ECE 512 Project Report

SSL client implementation with RSA method

Xiao Luo, Junhao Li

Table of Contents

1. Abstraction

This project will study and implement public key infrastructure on client side using RSA encryption method. A sever-client module is used to simulate the key exchange process between server and client. The content for the coming sections are: section 2 will introduce the complete SSL handshake process. Section 3 will illustrate our implementation of the project, including the modification and simplification we made on SSL handshake process, the methods we used to implement RSA and DH, and the difficulties we encounter during coding. The simulation result based on the modules is also demonstrated in section 3 along with the analysis of the result. Section 4 includes the conclusion part and the prospects of this project.

2. Theory Basis

2.1 SSL protocol

SSL (Secure Socket Layer) is a protocol designed by Netscape to guarantee network communication security and data integrity. SSL use X.509 certificates and asymmetric cryptography. This section will introduce SSL protocol. The server side process is shown below:

- Start communication：

  Client Hello: The client should send message including protocol version, cipher suite, session ID, compression method, etc., to the server.

  Server Hello: If the server find the corresponding cipher suite as the client send, it will send a "Server Hello" message to the client. This message should include the similar information as the message sent by client. If the process is error because of network, the server will send "handshake failure" back to the client.
- Server Certificate Authentication and Key exchange：

  Server Certificate: The server should also send its signed certificate to the client. This certificate will be sent soon after the server hello message.

  Server Key Exchange: The client uses the information received from server to authenticate server's identity. If the verification fails, SSL connection will not be established. If the verification process succeeds, a message will be sent by client. When the server receives this message, it will send pre-master key to client for key generation.
- Certificate Request (Optional):

  Sometimes the server need to authenticate the client identity. In this case, the client should sent its digital signature and send it together with the client certificate to the server. This process is similar to Server Certificate Authentication process.
- Serve Finish:

  This message indicates the "Hand Shaking" process on server side is complete.

The client side process is similar to the server's. We list some steps which differs with the server side below:

- Certificate Verify:

  If the certificate of the client is signed, then client has to send this certificate verify message for the authentication of the certificate to server.
- Client Certificate Authentication: (optional):

  Client Certificate: If the server requires the client to send its certificate, the client should send the related information in this period. Otherwise this part will not include any certificate information. In case the server requires client authentication, the serve will check whether the certificate authority of the client is on its trusted list. If the authority is not on the list, then the conversation ends.
- Client Key Exchange:

  Suppose we use RSA method, then this message should include pre-master key encrypted with the public key sent by the server. Then the server will decrypt the received prime key with its private key and use the prime key to generate the key pair for the coming conversation.

- Client Finish:
  The client then informs the server about the cryptography method and the master key for generating new cipher keys. Also the server will include a message informing the client that the SSL handshake process is finished. This message should be encrypted with the RSA public key.
- Server Finish:
  The server then informs the client about the cryptography method and the master key. Also the server will include a message informing the client that the SSL handshake process is finished. This message should be encrypted with the new cipher key.

The client and server will use the master secret to generate session keys which are used for encryption and decryption for the coming conversation. For our final project, we follow the basic process for SSL handshake with a little adjustment, the details will be discussed in part 3.

2.2 Diffie-Hellman Method

Diffie-Hellman key exchange (D.H method) is a classic method used in public-key cryptography. The basic key exchange process is as below:

(1) Client will first send a predefined number g to the server.

(2) Then the server will randomly pick a number a and g^a to Client.

(3) Client receives g^a and is able to compute a since Client knows g.

(4) Client randomly picks a number b and sends g^b to the Server.

(5) Server receives g^b and is able to compute b since Server knows g.

(6) Now Client and Server have a shared key gab and will use this key to encrypt and decrypt.

2.3 RSA Method

RSA algorithm is named after the initials for its founders: Ron Rivest, Adi Shamir and Leonard Adleman. It's a method based on difficulty of factoring and has been widely used in secure data transmission. So we choose RSA as the cryptography method for generating the session keys mentioned above. The basic key generation process is as below:

(1) Choose two distinct prime p and q of approximately same size.

(2) Compute $n = p \times q$ and $\Phi(n) = (p-1)\times(q-1)$.

(3) Choose a prime number e such that $e \in (1, \Phi(n))$. Also e and $\Phi(n)$ should be co-prime.

(4) Compute $d \equiv e^{-1}(\text{mod}(\Phi(n)))$.

Then we have public key pair: $<e,n>$ and $<d,n>$. The encryption and decryption method is shown below:

Encryption: $E(m) \equiv m^e(\text{mod}(n))$

Decryption: $D(c) \equiv c^d(\text{mod}(n))$

Below are some methods and simplification we use while implementing RSA:

- Although for security reason, p and q should be chosen as distinguished prime with similar bit-length, but should not be too small, we pick rather small numbers so that the exponentiation calculation will not be too large to compute.

- Since we pick rather small p and q, we calculate the co-prime of Φ (n) by simply iterating all the primes start at 2. Euclidian method is applied to determine prime condition.
- In order to find the inverse modular d, we implemented extended Euclid method. This function will solve the Bézout's identity, which will give the inverse modular for two particular numbers.

3. Implementation Results

3.1 Handshake Process

In order to simulate the information exchange in experiment. We made some simplifications on it. The first part of section 3 will introduce those changes and the next part will give the experiment result. Basically we simplified the certificate authentication process and the package format. The modification is described below:

- Client Hello

  The "Hello" message sent by the client only includes the protocol information (SSL version). Other functions like compression and reconnect can be achieved with specific library.
- Client Certificate Authentication

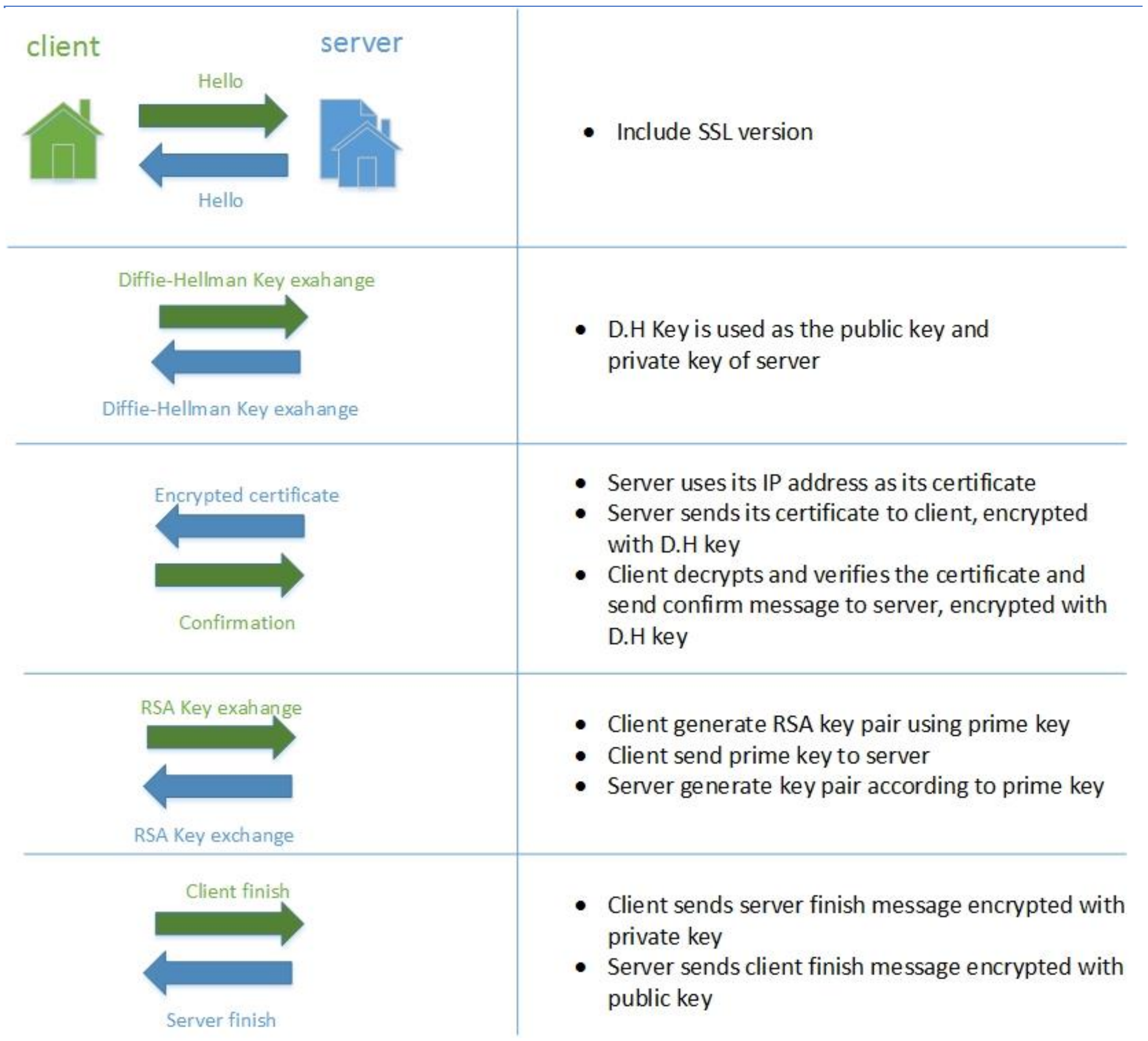
  In this part, we add the process of exchange D. H keys between client and server. Then we use D.H key.

  In addition, since real server's certificate and digital signature is much complicated and the focus of our project is the cryptography method, we use the IP address of the server as its digital signature. We encode server's IP address with the DH key and send this encrypted message to the client. If the client can use the DH key to decrypt this message and compare the decoded message with the server IP in its arguments, by which verifies the server identity.
- Key exchange

  The client will receive server's digital signature and should be able to decrypt the signature with the D.H key they share. Then the client generate two random prime number p and q as the master key for the RSA method, encode them with the server's public key (DH key) and send them to the server. The client will receive a message encoded with the RSA public key sent by server. The client should be able to decode information using the private RSA key and encode information with the RSA public key sent to server.
- Hand shake finish and start talking:

  When the handshaking process finishes, the client and server can change information pairwise and all the information is encrypted by RSA method.

The whole handshake process is as figure is illustrated below:



- Include SSL version

- D.H Key is used as the public key and private key of server

- Server uses its IP address as its certificate
- Server sends its certificate to client, encrypted with D.H key
- Client decrypts and verifies the certificate and send confirm message to server, encrypted with D.H key

- Client generate RSA key pair using prime key
- Client send prime key to server
- Server generate key pair according to prime key

- Client sends server finish message encrypted with private key
- Server sends client finish message encrypted with public key

Fig1. The whole hand shake process

3.2 Difficulties&solutions in programming

In RSA method, the decryption process require large number in power. For example, in our program, the public key pair is <209, 7> and the private key pair is <209,103>, which means the decryption process requires:

$$original = content^{103} \bmod 209$$

We tested several types to store the huge number and even the *unsigned long long* type is overflow (even greater than 10 power). It is a dilemma and we searched the Internet about any possible solutions. Finally we have found the equation about solving modular iteratively. The equation can be written as this:

$$x^{a+b} \bmod n = ((x^a \bmod n) \times (x^b \bmod n)) \bmod n$$

Applying this equations, we divide 103 power into six parts, including 5 parts of 20 power and 1 part of 3 power. For each 20-power part, we divide it into 4 part of 5 power. The final equation can be shown like that:

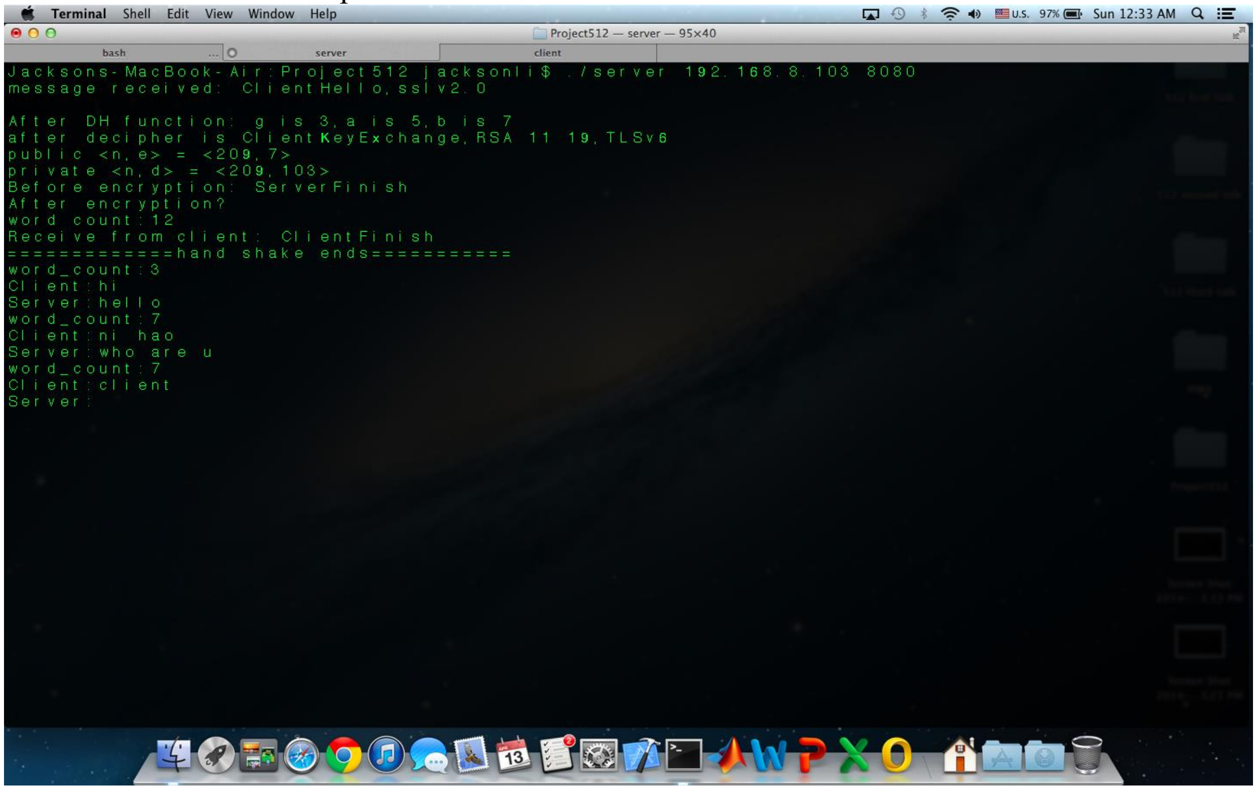
$$x^{103} \bmod n = (((x^5 \bmod n)^4 \bmod n) \times (x^3 \bmod n)) \bmod n$$

In real test, the program can decrypt correctly according to this equation.

The second problem which confused us for several days is that after encryption process. It will generate a *long long* type array to store the huge number. According to Socket protocol, the buffer is void* type which means it can be transformed into any type we want to send. But in real case, the client cannot receive the encrypted content. Even when we cast the type from *longlong** to *char**, only the first number was read by client. It is quite confusing since we do not clear the implementation of socket process in library. We tried many methods and finally found that because the buffer in server is not char* type any more, the system cannot determine the EOF position for that buffer and choose to treat it as one number. After that we modified the encryption function to count the word number of original content and tell the rio_writen call the exactly size of the buffer, the receiving process seems and client can receive the same content as the server sent.

3.3 Result Demonstration

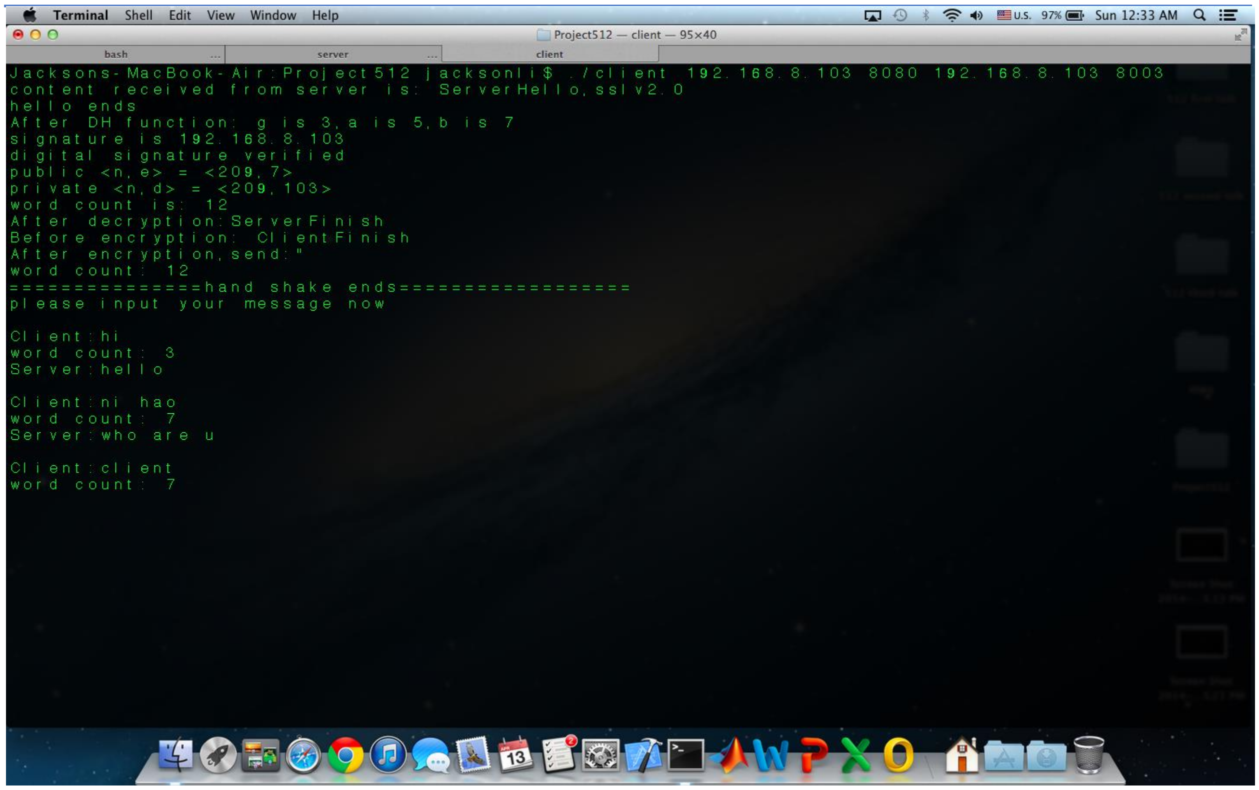
Here is the result in the experiment:



```
Jacksons-MacBook-Air:Project512 jacksonli$ ./server 192.168.8.103 8080
message received: ClientHello,sslv2.0

After DH function: g is 3,a is 5,b is 7
after decipher is ClientKeyExchange,RSA 11 19,TLSv6
public <n,e> = <209,7>
private <n,d> = <209,103>
Before encryption: ServerFinish
After encryption?
word count:12
Receive from client: ClientFinish
=============hand shake ends===========
word_count:3
Client:hi
Server:hello
word_count:7
Client:ni hao
Server:who are u
word_count:7
Client:client
Server:
```

Fig2a.Result (server side)



```
Jacksons-MacBook-Air:Project512 jacksonli$ ./client 192.168.8.103 8080 192.168.8.103 8003
content received from server is: ServerHello,sslv2.0
hello ends
After DH function: g is 3,a is 5,b is 7
signature is 192.168.8.103
digital signature verified
public <n,e> = <209,7>
private <n,d> = <209,103>
word count is: 12
After decryption:ServerFinish
Before encryption: ClientFinish
After encryption,send:"
word count: 12
===============hand shake ends==================
please input your message now

Client:hi
word count: 3
Server:hello

Client:ni hao
word count: 7
Server:who are u

Client:client
word count: 7
```

Fig2b.Result (client side)

Explanation:

Server side: In the first line, the server and client exchanges the D.H key. (In this case, it is 3, 5 and 7). The next line is the message received from client as the confirmation of cryptograph protocol. The third and fourth line is the D.H key exchange information, which is same as the key in client side. The 5th to 8th line is the “Finish” information encrypted by RSA method and message received from client and decryption result. The rest of results is the talking message, the transmission of those messages are all encrypted by RSA method and decrypted on the other side.

Client side: The first line is the message sent by server to inform the cryptograph protocol. Then it generates the D.H key and exchange them with server in the second line. The third line is the server signature message decrypted by D.H method, which is same to server IP address (192.168.9.103). The 4th to 10th line is the RSA key pair and “Finish” message encrypted by RSA method. The rest of results is the talking message, which corresponding to the result in server side.

3.4 Some trials on Open SSL library

We want to combine the client side program with the proxy so that we can run the proxy to connect the real website. We’ve tried OPENSSL to initial our SSL settings and implement SSL communication, but the server always responses “http 1.1/ 400 bad request” and we need to figure out the correct package format. We researched the actual headers and packet type of SSL protocol according to [1] as illustrated below:

The structure definition for SSL handshake is as below:

```
 struct {
      HandshakeType msg_type;
      uint24 length;
      select (HandshakeType) {
      case hello_request: HelloRequest;
      case client_hello: ClientHello;
      case server_hello: ServerHello;
      case certificate: Certificate;
      case server_key_exchange: ServerKeyExchange;
      case certificate_request: CertificateRequest;
      case server_hello_done: ServerHelloDone;
      case certificate_verify: CertificateVerify;
      case client_key_exchange: ClientKeyExchange;
      case finished: Finished;
            } body;
   } Handshake;
```

Message structure for SSL handshake is shown below:

```
enum {
      hello_request(0), client_hello(1), server_hello(2),
      certificate(11), server_key_exchange (12),
      certificate_request(13), server_hello_done(14),
      certificate_verify(15), client_key_exchange(16),
      finished(20), (255)
} HandshakeType;
```

The Client Hello message should follow the structure as below:

```
struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suites<2..2^16-1>;
    CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;
```

The definition of each part in the Client Hello is shown as below.

```
struct {
    uint8 major, minor;
} ProtocolVersion;
struct {
    uint32 gmt_unix_time;
    opaque random_bytes[28];
} Random;
```

gmt_unix_time is the number of seconds from at 0:00 on January 1, 1970 to the current time, in the format of standard UNIX 32-bit transmit clock. The random_bytes should be a random number which is 28 bytes long.

```
    opaque SessionID<0..32>
```

Session ID is defined by the server side, if the client do not have reusable session ID or do not wish to negocaite the safty parameter, this value should be left blank.

```
    uint8 CipherSuite[2];
```

Each CipherSuite represents a key exchange method; the CipherSuite should be arranged by priority.

```
 enum {
    null(0), (255)
} CompressionMethod;
```

The compression method should also be arranged according to priority and should always have a blank compression method so that the client and server can always reach an agreement.

The structure for key exchange message is as below:

```
struct {
  select (KeyExchangeAlgorithm) {
  case ec_key_agreement:
  Opaque ClientECDHEParams<1..2^16-1>;
  case ibs_key_agreement:
  Opaque ClientIBSDHParams<1..2^16-1>;
  case ecc_encryption:
  opaque ECCEncryptedPreMasterSecret<0..2^16-1>;
  case ibe:
  opaque IBEEncryptedPreMasterSecret<0..2^16-1>;
  case rsa:
  opaque RSAEncryptedPreMasterSecret<0..2^16-1>;
} exchange_keys;
} ClientKeyExchange;
```

The “RSAEncryptedPreMasterSecret” is the pre-master key for RSA and should be encode with the public key of the server. The next step we will try to figure out the exact value of parameters.

4. Conclusion

In this project, we successfully monitor the SSL handshake process by simulating the information exchange and data encryption on client and server side. The result demonstrates the effectiveness of D.H method and RSA method. The program can be used as an online information change software such as chatting or file exchange (if sever has storage management module). In addition, we have tried to use open SSL to connect website.

5. Reference

1. Tan Wuzheng, Huang Min. SSL VPN Technology Specification [Z]. China national Encryption Administration. 2010.08

2. Florian Kohlar and Sven Schäge and Jörg Schwenk. On the Security of TLS-DH and TLS-RSA in the Standard Model [OL]. Cryptology ePrint Archive: http://eprint.iacr.org/2013/367. 2013.06

3. Eric Rescorla. SSL and TLS: Designing and Building Secure Systems [M]. Addison-Wesley Professional. 2001: chapter 9

4. For introduction about RSA method, see: http://en.wikipedia.org/wiki/RSA_ (cryptosystem)