http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 13 (4): 601-613, 2014
ISSN 1812-5638 / DOI: 10.3923/itj.2014.601.613

Mechanized Verification of Security Properties of Transport Layer Security 1.2 Protocol with Crypto Verif in Computational Model

Bo Meng, Leyuan Niu, Yitong Yang and Zimao Li
School of Computer, South-Center University for Nationalities,
MinYuan Road # 708, HongShan Section, 430074, Wuhan, Hubei, China

Abstract: In modern society, many transactions have been processed through web-based applications. In order to protect those critical applications against attacks, Transport Layer Security (TLS) protocol has been implemented and widely deployed. The related literatures show that security analysis of TLS 1.2 protocol where cipher suite is RSA encryption has not been implemented with mechanized tool in computational model. Hence in this study, Blanchet calculus is used to analyze TLS 1.2 protocol where cipher suite is RSA encryption with mechanized tool crypto verif in computational model. The term, process and correspondence are used to model authentication in TLS 1.2 protocol where cipher suite is RSA encryption. The result shows that TLS 1.2 protocol where Cipher suite is RSA encryption has the pre master key confidentiality and authentication from server to client. The first mechanized analysis on TLS 1.2 protocol where Cipher suite is RSA encryption is implemented in computational model with active adversary in this study.

Key words: Verification, confidentiality, authentication, correspondence, protocol security

INTRODUCTION

With the rapid development of network technology and Web technology, Internet has a strong influence on all kinds of aspects in society. Many transactions have been processed through web-based applications. In order to protect the web-based applications against passive and active attacks, TLS protocol (http://tools.ietf.org/html/rfc5246) is widely deployed. The latest version is TLS 1.2. The objective of TLS 1.2 protocol is to provide confidentiality and authentication between two communicating parts. So, people have paid a close attention to analysis and verification of its security properties and want to get more confidence on it.

For the sake of analyzing and proving the security properties of security protocols and strengthening the confidence of the people, two approaches have been proposed form the beginning of the 1980s (Meng, 2011). One is symbolic model which is also called Dolev-Yao model and in which cryptographic primitives are abstracted as perfect black boxes. Until now lots of mechanized tools in this model have been developed, for example, Casper, Isabelle, ProVerif and Scyther. In 2005 Ogata and Futatsugi (2005) formally analyzed in symbolic model TLS protocol with CafeOBJ method based on equational reasoning. But the results of proof based on symbolic model are not quite clear.

The other approach is computational model based on complexity and probability. The attacker in computational model is modeled as a probabilistic polynomial-time machine. The computation model is more realistic but it is difficult to mechanized proof until the introduction of mechanized tool Crypto verif (Blanchet, 2008) which is the first mechanized tool with computational model. In 2012, (Jager *et al*., 2012) firstly analyzed by hand TLS protocol where Cipher suite is ephemeral Diffie-Hellman key exchange protocol in standard model. In 2013, (Fournet *et al*., 2013) used F7 refinement typechecker to verify security properties of TLS protocol implementation.

According to the related references, analysis of security of TLS 1.2 protocol where Cipher Suite is RSA encryption with mechanized tool in computational model is not found.

Owning to the previous analysis of TLS 1.2 protocol is not quite clear, in this study, Blanchet calculus is used to analyze TLS 1.2 protocol where Cipher suite is RSA encryption with mechanized tool Crypto verif.

CONTRIBUTION AND OVERVIEW

During the past several years TLS protocol bas been implemented and widely deployed in many web-based applications. For the sake of verifying the security properties of security protocols and improving the

Corresponding Author: Zimao Li, School of Computer, South-Center University for Nationalities,
MinYuan Road # 708, HongShan Section, 430074, Wuhan, Hubei, China Tel: 0086-18602707481

confidence on its security, symbolic model and computational model have been developed. Symbolic model is also called Dolev-Yao model; computational model is based on complexity and probability. The later model is more realistic owning to that the attacker is modeled as a probabilistic polynomial-time machine. According to the related references, until now it is not existence that security analysis of TLS 1.2 protocol where Cipher suite is RSA encryption with mechanized tool in computational model.

So analysis of security properties of TLS 1.2 protocol where Cipher suite is RSA encryption with mechanized tool in computational model plays an important role in security protocol field and is a significant work. Hence in this study, Blanchet calculus is used to analyze TLS 1.2 protocol where Cipher suite is RSA encryption with mechanized tool.

The main contributions of this study are summarized as follows:

- The status of analysis in TLS1.2 protocol including in symbolic model and in computational model is presented. Until now it is not existence that analysis security of TLS 1.2 protocol where Cipher suite is RSA encryption with mechanized tool in computational model
- Applying Blanchet calculus in computational model with active adversary for mechanized verification of TLS 1.2 protocol where Cipher suite is RSA encryption. Authentication is expressed by non-injective or injective correspondence. Figure 1 shows the model of mechanized verification of TLS 1.2 protocol where Cipher suite is RSA encryption
- The result shows that TLS 1.2 protocol where cipher suite is RSA encryption has confidentiality of pre master key and authentication from server to client. The first mechanized verification on TLS 1.2 protocol where cipher suite is RSA encryption in computational model of in active adversary is implemented in this study

RELATED WORK

In this part the status quo of the proof in TLS protocol and its implementation based on symbolic model and on computational model is presented. Until now there does not exist that analysis of TLS 1.2 protocol with mechanized tool in computational model.

In 1996, Wagner and Schneier (1996) analyzed Secure Sockets Layer (SSL) 3.0 which is the former version of TLS protocol with informal method. They found some active attacks which mainly include change cipher spec-dropping, Key exchange algorithm spoofing and version roll-back attack.

In symbolic model (Paulson, 1999) uses Isabelle to automatically analyze TLS protocol and proves its security claimed in TLS specification; in 2005 (Ogata and Futatsugi, 2005) formally analyzed TLS protocol with CafeOBJ method based on equational reasoning. The results are that TLS protocol has the security properties with the condition pre-master secrets cannot be leaked.

In computational model (Jonsson and Kaliski, 2002) present an analysis of a variant of TLS hand shake protocol where Cipher suite is RSA by hand. They argue that TLS should use Tagged Key-Encapsulation Mechanism (TKEM) 1 rather than TKEM2; In 2008, (Morrissey *et al*., 2008) analyzed TLS hand shake protocol in random model by hand. They firstly proposed a security model for key agreement protocol and then analyzed TLS hand shake protocol with the condition that Message Authentication Code (MAC) value is send in

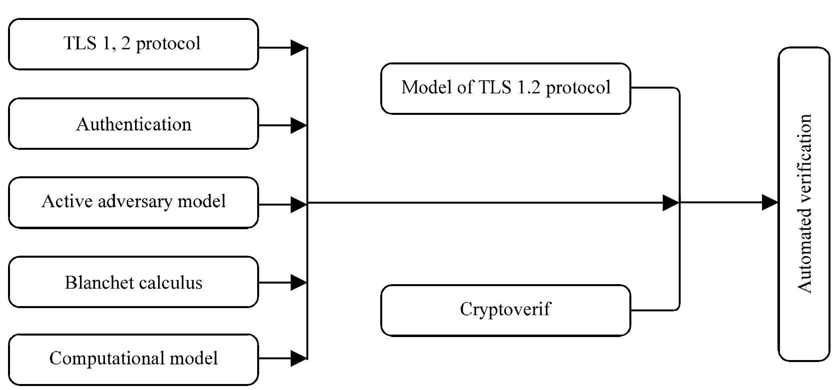

Fig. 1: Model of mechanized verification of TLS 1.2 protocol where Cipher suite is RSA encryption

the plaintext and showed that TLS key agreement protocol is secure. However the version of TLS protocol with Diffie-Hellman Key Exchange is not analyzed; In 2011 (Paterson *et al*., 2011) presented the scheme of length-hiding authenticated encryption to analyze security of TLS Record Protocol by hand. They found a new distinguishing attack against TLS where variable length padding and short MACs are used; in 2012 Jager *et al*. (2012) firstly analyzed by hand TLS protocol where Cipher suite is ephemeral Diffie-Hellman key exchange in the standard model. They introduced the authenticated and confidential channel establishment security model and showed that the combination of the TLS hand shake protocol with the TLS Record Layer can be proven secure.

Here the aspect of security properties of implementation of TLS protocol is discussed. In 2009 Chaki and Datta (2009) presented the first framework to automatically analyze authentication and secrecy of the hand shake in Open SSL. In 2012, (Bhargavan *et al*., 2012) firstly implemented TLS protocol 1.0 with language F#. And then they designed a model extraction method to extract the abstract model. Finally, they used Crypto verif to analyze security properties of TLS implementation in F#. In 2013 (Fournet *et al*., 2013) developed a new, verified, reference implementation of TLS protocol 1.2 with functional language F#. And then they used the F7 refinement type checker to verify its security properties and find a few new attacks.

REVIEW OF BLANCHET CALCULUS AND MECHANIZED PROOF TOOL CRYPTO VERIF

In this section there is a brief overview of Blanchet calculus (Blanchet, 2008) and the mechanized prover Crypto verif, formalize TLS 1.2 protocol using it (http://www.Crypto verif.ens.fr). Blanchet calculus is a probabilistic polynomial calculus and has been developed for the automated proof security protocols. In this calculus, messages are bitstrings and cryptographic primitives are functions operating on bitstrings. Blanchet calculus includes terms and processes.

The mechanized prover Crypto verif can directly prove security properties of cryptographic protocols in the computational model in which the cryptographic primitives are functions on bit-strings and the adversary is a polynomial-time Turing machine. It also can prove secrecy properties and events that can be executed only with negligible probability.

Crypto verif can works in two modes: A fully automatic and an interactive mode. The interactive mode, requires a Crypto verif user to provide commands that indicate the main game transformations. Crypto verif is sound about the security properties it shows in a proof but properties it cannot prove are not necessarily invalid.

REVIEW OF TLS 1.2 HAND SHAKE PROTOCOL

TLS 1.2 protocol mainly consists of hand shake protocol and Record protocol which play a very important role in TLS protocol. TLS Record Protocol is a layered protocol and accepts messages from higher level layer to be transmitted and partitions the data into blocks and transmits the result. Received data is processed inversely and then delivered to higher level clients. TLS 1.2 hand shake Protocol is one of the defined higher-level clients of TLS 1.2 Record protocol and is used to establish a secure session between server and client. In other words TLS 1.2 hand shake protocol can provide the authentication from server to client and confidentiality of pre master key. Figure 2 describes the messages exchanged between server and client in the simplification of TLS 1.2 hand shake protocol.

In TLS 1.2 hand shake protocol there are two communicating parties involved. One is client, the other is server. Client and server use TLS 1.2 hand shake protocol to share a common key and server use TLS 1.2 hand shake protocol to authenticate identity of client. The simplification of TLS 1.2 hand shake protocol contains seven messages which are client hello, Server hello, Server certificate, Server hello done, client key exchange, Client finished and Server finished:

$$\text{Client Hello:} = \text{client\_version} \,\|\, \text{client\_random} \,\|\, \text{session\_id} \,\|\, \text{cipher\_suites} \,\|\, \text{compression\_methods} \,\|\, \text{extensions} \quad (1)$$

When a client wants to connect to a server, TLS 1.2 hand shake protocol requires client to send the Client hello message. Client hello mainly includes client_version, client_random, session_id, cipher_suites, compression_methods and extensions. Client_version parameter describes the version of the TLS protocol through which the client wants to communicate with server during this session. Client_random parameter describes the structure generated by client who is used to prevent against replay attacks and to compute the master key. Session_id parameter is an important parameter in client hello message. It describes the Identity (ID) of a

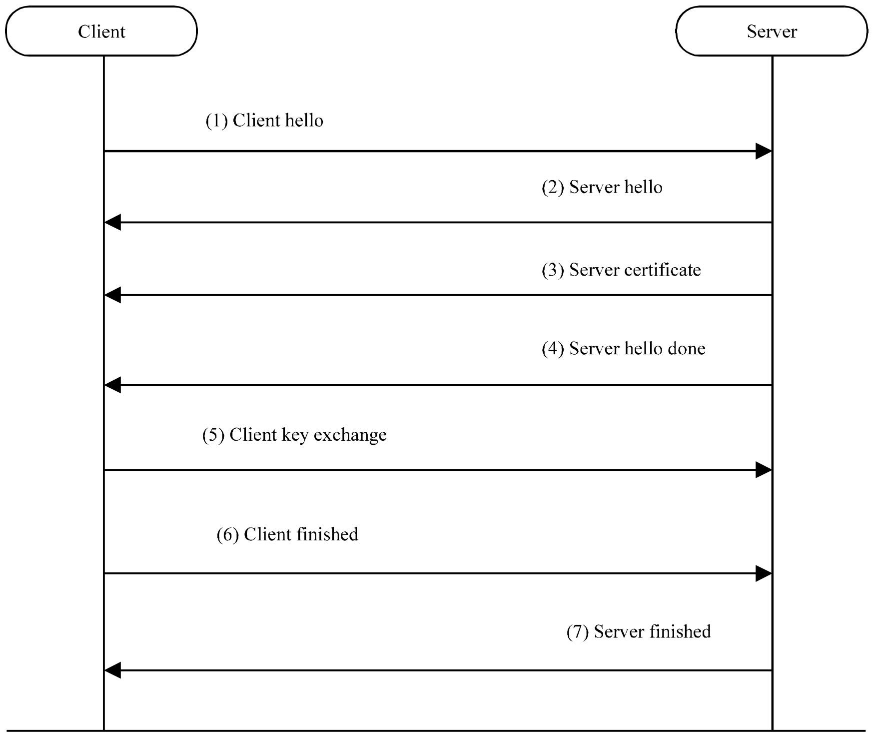

Fig. 2: TLS 1.2 hand shake protocol

session that the client wants to use for this connection. Cipher_suites parameter describes that a list of the cryptographic options supported by the client. Compression_methods parameter describes the compression methods supported by the client. Extensions parameter describes some extended functions need to be supported by server. The client generates Client hello message and sends it to server:

$$\text{Server hello: =c server\_version} \\ \|\text{server\_random}\|\text{session\_id}\| \\ \text{chiper\_suites}\|\text{compression\_methods}\|\text{extensions} \quad (2)$$

When the server receives the Client hello message, it then constructs the Server hello message. Server hello message mainly includes server _version, server_random, session_id, cipher_suites, compression_methods and extensions. Server generates message Server hello according to the message Client hello. Server_version parameter is generated with the condition that the lower of that required by the client in client_version in message Client hello and the highest supported by the server. Server_random parameter describes the structure generated by server which is used to prevent against replay attacks and to calculate the master key. Session_id parameter describes that the server finds the session id from the cache according to the session id from the Session_id in message Client hello. Cipher_suites parameter is from the cipher_suites in Client hello message. Here the value of Cipher_suites is RSA public encryption. Compression_methods parameter is from the compression_methods in Client hello message. Extensions parameter is from the extensions in Client hello message. After the server constructs message server hello, it sends it to the client.

$$\text{Server certificate: = version}\|\text{serial number} \\ \|\text{algorithm indentifier}\|\text{issuer}\|\text{utc time}\| \\ \text{server\_subject\_name} \\ \|\text{server\_subject\_key\_info}\|\text{signature} \quad (3)$$

After the server sends the message server hello to the client, he constructs the message Server certificate to the client. The message Server certificate is used to tell the public key of the server to the client. The message Server certificate mainly contains version, serial Number, algorithm identifier, issuer, utcTime, server_subject_name, server_subject_key_info and signature. Here the server_subject_name and server_subject_key_info parameters are mainly introduced. Server_subject_name parameter is the name of server which is the owner of certificate. Server_subject_key_info parameter mainly

```
Expand IND_CCA2_public_key_enc (rsakeyseed, rsapkey, rsaskey, cleartext , message, seed,
                                rsaskgen, rsapkgen, enc,dec, injbot, Z,Penc,Penccoll)
Expand Collision resistant_hashh (hashkey, hashinput, input , hash, phash)
Expand PRF_new (key seed, key, input, output, kgen, f, Pprf)
Define IND_CPA_sym_enc (key seed, key, cleartext, ciphertext, seed, kgen, enc,dec, injbot, Z, Penc)
Define PRF_new (key seed, key, input, output, kgen, f, Pprf)
{
 Param N1, N2, N3.

 Fun fkey, label, input: output
 Fun kgen (key seed): key

 Equiv N3 new r: keyseed; new labelc:label; N1 of (x: input): = f (kgen(r), labelc, x)
   <=(N3 *Pprf (time+(N3-1)*(time(kgen)+N1 *time (f, maxlength(x), N1, maxlength(x)=>
    N3 new r: keyseed; !N1 of (x:input): =
          Find[unique] j<=N1 such that defined (x[j],r2[j]) && otheruses(r2[j]) && x = x[j] then r2[j]
          Else new r2: output; r2
}
```

Fig. 3: Cryptographic assumptions

includes the public key of the server and the related information on the public key:

$$\text{Server hello done: = Server hello done} \quad (4)$$

After the server sends the message server certificate, he constructs the message server hello done and sends it to the client. The server hello done message is sent by the server to show that the end of the server hello and associated messages. After sending server hello done message, the server will wait for a client response:

$$\text{Client key exchange: = Client key exchange} \\ \|\text{Encrypted pre master secret} \quad (5)$$

After it receives the server hello done message, client key exchage message is sent by the client. Client key exchage message is the key message in TLS 1.2 hand shake protocol which is used to tell the pre Master key to the client in a secure way. Client key exchage message mainly includes key exchange algorithm and encrypted pre master secret parameters. With this message, the pre master key confidentiality is implemented by transmission of the RSA-encrypted pre Master key used here. Key exchange algorithm parameter describes the key exchange algorithm. The value of key exchange algorithm is RSA. Encrypted pre Master secret parameter describes the structure of pre Master key confidentiality. Pre_master_secret is generated by the client and is used to generate the master secret:

$$\text{Client finish: = Client finish} \quad (6)$$

Client finished message is always sent immediately to the server when the verification of the key exchange and authentication processes were successful. The server must verify that the contents are correct. Once the client has sent client finished message and received and validated the Server finished message from the server, the client may begin to send and receive application data over the connection:

$$\text{Server finished: = Server finished} \quad (7)$$

A Server finished message is always sent immediately to the client when the verification of the key exchange and authentication processes were successful. The client must verify that the contents are correct. Once the server has sent server finished message and received and validated the client finished message from the client, the server may begin to send and receive application data over the connection.

FORMALIZING TLS 1.2 HAND SHAKE PROTOCOL IN BLANCHET CALCULUS

When TLS 1.2 hand shake protocol is formalized it assumes that public-key encryption is indistinguishability under adaptive chosen ciphertext attacks. Public key signature is unforgeable against chosen-message attack. Symmetric encryption is indistinguishability under chosen-plaintext attacks and probabilistic symmetric encryption. Hash function including hash functions defined by us according to the requirements is collision resistant and unforgeability against adaptive chosen messages attacks. Figure 3 describes the cryptographic assumptions in our analysis of TLS 1.2 hand shake protocol.

```
Event server (output).
Event client (output).
Query x:output; event server (x)==>client (x)
Query x:output; inj:event server(x)==>inj: client(x)
Query secret pre Master secret
```

Fig. 4: Query events and secret

```
Let TLS 1.2 process = initator process (|(!n1 client process)
|!n2(server process)))
```

Fig. 5: TLS 1.2 hand shake protocol process

```
Let initator process =
    Start ();
    New seed one: rsakey seed;
    Let pkeyrsa: Rsapkey = rsapkgen (seed one) in
    Let skeyrsa: Rsaskey = rsaskgen (seed one) in
    New seed two: Keyseed;
    Let signpkey: Pkey = pkgen (seed two) in
    Let signskey: Skey = skgen (seed two) in
    New keyhash:hash key;
    c̄ pkeyrsa, signpkey, key hash
```

Fig. 6: Initator process

Here non-injective correspondences and injective correspondences are used to model authentication from server to client. Firstly non-injective correspondences are used: Event server(x) =>inj: client(x) is used to authenticate client by server. Then injective correspondences are used: Event inj: event server(x) => inj: client(x) is used to authenticate client by server. Figure 4 describes the events and correspondence. Query secret pre master secret is used to query the secrecy of pre master secret.

The complete formal model of TLS 1.2 hand shake protocol where Cipher suite is RSA encryption in Blanchet calculus is given in Figure. Figure 5-8 reports the basic processes include initator process, client process and server process in authentication and secrecy forming the model of TLS 1.2 protocol. The process TLS 1.2 process in Fig. 5 is assumed to run in interaction with an adversary which also models the network.

Initator process generates server's public key pkeyrsa and private key signp key for encryption in RSA scheme and public key signs key and private key signs key, for digital signature in RSA cryptosystem. Finally, it sends signp key hash by the public channel c.

In detail initator process in Fig. 6 firstly generates server's public key rsaky seed in the following procedure: Initator process receives a null message on channel start, sent by the adversary. Then, it chooses randomly with uniform probability a bitstring seed one in the type rsakey seed, by the construction new seed one: Rasaky seed. A type T, such as raskeyseed, aims at denoting a set of bitstrings. Then, initator process generates the server's public key pkeyrsa corresponding to the coins seed one, by calling the public-key generation algorithm rsapkgen (seed one). Similarly, initator proccess generates the secret key skeyrsa by calling rsaskgen(seed one).

After that initator process generates the server's public key signp key and private key signs key for digital signature in RSA scheme in the following procedure: Initator process chooses randomly with uniform probability a bitstring seed two in the type by the construction new seed two: Key seed. Then, initator process generates the server's public key signp key corresponding to the coinsseed two, by calling the public-key generation algorithm pkgen (seed two). Similarly, initator process generates the secret key signs key by calling skgen (seed two). At the same time it produces the hash key key hash in type hash key by the construct new key hash: Hash key. Finally, it sends pkeyrsa, signp key, hash by the public channel c. After sending this message, the control passes to the receiving process which is part of the adversary. Several processes are available, which represent the roles of client and server process. $(|(!^{n1}$client process$)|(!^{n2}$server process$))$ is the parallel composition of client process $|!^{n1}$client process, server process $|!^{n1}$Server process. It makes simultaneously available the processes.

Client process is modeled as client process in Blanchet calculus in Fig. 7. Client process generates the client-version Client version in type version by the construct new client version: Version, client-random client random in type by the construct new client random: random and cipher-suites client Cipher suitesin type cipher_suites by the construct new client cipher suites: Cipher_suites. And then it produces Client Hello message message one in type message using the function concat A (client version, client random, null, client cipher suites) by the construct let message one: Message = concat A (client version, client random, null, client phersuites) in. Finally, it sends message one, one message on by the public channel through the construct $\overline{c1}\langle$one, message one$\rangle$.

After that client process receives the message two, message two_s in type message from the public channel

```
Let client process =
    C ();
    (*Client hello*)
    New clientversion: Version;
    New clientrandom: Random;
    New client cipher suites: cipher_suites;
    Let message one: message = concat Aclient
    Version, clientrandom, null, clientCipher suites in
    c̄1⟨one, messageone⟩;
    c2(=two, message two_s:message);
    Let concatA (Agreement_version:version,server_random:random,
                 sessionid_s:session_id,suites_s:chiper_suites) =message two_s in
    C();
    c3 = (three, message three_s:message, signone_s:signature);
    Iif check(message three_s, signpkey, signone_s) then
    Let concatB (Certify Version: Certificate Version,
                 name_s:subjectname,certify_info: certificate) =message three_s in
    Let concatinfo(pkeyrsa_s:rsapkey, name_algorithm:algorithmname)=certify_info in
    c̄⟨⟩;
    c4 = (four, messageour_s:message);
    If messageour_s = Server hello done then
    (*Client exchange*)
    New pre Master secret:key;
    New r2:seed;
    Let messagefive:message = enc(key to clear text (pre Master secret), pkeyrsa_s, r2) in
    c̄5⟨five, messag efive⟩;
    (*Client Finished*)
    C ();
    Let c_s_random_c:input = concat prf (clientrandom, server_random) in
    Let hash message_c:hash input = cocat hash one(message one, message five) in
    Let key to output (Master secret_c:key)=f (premastersecret,mastersecret,
                                               c_s_random_c) in
    Let verify data: Output = f (mastesecret,clientfinished,
                                 hash(keyhash,hashmessage_c)) in
    Event client(verify data);
    c̄6⟨six, verify data⟩.
```

Fig. 7: Client process

c2 through the construct c2 (= two, message two_s: Message). And then it gets the version agreement_version in type version, server_random in type random, sessionid_s in type session_id and cipher_suites suites_s in type cipher_suites through the function concat A (Agreement_version: Version, server_random: random, sessionid_s: Session_id, suites_s: Cipher_suites) by the construct let concat A (Agreement_version: Version, server_random: Random, sessionid_s: Session_id, suites_s: Cipher_suites) = message two _s in. Client process sends the message = three, message three_s: Message, sign one_s: Signature in the public channel c3 by the construct ******.

And then client process gets the digital signature sign one_s: Signature of message message three_s: message through the public channel c3 by the construct c3 = (= three, message three_s: message, signone_s: signature. It uses the function check () to verify the digital signature signone_s: signature of messag emessage three_s: message. If the verification is successful, then it gets certificate version certify version: Certificate version, server subject name name_s: subject name and certificate certify_info: Certificate using the construct let concat B (Certify version: Certificate version, name_s: subject name, certify_info: Certificate) = message three_s: in. Then it gets the public key pkeyrsa_s: rsapkey of the server and the algorithm name_algorithm: algorithm name from certificatecertify_info: Certificate using the construct let concatinfo (pkeyrsa_s: rsapkey, name_algorithm: algorithm name) = certify_info.

After that client process receives the message message our_s: message through the public channel c4 by the constructc4 (= four, message our_s: Message). If the message message our_s: Message is server hello Done then it creates the pre master key pre master secret

```
Let Server process =
    c7 (=one, messageone_c: message);

    Let concat A (client_version:version,client_random:random,
                  id:session_id,ciphersuites:cipher_suites) = message one_c in

    (*Session*)
    find i<=N such that defined (sessionid [i] && (sessionid [I] =id) then
            let id_store: Session_id=id in
            if id = null then
            new session id: Session_id;
    let id_store = Session id in
            If client_version = Agreement version then
            new server  random: Random;
            Let message two: Message = concat A(Agreement version,server random, id_store, suites)i n
c8 ⟨two, message two⟩;
(*Server certificate*)
c ();
New name: subjectname;
New subject algorithm: Algorithmname;
Let certificate info: Certificate = concat info (pkeyrsa,subject algorithm) in
Let message three: Message = concat B (certify, name, Certificate info) in
new r1: Seed;
Let sign one: Signature = Sign (message three, signs key, r1) in
c9 ⟨three, message three, signone⟩;
(*Server hello done*)
c ();
c10 ⟨four, server hello don⟩
e ();
c11(= five, messagef ive_c: message);
Let injbot (key to clear text (pre Master_secret: Key)) = dec (message five_c, skeyrsa) in
c ⟨⟩;
(*Server finished*)
c12 (= six, verify data_fc: Output);
Let c_s_random_s: Input = concatprf  (client _random, server random) in
Let key to out put (Master secret_s: Key) = f(pre Master_secret, maste secret,c_s_random_s) in
Let hash message_fc: Hash input = cocat hash one (message one_c, message five_c) in
if f (Master secret_s, client finished, hash (key hash, hash message_fc)) = verify data_fc then
Let hash message: Hash input = cocat hash two (message two, message three, Server hello done) in
event server (verify data_fc);
c13 , (seven, f (Master secret_s, server finished, hash (key hash, hash message)))
```

Fig. 8: Server process

in type using the construct new pre Master secret: kay and uses the RSA encryption one() to encrypt it with public key pkeyrsa_s by the construct let message five: Message = one (key toclear text (pre Master secret), pkeyrsa_s, s2 and gets the message message five in type message. Finally, it sends message message five by the public channel c5 using the construct $\overline{c3}$ ⟨= three, message three_s: message, sign one_s: Signature⟩.

Client process uses the function concatprf() to compute the hash input_s_random_c: Input of message client random, server_random by the construct let c_s_random_c: Input = concatprf (client random, server_random) in. And then it uses the function cocat hash one () to calculate the hash input cocathashone of message message one, message five by the construct let hash message_s: Hash input = cocat hash one (message one, message five) in. Client process gets the master key master secret_c: Key and uses the PRF function f() to generate the master key master secret_keyc: Key by the construct let key to output (message secret_c: Key) = f (pre master secret, master secret, c_s_random_c in. Also it use the PRF function f() to compute by the construct let verify data: Output = f(master secret_c, client finished, hash (key, hash, hash message_c)). Finally, it executes the event event client (verify data) and send it through the public channel c6 by the construct $\overline{c6}$ ⟨six, verify data⟩.

Server process is modeled as server process in Blanchet calculus in Fig. 8. Server process receives the message one, message one_c: Message from the public channel c7 through the construct c7 (= one, message one_c: Message). And then it uses the function concat A to parse the message nessage one_c and gets client versionclient version: Version, client random client_random: Random, session id id: Session_id and cipher suite cipher suites: Cipher_suites by the construct let concat A(client_version: Version, client_random: random, id, cipher_suites)-message one_c in Server process checks id whether it defined and is equal to sessionid or not by the construct find I<=N such that defined (session id [i] && (session id [i] = id). If the result is ok, then it stores id into id_store: Session_id using the construct let id_store: Session_id = id in. If is null, then server process generates a new session id in type session_id by the construct new session id: session_id and stores it into id_store by the construct let id_store = session id. At the same time if client_version is equal to agreement version, then it creates a random number server random by the construct new server random: random and use the function cancat A () to compute the message message two: Message by the construct let message two: message two: Message = concat A (Agreement version, server random, id_store, suites in. Server process sends message message two by the public channel c8 through the construct $\overline{c8}$.

Server process generates the subject name name: Subject name of certificate by the construct new name: Subject name and the subject algorithm of certificate subject algorithm: Algorithm name by the construct new subject algorithm: Algorithm name. At the same time it uses function concatinfo () to generate certificate information certificate info: Certificate by the construct let certificate info: Certificate = concatinfo (pkeyrsa, subject algorithm in and uses the function concat B() to compute the message message three: Message by the construct let message three: Message = concat B (certify name, certificate info) in. And then it generates the digital signature sign one: signature of the message message three: Message using the digital signature function sign () by the construct let sign one: Signature = sign (message three, signs key, r1) in. It also sends message three, sign one through the public channel c9 by the construct $\overline{c9}$⟨three, message three, sign one⟩ and sends four, server Hello Done through the public channel c10 by the construct $\overline{c10}$.

Server process receives the message message five_c: Message from the public channel c11 by the construct c11 (= five, message five_c: message). It uses the RSA decryption dec() to decrypt message five: Message and gets the pre master key pre master_secret: key by the construct let injbot (key to clear text (pre master_secret: key)) = dec (message five_c, skeyrsa) in.

After that server process receives the messageverify data_fc: Output from the public channel c12 with the constructc12 (= six, verify data_fc: Output). It uses the function concatprf to compute the c_s_random_s by the construct let c_s_random_s: input = concatprf (client_random, server random) in and uses the PRF function to calculate the master key master secret_s: key by the construct let key to output (master secret_s: key) = f(pre master_secret, master secret, c_s_random_s) in. At the same time it computes the hash input hash message_fc by the function cocat hash one() through the construct let hash message_fc: Hash point = cocat hash one (message one_c, message five_c) in. If verify data_fc is equal to output of the function f(master secret_s, client finished, hash (key hashm hash message_fc)), then it uses the function cocat hash two () to compute the hash input hash message: hash input by the construct let hash messageL hash input = cocat hash two (message two, message three, server hello, done) in. Finally, it executes the event event server (verify data_fc) and sends the message f(master secret_s, server finished, hash(key hash, hash message)) by the public channel c13 by the construct $\overline{c13}$ (seven, f(master secret_s, server finished, hash (key hash, hash message))).

MECHANIZED VERIFICATION OF SECRECY AND AUTHENTICATION IN TLS 1.2 HAND SHAKE PROTOCOL WITH CRYPTO VERIF

The inputs of Crypto verif have two formats. One is channels Front-end. The other is oracles Front-end. In both cases, the output of Crypto verif is same. The main difference between the two inputs is that the oracle front-end is based on oracles and the channel front-end is based on channels.

In this study, channel Front-end is used as the input of Crypto verif. Hence formal model of TLS 1.2 hand shake protocol where Cipher suite is RSA encryption must transform into the syntax of Crypto verif and generate the Crypto verif inputs in the form of channels Front-end.

Here non-injective correspondences and injective correspondences in Table 1 are used to model the authentication from server to client. First non-injective correspondences are used: event event server(x) ==> client(x) authenticate client by server. And then Injective correspondences are then used: event inj: event server(x)==>inj client(x) authenticate client by server.

From Fig. 9-11, the inputs of verification of authentication and secrecy in Crypto verif are presented. The analysis was executed by Crypto verif and succeeded.

Table 1: Correspondences in TLS1.2 protocol

Correspondences
Event server(x)==>client(x)
Inj: Event server(x)==> inj: Client (x)

```
param N

Type version [large,  fixed,  bounded]
Type random [large, fixed, bounded]
Type seed [fixed]
Type rsakeyseed [large, fixed]
Type rsapkey [bounded]
Type rsaskey [bounded]
Type clear [textlarge, bounded]
Type ciphertext [large]
Type keyseed [large, fixed]
Type pkey [bounded]
Type skey [bounded]
Type signinput [fixed]
Type signature [bounded]
Type hashkey [fixed]
Type hashinput [fixed]
Type hashoutput [bounded, fixed]
Type label [fixed]
Type keyfixed, [bounded]
Type session_id [large, fixed]
Type cipher_suites [large, fixed, bounded]
Type CertificateVersion [large, bounded]
Type Certificate [large]
Type subjectname [large, fixed]
Type algorithmname [large, fixed]
Type Exchange algorithm [large, fixed]
Type pre Master secret [large, fixed, bounded]
Type message [large, fixed, bounded]
Type host [bounded]
Type input [large, bounded]
Type output [fixed, bounded]

Const Agreementversion: Version. (*protocol version*)
Const suites:cipher_suites.(*cryptographic suite*)
Const certify: certificate version.(*certifi cate verson*)
Const server hello done:message.(*Server hello done*)
Const mastesecret: Label.
Const server finished: Label.(*h:label"server finished"*)
Const client finished:label.(*h:label"client finished"*)
Const null:session_id.(*session_id is null*)
Const one:host
Const two:host
Const three:host
Const four:host
Const five:host
Const six:host
Const seven:host

Fun key to clear text (key): Clear text [compos]
Fun key to out put (key): Output [compos]
Fun concat A (Version, Random, Session_id, Cipher_suites): Message [compos]
Fun concat B (Certificate  version, subjectname, certificate): Message [compos]
Fun concat info(rsapkey, algorithmname): Certificate [compos]
Fun concatprf [random, random]: Input [compos]
Fun cocat hash one (message, message): Hash input [compos]
Fun cocat hash two (message, message, message): Hash input [compos]

Proba penc
Proba penccoll
Proba psign
Proba psig
ncoll
Proba phash
Proba pprf
```

Fig. 9: Inputs of TLS1.2 protocol in cryp to verif

```
Define PRF_new (key seed, key, input, output, kgen, f, Pprf) {
Param N1, N2, N3.
Fun f (key, label, input): Output.
Fun kgen (key seed): Key.
Equiv N3 new r: key seed; new labelc: Label; N1 of (x:input): = f (kgen(r), labelc, x)
   <=(N3 *Pprf (time+(N3-1)*(time(kgen) + N1 *time (f, max length (x))), N1, max length (x)))=>
    N3 new r: Key seed;  N1 of(x:input): =
              Find [unique] j<=N1 such that defined (x[j],r2[j]) and other uses(r2[j]) and x = x[j] then r2[j]
              Else new r2: output;  r2.
}
Expand IND_CCA2_public_key_enc (rsakey seed, rsapkey, rsaskey, cleartext, message,
Seed, rsaskgen, rsapkgen, enc,dec, injbot, Z,Penc,Penccoll).
Expand UF_CMA_signature (key seed, pkey, skey, message, signature, seed, skgen, pkgen, sign, check,
Psign,Psigncoll).
Expand collision resistant_hash(hashkey, hashinput, input, hash,Phash).
Expand PRF_new (key seed, key, input, out put, kgen, f,Pprf ).
Event server (output). event client (output).
Query x:output; event server (x)==>client(x).query x:output; inj:event server(x)==>inj:client(x).
Query secret premastersecret.
Chaunel start, c, c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13
```

Fig. 10: Continue

```
Llet Client process=
    in(c, ());
    (*Client Hello*)
        New client version: Version; new client random: Random;  new client cipher suites: Cipher_suites;
        Let message one: Message = concat A (client version, client random, null, client cipher suites) in
     Out (c1, (one, message one));
        In(c2, (= two, message two_s: Message));
        Let concat A (Agreement_version: Version, server_random: Random,
Session id_s: Session_id, suites_s: Cipher_suites) = message two_s in
        Out(c, ());
        In (c3, (=three, messagethree_s:message, signone_s:signature));
        If check(messagethree_s, signpkey, signone_s) then
        Let concat B (Certify version: Certificate version,
Name_s: Subject name, certify_info: Certificate) = message three_s in
        let concat info(pkeyrsa_s: Rsapkey, name_algorithm: Algorithmname) = certify_info in
        out(c, ());
        In(c4, (=four, message our_s: Message));
        If message our_s = server hello done then
(Client exchange)
        New premaster secret:key; new r2: seed;
        Let message five: message = enc (key to clear text (prem
        [fixed [fixed* *
        [fixed[fixed astersecret), pkeyrsa_s, r2) in
        out(c5, (five, message five));
(Client finished)
        in(c, ());
        Let c_s_random_c: Input = concat prf (client random, server_random) in
        Let hash message_c: Hash input = cocat hash one (message
        [fixed[fixed* *
        [fixed[fixed geone, message five) in
        Let key to output (masters ecret_c: Key) = f (pre master secret, maste secret, c_s_random_c) in
        Let verify data: Output = f(maste rsecret_c, client finished, hash (key hash, hash message_c)) in
        Event client (verify data);
        out(c6, (six, verify data)).
```

Fig. 10: Inputs of TLS1.2 protocol in cryp to verif

```
Let server process=
    ¬ in(c7,  (=one, message one_c:message));
        Let concat A (client_version: Version, client_random: Random, id: Session_id,
Cipher suites: Cipher_suites) = message one_c in
    (* SessionId *)
        Find i<=N suchthat defined (session id[i]) && (session id[i] = id) then let id_store: Session_id = id in if id = null then
        New sessionid:session_id;
        Let id_store = session id in if client_version=Agreement version then new server random: Random;
        Let message two: Message = concat A (Agreement version, server random, id_store, suites) in out (c8, (two, message two));
    (* Server certificate *)
        In(c, ());  new name: Subject name;  new subject algorithm: Algorithmname;
        Let Certificate info: Certificate = concatinfo (pkeyrsa, Agreement version) in
        Let message three: Message = concatB(certify, name, Certificateinfo) in new r1:seed;
        Let signone: Signature = sign (message three, signskey, r1) in out(c9, (three, message three, signone));
    (* Server hello done *)
        In(c, ());
        Out(c10, (four, server hello done));
        In(c11, ( = five, message five_c:message));
        Let injbot (key to clear text (pre Master_secret:key)) = dec (messagefive_c, skeyrsa) in out(c, ());
(* Server finished *)
        In(c12, ( = six, verifydata_fc:output));
        Let c_s_random_s: Input = concatprf(client_random, server random) in
        Let key to output (Master secret_s: Key) = f(pre Master_secret, mastesecret, c_s_random_s) in
```

Fig. 11: Continue

```
        Let hash message_fc: Hash input = cocat hash one (message one_c, message five_c) in
        if f(mastersecret_s, clientfinished, hash(keyhash, hash message_fc)) = verifydata_fc then
        let hash message:hash input = cocathash two(message two, message three, server hello done) in event server (verify data_fc);
        out(c13, (seven, f(maste rsecret_s, server finished, hash (key hash, hash message)))).
process
    In(start, ());
    New seedone:rsakey seed;
    Let pkeyrsa: Rsapkey = rsapkgen (seed one) in
    Let skeyrsa: Rsaskey = rsaskgen (seed one) in new seedtwo:key seed;
    Let signpkey: Pkey = pkgen (seed two) in
    Let signskey: Skey = skgen (seed two) in new keyhash:hashkey;
    Out(c, (pkeyrsa, signpkey, keyhash));
    ((!N Client process)|(!N Server process))
```

Fig. 11: Inputs of TLS1.2 protocol in crypto verif

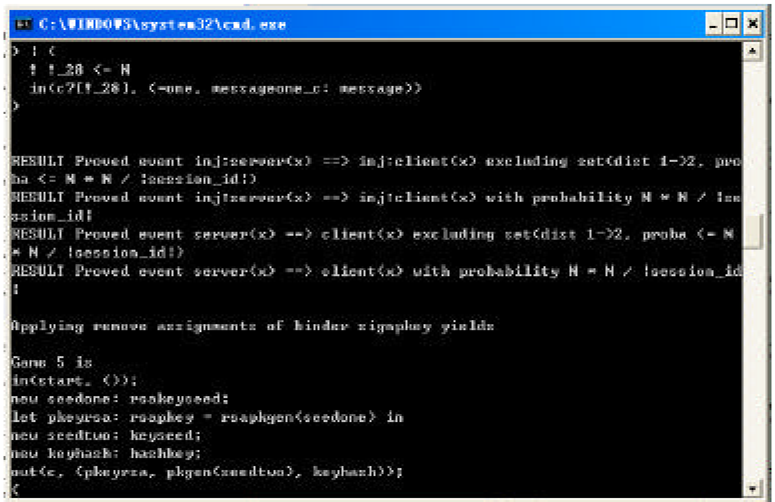

Fig. 12: Authentication of TLS 1.2 protocol in crypto verif

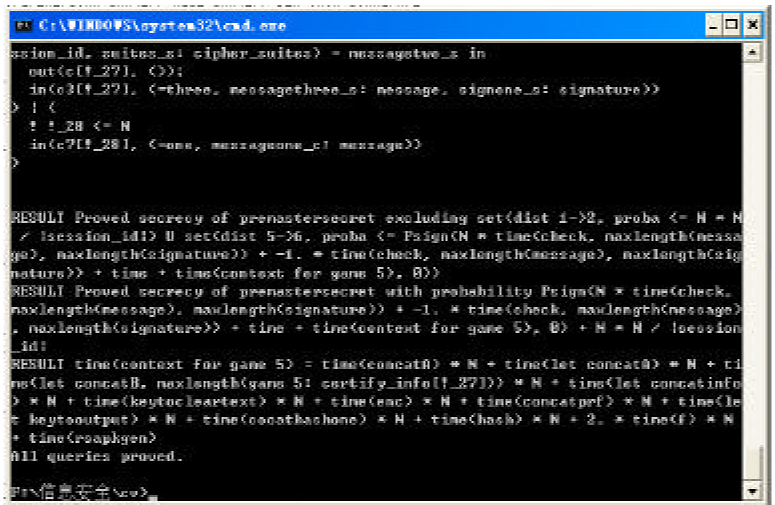

Fig. 13: Secrecy of TLS 1.2 protocol in crypto verif

The results show in Fig. 12 and 13. TLS 1.2 protocol where Cipher Suite is RSA encryption is proved to guarantee authentication from server to client and the secrecy of pre master key in computation model.

CONCLUSION

During the past several years TLS protocol bas been implemented and deployed widely in many web-based applications. According to the related references, until now it is not found that security analysis of TLS 1.2 protocol where Cipher suite is RSA encryption with mechanized tool in computational model. Hence, in this study, Blanchet calculus in computational model is used to analyze TLS 1.2 protocol where Cipher suite is RSA encryption with mechanized tool. The result shows that it has confidentiality of pre master key and authentication from server to client. The first mechanized verification on TLS 1.2 protocol where Cipher suite is RSA encryption in computational model of in active adversary is executed.

In the near future the verification of the Java implementation of TLS 1.2 protocol in computational model is very interesting.

ACKNOWLEDGMENT

This study was supported in part by Natural Science Foundation of The state Ethnic Affairs Commission of PRC under the Grants No: 12ZNZ008, titled “Automatic Verification of Cryptographic Security in Security Protocol Java Code”, conducted in Wuhan, China from 1/1/2013 to 30/12/2013.

REFERENCES

Bhargavan, K., C. Fournet, R. Corin and E. Zalinescu, 2012. Verified cryptographic implementations for TLS. ACM Trans. Inform. Syst. Secur., Vol. 15. 10.1145/2133375.2133378

Blanchet, B., 2008. A computationally sound mechanized prover for security protocols. IEEE Trans. Dependable Secure Comput., 5: 193-207.

Chaki, S. and A. Datta, 2009. ASPIER: An automated framework for verifying security protocol implementations. Proceedings of the 22nd IEEE Computer Security Foundations Symposium, July 8-10, 2009, Port Jefferson, New York, pp: 172-185.

Fournet, C., K. Bhargavan, M. Kohlweiss, A. Pironti and P.Y. Strub, 2013. An implementation of TLS 1.2 with verified cryptographic security. Proceedings of the 2nd International Conference on Principles of Security and Trust, March 16-24, 2013, Rome, Italy, pp: 17-17.

Jager, T., F. Kohlar, S. Schage and J. Schwenk, 2012. On the security of TLS-DHE in the standard model. Proceedings of the 32nd Annual Cryptology Conference on Advances in Cryptology, August 19-23, 2012, Santa Barbara, CA., USA., pp: 273-293.

Jonsson, J. and B.S. Jr. Kaliski, 2002. On the security of RSA encryption in TLS. Proceedings of the 22nd Annual International Cryptology Conference on Advances in Cryptology, August 18-22, 2002, Santa Barbara, CA., USA., pp: 127-142.

Meng, B., 2011. A survey on analysis of selected cryptographic primitives and security protocols in symbolic model and computational model. Inform. Technol. J., 10: 1068-1091.

Morrissey, P., N.P. Smart and B. Warinschi, 2008. A modular security analysis of the TLS hand shake protocol. Proceedings of the 14th International Conference on the Theory and Application of Cryptology and Information Security, December 7-11, 2008, Melbourne, Australia, pp: 55-76.

Ogata, K. and K. Futatsugi, 2005. Equational approach to formal analysis of TLS. Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, June 10, 2005, Columbus, OH., USA., pp: 795-804.

Paterson, K.G., T. Ristenpart and T. Shrimpton, 2011. Tag size does matter: Attacks and proofs for the TLS record protocol. Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security, December 4-8, 2011, Seoul, South Korea, pp: 372-389.

Paulson, L.C., 1999. Inductive analysis of the internet protocol TLS. ACM Trans. Inform. Syst. Secur., 2: 332-351.

Wagner, D. and B. Schneier, 1996. Analysis of the SSL 3.0 protocol. Proceedings of the 2nd USENIX Workshop on Electronic Commerce, November 18-21, 1996, Oakland, California, pp: 29-40.