ii, tResult="", result="", llst="";
46:   var lrArr, keyV, tmp01, tmp02;
47:
48:   keyV = myParseSt(keySt);
49:   tty = str2bigInt(keyV[0],10,5);
50:   ttp = str2bigInt(keyV[1],10,5);
51:
52:   lrArr = myParseSt(msgSt);
53:   ttc = str2bigInt(lrArr[0],10,5);
54:   for (ii=0; ii < lrArr.length -1; ii++)  {
55:     tmp01 = dup(ttc);
56:     powMod(tmp01,tty,ttp);
57:     inverseMod(tmp01,ttp);
58:     ttd = str2bigInt(lrArr[ii+1],10,5);
59:     multMod(tmp01,ttd,ttp);
60:     result  = bigInt2str(tmp01,10);
61:     tresult = parseInt(result,10);
62:
63:     tempSt= String.fromCharCode(
64:       ((tresult >>> 24) & 0xff), ((tresult >>> 16) & 0xff),
65:       ((tresult >>>  8) & 0xff), ( tresult         & 0xff));
66:     llst += tempSt;
67:   }
68:   return llst;
69:  }
70:  var testSt = "2572982667860 1819681924661 2309339659009 "+
71:         "1939697808488 279778950096 2596264233734 992925380914";
72:  document.getElementById("in_mesg").value=testSt;
73: </script>
74: </body>
```

This page contains two functions namely: `decrypt_fun()` and `elgamal_de()`. The first function controls the operation of the decryption and basically the same as the encryption `encrypt_fun()` in ex08-06.htm. The only difference is that the ElGamal decryption function `elgamal_de()` is called in line 39. When the function `elgamal_de()` is executed with the secret-key and ciphertext information, decryption is performed.

First, the secret-key information stored in variable `keySt` is parsed into individual strings and convert to big integers `[tty, ttp]` (see lines 48-50). Also, the encrypted message in `msgSt` is parsed and into an array of strings `lrArr[]`. The first element of `lrArr[0]` contains the value of the decryption constant c and therefore extracted into big integer `ttc` in line 53. The for-loop in lines

54-67 extracts each remaining element `lrArr[ii]` and convert it to decryption constant `d[ii]`. The statements in lines 55-59 are used to perform the decryption:

$$m[i] = \frac{d[ii]}{c^{y}} \mod p$$

The result is stored in variable `tmp01` in line 59. The next is to convert this big number format `tmp01` (or `m[ii]`) into ordinary number 32-bit format. Finally, the statement in line 63-63 convert this 32-bit integer back to characters and appended into `llst` and return to the function caller. As a result the decrypted message will appear at the second text area of the page.

For a demonstration example, a testing encryption string is included in lines 70-72. This string is copied from the result of ex08-06.htm so that the plaintext can be obtained in this ex08-07.htm example. A screen shot is shown in Fig. 08.11.

If you have a big prime `p` such as more than 100 digits, you can group more characters into one big integer. In this case, you may need to replace the following statement with the appropriate routines in the APM package: BigInt.js.

```
lrArr= (msgSt.charCodeAt(m++) << 24)|(msgSt.charCodeAt(m++) << 16)|
       (msgSt.charCodeAt(m++) << 8) | msgSt.charCodeAt(m++);
```

Apart from the privacy or confidentiality of message, another essential application of public-key schemes is digital signature and data integrity which will be discussed in next section.

8.3.4 Using ElGamal Scheme For Digital Signatures and Data Integrity

The idea of digital signature is simple. It is similar to put a signature on a paper. The main difference is that the paper is in electronic document form. When you sign a paper or contract with your signature, the purpose is make it unique and binding you with the document. A legal process can then use the signature to identify you. Based on this fact, trust can be built within legal systems covering all business activities.

The basic principle for encryption is that you don't want the document readable by hiding the contents. The digital signature process is different. Basically, you want to construct a way so that the recipient can identify the document sender. One of the naturally ways to do that is to sign the document with your secret-key.

The ElGamal Digital Signature and Data Integrity

Suppose the public and private keys of Bob are : `(Y,g,p)` and `y` respectively. The following procedure is used to perform ElGamal digital signature and signing a document:

- Let the message be `m`
- Pick a random number `k` relatively prime to (`p-1`)
- Compute the ElGamal signature constants `s1`, and `s2` as follows:

$$s_1 = g^{k} \mod p \qquad s_2 = \frac{m - s_1 y}{k} \mod (p - 1)$$

The constant s_2 is similar to signing the message `m` with the secret-key `y`.

- Send the information {`m,s1,s2`} to the recipient.
- The recipient can verify the sender by computing the constants `r1` and `r2`:

$$r_1 = g^m \mod p \qquad r_2 = Y^{s_1} \ {s_1}^{s_2} \mod p$$

- If `r1 = r2`, the message m must be sent by the owner of (`Y,g,p`) and y

Consider the constant `r2`. If you substitute the definitions of `Y`, `s1`, and `s2` into `r2`, you have:

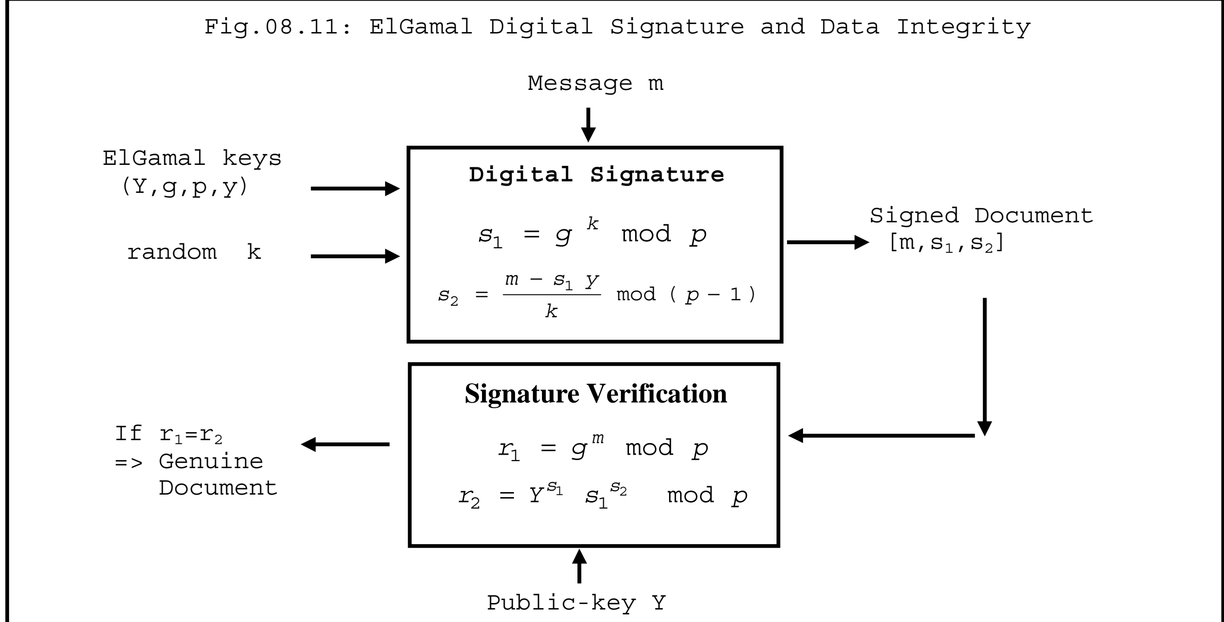
$$\begin{aligned} r_2 &= Y^{s_1} \ {s_1}^{s_2} \mod p \\ &= \left(g^y\right)^{s_1} \left(g^k\right)^{\frac{m - y s_1}{k}} \mod p \\ &= g^m \mod p \end{aligned}$$

This result is the constant r1. In other words, the signed document {m, s_1, s_2} can be verified by the public-key information (`Y,g,p`) by computing the constants r1, and r2. If they are equal, the signed document {m, s_1, s_2} , in an ideal case, must be sent by the owner of the public-key {`Y,g,p`}.

In practice, the message m is usually the hash-value (or digest) of a document. By signing the hash-value using the secret-key, the digest of the document is protected by the public-key encryption. In this case, no alteration can be made and, therefore, data integrity is maintained.

The numbers s_1 and s_2 can be considered as the signature of the owner of keys (`Y,g,p,y`) signing on the document m. This digital signature process is illustrated in Fig.08.11.

Fig.08.11: ElGamal Digital Signature and Data Integrity



Consider a practical example. Suppose the public and secret keys of Bob are:

```
(Y,g,p) = (65079,5,170003)        y = 520
```

Let the message `m=72101` and the selected number k relatively prime to `(p-1)` is `k=70001`.

$$s_1 = 5^{70001} \bmod 170003 = 119060 \qquad s_2 = \frac{72101 - 520(119060)}{70001} \bmod 170002 = 122563$$

The signed document is therefore {`72101,119060,122563`}. These values can be sent to the recipient for verification. To verify the signature, the constants `r1` and `r2` are computed as follows:

$$r_1 = 5^{72101} \bmod 170003 = 135849$$

$$r_2 = \left(65079^{119060} \bmod 170003\right)\left(119060^{122563} \bmod 170003\right) \bmod 170003 = 135849$$

From this example, we know that the signed document {`72101,119060,122563`} , in an ideal case, must be sent by the owner of the public-key `(65079,5,170003)`

8.3.5 Implementation of the Digital Signature Scheme

The ElGamal digital signature process has two parts. The first part is to generate the signature value pair `[s1,s2]` of a plaintext number `m`. In practice, the number m is usually the hash-value (or digest) of a document. The formulae governing values `s1` and `s2` are provided in Fig.08.08. Therefore, the implementation of ElGamal signature is simply a process to program these two formulas into a Web page. Consider the example below:

```
Example: ex08-08.htm - The ElGamal Digital Signature              (Part I)

 1: <head><title>The ElGamal Digital Signature</title></head>
 2: <style>
 3:  .butSt{font-size:16pt;width:70px;height:35px;
 4:           font-weight:bold;background:#ddffdd;color:#ff0000}
 5:  .txtSt{font-size:14pt;width:240px;height:35px;font-weight:bold;
 6:         background:#dddddd;color:#ff0000}
 7: </style>
 8: <body style="font-family:arial;font-size:20pt;text-align:center;
 9:       background:#000088;color:#ffff00">
10:  The ElGamal Digital Signature [s1,s2] <br />
11: <span style="font-size:16pt"><br />
12:  s1 = g <sup>k</sup> mod p     &    
13:  s2 = ( m - y*s1 ) / k     mod (p-1) </span>
14: <form action="">
15: <table style="font-size:16pt;width:620px" cellspacing="5" align="center">
16:  <tr><td>Plaintext Number m</td><td>
17:          <input type="text" id="tm" size="32" maxlength="32"
18:           class="txtSt" value="072101" /></td></tr>
19:  <tr><td>Signature number k:</td><td >
20:           <input type="text" id="tk" size="32" maxlength="32"
21:            class="txtSt" value="70001" /></td></tr>
22:  <tr><td>The Public-Key Y: </td><td >
23:          <input type="text" id="tY" size="32" maxlength="32"
```

```
24:           class="txtSt" value="65079" /></td></tr>
25:  <tr><td>The generator g: </td><td >
26:          <input type="text" id="tg" size="32" maxlength="32"
27:           class="txtSt" value="5" /></td></tr>
28:  <tr><td>The Prime p: </td><td>
29:          <input type="text" id="tp" size="32" maxlength="32"
30:           class="txtSt" value="170003" /></td></tr>
31:  <tr><td>The secret-key y: </td><td>
32:          <input type="text" id="tyy" size="32" maxlength="32"
33:           class="txtSt" value="520" /></td></tr>
34:  <tr><td>Encrypted Numbers [s1,s2] </td><td>
35:          <input type="text" id="result" size="32"
36:           maxlength="32"  class="txtSt" value="" />
37:          <input type="button" class="butSt" value="OK"
38:           onclick="elgamal_sig()" /></td></tr>
39: </table></form>
```

This XHTML code generates seven text boxes and one `OK` button. The first 6 boxes are used to get the information `[m,k,Y,g,p,y]` respectively. When the `OK` button is clicked, the function `elgamal_sig()` is run and the signature pair `[s1,s2]` are generated in the last text box. The details of the function `elgamal_sig()` is given the second part of the example.

```
Example: Continuation of ex08-08.htm                              (Part II)

40: <script src="BigInt.js"></script>
41: <script>
42:  function elgamal_sig()
43:  {
44:    var ttm,ttk,ttY,ttg,ttp,tty,tmp01,tmp02,s1,s2;
45:    ttm = str2bigInt(document.getElementById("tm").value,10,0);
46:    ttk = str2bigInt(document.getElementById("tk").value,10,0);
47:    ttY = str2bigInt(document.getElementById("tY").value,10,0);
48:    ttg = str2bigInt(document.getElementById("tg").value,10,0);
49:    ttp = str2bigInt(document.getElementById("tp").value,10,0);
50:    tty = str2bigInt(document.getElementById("tyy").value,10,0);
51:
52:    s1 = dup(ttg);                 powMod(s1,ttk,ttp);
53:    oneInt = int2bigInt(1,1,1);    sub(ttp,oneInt);
54:    tmp01 = dup(tty);              mult(tmp01,s1);
55:    tmp02 = dup(ttm);
56:    if (greater(tmp02,tmp01)) {
57:         sub(tmp02,tmp01);
58:         tmp03 = tmp02; sflag = 0;
59:    } else {
60:         sub(tmp01,tmp02);
61:         tmp03 = tmp01; sflag = 1;
62:    }
63:    s2 = dup(ttk);
64:    inverseMod(s2,ttp);
65:    multMod(s2,tmp03,ttp);
66:    if (sflag ==1) {
67:       ts2 = dup(ttp);
68:       sub(ts2,s2); s2=dup(ts2);
69:    }
70:    llst = "[ "+bigInt2str(s1,10)+" , "+bigInt2str(s2,10)+" ]";
71:    document.getElementById("result").value = llst;
72:  }
73: </script>
74: </body>
```

Line 40 includes the APM package BigInt.js into the page so that precision computations are available. The statements in lines 45-50 get the information `[m,k,Y,g,p,y]` from the XHTML interface. The first signature constant `s1` is calculated by line 52 using the formula:

$$s_1 = g^{k} \bmod p$$

Line 53 computes the value (`p-1`). It generates the number 1 in big integer format and then subtracts the prime (i.e. `ttp`) by 1. The remaining statements are used to compute the second signature constant `s2` using:

$$s_2 = \frac{m - s_1 y}{k} \bmod (p - 1)$$

First, the values of `m` and s_1y are computed. In order to avoid negative number, a comparison statement is used in lines 56-62. This statement determines whether m is greater than s_1y and performs the non-negative subtraction $| m-s_1y |$. If the result stored in variable `tmp03` is supposed a negative number, the flag `sflag` is set to 1. Lines 64-65 perform the computation:

$$\frac{| m - s_1 y |}{k} \bmod (p - 1)$$

If this number is supposed to be negative, the `sflag` is 1. In this case, lines 67-68 subtract this value by the prime `ttp` to produce the result of `s2`. Both values `s1` and `s2` are output by the statement in lines 70-71. The document m and the signature `[s1,s2]` are then sent to the recipient for verification. A screen shot of this example is shown in Fig. 08.12.

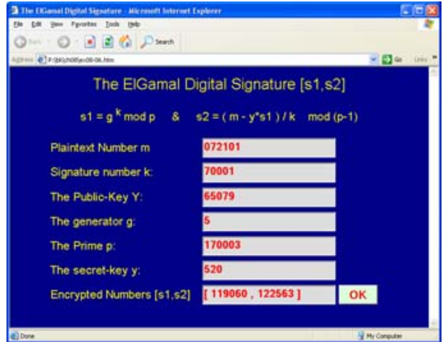


Fig.08.12: ElGamal Digital Signature

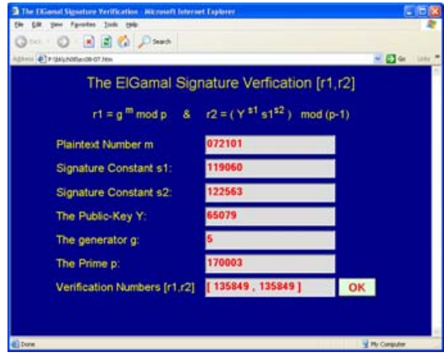


Fig.08.13: Signature Verification

The values [`s1,s2`] are generally regarded as the signature of the message `m`. In practice, the message m and the signature are sent to the recipient. When the recipient receives the signature of the document in the format {`m,s1,s2`}, he/she can perform the verification process to identify the sender of the document. Consider the following Web page.

```
Example: ex08-09.htm - Digital Signature Verification            (Part I)

 1: <head><title>The ElGamal Signature Verification</title></head>
 2: <style>
 3:   .butSt{font-size:16pt;width:70px;height:35px;
 4:            font-weight:bold;background:#ddffdd;color:#ff0000}
```

```
 5:  .txtSt{font-size:14pt;width:240px;height:35px;font-weight:bold;
 6:         background:#dddddd;color:#ff0000}
 7: </style>
 8: <body style="font-family:arial;font-size:20pt;text-align:center;
 9:       background:#000088;color:#ffff00">
10:  The ElGamal Signature Verfication [r1,r2] <br />
11:  <span style="font-size:16pt"><br />
12:  r1 = g <sup>m</sup> mod p     &    
13:  r2 = ( Y <sup>s1</sup> s1<sup>s2</sup> )   mod (p-1)</span>
14: <form action="">
15: <table style="font-size:16pt;width:620px" cellspacing="5" align="center">
16:  <tr><td>Plaintext Number m</td><td>
17:          <input type="text" id="tm" size="32" maxlength="32"
18:           class="txtSt" value="072101" /></td></tr>
19:  <tr><td>Signature Constant s1: </td><td>
20:          <input type="text" id="ts1" size="32" maxlength="32"
21:           class="txtSt" value="119060" /></td></tr>
22:  <tr><td>Signature Constant s2: </td><td>
23:          <input type="text" id="ts2" size="32" maxlength="32"
24:           class="txtSt" value="122563" /></td></tr>
25:  <tr><td>The Public-Key Y: </td><td >
26:          <input type="text" id="tY" size="32" maxlength="32"
27:           class="txtSt" value="65079" /></td></tr>
28:  <tr><td>The generator g: </td><td >
29:          <input type="text" id="tg" size="32" maxlength="32"
30:           class="txtSt" value="5" /></td></tr>
31:  <tr><td>The Prime p: </td><td>
32:          <input type="text" id="tp" size="32" maxlength="32"
33:           class="txtSt" value="170003" /></td></tr>
34:  <tr><td>Verification Numbers [r1,r2] </td><td>
35:          <input type="text" id="result" size="32"
36:           maxlength="32"  class="txtSt" value="" />
37:          <input type="button" class="butSt" value="OK"
38:           onclick="elgamal_sig2()" /></td></tr>
39: </table></form>
```

This XHTML code acts as an interface for the ElGamal signature verification process. It generates seven text boxes and one `OK` button. The first six boxes are used to obtain the information {`m,s1,s2,Y,g,p`} respectively. When the `OK` button is clicked, the function `elgamal_sig2()` is activated to perform signature verification. The verification constants [`r1,r2`] will be displayed in the last text box declared in lines 35-36. The details of this function are given in the second of the page.

```
Example: Continuation of ex08-09.htm                              (Part II)

40: <script src="BigInt.js"></script>
41: <script>
42:  function elgamal_sig2()
43:  {
44:    var ttm,tts1,tts2,ttY,ttg,ttp,tmp01,tmp02,r1,r2;
45:    ttm = str2bigInt(document.getElementById("tm").value,10,0);
46:    tts1 = str2bigInt(document.getElementById("ts1").value,10,0);
47:    tts2 = str2bigInt(document.getElementById("ts2").value,10,0);
48:    ttY = str2bigInt(document.getElementById("tY").value,10,0);
49:    ttg = str2bigInt(document.getElementById("tg").value,10,0);
50:    ttp = str2bigInt(document.getElementById("tp").value,10,0);
```

```
51:
52:    powMod(ttg,ttm,ttp);       r1 = dup(ttg);
53:    tmp01 = dup(ttY);
54:    powMod(tmp01,tts1,ttp);
55:    tmp02 = dup(tts1);
56:    powMod(tmp02,tts2,ttp);
57:    multMod(tmp01,tmp02,ttp); r2 = dup(tmp01);
58:
59:    llst = "[ "+bigInt2str(r1,10)+" , "+bigInt2str(r1,10)+" ]";
60:    document.getElementById("result").value = llst;
61:  }
62: </script>
63: </body>
```

First, the information {m,s1,s2,Y,g,p} are captured and converted into big integers in lines 45-50. The statement in line 52 calculates the first verification constant r1:

$$r_1 = g^m \bmod p$$

The statements in lines 53-57 compute the second verification constant r2:

$$r_2 = Y^{s_1} \; s_1^{s_2} \quad \bmod p$$

These two values are displayed by the lines 59-60 as [r1,r2] pair. When r1=r2, we know that the message m was really sent by the owner of the public-key {Y,g,p}. A screen shot is shown in Fig.08.13.

8.4 The RSA Scheme, Digital Signature and Hybrid Encryption

8.4.1 The RSA Public-Key Algorithm and Challenge

RSA is another popular public-key encryption/decryption scheme. It was invented in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman and is called RSA algorithm. The scheme works by generating a public-key and a secret-key. Both keys can be used to encrypt messages. When one key is used to encrypt, the other one must be applied for the decryption. The purpose of the public-key is to make it available to anyone. In an ideal case, only the owner of the secret-key is able decrypt the ciphertext.

As many other public-key technologies, the RSA public-key infrastructures and algorithms consist the following three parts:

- Generating the Keys
- Message Encryption and Decryption
- Digital Signatures and Authentications

We will discuss them one-by-one in this section. First, let's consider how to generate the RSA keys.

Generating the RSA Public and Secret Keys

The operation of the scheme is described by the following steps:

- Select two large primes, p and q.

- Compute the values `n=pq`, and `phi=(p-1)(q-1)`
- Choose a number, $e < n$ such that `e` and `(p-1)(q-1)` have no common factors except 1, i.e.

$$\gcd\left(e, (p-1)(q-1)\right) = 1$$

  where `gcd` is the greatest common divisor. In practice, `[e,n]` is the public-key

- Compute another number `d` such that

$$d = \frac{1}{e} \mod (p-1)(q-1) \quad \text{or} \quad ed - 1 = k(p-1)(q-1)$$

  In other words `(ed - 1)` is divisible by `(p-1)(q-1)`. The existence of d is guaranteed by the Chinese Remainder Theorem. The value `[d,n]` is the secret-key.

Given an arbitrary integer $m \neq 0 \mod n$ and using Fermat's Little theorem, we have the encryption/ decryption formula:

$$m^{(p-1)(q-1)} = 1 \mod n \quad \Rightarrow \quad \left(m^e\right)^d = m \mod n$$

As long as the prime factors `p` and `q` are not known or `n` cannot be factorized easily, the scheme is secure. In other words, one may not be easy to derive the secret-key `[d,n]` from the information of the public-key `[e,n]` or vice versa. For this reason, the prime factors `p` and `q` are usually kept with the secret-key or destroyed.

The security strength of RSA algorithm lays on the fact that factorization of a big number `n` into two primes `p` and `q` are not easy. If one could factor `n` into `p` and `q`, then the secret-key `[d,n]` can be obtained by the public-key information `[e,n]`.

The RSA Encryption/Decryption

One of the best ways to demonstrate how to perform encryption/decryption with RSA keys is by example. Suppose Bob and Sue has the following RSA public and secret keys:

$$Bob_P = [e_b, n_b] \qquad Sue_P = [e_s, n_s]$$
$$Bob_S = [d_b, n_b] \qquad Sue_S = [d_s, n_s]$$

When Bob wants to send a message `m` to Sue, Bob creates the ciphertext c_s by applying the public-key Sue_P and the following formula:

$$c_s = m^{e_s} \mod n_s$$

When the ciphertext arrives, Sue uses her secret-key Sue_S and the formula below to decrypt the message:

$$m = c_s^{d_s} \mod n_s$$

The chosen relationship between e and d in the encryption formula ensures that Sue correctly recovers m. In an ideal case, Sue is the only owner of the secret-key `d`, only Sue can decrypt it.

The RSA Digital Signatures and Authentications

Suppose Bob wants to send a message m to Sue in such away that the identity of the sender, Bob in this case, could be verified by Sue. To do that, Bob can sign the message m using his secret-key $[d_b, n_b]$. The signing process is done by the formula:

$$s_b = m^{d_b} \bmod n_b$$

Bob sends both the message m and the signature s_b to Sue. The authentication process can be checked by Sue using Bob's public-key $[e_b, n_b]$ and the formula:

$$m = s_b^{e_b} \bmod n_b$$

If the plaintext m is recovered, it must be sent by Bob.

As you can see from this process, authentication take place without any sharing of private-keys. In other words, the security of the private-keys is no compromised. Each person uses only other people's public keys and his or her own private key. Anyone can send an encrypted message to the one intended. To verify a signed message, only public-key of the sender is needed.

The RSA Challenges and Attacks

One of the nightmares for most public-key algorithms is that an attacker to discover the private-key corresponding to a given public-key. This would enable the attacker to read all messages encrypted with the public key. Even worse, the attacker can forge signatures and the entire security of authentication and data integrity are compromised. Two of these attacks on RSA scheme are summarized below:

- Factorizing the Public Modulo n to Primes p and q:

  If the public modulo can be factorized into primes p and q, the attacker can easily get d, the private-key exponent from the public-key exponent. The RSA scheme is based on the assumption that factoring n is difficult.

- Finding the e-th Roots Mod n

  Another way to crack the RSA scheme is to find a technique to compute the *e*-th roots mod n. That is to solve the following equation for the plaintext m:

  $$c = m^{e} \bmod n$$

  This is a direct method and would allow someone to recover plaintext messages. Similarly, by solving the digital signature equation below

  $$m = s^{e} \bmod n$$

  forge signatures can be done even without knowing the private-key. This attack is believed not related to the factorization of n. However, there is no known effective algorithm to find the e-th roots of mod n.

Now, let's see how to generate RSA public and secret keys and more importantly how to use them to perform encryption/decryption on messages:

8.4.2 Generating the RSA Public and Secret Keys

The basic construction of RSA public and secret keys is simple. Basically, you need two prime numbers p, q, and a selected public-key component e. The secret-key d is generated by the inverse modulo formula:

$$d = \frac{1}{e} \mod (p-1)(q-1)$$

The public-key and secret-key are [e,n] and [d,n] where n=pq. The security of the algorithm relies on the difficulty of factoring n. In order to have strong security strength, the primes p and q must be big enough to discourage attackers using prime factorization attacks.

For example, consider a 29 digits number n=83729636702158658998761175349. This number can be factorized into two primes less than half a second on a low end portable PC. The result are two 15 digit primes p and q, i.e.

```
n  =  83729636702158658998761175349
   =  (976746753830539)*(85722974121791)
```

In the real world, any prime number less than 20 digits (~70-bit) used on RSA algorithm are trivially easy to crack. A prime number with 100 digits (~300-bit) can only provide a very limited security-strength when under attack. For a descent protection, the prime number must be more than 200 digits (~700-bit). By using the APM package: BigInt.js, a random70-bit prime number can be obtained by the function calls below:

```
prime_p = int2bigInt(0,70,0);
randTruePrime(prime_p,70)
```

The first statement converts the number 0 into a big integer called prime_p with at least 70-bit. This is necessary simply because to make sure that the variable prime_p is big enough to store the prime number of 70-bit length. The second statement is to generate a random true prime to prime_p. With the knowledge of bit-length, the RSA keys can be practically described by the following algorithm:

- Given bit-length b, generate the random primes p and q of b bit-length.
- Given public-key e, compute the secret-key d by using the formula above.

Consider the page as follows:

```
Example: ex08-10.htm - Generating RSA Public and Secret Keys      (Part I)

 1: <head><title>Generating RSA Keys</title></head>
 2: <style>
 3:  .butSt{font-size:16pt;width:70px;height:35px;
 4:           font-weight:bold;background:#ddffdd;color:#ff0000}
 5:  .txtSt{font-size:14pt;width:500px;height:35px;font-weight:bold;
 6:         background:#eeeeee;color:#ff0000}
 7: </style>
 8: <body style="font-family:arial;font-size:20pt;text-align:center;
 9:       background:#000088;color:#ffff00">
10:  Generating The RSA Public and Secret Keys<br />
11:  <span style="font-size:16pt"><br />d = 1 / e    
12:       mod    (p-1)(q-1)</span><br /><br/>
13: <form action="">
14: <table style="font-size:16pt;width:660px" cellspacing="5" align="center">
15:  <tr><td>The bit-length b</td><td>
```

```
16:          <input type="text" id="tb" size="32" maxlength="32"
17:           class="txtSt" style="width:50px" value="70" /></td>
18:      <td>Public-Key e</td><td>
19:          <input type="text" id="te" size="32" maxlength="32"
20:           class="txtSt" style="width:280px" value="65537" />
21: </table><br />
22: <table style="font-size:16pt;width:660px" cellspacing="5" align="center">
23:  <tr><td colspan="2"> Generate the RSA Secret-Key:   
24:      <input type="button" class="butSt" value="OK"
25:        onclick="rsa_keys()" /></td></tr>
26:  <tr><td>Prime p</td><td>
27:      <input type="text" id="tp" size="32" maxlength="32" readonly
28:        class="txtSt" style="color:#008800" value="" /></td></tr>
29:  <tr><td>Prime q</td><td >
30:      <input type="text" id="tq" size="32" maxlength="32" readonly
31:        class="txtSt" style="color:#008800" value="" /></td></tr>
32:  <tr><td>n = pq</td><td >
33:      <input type="text" id="tn" size="32" maxlength="42" readonly
34:        class="txtSt" style="color:#008800" value="" /></td></tr>
35:  <tr><td>Secret-Key d</td><td>
36:       <input type="text" id="td" size="32" maxlength="42" readonly
37:        class="txtSt" style="color:#008800" value="" /></td></tr>
38: </table></form>
```

This XHTML code contains two input boxes, one `OK` button, and four output boxes. The two input boxes in lines 16-17 & 19-20 allow a user to fill in the bit-length `b` and the public-key `e`. When the `OK` button in lines 24-25 is clicked, the function `rsa_keys()` is executed. As a result, the random primes `p` & `q`, the product `n=pq`, and the secret-key `d` are displayed in the last four read only output boxes respectively.

The details of the function `rsa_keys()` are given in the second part of the example.

```
Example: Continuation of ex08-10.htm                              (Part II)

39: <script src="BigInt.js"></script>
40: <script>
41:  var rsa_b, rsa_e, rsa_p, rsa_q, rsa_n;
42:  function rsa_keys()
43:  {
44:    rsa_b = parseInt(document.getElementById("tb").value);
45:
46:    if (rsa_b < 5) alert("Bit Length Cannot Less Than 5");
47:    else {
48:      rsa_p=int2bigInt(0,rsa_b,0);
49:      rsa_q=int2bigInt(0,rsa_b,0);
50:      rsa_e = str2bigInt(document.getElementById("te").value,10,0);
51:      rsa_primes();
52:      rsa_n();
53:      rsa_d();
54:    }
55:  }
```

The function `rsa_keys()` in lines 42-55 controls the key generation process. First, a number of global variables are declared and they are:

`rsa_b, rsa_e, rsa_p, rsa_q, and rsa_n`

They are used to store the corresponding values of `b,e,p,q,` and `n` in the RSA public and secret key formula. Line 44 captures the b bit-length as variable `rsa_b`. To avoid some computational short-circuit behaviour, bit-length less than 5 is not allowed. If `rsa_b` is more than or equals to 5, the

statements in lines 48-53 are executed. Lines 48-49 create `rsa_b` bit integers `rsa_p1` and `rsa_q1`. They are used to store the two random primes. The public-key information is captured and converted to big integer `rsa_e`. Once `rsa_b` and `rsa_e` are available, the following functions are run:

`rsa_primes()` -- Generate two random primes p and q of `b` bit-length.
`rsa_n()` -- Compute the modulus `n=p*q`
`rsa_d()` -- Compute the secret-key `d`

These three functions are defined in the third part of the example.

```
Example: Continuation of ex08-10.htm                              (Part III)

56:  function rsa_primes()
57:  {
58:    var tmp;
59:    while (1) {
60:      randTruePrime(rsa_p,rsa_b);
61:      tmp=dup(rsa_p);
62:      mod(tmp,rsa_e,0);
63:      if (!equalsInt(tmp,1))
64:      break;
65:    }
66:    document.getElementById("tp").value=bigInt2str(rsa_p,10);
67:    while(1) {
68:       randTruePrime(rsa_q,rsa_b);
69:       tmp=dup(rsa_q);
70:       mod(tmp,rsa_e,0);
71:       if (!equals(rsa_p,rsa_q) && !equalsInt(tmp,1))
72:       break;
73:    }
74:    document.getElementById("tq").value=bigInt2str(rsa_q,10);
75:  }
76:  function rsa_n()
77:  {
78:    var s = rsa_p.length + rsa_q.length-1;
79:    var tmp1;
80:    tmp1=int2bigInt(0,10,s)
81:    copy(tmp1,rsa_p);
82:    mult(tmp1,rsa_q);
83:    document.getElementById("tn").value=bigInt2str(tmp1,10);
84:  }
85:  function rsa_d()
86:  {
87:    var tmp1, tmq1, phi;
88:    var errorSt="Public-key e not invertible. Try a different prime";
89:    var s = rsa_p.length + rsa_q.length-1;
90:    phi=int2bigInt(0,10,s)
91:    tmpe = int2bigInt(0,10,s)
92:    tmp1 = dup(rsa_p);    tmq1 = dup(rsa_q);
93:    addInt(tmp1,-1);      addInt(tmq1,-1);
94:    copy(phi,tmp1)
95:    mult(phi,tmq1);
96:    copy(tmpe,rsa_e);
97:    s=inverseMod(tmpe,phi);
98:    if (!s)
99:     document.getElementById("td").value=errorSt;
100:    else
101:     document.getElementById("td").value=bigInt2str(tmpe,10);
102:  }
103: </script>
104: </body>
```

The function `rsa_primes()` generates the two primes `p` and `q` with `b` bit-length. In order to satisfy the greatest common divisor criteria,

$$\gcd\left(e, (p-1)(q-1)\right) = 1$$

lines 61-62 computes the value "`p mod e`". If this value is 1, the value (`p-1`) is divisible by `e`. In this case, the `gcd` condition above is not satisfied. The prime `p` is rejected and a new one is needed as illustrated by the while-loop in lines 59-65. The search is continued until a proper prime `p` is found. The prime is then output the first display box by line 66. Note that variables in this example are prefix with "`rsa`" string. For example `p` is represented by `rsa_p`.

The while-loop in 67-73 search for another random prime q using the same argument as prime `p`. In addition to the `gcd` criteria, prime `q` should not equal to prime `p`. When these two conditions are satisfied, this prime `q` is output the second display box by line 74.

The function `rsa_n()` is to compute the product of `p` and `q`. Before the actual multiplication, the statement in line 78 computes the necessary bit-length to hold the product. The result `n=pq` is output to the third display box by line 83.

The secret-key is computed by the function `rsa_d()` in lines 85-102 using the formula:

$$d = \frac{1}{e} \mod (p-1)(q-1)$$

Lines 92-93 compute the values `p-1`, and `q-1`. The product `(p-1)(q-1)` is calculated by lines 94-96 and stored in variable `phi`. The function in 97 computes the inverse of `e mod phi`. The result is `d` stored in variable `tmpe`. The variable `s` contains the operation status. If there is an error, the error message `errorSt` is printed to indicate that the value `e` is not invertible. If there is no error, the value of `d` is output to the last text box by line 101. A screen shot of this example is shown in Fig.08.14.

Fig.08.14: Generating RSA Keys

From Fig.08.14, we can see that the 70-bit primes `p` and `q` are 21 digits. The values of `n=pq` and `phi=(p-1)(q-1)` are 42 digits. In order to prevent easy prime factorization of `n`, both the primes should be more than 70-bit. For secure public-key encryptions, you do need big numbers.

8.4.3 Message Encryption/Decryption using the RSA Scheme

Given the following public-key and secret-key pair

```
Public-key = [ e , n ]       secret-key = [ d , n ]
```

The message encryption and decryption are governed by the following formula:

$$c = m^{e} \bmod n \qquad m = c^{d} \bmod n$$

where m is the plaintext message operated as number. To perform the real message encryption/ decryption, we following the approach as described for the ElGamal situation. A long message is grouped into elements of 4 characters. Each element (4 characters) is represented by a 32-bit integer m so that the first equation above can be applied to encrypt the data.

To build a simple demonstration page, the following XHTML interface is used:

```
Example: ex08-11.htm - Message Encryption Using RSA Scheme        (Part I)

 1: <head><title>RSA Message Encryption</title></head>
 2: <style>
 3:  .butSt{font-size:14pt;width:250px;height:30px;
 4:           font-weight:bold;background:#dddddd;color:#ff0000}
 5: </style>
 6: <body style="font-family:arial;font-size:22pt;text-align:center;
 7:       background:#000088;color:#ffff00">
 8: Message Encryption Using <br />RSA Public-Key Scheme
 9: <form action="">
10: <table style="font-size:14pt;width:720px" align="center" cellspacing=5>
11:  <tr><td>Input Message: </td></tr>
12:  <tr><td><textarea rows="5" cols="40" id="in_mesg"
13:       style="font-size:14pt;width:700px;height:150px;font-weight:bold;
14:       background:#dddddd" >Meet Me At 2pm Tomorrow</textarea></td></tr>
15:  <tr><td ><br />Enter RSA Keys (Public-Key or Secret-Key)</td></tr>
16:  <tr><td><input type="text" id="key_v" size="100" maxlength="100"
17:       style="font-size:14pt;width:700px;height:35px;font-weight:bold;
18:       background:#dddddd;color:#ff0000" value="" /></td></tr>
19:  <tr><td ><br />Encrypt The Message    
20:       <input size="20" type="button" class="butSt" style="width:180px"
21:       value="OK" onclick="rsa_en()" /></td></tr>
22:  <tr><td><textarea rows="5" cols="40" id="out_mesg" readonly
23:       style="font-size:14pt;font-weight:bold;width:700px;
24:       height:150px;background:#aaffaa"></textarea></td></tr>
25: </tbody></table>
26: </form>
```

This XHTML code displays a simple interface for the message encryption containing two text areas, one text box and one `OK` button. The plaintext is input in the first text area defined in lines 12-14. Since you can use either the public-key `[c,n]` or secret-key `[d,n]` for encryption, the key is input into the text box (see lines 16-17). When the `OK` button is pressed, the function `rsa_en()` is run and the encrypted message appears in the text area at the bottom of the browser window. The construction of the function `rsa_en()` is given in the second part of the example.

```
Example: Continuation of ex08-11.htm                              (Part II)

27: <script src="hexlib.js"></script>
28: <script src="BigInt.js"></script>
29: <script>
30:  var kSt="65537 5816988584351038538232432739075597652351 77";
31:  document.getElementById("key_v").value = kSt;
32:
33:  function rsa_en()
```

```
34:  {
35:    var keySt="",message="",llst="", keyV;
36:    document.getElementById("out_mesg").value = llst;
37:    key = document.getElementById("key_v").value;
38:    message = document.getElementById("in_mesg").value;
39:    llst = rsa_enf(key, message);
40:    document.getElementById("out_mesg").value = llst;
41:  }
42:
43:  function rsa_enf(keySt, msgSt)
44:  {
45:    var m=0,len = msgSt.length;
46:    var ret="", ttm, lrArr, keyV, tmp;
47:    keyV = myParseSt(keySt);
48:    tte = str2bigInt(keyV[0],10,keyV[0].length);
49:    ttn = str2bigInt(keyV[1],10,keyV[1].length);
50:    tmp = int2bigInt(0,10,keyV[1].length);
51:    while (m < len) {
52:      lrArr= (msgSt.charCodeAt(m++) << 24)|(msgSt.charCodeAt(m++) << 16)|
53:             (msgSt.charCodeAt(m++) << 8) | msgSt.charCodeAt(m++);
54:      ttm = int2bigInt(lrArr,10,0);
55:      copy(tmp,ttm);
56:      powMod(tmp,tte,ttn);
57:      ret += bigInt2str(tmp,10) +" ";
58:    }
59:   return ret;
60:  }
61: </script>
62: </body>
```

First, a testing key is defined in lines 30-31 and put into the input box for the key.

```
public-key [c,n] = [65537  581698858435103853823243273907559765235177]
```

This key is the public-key generated by ex08-10.htm. The controlling function `rsa_en()` captures the key and plaintext in lines 37-38 and call the encryption engine `rsa_enf()` to encrypt the data. The result is returned to variable `llst` and display by the statement in line 40.

The encryption engine takes the key (`keySt`) and plaintext (`msgSt`) as input. The contents of the key are extract by lines 47-50 into components `[tte,ttn]`. For every 4 characters of the plaintext, the statements in lines 52-53 are used to group them into 1 32-bit integer in variable `lrArr`. This value is converted to big integer as `ttm` (see line 54-55). The power modulo function in line 56 compute the ciphertext stored in variable `tmp`. This ciphertext value is converted to string and appended to the result and returned to the function caller at the end. A screen shot is shown in Fig.08.15.

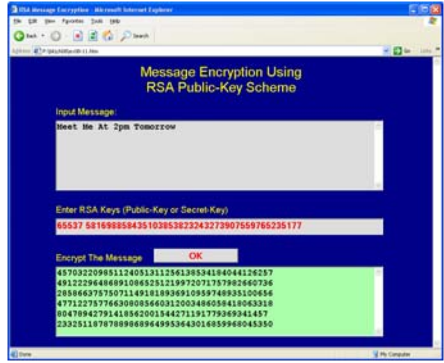


Fig.08.15: RSA Message Encryption

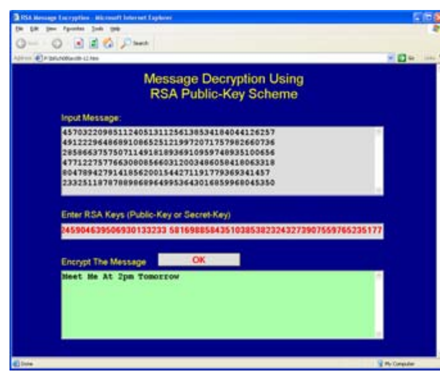


Fig.08.16: RSA Message Decryption

From Fig.08.15, we see that the plaintext "`Meet Me At 2pm Tomorrow`" contains 23 characters. This string is grouped into 6 elements and each element is a 32-bit integer. After the encryption, the first element representing the word "`Meet`" turns out to be:

```
457032209851124051311256138534184044126257
```

This is a big integer of 42 digits. It is easy to see that the size of the output depends on the size of the modulo `n=pq`. Since the modulus `n` is 42 digits, you would expect the encryption results for each element are normally 42 digits as well. For more secure encryption or longer `n`, the ciphertext will become bigger. This is one of the disadvantages for many public-key encryptions. Also bigger the number will complicate and slow down the computation. One compensation technique may be to group more characters into one bigger element using APM package.

For a decryption page, we are using the same XHTML interface as in part I of ex08-11.htm. Let's make a copy of this XHTML code and call it part I of ex08-12.htm. Inside this new page, modify the following lines:

```
8: Message Decryption Using <br />RSA Public-Key Scheme
21:       value="OK" onclick="rsa_de()" /></td></tr>
```

The first modification is for display and information only. When the `OK` button is clicked, the function `rsa_de()` in line 21 will be called to perform decryption. The details of this function are given in the second part of the page.

```
Example: ex08-12.htm - Message Decryption Using RSA Scheme        (Part II)

27: <script src="hexlib.js"></script>
28: <script src="BigInt.js"></script>
29: <script>
30:  var msgSt="457032209851124051311256138534184044126257 "+
31:            "491222964868910865251219972071757982660736 "+
32:            "285866375750711491818936910959748935100656 "+
33:            "477122757766308085660312003486058418063318 "+
34:            "80478942791418562001544271191779369341457 "+
35:            "233251187878898689649953643016859968045350 ";
36:  document.getElementById("in_mesg").value = msgSt;
37:  var kSt ="210447227268509580452459046395069301332333 "+
38:             "581698858435103853823243273907559765235177";
39:  document.getElementById("key_v").value = kSt;
40:
41:  function rsa_de()
42:  {
43:    var keySt="",message="",llst="", keyV;
44:    document.getElementById("out_mesg").value = llst;
45:    key = document.getElementById("key_v").value;
46:    message = document.getElementById("in_mesg").value;
47:    llst = rsa_def(key, message);
48:    document.getElementById("out_mesg").value = llst;
49:  }
50:
51:  function rsa_def(keySt, msgSt)
52:  {
53:    var m=0,len = msgSt.length;
54:    var ii,ttc="",ttd,ttn,ret="",tempSt,result,tresult,lrArr;
55:    keyV = myParseSt(keySt);
56:    ttd = str2bigInt(keyV[0],10,keyV[0].length);
```

```
57:    ttn = str2bigInt(keyV[1],10,keyV[1].length);
58:    lrArr = myParseSt(msgSt);
59:    for (ii=0; ii < lrArr.length; ii++)  {
60:      ttc = str2bigInt(lrArr[ii],10,lrArr[ii].length);
61:      powMod(ttc,ttd,ttn);
62:      result  = bigInt2str(ttc,10);
63:      tresult = parseInt(result,10);
64:      tempSt= String.fromCharCode(
65:        ((tresult >>> 24) & 0xff), ((tresult >>> 16) & 0xff),
66:        ((tresult >>>  8) & 0xff), ( tresult         & 0xff));
67:     ret += tempSt;
68:   }
69:   return ret;
70:  }
71: </script>
72: </body>
```

Lines 30-35 define a testing ciphertest and put into the first text box by the statement in line 36. This ciphertext is obtained from ex08-11.htm. A testing secret-key also provided by lines 37-39.

```
[d,n] =   [ 2104472272685095804524590463950693013323 3
              5816988584351038538232432739075597652351 77 ]
```

This key is the corresponding secret-key used in example ex08-11.htm so that the following original plaintext is expected:

```
"Meet Me At 2pm Tomorrow"
```

The statements in lines 45-46 inside function `rsa_de()` captures the input and key strings. Then the decryption engine `rsa_def()` is called. The decrypted result is display at the bottom of the text area by line 48.

The decryption engine `rsa_def()` is defined in lines 51-70. This function takes the key and message strings as input. The decryption key components `[d,n]` is extracted from the key string by lines 55-57. Each big number in the input ciphertext is parsed into big integer variable `ttc`. The power modulo function `powMod()` in line 61 performs the decryption. Although the result is a big integer, we know that it is a 32-bit integer and can be converted into a string of 4 characters by the statement in lines 64-66. This string is added into the variable ret and returned to the function called by line 69. A screen shot of this example is shown in Fig.08.16.

Note that the RSA encryption and decryption formulas are symmetric. In other words, both the public and secret key can be used for encryption. If you use one key for encryption, the other key must be used for decryption. This feature is the foundation of digital signature and creates a new application chapter for security application. From now on, remember that both keys can be used for encryption.

8.4.4 Sending and Receiving Secure Message with RSA Digital Signature

In this section, we consider how to use RSA scheme to construct a system to send and receive secure message with signatures. The main idea is described as follows:

Sending a secure message with digital signature:

- Encrypt the document using the sender's secret-key.
  (Signing the document)
- Encrypt the document again using the receiver's public-key

Receiving the secure message:

- Decrypt the document using the receiver's secret-key.
- Decrypt the document again using the sender's public-key
  (This process is equivalent to verifying the sender's digital signature)

Given the RSA `Public-key=[e,n]` and secret-key `=[d,n]`, the encryption/decryption on message m are governed by the following equations:

Encryption: $c = m^e \bmod n$ Decryption: $m = c^d \bmod n$

Remember that both keys can be used for encryption. If you use one key for encryption, the other key must be used for decryption. To build the page, we first use the following XHTML interface.

```
Example: ex08-13.htm - Sending Secure Message With Signature      (Part I)

 1: <head><title>Sending RSA Secure Message</title></head>
 2: <style>
 3:  .butSt{font-size:14pt;width:250px;height:30px;
 4:          font-weight:bold;background:#dddddd;color:#ff0000}
 5: </style>
 6: <body style="font-family:arial;font-size:22pt;text-align:center;
 7:      background:#000088;color:#ffff00">
 8: Sending RSA Secure Message with <br />Digital Signature
 9: <form action="">
10: <table style="font-size:14pt;width:720px;text-align:left" align=center>
11:  <tr><td>Input Message: </td></tr>
12:  <tr><td><textarea rows="5" cols="40" id="in_mesg"
13:      style="font-size:14pt;width:700px;height:150px;font-weight:bold;
14:      background:#dddddd" >Meet Me At 2pm Tomorrow</textarea></td></tr>
15:  <tr><td ><br />Sender's RSA Secret-Key -- Signing Document</td></tr>
16:  <tr><td><input type="text" id="s_key" size="100" maxlength="100"
17:      style="font-size:14pt;width:700px;height:35px;font-weight:bold;
18:      background:#dddddd;color:#ff0000" value="" /></td></tr>
19:  <tr><td >Reciever's RSA Public-Key -- Intended Receiver</td></tr>
20:  <tr><td><input type="text" id="p_key" size="100" maxlength="100"
21:      style="font-size:14pt;width:700px;height:35px;font-weight:bold;
22:      background:#dddddd;color:#ff0000" value="" /></td></tr>
23:  <tr><td ><br />Encrypt The Message    
24:      <input size="20" type="button" class="butSt" style="width:180px"
25:      value="OK" onclick="rsa_en()" /></td></tr>
26:  <tr><td><textarea rows="5" cols="40" id="out_mesg" readonly
27:      style="font-size:14pt;font-weight:bold;width:700px;
28:      height:150px;background:#aaffaa"></textarea></td></tr>
29: </table>
30: </form>
```

This interface contains two text areas, two text boxes, and on `OK` button. The first text area allows the sender to input a message. The first text box is for sender's secret-key so that the message can be signed. The second text box is for receiver's public-key so that the signed message can be encrypted and dedicated for the intended receiver. Once the `OK` button is clicked, the message will be signed and encrypted for that particular receiver. In an ideal situation, one the intended receiver can decrypt the message and use the public-key of the sender to identify the signature.

The processing engine is the function `rsa_en()` (see line 25). This function is given in the second part of the page.

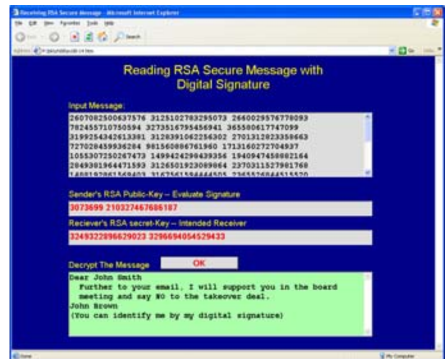
```
Example: Continuation of ex08-13.htm                              (Part II)

31: <script src="hexlib.js"></script>
32: <script src="BigInt.js"></script>
33: <script>
34:  var skSt="42440281951699 210327467686187";
35:  var pkSt="3830539 329669405452943 3";
36:  document.getElementById("s_key").value = skSt;
37:  document.getElementById("p_key").value = pkSt;
38:
39:  function rsa_en()
40:  {
41:   var skey="",pkey="",message="",llst="";
42:   document.getElementById("out_mesg").value = llst;
43:   skey = document.getElementById("s_key").value;
44:   pkey = document.getElementById("p_key").value;
45:   message = document.getElementById("in_mesg").value;
46:   llst = rsa_enf(skey,pkey,message);
47:   document.getElementById("out_mesg").value = llst;
48:  }
49:  function rsa_enf(skeySt,pkeySt,msgSt)
50:  {
51:   var m=0,len = msgSt.length,ttm,tts,ttp,ttns,ttnp;
52:   var ret="",lrArr,pkeyV,skeyV,tmp;
53:   skeyV = myParseSt(skeySt);
54:      tts = str2bigInt(skeyV[0],10,skeyV[0].length);
55:      ttns = str2bigInt(skeyV[1],10,skeyV[1].length);
56:   pkeyV = myParseSt(pkeySt);
57:      ttp = str2bigInt(pkeyV[0],10,pkeyV[0].length);
58:      ttnp = str2bigInt(pkeyV[1],10,pkeyV[1].length);
59:   tmp = int2bigInt(0,10,pkeyV[1].length);
60:   while (m < len) {
61:     lrArr= (msgSt.charCodeAt(m++) << 24)|(msgSt.charCodeAt(m++) << 16)|
62:            (msgSt.charCodeAt(m++) << 8) | msgSt.charCodeAt(m++);
63:     ttm = int2bigInt(lrArr,10,5);
64:     copy(tmp,ttm);
65:     powMod(tmp,tts,ttns);
66:     powMod(tmp,ttp,ttnp);
67:     ret += bigInt2str(tmp,10) +" ";
68:   }
69:   return ret;
70:  }
71: </script>
72: </body
```

To create a testing sample for this page, lines 34-35 store the secret-key of a sender and the public-key of a receiver. These two keys are put into the text boxes by lines 36-37. When the `OK` button is pressed, the `rsa_en()` function is run. The secret-key, public-key, and message are captured by lines 43-45. These strings are input to the function `rsa_enf()` for the signing and encryption processes. The result is returned to variable `llst` and printout to the text area at the bottom of the screen.

Inside the function `rsa_enf()`, the components of sender's secret-key `[d,n]` are extracted to variables `tts` and `ttns` (see lines 53-55). The components of receiver's public-key `[e,n]` are extracted to variables `ttp`, and `ttnp` (see lines 56-58). The while-loop is used for processing every 4 characters of the plaintext into a 32-bit integer. This integer converts to big integer `ttm` by lines 63-64. The first power modulo function in line 65 is equivalent to signing the message using the secret-

key of the sender. The next power modulo function (see line 66) performs the encryption on the signed message so that only the intended person can decrypt and read. The result is converted into string and added into variable ret and return to the function caller at line 69.

Suppose you have a sensitive letter, you can sign and encrypt it as shown in Fig.08.17.



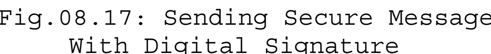


Fig.08.17: Sending Secure Message With Digital Signature

Fig.08.18: Receiving Secure Message With Digital Signature

For decryption and verifying the signature process, a similar page is needed. To construct this page, we first make a copy of the part I of ex08-13.htm and call it ex08-14.htm. Inside this new page, modify the following lines:

```
8: Receiving RSA Secure Message with <br />Digital Signature
15:  <tr><td ><br />Sender's RSA Public-Key -- Evaluate Signature</td></tr>
16:  <tr><td><input type="text" id="p_key" size="100" maxlength="100"
19:  <tr><td >Reciever's RSA secret-Key -- Intended Receiver</td></tr>
20:  <tr><td><input type="text" id="s_key" size="100" maxlength="100"
25:       value="OK" onclick="rsa_de()" /></td></tr>
```

Line 8 changes the displayed title. Lines 15-16 ask the user to input the sender's public-key so that the signature of sender can be identified properly. Lines 19-20 is for user's (or receiver's) secret-key so that the message can be decrypted. When the `OK` button is clicked, the function `rsa_de()` is run for the entire process. This function is given in the second part of the page.

```
Example: Continuation of ex08-14.htm                              (Part II)

31: <script src="hexlib.js"></script>
32: <script src="BigInt.js"></script>
33: <script>
34:  var pkSt="3073699 210327467686187";
35:  var skSt="3249322896629023 3296694054529433";
36:  var cSt = "117609435150577 261542301093665 118076629232528 "+
37:            "1793254257998247 1709547461124399 1542346838415894 ";
38:  document.getElementById("p_key").value = pkSt;
39:  document.getElementById("s_key").value = skSt;
40:  document.getElementById("in_mesg").value = cSt;
41:
42:  function rsa_de()
43:  {
44:   var skey="",pkey="",message="",llst="";
45:   document.getElementById("out_mesg").value = llst;
46:   skey = document.getElementById("s_key").value;
47:   pkey = document.getElementById("p_key").value;
48:   message = document.getElementById("in_mesg").value;
49:   llst = rsa_def(skey,pkey,message);
```

```
50:   document.getElementById("out_mesg").value = llst;
51: }
52: function rsa_def(skeySt,pkeySt,msgSt)
53: {
54:  var m=0,len = msgSt.length,ttc,tts,ttns,ttp,ttnp,ttn;
55:  var ii,ret="",tempSt,result,tresult,lrArr;
56:  skeyV = myParseSt(skeySt);
57:    tts = str2bigInt(skeyV[0],10,skeyV[0].length);
58:    ttns = str2bigInt(skeyV[1],10,skeyV[1].length);
59:  pkeyV = myParseSt(pkeySt);
60:    ttp = str2bigInt(pkeyV[0],10,pkeyV[0].length);
61:    ttnp = str2bigInt(pkeyV[1],10,pkeyV[1].length);
62:  lrArr = myParseSt(msgSt);
63:  for (ii=0; ii < lrArr.length; ii++)  {
64:    ttc = str2bigInt(lrArr[ii],10,lrArr[0].length);
65:    powMod(ttc,tts,ttns);
66:    powMod(ttc,ttp,ttnp);
67:    result  = bigInt2str(ttc,10);
68:    tresult = parseInt(result,10);
69:    tempSt= String.fromCharCode(
70:      ((tresult >>> 24) & 0xff), ((tresult >>> 16) & 0xff),
71:      ((tresult >>>  8) & 0xff), ( tresult         & 0xff));
72:    ret += tempSt;
73:  }
74:  return ret;
75: }
76: </script>
77: </body>
```

For a testing sample, lines 34-37 store the sender's public-key, receiver's secret-key and a sample of ciphertext. These values are put into the appropriated text area and boxes. When the OK button is pressed, the function `rsa_de()` is run. This function will capture the keys and ciphertext into variables `skey`, `pkey`, and `msgSt` respectively. The function call `rsa_def()` in line 49 performs the entire process. The result is returned to variable llst and output to the text area at the bottom of the browser window.

Inside the function `rsa_def()`, the sender's public-key components `[e,n]` are extracted into variables `tts`, and `ttns`. Similarly, the receiver's secret-key components `[d,n]` are extracted into variables `ttp` and `ttnp` respectively. The for-loop in lines 63-73 will process each big number in the ciphertext. The first power modulo function in line 65 is to decrypt the message first using the receiver's secret-key. After the decryption, the sender's public-key is used to identify the signature. The result is the plaintext in big number format. The statements in lines 68-72 convert the big number into character format and added into variable ret. After the entire process, the plaintext is stored in ret and returned to the function caller at line 74 for display purpose. Using this example, the decryption and signature verifying result of ex08-13.htm is shown in Fig.08.18. As you can see from this figure, the original sensitive letter is displayed in the text area at the bottom of the browser window.

8.4.5 Building a Hybrid Encryption Scheme: RSA + AES

Since the invention of public-key schemes in the early of 70s, public-key encryptions have provided a solution to a number of difficult problems such as key-exchange and digital signatures involving modern cryptology and information security. However, from the experience of last few sections, some of the disadvantages of public-key schemes are:

- Calculations are, in general, involving big numbers. Even you have good APM package, the computations are slow.
- Encrypted messages are bigger than conventional symmetric-key encryption.
- Security-strength of public-key schemes are generally weaker than convention encryption.

These disadvantages make it neither ideal nor efficient to encrypt even a small document. Together with weaker security-strength, the applications of public-key schemes are restricted. To compensate this problem, the idea of hybrid encryption established.

General speaking, hybrid encryption is a combination of conventional symmetric-key and public-key schemes. The main purpose is to take the advantages of both crypto-systems and solving security problems.

One of the popular applications of hybrid encryption is described below:

- Perform message encryption/decryption using symmetric-key methods
- Protect and transmit the symmetric-key password using public-key encryption

In this case, the plaintext message is protected by a much tougher symmetric-key encryption. If you want to transmit the encrypted message to someone, all you need is to encrypt the password using public-key of the receiver.

Based on this idea, a hybrid encryption is developed in this section using the MD5, AES and RSA schemes. The structure is simple. Given a character message m, a password or key is constructed by the following command:

```
key = md5(m +Date()+Math.random())
```

This will combine the message m, date and a random number together. After the `md5()` function, a message digest value (128-bit) is obtained as key. This key is used as the password to encrypt the message m with AES scheme, i.e.

```
c0 = AES(m,key)
```

When you want to send the ciphertext c0 to someone such as Bob, you can encrypt the key with Bob's public-key Bob_P, i.e.

```
c1= RSA(key,Bob_P)
```

After concatenate `c1` and `c0` together, the result is send to Bob for decryption. Consider the page below:

```
Example: ex08-15.htm - Message Encryption With A Hybrid Scheme    (Part I)

 1: <head><title>Hybrid Message Encryption</title></head>
 2: <style>
 3:  .butSt{font-size:14pt;width:250px;height:30px;
 4:       font-weight:bold;background:#dddddd;color:#ff0000}
 5:  .txtArea{font-size:14pt;width:780px;height:130px;font-weight:bold;
 6:       background:#dddddd}
 7: </style>
 8: <body style="font-family:arial;font-size:22pt;text-align:center;
 9:       background:#000088;color:#ffff00">
```

```
10: Message Encryption Using <br />A Hybrid Scheme: (RSA + MD5 + AES)
11: <form action="">
12: <table style="font-size:14pt;width:800px" align="center" cellspacing=5>
13:  <tr><td>Input Message: </td></tr>
14:  <tr><td><textarea rows="5" cols="40" id="in_mesg"
15:       class="txtArea">Meet Me At 2pm Tomorrow</textarea></td></tr>
16:  <tr><td >Enter RSA Keys (Public-Key or Secret-Key)</td></tr>
17:  <tr><td><textarea rows="5" cols="40" id="key_v" class="txtArea"
18:       style="height:60px"></textarea></td></tr>
19:  <tr><td >Encrypt The Message    
20:       <input size="20" type="button" class="butSt" style="width:180px"
21:       value="OK" onclick="hyb_en()" /></td></tr>
22:  <tr><td><textarea rows="5" cols="40" id="out_mesg" class="txtArea"
23:       readonly style="background:#00ee00"></textarea></td></tr>
24: </tbody></table>
25: </form>
```

This XHTML codes in this page generates three text areas. The first text area is for the plaintext. The second text area allows a user to input the RSA public-key `[e,n]` of the recipient. When the `OK` button is clicked at line 21, the function `hyb_en()` is run. As a result, a session key combining the MD5 hash, plaintext message, date, and a random number is generated. This key is used to encrypt the plaintext using AES scheme. The encrypted result is displayed in the text area at the bottom of the browser window. The function `hyb_en()` is given in the second part of the page.

```
Example: Continuation of ex08-15.htm                          (Part II)

26: <script src="hexlib.js"></script>
27: <script src="ex06-09.js"></script>
28: <script src="ex03-05.js"></script>
29: <script src="BigInt.js"></script>
30: <script>
31:  var kSt="5310075487163797179490709 "+
32:          "349309920095549615294937187063756384577";
33:  document.getElementById("key_v").value = kSt;
34:
35:  function hyb_en()
36:  {
37:    var keySt="",message="",llst="", keyV;
38:    document.getElementById("out_mesg").value = llst;
39:    key = document.getElementById("key_v").value;
40:    message = document.getElementById("in_mesg").value;
41:    llst = hyb_enf(key, message);
42:    document.getElementById("out_mesg").value = llst;
43:  }
44:  function hyb_enf(keySt, msgSt)
45:  {
46:    var m=0,len = msgSt.length;
47:    var ret="", ttm, lrArr, keyV, tmp;
48:    keyV = myParseSt(keySt);
49:      tte = str2bigInt(keyV[0],10,keyV[0].length);
50:      ttn = str2bigInt(keyV[1],10,keyV[1].length);
51:    tmp = int2bigInt(0,10,keyV[1].length+1);
52:    sessionKey = md5(msgSt + Date() + Math.random());
53:
54:    sessB = str2bigInt(sessionKey,16,sessionKey.length);
55:    if (greater(sessB,ttn)) mod(sessB,ttn);
56:
```

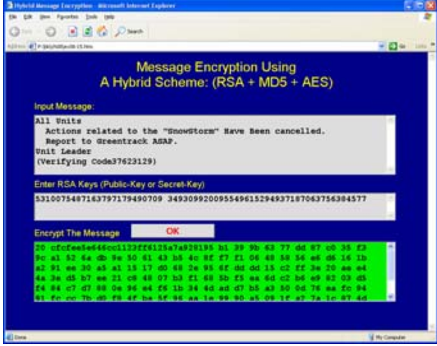
```
57:     keySt = bigInt2str(sessB,16);
58:     sessionKey = keySt.toLowerCase();
59:
60:     llst = byteStToHex(aes(sessionKey, msgSt,true));
61:     tmp = str2bigInt(sessionKey,16,sessionKey.length);
62:     powMod(tmp,tte,ttn);
63:     otmpSt = bigInt2str(tmp,16);
64:     otmpSt = otmpSt.toLowerCase();
65:     ret = byteToHex(otmpSt.length)+" "+otmpSt+" "+llst;
66:     return ret;
67:   }
68: </script>
69: </body>
```

First, the MD5 and AES codes are included into the page in lines 27-28. Lines 31-33 store values of a testing public-key `[e,n]`, this key will be put into the interface page and used to encrypt a session key generated by the program. When the OK button is clicked, the function `hyb_fun()` run. First, this function captures the plaintext and the encryption key information into variables message and key respectively. These values are input to the encryption engine `hyb_enf()` in line 41. This function is defined in lines 44-67.

First, the information `[e,n]` of the key string `keySt` is extracted by lines 48-50 into variables `tte` and `ttn`. The next step is to build an session key (see line 52):

```
sessionKey = md5(msgSt + Date() + Math.random());
```

This key is a MD5 hash value combining the plaintext, current date and a random number. The hash is a 128 bit value in hexadecimal character format. When this value converted into number `sessB` by line 54, it can be bigger than the modulus of the public-key `ttn`. Therefore the conditional-if statement in line 55 to make sure that `sessB` is smaller than `ttn`. Once this value is available, it is converted into string (lower case string) and used to encrypt the plaintext with AES scheme (see lines 60). Now, the remaining task is to encrypt the session key by the RSA public-key `[e,n]`. The encryption result is stored in variable `otmpSt`. This `otmpSt` together with the AES encryption `llst` are returned to the function caller by line 65 so that all information are include in the ciphertext. In practice, this may not be an ideal situation since the intruder can extract the session key from the ciphertext. This may compromise the security of the scheme. Under the Kerchkoff principle, secrecy of the scheme should not be assumed. The design here is mainly for our demonstration and educational purposes. A screen shot of this hybrid encryption is shown in Fig.08.19.



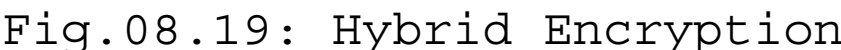


Fig.08.19: Hybrid Encryption

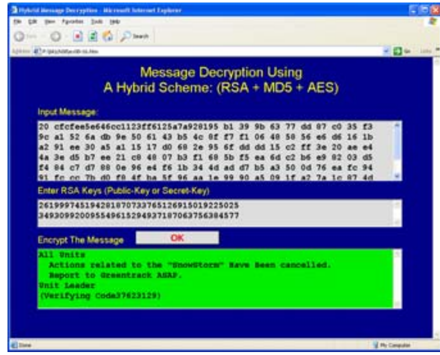


Fig.08.20: Hybrid Decryption

For the decryption, the same interface is used. Let's make a copy of part I of ex08-15.htm and call it ex08-16.htm. Inside this new page, modify the following two lines:

```
10: Message Decryption Using <br />A Hybrid Scheme: (RSA + MD5 + AES)
21:       value="OK" onclick="hyb_de()" /></td></tr>
```

Line 10 changes the displaying title. When the `OK` button is clicked, the decryption function `hyb_de()` is called at line 21 instead. This function is given in the second part of the page.

```
Example: ex08-16.htm - Message Encryption With A Hybrid Scheme    (Part II)

26: <script src="hexlib.js"></script>
27: <script src="ex06-09.js"></script>
28: <script src="BigInt.js"></script>
29: <script>
30:  var msgSt="20 bb4fc4dc69258c71575137bd5b1a9022 "+
31:            "95 2b a5 05 46 ce d0 f3 8a 77 20 f1 37 80 ae "+
32:            "74 24 df 17 bd b7 cf 5a 6f 16 83 e0 "+
33:            "c5 20 a1 91 c9 ";
34:  document.getElementById("in_mesg").value = msgSt;
35:  var kSt ="2619997451942818707337651269150l9225025 "+
36:           "349309920095549615294937187063756384577";
37:  document.getElementById("key_v").value = kSt;
38:
39:  function hyb_de()
40:  {
41:    var keySt="",message="",llst="", keyV;
42:    document.getElementById("out_mesg").value = llst;
43:    key = document.getElementById("key_v").value;
44:    message = document.getElementById("in_mesg").value;
45:    llst = hyb_def(key, message);
46:    document.getElementById("out_mesg").value = llst;
47:  }
48:
49:  function hyb_def(keySt, msgSt)
50:  {
51:    var m=0,len = msgSt.length;
52:    var ii,ttc="",ttd,ttn,ret="",tempSt,result,tresult,lrArr;
53:    keyV = myParseSt(keySt);
54:      ttd = str2bigInt(keyV[0],10,keyV[0].length);
55:      ttn = str2bigInt(keyV[1],10,keyV[1].length);
56:    ind = parseInt(msgSt.substring(0,2),16);
57:    sessionKey = msgSt.substring(3,3+ind);
58:    ttc = str2bigInt(sessionKey,16,sessionKey.length);
59:    powMod(ttc,ttd,ttn);
60:    sessionKey = bigInt2str(ttc,16);
61:    sessionKey = sessionKey.toLowerCase();
62:
63:    msgSt = msgSt.substring(3+ind,msgSt.length);
64:    message = hexStToByteSt(msgSt)
65:    ret = aes(sessionKey, message, false);
66:   return ret;
67:  }
68: </script>
69: </body>
```

Lines 30-37 store values of a testing ciphertext and secret-key `[d,n]` of the receiver. When the `OK` button is clicked, the function `hyb_de()` captures the information of the plaintext and key (see lines 43-44) and perform the decryption by calling the engine `hyb_def()` at line 45.

The decryption engine `hyb_def()` is defined in lines 49-67. First, the information of the decryption key `[d,n]` are extracted into variables `ttd` and `ttn` respectively. The encrypted session key is

extracted by lines 56-57. After the RSA decryption process in lines 58-61, the session key is obtained in variable `sessionKey`. The AES encrypted message is extracted by line 63. This ciphertext is decrypted by the `sessionKey` in line 65. The result or the plaintext is stored in variable ret and returned to the function caller for display purposes. A screen shot of this example in action is shown in Fig.08.20. From this figure, you can see that, the plaintext from Fig.08.19 is appeared at the decryption box in Fig.08.20.

The public-key schemes discussed thus far are all based on the number theory and factorization over Galois Fields (GF). The next popular public-key that we are going to introduce is slightly different. It is depended on a special kind of curves called “Elliptic Curves”.

8.5 Elliptic Curves and Public-Key Encryption/Decryption

8.5.1 What are Elliptic Curves?

An elliptic curve is the set of solutions (x,y) to an equation of the form:

$$y^2 = x^3 + ax + b \qquad \text{(Elliptic Curve)}$$

where a and b are constants. Note that the right hand side is a special cubic polynomial. The left hand side is a quadratic term. In order to prevent repeated root on the right hand side, the constants a and b are restricted by:

$$4a^3 + 27b^2 \neq 0$$

Despite its simple form, elliptic curves are studied for many years and have many significant applications in the history of mathematics. May be one of the most interesting results related to application of elliptic curves is that they are used to prove the “Fermat’s Last Theorem”. Elliptic curve encryption/decryption or cryptography was proposed and studied by Victor Miller and Neal Koblitz in the mid 1980s and quickly becomes one of the popular subjects in cryptology.

In general, public-key encryption/decryption with elliptic curves, in many cases, can provide faster algorithm, smaller keys and keeping the same level of security. The advantages come from using a different kind of mathematical entity called group for public-key arithmetic. We will discuss this shortly. First, let’s see some elliptic curves below:

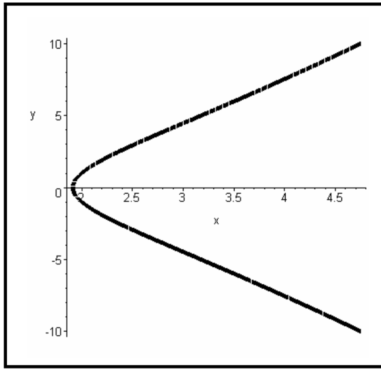


Fig.08.21: $y^2 = x^3 - 7$

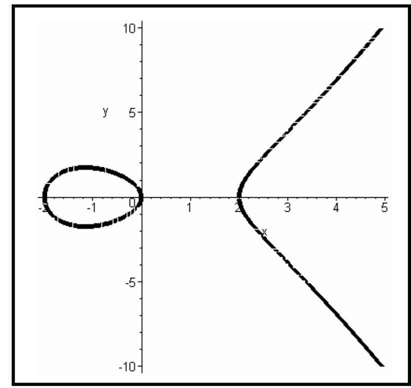


Fig.08.22: $y^2 = x^3 - 4x$

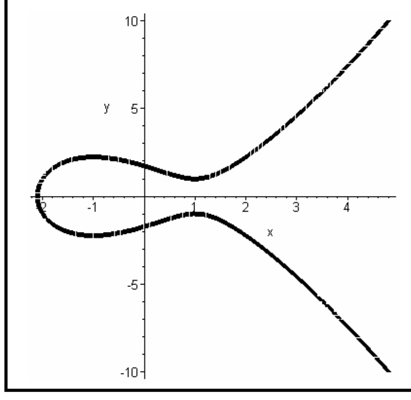


Fig.08.23 $y^2 = x^3 - 3x + 3$

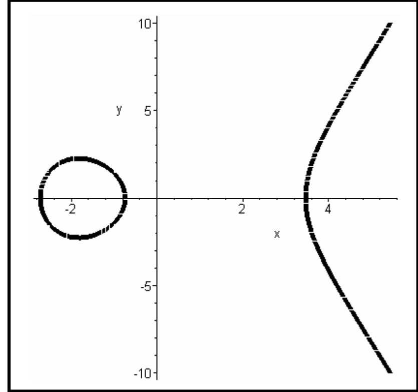


Fig.08.24 $y^2 = x^3 - 10x - 7$

One of the interesting and powerful features of elliptic curves is that by using a special "add" operation, any two points add together will result a third point on the same curve.

Adding Points on an Elliptic Curve

Given two points `u=(x1,y1)` and `v=(x2,y2)` on the elliptic curve, the point `u+v` is calculated by:

- Draw a straight line through `u` and `v`. and find the third intersecting point `w`.
- Draw a vertical line through `w` (and `O`) and find the third intersecting point `u+v`.

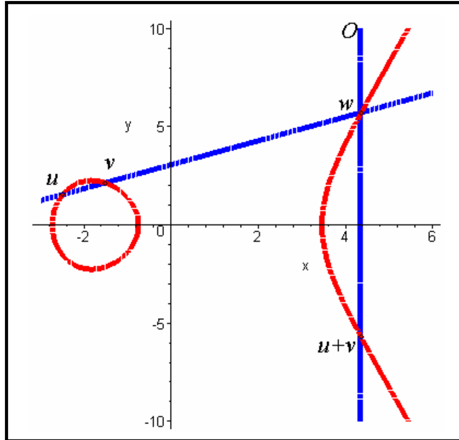


Fig.08.22: Adding Two points on Elliptic Curve

To make this rule works or well-defined, we assume that

- An extra imaginary point O on the curve, also known as the identity element, or the point at infinity. It has no specific `(x,y)` coordinates, but one might imagine that its location is infinitely high above the curve where all vertical lines converge.

- A line tangent to a point on the curve is said to intersect the point twice. Think of the tangent as the limit of a line through two distinct points as the points approach each other.

If we consider the `O` is the identity element, an elliptic curve with the addition operation forms a group.

Based on the elliptic curve addition, scalar multiplications are also defined. Given a scalar integer s and a point u on an elliptic curve, the "Scalar Multiplication" is defined by the following addition:

$$(s)\, u = \underbrace{u + u + \ldots + u}_{(s\ times)}$$

Now, let's consider how to compute addition of two points on an elliptic curve. Suppose `u=(x1,y1)` and `v=(x2,y2)` are two point on an elliptic curve `E`. The addition result is another point `w=(x3,y3)` on the same curve calculated by the following "`Addition Formula`" below:

$$x_3 = \lambda^2 - x_1 - x_2 \qquad y_3 = \lambda\,(x_1 - x_3) - y_1 \qquad \lambda = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} & for\ x_1 \neq x_2 \\ \dfrac{3\,x_1^2 + a}{2\,y_1} & for\ x_1 = x_2 \end{cases}$$

For encryption and decryption, we always want integer operations and therefore elliptic curves over finite fields are considered. The most popular finite fields for this are the integers modulo a prime number p. With this formula, the computations are carried out simply with `a (mod p)` operation. Now, let's see how to use elliptic curve to perform encryption and decryption.

8.5.2 Elliptic Curve Cryptography (ECC)

The subject to use elliptic curves for encryption/decryption is called "Elliptic Curve Cryptography" (ECC). For this purpose, all points on an elliptic curve are considered as integers over a finite field or more precisely a field generated by a large prime number. Instead of real (or floating point) numbers, computations (or additions) are perform with modulus of a prime number p. The use of elliptic curves over finite fields as a basis for encryption/decryption was first suggested by Koblitz,

Given a prime number p, the elliptic curve E over the finite field GF(p) is given by:

$$E : \ y^2 = x^3 + ax + b \qquad \bmod p$$

where `x,y,a,b` are elements of `{1,2,3,..,p-1}`. In computational form, an elliptic curve in `GF(p)` is uniquely defined by elements `[a,b,p]` and therefore is represented by `E:[a,b,p]`

In general, ECC schemes are public-key based. In other words, a public-key and a secret-key are generated. Normally, the public-key is used for encryption and the secret-key is for the decryption.

Generating the Public and Secret Keys of a ECC scheme

To generate the ECC public and secret keys, the following steps are required:

- Select a base point called B on the curve.
- Pick a random integer, k (secret-key) and kept secret.
- Compute the point K (public-key) by the scalar multiplication: K = k*B.

The public and secret keys of the elliptic curve system are:

```
Secret-Key: [k,B,a,b,p]      and       Public-Key:  [K,B,a,b,p]
```

Note that the secret-key k is a value whereas the public-key K is a point with x-y coordinates. In practice and in terms of x-y co-ordinate values, we have:

```
Secret-Key:[k,Bx,By,a,b,p]     and     Public-Key:[Kx,Ky,Bx,By,a,b,p]
```

All values are computed against modulo `p` (or `mod p`). The difficulty in obtaining the private key from the public key is based on the discrete log problem (DLP) for elliptic curves. The DLP states that given a point K and a base point B, it is difficult to find an integer k such that:

```
K = k*B   mod  p
```

To compare the security-strength, the NIST and ANSI X9 recommend the following key size for RSA, ECC (Elliptic Curve Cryptology) and Symmetric-key:

Encryption Schemes	Minimum Key Size
RSA Public-Key	1024 Bits
Elliptic Public-Key	160 Bits
Symmetric-Key	80 Bits

From this table, the security-strength of elliptic curve is much stronger than RSA scheme. When compare to symmetric-key encryption, elliptic curve is achieving half of the strength. Now, we consider how to performed encryption/decryption on points of an elliptic curve.

Encryption and Decryption on Elliptic Curves

Suppose Sue wants to send a sensitive message to Bob with the following public and secret keys:

```
Bob's Public-Key: [K,B,a,b,p]     and    Bob's Secret-Key [k,B,a,b,p]
```

Suppose the message is represented as a point `M` on the elliptic curve `E:[a,b,p]`. Sue can use Bob's public-key to encrypt the message `M` as follows:

- Select a random integer, `r`, and compute the ciphertext pair: `[c0 c1]` where

```
c0 = r*B     c1 = M + r*K
```

- Send the ciphertext `[c0 c1]` to Bob
- To decrypt the ciphertext, Bob multiply the first component by his secret-key `k`, and subtract from the second component, i.e.

```
c1 - k*c0 = (M + r*K ) - k*(r*B) = M + r*(k*B) - k*(r*B) = M
```

From these steps, we can see that both the encryption and decryption operations are relied on the addition and scalar multiplication of points on the elliptic curve.

8.5.3 Adding Two Points On An Elliptic Curve

Suppose we have an elliptic curve E defined by the equation over a finite field `GF(p)`:

$$E : y^2 = x^3 + ax + b \mod p \quad \text{where} \quad 4a^3 + 27b^2 \neq 0$$

Computationally, this curve is uniquely defined by the values `E:[a,b,p]`. For security strength, the value `p` is assumed to be a big prime number.

From section 8.5.4, we know that to generate the public and secret key of this ECC system, implementation on the “Scalar Multiplication” of point on `E` is necessary. Also, scalar multiplication computation is carried out by the “Addition Formula”. Consider the following example:

```
Example: ex08-17.htm - Adding Two Points On An Elliptic Curve     (Part I)

 1: <head><title>Adding Two Points On An Elliptic Curve</title></head>
 2: <style>
 3:  .butSt{font-size:16pt;width:70px;height:35px;
 4:           font-weight:bold;background:#ddffdd;color:#ff0000}
 5:  .txtSt{font-size:14pt;width:500px;height:35px;font-weight:bold;
 6:         background:#eeeeee;color:#ff0000}
 7:  .txtArea{font-size:14pt;width:630px;height:100px;font-weight:bold;
 8:       background:#aaffaa}
 9: </style>
10: <body style="font-family:arial;font-size:20pt;text-align:center;
11:       background:#000088;color:#ffff00">
12:  Adding Two Points P[x y] and Q[x y] <br />On An Elliptic Curve<br />
13:  <span style="font-size:16pt"><br />E:    y<sup>2</sup>
14:      = x<sup>3</sup> + ax +b     mod    p
15:  </span><br /><br/>
16: <form action="">
17: <table style="font-size:16pt;width:680px" cellspacing="5" align="center">
18:  <tr><td style="width:130px">Point P:[x y]</td><td>
19:      <input type="text" id="tp" size="132" maxlength="132"
20:        class="txtSt" style="color:#008800" value="0 18" /></td></tr>
21:  <tr><td>Point Q:[x y]</td><td >
22:      <input type="text" id="tq" size="132" maxlength="132"
23:        class="txtSt" style="color:#008800" value="18 11" /></td></tr>
24:  <tr><td>E: [a b p]</td><td >
25:      <input type="text" id="te" size="132" maxlength="142"
26:        class="txtSt" style="color:#008800" value="4 14 31" /></td></tr>
27:  <tr><td colspan="2"> Calculating (P+Q)[x y] on Curve E   
28:      <input type="button" class="butSt" value="OK"
29:        onclick="add_fun()" /></td></tr>
30:  <tr><td colspan="2"><textarea rows="5" cols="40" id="out_mesg"
31:        class="txtArea" readonly ></textarea></td></tr>
32: </table></form>
33: <script src="BigInt.js"></script>
34: <script src="hexlib.js"></script>
35: <script src="elliptic.js"></script>
```

This XHTML code is the interface part of the example. It generates three text boxes, one push button, and one text area. The three text boxes are designed to obtain the points `P`, `Q`, and the elliptic curve E from the user. When the OK button is clicked, the function `add_fun()` is run (see line 29). This function performs the addition of points `P` and `Q`. The resulting point will be displayed in the text area at the bottom of the browser window.

In addition to the usual script files BigInt.js and hexlib.js, a new script called elliptic.js is included in line 35. This file contains all functions necessary to handle elliptic curves for the rest of this chapter.

The function `add_fun()` is provided in the second part of the example.

```
Example: Continuation of ex08-17.htm                              (Part II)

36: <script>
37:   pSt = "1773056069879938 1092467599646463";
38:   qSt = "1773056069879938 1092467599646463";
39:   eSt = "589758509 280934777552806 2237171265681283";
40:   document.getElementById("tp").value = pSt;
41:   document.getElementById("tq").value = qSt;
42:   document.getElementById("te").value = eSt;
43: function add_fun()
44: {
45:   var tmpSt,ta,tb,tp,tPx,tPy,tQx,tQy,tRx,tRy,lDigits=10;
46:   ta   = int2bigInt(0,10,lDigits);     tb   = int2bigInt(0,10,lDigits);
47:   tp   = int2bigInt(0,10,lDigits);
48:   tPx  = int2bigInt(0,10,lDigits);     tPy  = int2bigInt(0,10,lDigits);
49:   tQx  = int2bigInt(0,10,lDigits);     tQy  = int2bigInt(0,10,lDigits);
50:   tRx  = int2bigInt(0,10,lDigits);     tRy  = int2bigInt(0,10,lDigits);
51:
52:   tmpSt = document.getElementById("te").value;
53:   eleV=myParseSt(tmpSt);               ta=str2bigInt(eleV[0],10,lDigits);
54:   tb=str2bigInt(eleV[1],10,lDigits);   tp=str2bigInt(eleV[2],10,lDigits);
55:   while (negative(ta)) add(ta,tp);     while (negative(tb)) add(tb,tp);
56:
57:   tmpSt=document.getElementById("tp").value;
58:   eleV1=myParseSt(tmpSt);
59:   tPx=str2bigInt(eleV1[0],10,lDigits); tPy=str2bigInt(eleV1[1],10,lDigits);
60:
61:   tmpSt=document.getElementById("tq").value;
62:   eleV2=myParseSt(tmpSt);
63:   tQx=str2bigInt(eleV2[0],10,lDigits); tQy=str2bigInt(eleV2[1],10,lDigits);
64
65:   elliptic_add(tPx,tPy,tQx,tQy,ta,tb,tp,tRx,tRy);
66:   document.getElementById("out_mesg").value =
67:     bigInt2str(tRx,10) +" "+ bigInt2str(tRy,lDigits);
68: }
69: </script>
70: </body>
```

To test this page, two sample points and an elliptic curve are provided in lines 37-39. For example, the string `pSt` in line 37 representing the point P contains the x-y coordinates of the point P:

```
P[x y] = [1773056069879938 1092467599646463]
```

In general, we will use this `P[x y]` convention to represent points on elliptic curve down to x-y coordinate level. The elliptic curve E is defined by the `[a,b,p]` values given in line 39:

```
E[a,b,p]=[589758509 280934777552806 2237171265681283]
```

The strings `P`,`Q` and `E` are put into the necessary text boxes given in the XHTML code in part I. When the `OK` button is clicked, the function `add_fun()` in lines 43-67 is run. Since we are dealing with big integers, all variables are defined in big integer format (see lines 45-50).

The values of the elliptic curve `E` are obtained and extracted into variables `ta`, `tb`, and `tp` in lines 52-55. The while-loops in lines 55 are used to handle negative value cases. The point `P` are obtained and

extracted into variables `tPx` and `tPy` in lines57-59. Similarly, point `Q` are obtained and stored in variables `tQx` and `tQy` (see lines 61-63). The function call in line 65 performs the actual point addition of points `P:[tPx tPy]` and `Q:[tQx tQy]` on curve `E[ta,tb,tp]`

```
elliptic_add(tPx,tPy,tQx,tQy,ta,tb,tp,tRx,tRy);
```

The result is another point `R` on the curve represented by the x-y coordinates `tRx` and `tRy`. This function is given by the script file elliptic.js below:

```
Example: ECMAScript File elliptic.js For Elliptic Curves          (Part I)

 1: function elliptic_add(ttPx,ttPy,ttQx,ttQy,tta,ttb,ttp,ttRx,ttRy)
 2: {
 3:  var dPxQx,dPyQy,aPxQx,dQxRx,tQx,idPxQx,slop,idQ2y,gDigits=10;
 4:  dPxQx  = int2bigInt(0,10,gDigits);  dPyQy = int2bigInt(0,10,gDigits);
 5:  aPxQx  = int2bigInt(0,10,gDigits);  dQxRx = int2bigInt(0,10,gDigits);
 6:  tQx    = int2bigInt(0,10,gDigits);  slop  = int2bigInt(0,10,gDigits);
 7:  idPxQx = int2bigInt(0,10,gDigits);  iQ2y  = int2bigInt(0,10,gDigits);
 8:  Rx     = int2bigInt(0,10,gDigits);  Ry    = int2bigInt(0,10,gDigits);
 9:  if (!equals(ttPx,ttQx) && !equals(ttPy,ttQy))
10:  {
11:    copy(dPxQx,ttPx);      //Px - Qx
12:    sub(dPxQx,ttQx);       while (negative(dPxQx)) add(dPxQx,ttp);
13:    copy(dPyQy,ttPy);      //Py - Qy
14:    sub(dPyQy,ttQy);       while (negative(dPyQy)) add(dPyQy,ttp);
15:
16:    copy(idPxQx,dPxQx);    inverseMod(idPxQx,ttp);   // 1/(Px-Qx) mod p
17:    copy(aPxQx,ttPx);      add(aPxQx,ttQx);          //Px + Qx
18:    copy(slop,dPyQy);      multMod(slop,idPxQx,ttp); //(py-Qy)/(Px-Qx)
19:
20:    copy(Rx,slop);         multMod(Rx,Rx,ttp);
21:    sub(Rx,aPxQx);         while (negative(Rx)) add(Rx,ttp);
22:    mod(Rx,ttp);
23:    copy(dQxRx,ttQx);
24:    sub(dQxRx,Rx);         while (negative(dQxRx)) add(dQxRx,ttp);
25:
26:    multMod(dQxRx,slop,ttp); copy(Ry,dQxRx);
27:    sub(Ry,ttQy);          while (negative(Ry)) add(Ry,ttp);
28:    mod(Ry,ttp);
29:  }
30:  if (equals(ttPx,ttQx) && equals(ttPy,ttQy))
31:  {
32:     var two = int2bigInt(2,10,gDigits);  three = int2bigInt(3,10,gDigits);
33:     copy(iQ2y,ttQy);     multMod(iQ2y,two,ttp);  inverseMod(iQ2y,ttp);
34:
35:     copy(Rx,ttQx);       multMod(Rx,Rx,ttp);      multMod(Rx,three,ttp);
36:     add(Rx,tta);         multMod(Rx,iQ2y,ttp);    copy(Ry,Rx);
37:     multMod(Rx,Rx,ttp);
38:
39:     copy(tQx,ttQx);      multMod(tQx,two,ttp);
40:     sub(Rx,tQx);         while (negative(Rx)) add(Rx,ttp);
41:
42:     copy(dQxRx,ttQx);
43:     sub(dQxRx,Rx);          while (negative(dQxRx)) add(dQxRx,ttp);
44:     multMod(Ry,dQxRx,ttp);
45:     sub(Ry,ttQy);           while (negative(Ry)) add(Ry,ttp);
46:     mod(Ry,ttp);
47:  }
48:  copy(ttRx,Rx); copy(ttRy,Ry);
49: }
50:
```

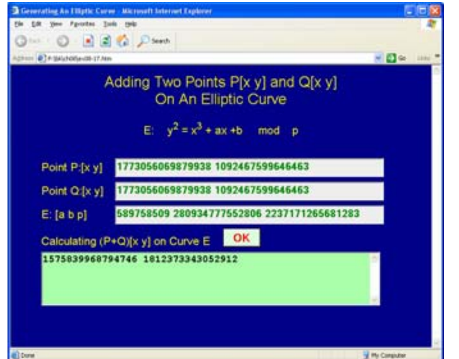
This script is to implement the following "Addition" formulas of two points `P` and `Q` on elliptic curve `E`. The result is a third point `R:[x y]` on curve `E`.

$$R_x = \lambda^2 - P_x - Q_x$$
$$R_y = \lambda (Q_x - R_x) - Q_y$$

$$\lambda = \begin{cases} \dfrac{P_y - Q_y}{P_x - Q_x} & \text{for } P \neq Q \\[2ex] \dfrac{3Q_x^2 + a}{2Q_y} & \text{for } P = Q \end{cases}$$

The implementation is carried out using the routines for big integers. Lines 9-29 handle the situation when $P \neq Q$. First, the value λ is computed by lines 11-18 and stored in variable slop. Lines 20-24 calculate the x-coordinate of point `R` (i.e R_x). The statements in lines 26-28 compute the y-coordinate of `R` (i.e. R_y). Similarly, lines 30-49 calculate the x-y coordinates of the point `R` when `P=Q` situation.

Note that the implementation above is simple for easy understanding purposes. There is no code to handle the situation where the x-coordinates of `P` and `Q` are equal. In this case, the function should return infinity (or point `O`). Also, there is no additional arrangement for infinity points. For a more professional approach, all points on the elliptic curve should have a status indicating whether it is the infinity point. A screen shot of this example in action is shown in Fig. 08.23.



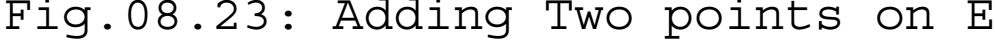


Fig.08.23: Adding Two points on E

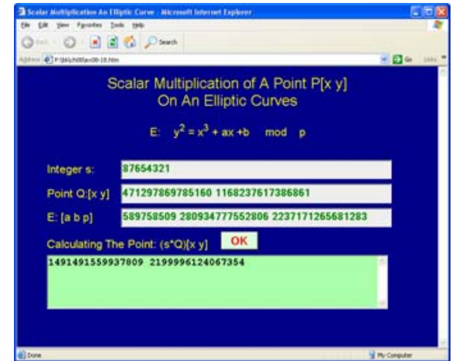


Fig.08.24: Scalar Multiplication on E

With this "Addition" routine, "Scalar Multiplication" on elliptic curve is implemented in the next section. As a result, secret-key and public-key of an ECC system are generated.

8.5.3 Scalar Multiplication and Generating the Keys for ECC

Given a point `P[x y]` on an elliptic curve `E[a,b,p]`, the scalar multiplication of `P` by an integer `s` is computed by the following addition formulas on an elliptic curve.

$$(s)\,P_x = \underbrace{P_x + P_x + \ldots + P_x}_{(s\ times)} \qquad (s)\,P_y = \underbrace{P_y + P_y + \ldots + P_y}_{(s\ times)}$$

There are a number of ways to implement the series of additions. The simplest may be using a for-loop and calling the addition routine in previous section. However, using a for-loop may be too slow for many ECC applications. For a faster implementation, we use the algorithm and pseudo-code below:

```
Example: Pseudo-Code For Scalar Multiplication On An Elliptic Curve

 1: Functiuon elliptic_mult(s, P, E, W)
 2: {
 3:  var T1, T2, d, r;                     //T1 & T2 variables for points
 4:  T2=P;                                 //Assign point P to variable T2
 5:  d = integer(s/2);   r = s%2;          //Qoutient (d) and Remainder (r)
 6:  while (d > 0) {                       //While-loop
 7:   T1 = T2;  T2 = elliptic_add(T1,T1);  //T2 is point P in power of 2
 8:   if (r==1) W = elliptic_add(W,T1);    //Adding T1 to point W
 9:   if (d==1) W = elliptic_add(W,T2);    //Adding T2 to point W
10:   d = integer(s/2);   r = s%2;         //New d, r and continue while-loop
11:  }
12: }
```

This algorithm may not be the fastest. It is easy to read and can be used to compute a series of additions in the speed of power of 2. It is equivalent to break the integer `s` down into a sequence of power of 2, remainders and then add them together.

To implement this scalar multiplication, let's make a copy of part I of ex08-17.htm and call it ex08-18.htm. Inside this new page, modify the following lines:

```
 12:  Scalar Multiplication of A Point P[x y]<br/>On An Elliptic Curves<br/>
 18:  <tr><td style="width:130px">Integer s:</td><td >
 19:      <input type="text" id="ts" size="132" maxlength="132"
 27:  <tr><td colspan="2"> Calculating The Point: (s*Q)[x y]   
 29:        onclick="mult_fun()" /></td></tr>
```

The first 4 lines above are used to change the display. When the `OK` button is clicked, the function `mult_fun()` is run to perform the scalar multiplication on an elliptic curve. This function is defined in the part II of ex08-18.htm.

```
Example: ex08-18.htm - Scalar Multiplication On An Elliptic Curve (Part II)

36: <script>
37:   sSt = "87654321";
38:   qSt ="471297869785160 1168237617386861";
39:   eSt = "589758509 280934777552806 2237171265681283";
40:   document.getElementById("ts").value = sSt;
41:   document.getElementById("tq").value = qSt;
42:   document.getElementById("te").value = eSt;
43: function mult_fun()
44: {
45:   var tmpSt,ta,tb,tp,tx,tPx,tPy,tRx,tRy,lDigits=10;
46:   ta   = int2bigInt(0,10,lDigits);     tb   = int2bigInt(0,10,lDigits);
47:   tp   = int2bigInt(0,10,lDigits);     ts   = int2bigInt(0,10,lDigits);
48:   tQx  = int2bigInt(0,10,lDigits);     tQy  = int2bigInt(0,10,lDigits);
49:   tRx  = int2bigInt(0,10,lDigits);     tRy  = int2bigInt(0,10,lDigits);
50:
51:   tmpSt = document.getElementById("te").value;
52:   eleV = myParseSt(tmpSt);             ta = str2bigInt(eleV[0],10,10);
53:   tb = str2bigInt(eleV[1],10,10);      tp = str2bigInt(eleV[2],10,10);
54:   while (negative(ta)) add(ta,tp);     while (negative(tb)) add(tb,tp);
55:
56:   tmpSt = document.getElementById("tq").value;
57:   eleV1 = myParseSt(tmpSt);
```

```
58:   tQx = str2bigInt(eleV1[0],10,10);    tQy = str2bigInt(eleV1[1],10,10);
59:
60:   tmpSt = document.getElementById("ts").value;
61:   ts = str2bigInt(tmpSt,10,0);
62
63:   elliptic_mult(ts,tQx,tQy,ta,tb,tp,tRx,tRy);
64:   document.getElementById("out_mesg").value =
65:     bigInt2str(tRx,10) +" "+ bigInt2str(tRy,10);
66: }
67: </script>
68: </body>
```

Lines 37-42 provide a testing sample for the scalar multiplication. The function `mult_fun()` is defined in lines 43-66. The values of the elliptic curve `E` is obtained and extracted into variables `ta`, `tb`, and `tp` (see lines 51-54). The x-y coordinates of the point `Q` is extracted into variables `tQx` and `tQy` (see lines 56-58). The scalar `s` is obtained by lines 60-61. The function call in line 63:

```
elliptic_mult(ts,tQx,tQy,ta,tb,tp,tRx,tRy);
```

performs the scalar multiplication of s on the point `Q[x y]`. The result is another point R on E with x-y coordinates `R[tRx tRy]`. The scalar multiplication function `elliptic_mult()` is given in the script file elliptic.js.

```
Example: ECMAScript File elliptic.js For Elliptic Curves          (Part II)

51: function elliptic_mult(tts,ttPx,ttPy,tta,ttb,ttp,ttRx,ttRy)
52: {
53:   var r1x,r1y,r2x,r2y,wx,wy,d,r,zero,one,two,tm0,tm1,gDigits=10;
54:   r1x = int2bigInt(0,10,gDigits);  r1y  = int2bigInt(0,10,gDigits);
55:   r2x = int2bigInt(0,10,gDigits);  r2y  = int2bigInt(0,10,gDigits);
56:   wx  = int2bigInt(0,10,gDigits);  wy   = int2bigInt(0,10,gDigits);
57:   d   = int2bigInt(0,10,gDigits);  r    = int2bigInt(0,10,gDigits);
58:   zero= int2bigInt(0,10,gDigits);  one  = int2bigInt(1,10,gDigits);
59:   two = int2bigInt(2,10,gDigits);
60:   tm0 = int2bigInt(0,10,gDigits);  tm1  = int2bigInt(0,10,gDigits);
61:   Rx  = int2bigInt(0,10,gDigits);  Ry   = int2bigInt(0,10,gDigits);
62:
63:   copy(r2x,ttPx); copy(r2y,ttPy);    sub(wx,one); sub(wy,one);
64:   copy(tm0,tts);  divide(tm0,two,d,r);
65:   while( greater(d,zero)) {
66:      copy(r1x,r2x);          copy(r1y,r2y);
67:      elliptic_add(r1x,r1y,r1x,r1y,tta,ttb,ttp,ttRx,ttRy);
68:      copy(r2x,ttRx);         copy(r2y,ttRy);
69:      if( equals(r,one)) {
70:        if ( negative(wx) && negative(wy)) {
71:            copy(wx,r1x);     copy(wy,r1y);
72:        } else {
73:            elliptic_add(wx,wy,r1x,r1y,tta,ttb,ttp,ttRx,ttRy);
74:            copy(wx,ttRx);    copy(wy,ttRy);
75:        }
76:      }
77:      if( equals(d,one)) {
78:        if ( negative(wx) && negative(wy)) {
79:            copy(wx,r2x);     copy(wy,r2y);
```

```
80:         } else {
81:             elliptic_add(wx,wy,r2x,r2y,tta,ttb,ttp,ttRx,ttRy);
82:             copy(wx,ttRx);    copy(wy,ttRy);
83:         }
84:       }
85:       copy(tm0,d);             divide(tm0,two,d,r);
86:    }
87:    copy(ttRx,wx); copy(ttRy,wy);
88: }
```

The script function `elliptic_mult()` in lines 51-88 is a direct implementation of the pseudo-code for scalar multiplication. The x-y coordinates of point `P[tPx tPy]` are copied to variables `R2[r2x r2y]` in line 63. Also, the point `W[wx wy]` is assign an initial value -1. After the scalar `tts` is copied to variable tm0, the quotient and remainder `of tm0 /2` are obtained as `d` and `r` respectively (see line 64). While `d` is bigger than zero, the point `R1[r1x r1y]` is set to the value of point `R2[r2x r2y]`. The point `R2` is then calculated as elliptic addition `R1+R1` by lines 67-68. If the remainder `r` is 1, the point `R1` is added into point `W` (see lines 69-76). If the quotient `d` is 1, the point `R2` is added into point `W` in lines 77-84. The statements in line 85 compute the quotient d and remainder `r` again for the while-loop. When the while-loop is exhausted (i.e. `d=0`), the scalar product (or resulting) point `W[wx wy]` is copy to point `R[ttRx ttRy]` and returned to the function caller. A screen shot of this example is shown in Fig.08.24.

Given a random integer k (secret-key) and a base point `B[x y]` on the elliptic curve `E[a b p]`, the public-key `K` of the ECC is generated by: `K = k*B` which is simply a scalar multiplication. Therefore, to develop a page to generate ECC secret and public keys is easy. All we have to do is to make a complete copy of ex08-18.htm and call it ex08-19.htm. Inside this new page, modify the following lines:

```
12:  Generating The Public-Key K[x y]<br/>For An ECC System<br/>
13:  <span style="font-size:16pt"><br />
14:     K[x y] = (k *B)[x y]
18:  <tr><td style="width:130px">Secret-Key k:</td><td >
21:  <tr><td>Point B:[x y]</td><td>
27:  <tr><td colspan="2"> Generating The Public-Key K[x y]    
```

Yes! All you have to do is to change 6 lines. In fact, they are for display purpose only and have no impact to the calculation. A screen shot of this example is shown in Fig.08.25.

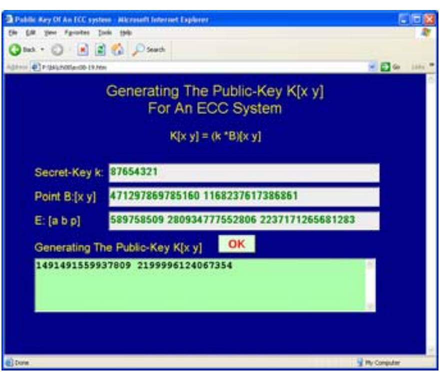


Fig.08.25: Generating Public-Key

With the secret-key and public-key generated, it's time to consider how to perform encryption/ decryption for an ECC system.

8.5.5 Encryption/Decryption with Elliptic Curves

Suppose we have the public-key and secret-key information below:

```
Public-Key: [K,B,a,b,p]     and     Secret-Key [k,B,a,b,p]
```

The encryption of a message M represented as a point on the elliptic curve `E[a,b,p]` is given by:

```
c0 = r*B     c1 = M + r*K             (ECC Encryption)
```

where `r` is a random integer. From the addition and scalar multiplication of elliptic curve, the ciphertext pair `[c0,c1]` are, in fact, two points on the curve `E[a,b,p]`.

To perform the decryption and obtaining the plaintext M back, the following formula is used:

```
c1 - k*c0 = M                         (ECC Decryption)
```

Note that the integer r in the encryption process plays no part in the decryption process. To implement the ECC encryption, consider the page as follows:

```
Example: ex08-20.htm – Elliptic Curve Encryption                  (Part I)

 1: <head><title>ECC Encryption</title></head>
 2: <style>
 3:  .butSt{font-size:16pt;width:70px;height:35px;
 4:           font-weight:bold;background:#ddffdd;color:#ff0000}
 5:  .txtSt{font-size:14pt;width:500px;height:35px;font-weight:bold;
 6:         background:#eeeeee;color:#ff0000}
 7:  .txtArea{font-size:14pt;width:630px;height:100px;font-weight:bold;
 8:       background:#aaffaa}
 9: </style>
10: <body style="font-family:arial;font-size:20pt;text-align:center;
11:       background:#000088;color:#ffff00">
12:  Encryption Using Elliptic Curves<br />
13:  <span style="font-size:16pt"><br />
14:   c0 = (r*B)[x y]      c1= (M + r * K)[x y]
15:  </span><br /><br/>
16: <form action="">
17: <table style="font-size:16pt;width:700px" cellspacing="5" align="center">
18:  <tr><td>Message M:[x y]</td><td>
19:      <input type="text" id="tm" size="132" maxlength="132"
20:        class="txtSt" style="color:#008800" value="" /></td></tr>
21:  <tr><td>Integer r:</td><td >
22:      <input type="text" id="tr" size="132" maxlength="132"
23:        class="txtSt" style="color:#008800" value="" /></td></tr>
24:  <tr><td>Public-Key K[x y]</td><td>
25:      <input type="text" id="tk" size="132" maxlength="132"
26:        class="txtSt" style="color:#008800" value="" /></td></tr>
27:  <tr><td>Point B:[x y]</td><td>
28:      <input type="text" id="tb" size="132" maxlength="132"
29:        class="txtSt" style="color:#008800" value="" /></td></tr>
30:  <tr><td>E: [a b p]</td><td >
31:      <input type="text" id="te" size="132" maxlength="142"
32:        class="txtSt" style="color:#008800" value="" /></td></tr>
33:  <tr><td colspan="2"> Generating The Public-Key  
34:      <input type="button" class="butSt" value="OK"
35:        onclick="en_fun()" /></td></tr>
```

```
36:  <tr><td colspan="2"><textarea rows="5" cols="40" id="out_mesg"
37:        class="txtArea" readonly ></textarea></td></tr>
38: </table></form>
39: <script src="BigInt.js"></script>
40: <script src="hexlib.js"></script>
41: <script src="elliptic.js"></script>
```

This page fragment is the interface part of the example containing five text boxes, one `OK` button and one text area. The first two text boxes are for the plaintext `M` and an arbitrary integer `r`. The remaining three text boxes are for the public-key `K`, base point `B`, and elliptic curve `E[a,b,p]`. Once the `OK` button is clicked, the function `en_fun()` is run (see line 35) and the ciphertext values `c0` and `c1` are displayed in the text area at the bottom of the browser window. This encryption function `en_fun()` is given in the part II of the example.

```
Example: Continuation of ex08-20.htm

42: <script>
43:   mSt = "1435965740019437 530943628671512";
44:   eSt = "589758509 280934777552806 2237171265681283";
45:   rSt = "7561",
46:   bSt = "471297869785160 1168237617386861";
47:   kSt = "2127814482883134 2049972171869118";  //Secret-Key: 1568
48:   document.getElementById("tm").value = mSt;
49:   document.getElementById("tr").value = rSt;
50:   document.getElementById("tk").value = kSt;
51:   document.getElementById("tb").value = bSt;
52:   document.getElementById("te").value = eSt;
53:
54: function en_fun()
55: {
56:   var tmpSt,ta,tb,tp,tBx,tBy,tQx,tQy,outSt="",lDigits=10;
57:   ta   = int2bigInt(0,10,lDigits);   tb   = int2bigInt(0,10,lDigits);
58:   tp   = int2bigInt(0,10,lDigits);   tr   = int2bigInt(0,10,lDigits);
59:   tMx  = int2bigInt(0,10,lDigits);   tMy  = int2bigInt(0,10,lDigits);
60:   tBx  = int2bigInt(0,10,lDigits);   tBy  = int2bigInt(0,10,lDigits);
61:   tKx  = int2bigInt(0,10,lDigits);   tKy  = int2bigInt(0,10,lDigits);
62:   tmx  = int2bigInt(0,10,lDigits);   tmy  = int2bigInt(0,10,lDigits);
63:   tRx  = int2bigInt(0,10,lDigits);   tRy  = int2bigInt(0,10,lDigits);
64:
65:   tmpSt = document.getElementById("te").value;
66:   eleV = myParseSt(tmpSt);           ta = str2bigInt(eleV[0],10,10);
67:   tb = str2bigInt(eleV[1],10,10);     tp = str2bigInt(eleV[2],10,10);
68:   while (negative(ta)) add(ta,tp);   while (negative(tb)) add(tb,tp);
69:
70:   tmpSt = document.getElementById("tb").value;
71:   eleV1 = myParseSt(tmpSt);
72:   tBx = str2bigInt(eleV1[0],10,10);  tBy = str2bigInt(eleV1[1],10,10);
73:
74:   tmpSt = document.getElementById("tr").value;
75:   tr = str2bigInt(tmpSt,10,0);
76:   elliptic_mult(tr,tBx,tBy,ta,tb,tp,tRx,tRy);
77:   outSt="Ciphertext c0=\n"+bigInt2str(tRx,10)+" "+bigInt2str(tRy,10)+"\n";
78:
79:   tmpSt = document.getElementById("tm").value;
80:   eleV1 = myParseSt(tmpSt);
81:   tMx = str2bigInt(eleV1[0],10,10);  tMy = str2bigInt(eleV1[1],10,10);
82:
```

```
83:   tmpSt = document.getElementById("tk").value;
84:   eleV1 = myParseSt(tmpSt);
85:   tKx = str2bigInt(eleV1[0],10,10);  tKy = str2bigInt(eleV1[1],10,10);
86:
87:   elliptic_mult(tr,tKx,tKy,ta,tb,tp,tRx,tRy);
88:     copy(tmx,tRx); copy(tmy,tRy);
89:   elliptic_add(tMx,tMy,tmx,tmy,ta,tb,tp,tRx,tRy);
90:
91:   outSt+="Ciphertext c1=\n"+bigInt2str(tRx,10)+" "+bigInt2str(tRy,10)+"\n";
92:   document.getElementById("out_mesg").value = outSt;
93: }
94: </script>
95: </body>
```

Lines 43-52 provide a testing sample of plaintext `M`, an arbitrary integer `r`, and the public-key information `{K,B,E[a,b,p]}`. Once the OK button is pressed, the function `en_fun()` in lines 54-93 is activated to encrypt the plaintext. Lines 65-68 obtain the information of the elliptic curve `E[a,b,p]`. The statements in lines 70-72 get the x-y coordinates of the base point `B[x y]`. The arbitrary integer `r` is obtained by lines 74-75. As soon as these information are available, the function in line 76 computes the first ciphertext point `c0[x y]= (r*B)[x y]` on `E[a,b,p]`, i.e.

```
elliptic_mult(tr,tBx,tBy,ta,tb,tp,tRx,tRy);
```

Since `c0` is a point on `E`, the x-y coordinates are stored in `[tRx, tRy]`. This point `c0` is appended to the string variable `outSt` at line 77 for display purposes.

The x-y coordinates of plaintext `M[x y]` is captured by lines 79-81. Together with the public-key `K[x y]`, the scalar multiplication in line 87 computes the x-y coordinates of `(r*K)[x y]`. By adding the plaintext `M` in line 89, the second ciphertext `c1` is obtained, i.e.

```
c1[x y] = (M + r*K)[x y]
```

This point `c1` is appended into string variable `outSt` at line 91 and displayed at the bottom of the browser window by the statement in line 92. A screen shot of this example is shown in Fig.08.26

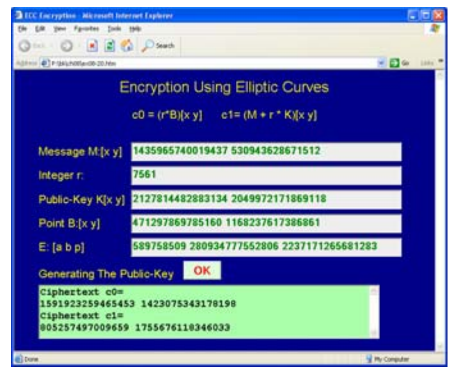


Fig.08.26: ECC Encryption

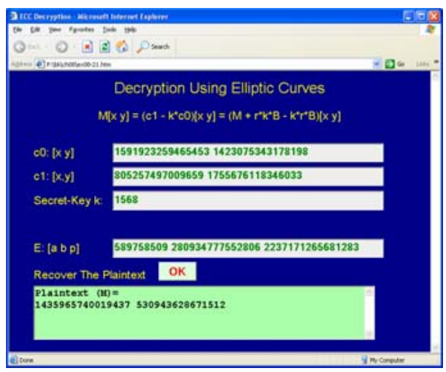


Fig.08.26: ECC Decryption

For the ECC decryption, we use a very similar interface as described in ex08-20.htm. Therefore let's make a copy of the part I of ex08-20.htm and call it ex08-21.htm. Inside this new page, modify the following lines:

```
12:  Decryption Using Elliptic Curves<br />
14:   M[x y] = (c1 - k*c0)[x y] = (M + r*k*B - k*r*B)[x y]
18:  <tr><td>c0: [x y] </td><td>
19:      <input type="text" id="c0" size="132" maxlength="132"
21:  <tr><td>c1: [x,y] </td><td >
22:      <input type="text" id="c1" size="132" maxlength="132"
27:  <tr style="visibility:hidden"><td>Point B:[x y]</td><td>
35:        onclick="de_fun()" /></td></tr>
```

The first two lines change the display text. Lines 18-19 generate a text box for the first ciphertext `c0[x y]`. Similarly, lines 21-22 generate a text box for the second ciphertext `c1[x y]`. For decryption, the base point B is not needed. By putting the attribute visibility:hidden in line 27, the entire row of the base point will disappear. When the `OK` button is clicked, the function `de_fun()` in line 35 is activate to perform decryption on `c0` and `c1`. This decryption function `de_fun()` is defined in the part II of ex08-21.htm below:

```
Example: ex08-21.htm - ECC Decryption                          (Part II)

42: <script>
43:   c0St = "1591923259465453 1423075343178198";
44:   c1St = "805257497009659 1755676118346033";
45:   kSt = "1568";      // K[x y]="2127814482883134 2049972171869118"
46:   eSt = "589758509 280934777552806 2237171265681283";
47:   document.getElementById("c0").value = c0St;
48:   document.getElementById("c1").value = c1St;
49:   document.getElementById("tk").value = kSt;
50:   document.getElementById("te").value = eSt;
51:
52: function de_fun()
53: {
54:   var tmpSt,ta,tb,tp,c0x,c0y,c1x,c1y,tk,outSt="",elev1,lDigits=10;
55:   ta   = int2bigInt(0,10,lDigits);   tb   = int2bigInt(0,10,lDigits);
56:   tp   = int2bigInt(0,10,lDigits);   tk   = int2bigInt(0,10,lDigits);
57:   c0x  = int2bigInt(0,10,lDigits);   c0y  = int2bigInt(0,10,lDigits);
58:   c1x  = int2bigInt(0,10,lDigits);   c1y  = int2bigInt(0,10,lDigits);
59:   tRx  = int2bigInt(0,10,lDigits);   tRy  = int2bigInt(0,10,lDigits);
60:
61:   tmpSt = document.getElementById("te").value;
62:   eleV1 = myParseSt(tmpSt);
63:   ta=str2bigInt(eleV1[0],10,lDigits);  tb=str2bigInt(eleV1[1],10,lDigits);
64:   tp=str2bigInt(eleV1[2],10,lDigits);
65:   while (negative(ta)) add(ta,tp);     while (negative(tb)) add(tb,tp);
66:
67:   tmpSt = document.getElementById("c0").value;
68:   eleV1 = myParseSt(tmpSt);
69:   c0x=str2bigInt(eleV1[0],10,lDigits); c0y=str2bigInt(eleV1[1],10,lDigits);
70:
71:   tmpSt = document.getElementById("c1").value;
72:   eleV1 = myParseSt(tmpSt);
73:   c1x=str2bigInt(eleV1[0],10,lDigits); c1y=str2bigInt(eleV1[1],10,lDigits);
74:
75:   tmpSt = document.getElementById("tk").value;
76:   tk = str2bigInt(tmpSt,10,lDigits);
77:
78:   elliptic_mult(tk,c0x,c0y,ta,tb,tp,tRx,tRy);
79:
80:   multInt(tRy,-1); if (negative(tRy)) add(tRy,tp); //Finding minus point
81:   copy(c0x,tRx); copy(c0y,tRy);
```

```
82:   elliptic_add(c1x,c1y,c0x,c0y,ta,tb,tp,tRx,tRy);
83:
84:   outSt+= "Plaintext (M)= \n"+
85:      bigInt2str(tRx,10) + " "+ bigInt2str(tRy,10) +"\n";
86:   document.getElementById("out_mesg").value = outSt;
87: }
88: </script>
89: </body>
```

Lines 43-50 provide a testing sample for the decryption. The elliptic curve information are captured by lines 61-65. The ciphertext pair `c0[x y]` and `c1[x y]` are obtained by lines67-73. Together with the secret-key k, the function in line 78 computes the value:

```
( k * c0 )[x  y]
```

To convert this point into negative, we need to multiply -1 onto the y-coordinate as illustrated in lines 80. Therefore, the function `elliptic_add()` in line 82 effectively adds the points `c1` and `-(k*c0)` together to get the plaintext `M`, i.e.

```
M[x y] = (c1 -  k * c0 )[x  y]
```

A screen shot of this example in action is shown in Fig.08.27. From this figure, we can see that if the ciphertext `c0` and `c1` in ex08-20.htm is used, the same plaintext `M[x y]` is obtained.