CS3211 Student Project

TLS Verification

Group 2

Fong Kuo Xin Anthony (A0229378N)

Nigel Lee (A0230647H)

Teo Chen Ning (A0230521Y)

Table of contents

Introduction

TLS is a cryptographic protocol that provides end-to-end security of data sent between applications over the Internet. It is widely used as cryptographic protocols designed to provide encryption security for data transmitted over a computer network. Today, TLS is one of the de-facto standards used by web browsers for encryption of data.

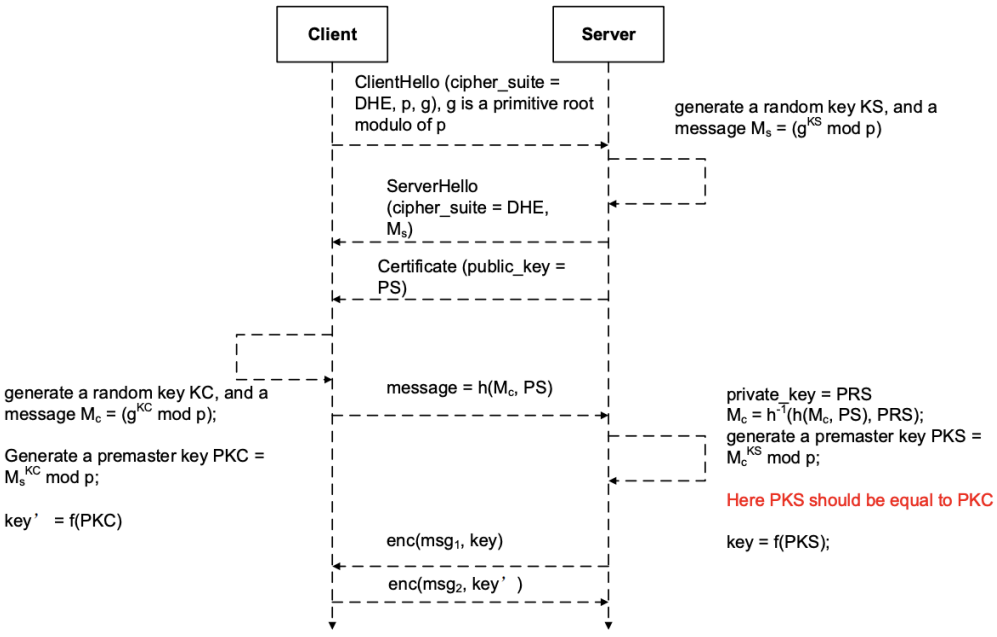
Objective

The aim of this project is to conduct a formal verification on a TLS model using the PAT software. The project involves creating concurrent models written in CSP with C# helper libraries, and testing the validity of the TLS model in providing a secure communication protocol. In this project, we will simulate different kinds of attackers in a variety of attack scenarios to test the TLS protocol.

Related works

There has been a great deal of work surrounding the verification of the TLS protocol. The most noteworthy of works include the work by Cremers et al [1], which offers a symbolic model and accompanying analysis of the TLS specification using the Tamarin prover. To the best of our knowledge, there has yet to exist a publication which verifies TLS using PAT, which is the goal of our current project.

TLS mechanism



TLS handshake with DHE exchange (credits: Wikipedia)

The TLS mechanism implemented in this project is a simplified, abstracted version of the Diffie-Helman key exchange method to simulate the process of a TLS connection.

First, the client generates two random values p and g, where p is a prime number and g is a primitive root modulo of prime number p. The client then sends these numbers to the server. The server generates a new random key KS, and uses that to create a server message M_s where $M_s = (g^{KS} \bmod p)$. In essence, if an attacker gets hold of M_s, it is nearly impossible for them to reverse guess KS, since prime factorization is computationally expensive.

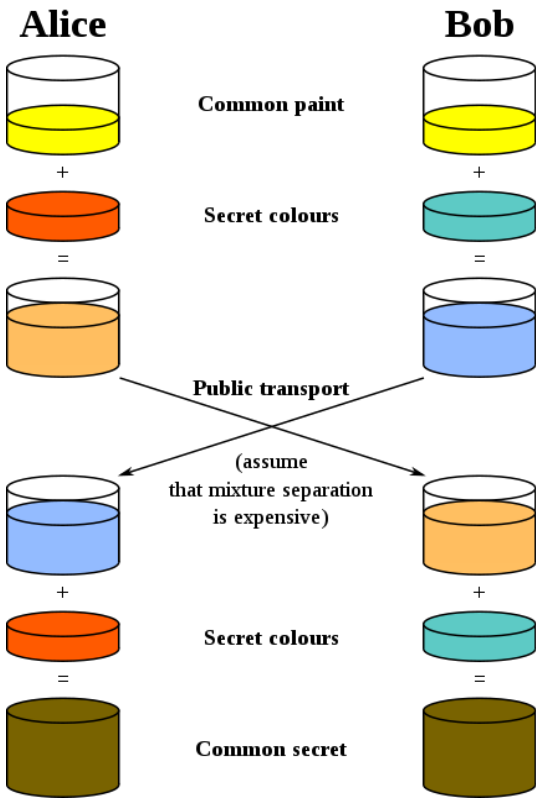
The server can then send M_s to the client. Here, the client also generates its own random key KC, and the client message $M_c = (g^{KC} \bmod p)$. Similarly, if the attacker gets ahold of M_c, it is difficult to guess the value of KC. This is sent back to the server.

Now, the server has M_c and the client has M_s. They will each use this message to generate a pre-master key, a shared key between the client and the server which can be used to encrypt messages. The client generates a pre-master key $PKC = M_s^{KC} \bmod p$, and the server generates a pre-master key $PKS = M_c^{KS} \bmod p$. The key that they generate will be the same (PKC = PKS), since they both send the parameters used to one another. We can use the fact that PKC = PKS to verify that the connection is successfully encrypted later on in our model.

Here, both the client and the server now have a shared key, PKC and PKS respectively. They can now use this shared key to encrypt all outgoing messages to one another without others being able to read the message. Since the randomly generated key KC and KS never leave the client and server respectively in an unmodded form, it would be extremely hard for the attacker to guess the shared key from M_c, M_s, g and p.

In a simple analogy, one can imagine that both the client and the server choose a random color of paint each, for example, blue for the client, red for the server. They then each mix this color into a bucket of black paint, and send it to each other. Someone that steals the bucket of paint will see a mixed color, but be unable to discern the original color mixed into it. In other words, they see a mixed color, but can’t tell that either blue or red was mixed into it.

The client and the server receive the bucket of mixed paint from the other party, and add in their own secret colors (blue and red respectively) into the bucket. Now they each have the same colors in their bucket (blue, red and black), but a malicious third party only sees a bucket of mixed colors and is unable to guess that there is blue and red in the paint bucket.



Depiction of the TLS protocol as paint (credits: Wikipedia)

In order to guard against repeated attacks from an attacker, the Diffie-Helman key exchange can be modified to regenerate the random keys KS and KC after a short, random period of time. This will ensure that the shared key is renewed often, making it extremely challenging for the attacker to make a repeated attack. In this manner, perfect forward secrecy can be maintained.

TLS objectives

The two main objectives of TLS are to maintain perfect forward secrecy, as well as to establish a trusted, encrypted communication between two unauthenticated parties.

For example, we could trivially have a public key and private key for both client and the server. The server can simply send the public key to the client, which the client uses to encrypt all communications, ensuring that no one can read the messages sent to the server. However, this implementation introduces two important flaws:

1. In the case where the server does not send a public key to the client, encryption cannot complete. Cases like this occur when perhaps the server or the client wishes to remain anonymous and thus does not send a certificate or public key.
2. If the private key of the server/client is accidentally leaked or hacked, then all previous communications are immediately exposed since all communications are encrypted by the same corresponding public key.

The Diffie-Helman ephemeral method solves this through the generation of random keys on both the client and the server side. In the case of 1, both parties can simply send the modded version of the key so they both end up with the shared key without any other information required in order to communicate securely (of course, we usually want an authenticated server).

In case 2, since random keys are re-generated after a certain amount of time, a compromise of the shared key will only result in all communications for one session being compromised. In the next session, a new pair of randomly generated key will be created, resulting in perfect forward secrecy.

TLS security properties

To verify the TLS protocol, we encode the claimed security properties of TLS 1.3 as lemmas. In this project, we strive to verify the following lemmas:

1. The client and server should establish the same session key after key exchange.
2. If the client or server has established a session key with an authenticated peer, the attacker would not be able to guess the session key being used.
3. Even if the private key of the server is stolen by the attacker, the attacker would not be able to guess the session key for the next phase or iteration.

The verification of the lemmas will be covered in the Verification of TLS security principles section.

Methodology

Modelling the protocol

Simplifications and assumptions

In order to simplify the modelling of the TLS protocol in PAT, we also make a few assumptions:

1. **Simplify the encryption to be replaced by multiplication and division.**

   Usually the server sends its public key to the client so the client can encrypt messages with it before sending it to the server. This adds another layer of protection, and acts as a guarantee to the client that it is communicating with the correct server (since only the server will be able to decrypt the message).

   For this case, we simplify the encryption process to a simple multiplication and division which we will show later. We make the assumption that it is impossible (or improbable) to guess the message if it is encrypted.

2. **Modify the bounds for generating a random key**

   To ensure that the generated pre-master key fits within the PAT data type, we artificially limit the bounds of the random number key. Note that this has unintended

consequences, such as the random keys being able to be brute forced through random iteration and guessing by PAT.

```
#define randomKeyLowerBound 3;
#define randomKeyUpperBound 10;
```

Code snippet for defining bounds

3. **Skip the verification of public certificate**

   In our study, we skip the verification of the public certificate in order to simplify the TLS protocol.

Storing state

We model the state of the TLS client-server connected via various global variables, some of which are shown below:

```
#define NoOfClients 1;
#define randomKeyLowerBound 3;
#define randomKeyUpperBound 9;

// Define cipher suite (we only use DHE)
var cipherSuite = DHE;

// Currently we hard code to have 3 clients supported

// Define p and g variables for all clients
var p = [5]; // p[clientID] = p value of client with clientID, p is a prime number
var g = [3]; // g[clientID] = g value of client with clientID, g is a primitive root modulo
of p

// Define random keys of server and client
var serverRandomKey[NoOfClients]; // serverRandomKey[clientID] = server random key (KS)
corresponding to clientID
var clientRandomKey[NoOfClients]; // clientRandomKey[clientID] = client random key (KC)
corresponding to clientID

// Define premaster keys of server and client
var serverPreMasterKey[NoOfClients] = [-1]; // serverRandomKey[clientID] = server random key
(KS) corresponding to clientID
var clientPreMasterKey[NoOfClients] = [-2]; // clientRandomKey[clientID] = client random key
(KC) corresponding to clientID

// Define messages sent by server and client
var serverMessage[NoOfClients]; // serverMessage[clientID] = server message (Ms)
corresponding to clientID
var clientMessage[NoOfClients]; // clientMessage[clientID] = client message (Mc)
corresponding to clientID

// Track clients connected to the server
var clientConnectedTo[NoOfClients];
var serverConnectedTo[NoOfClients];

// Store server certs
var serverPublicKeys[NoOfClients];
```

```
// Store messages sent between client and server
var clientMessagesToSend[NoOfClients] = [24]; // Stores unencrypted messages
var clientMessagesSent[NoOfClients]; // Stores encrypted messages with premaster key
var clientMessagesReceived[NoOfClients]; // Stores decrypted messages received from server
var serverMessagesToSend[NoOfClients] = [42]; // Stores unencrypted messages
var serverMessagesSent[NoOfClients]; // Stores encrypted messages with premaster key
var serverMessagesReceived[NoOfClients]; // Stores decrypted messages received from client

// Global state for pub and priv keys (client doesn't have as it's
unauthenticated/anonymous)
//var clientPublicKey[NoOfClients];
//var clientPrivateKey[NoOfClients];
var serverPublicKey = 3003; // server only needs 1 key pair
var serverPrivateKey = 3003;

// Attacker variables (Attacker always knows P and G since they are sent unencrypted over
the network)
var attackerRandomKey[NoOfClients]; // Attacker will generate random keys if it doesn't know
any
var attackerClientMessage[NoOfClients]; // Attacker can unencrypt clientMessage hashed with
server's public key if it has server's private key
var attackerClientRandomKey[NoOfClients];
var attackerServerRandomKey[NoOfClients];
var attackerPreMasterKey[NoOfClients] = [-3];
```

In our program, we also model the physical network in which data is transmitted as a channel in PAT, in which all parties have the ability to listen to. This models the real world use case whereby clients, servers and potentially attackers can listen to what is being transmitted through the network.

```
channel network 0; // 0 buffer = synchronous
```

External libraries

We have created two custom external libraries in PAT in order to handle the encryption/decryption via public/private key, as well as a random number generator in order to stimulate the generation of the random key on both the client and the server side.

Encryption/decryption

As mentioned, we have simplified encryption and decryption to multiplication with the public key (to encrypt) and division with the private key (to decrypt) respectively.

```
public class SCrypt
  {
       public static int encrypt(int message, int publicKey)
       {
             return (int)message * publicKey;
       }

       public static int decrypt(int encryptedMessage, int privateKey)
       {
             return (int)(encryptedMessage / privateKey);
       }
  }
```

```

```

Random key generator

Our random key generator takes in a lower bound and upper bound to generate numbers for.

```
public class RandomNumber
    {
            public static int randomNumber(int lowerBound, int upperBound)
        {
            Random rd = new Random();

            int randNum = rd.Next(lowerBound, upperBound);
            return randNum;
        }
    }
```

Modelling the client

In the TLS protocol, there is a defined client and server architecture and they engage in the TLS handshake process to set up an equal encryption key. We model the client and server as two separate processes in the PAT program. PAT utilizes the CSP idea to model the interaction between 2 different processes.

Here, we model the Client process:

```
// ====================================
// Client processes
// ====================================
Client(clientID) =
       GenerateP{p[clientID] = p[clientID]} ->
       GenerateG{g[clientID] = g[clientID]} ->
       network!clientID.clientID -> // Send clientID, p and g to server
       network!clientID.p[clientID] ->
       network!clientID.g[clientID] ->
       ClientReceiveServerHello(clientID);
```

First, the client generates P and G, and sends its ID (IP), P and G to the server unencrypted.

```
// Get server certificate for verification
ClientReceiveServerHello(clientID) =
       network?clientID.cipherVal -> // Get cipherval from server
       network?clientID.serverMsg{serverMessage[clientID] = serverMsg} -> // Recieve server
msg
       network?clientID.serverCert{serverPublicKeys[clientID] = serverCert} ->
       ClientGeneratePreMaster(clientID);
```

The client then waits for the server's message M_s, whereby it is now able to generate its shared premaster key.

```
ClientGeneratePreMaster(clientID) =
       GenerateClientRandomKey{clientRandomKey[clientID] = call(randomNumber,
randomKeyLowerBound, randomKeyUpperBound)} ->
```

```
       GenerateClientPreMasterKey{clientPreMasterKey[clientID] = call(Pow,
serverMessage[clientID], clientRandomKey[clientID]) % p[clientID] } -> // Compute client
premaster key
       GenerateClientMessage{clientMessage[clientID] = call(Pow, g[clientID],
clientRandomKey[clientID]) % p[clientID] } -> // Compute client message to send to the
server
       HashClientMessage{clientMessage[clientID] = call(encrypt, clientMessage[clientID],
serverPublicKeys[clientID])} -> // Hash message with server's public key before sending
       network!clientID.clientMessage[clientID] -> // Send hashed client message to server
       ClientConnected(clientID);
```

The client then proceeds to generator its own random key and the shared premaster key. It generates the client message M_c and encrypts it with the server's public key before sending it to the server.

```
// Do a SYN ACK
ClientConnected(clientID) =
       network?clientID.msg1{clientMessagesReceived[clientID] = call(decrypt, msg1,
clientPreMasterKey[clientID])} -> // Receive encrypted server message encrypted with
premasterkey
       EncryptClientMessage{clientMessagesSent[clientID] = call(encrypt,
clientMessagesToSend[clientID], clientPreMasterKey[clientID])} -> // Encrypt reply using
premasterkey and send to server
       network!clientID.clientMessagesSent[clientID] -> // Send client encrypted message to
server
       Skip();
```

Finally, we simulate the client sending and receiving a message encrypted using the shared premaster key to and from the server.

Modelling the server

We also model the Server process, which listens to the client and establishes a connection.

```
// ====================================
// Server processes
// ====================================
Server() =
       network?client.clientID -> // Receive clientID, p and g from client
       network?client.P ->
       network?client.G ->
       GenerateServerRandomKey{serverRandomKey[clientID] = call(randomNumber,
randomKeyLowerBound, randomKeyUpperBound) } ->
       GenerateServerMessage{serverMessage[clientID] = call(Pow, G,
serverRandomKey[clientID]) % P} ->
       ServerHello(client, clientID);

// Send server hello back to client
ServerHello(client, clientID) =
       network!client.cipherSuite -> // Send cipher suite chosen (DHE)
       network!client.serverMessage[clientID] -> // Send server message to client
       ServerSendCert(client, clientID);
```

The server first receives P and G from the client, and generates the server random key. It then forms the server message M_s and sends it unencrypted to the client.

```
// Send server pub key to verify itself to the client
ServerSendCert(client, clientID) =
	network!client.serverPublicKey -> // Send server cert over
	ServerGeneratePreMaster(client, clientID);
```

The server also sends its certificate and public key to the client to encrypt future transmissions and to secure its identity.

```
// Generate server's premaster key
ServerGeneratePreMaster(client, clientID) =
	network?client.clientMessageReceived ->
	UnhashClientMessage{clientMessage[clientID] = call(decrypt, clientMessageReceived,
serverPrivateKey)} -> // Read hashed client's message from client
	GenerateServerPreMasterKey{serverPreMasterKey[clientID] = call(Pow,
clientMessage[clientID], serverRandomKey[clientID]) % p[clientID]} ->
	EncryptServerMessage{serverMessagesSent[clientID] = call(encrypt,
serverMessagesToSend[clientID], serverPreMasterKey[clientID])} -> // Encrypt message using
premasterkey and send to client
	network!client.serverMessagesSent[clientID] -> // Send server encrypted message to
client
	ServerConnected(client, clientID);
```

The server then waits for the encrypted client's message M_c and uses it to generate its own shared premaster key after decrypting it with its private key. We then simulate the server sending an encrypted message using the shared premaster key to the client and receiving a similar message back.

```
// Update server connection and send message
ServerConnected(client, clientID) =
	network?client.msg2{serverMessagesReceived[clientID] = call(decrypt, msg2,
serverPreMasterKey[clientID])} -> // Receive encrypted server message encrypted with
premasterkey
	ServerConnectedTo{serverConnectedTo[clientID] = 1} ->
	Skip();
```

Finally, we log the server connection between client and server to continue communication, simulating a keep alive connection in a HTTPS transmission.

Modelling the attacking model

Similar to the client and server, we model the attacker as a separate process who is able to listen into the network (channel in PAT). In our project, we consider five different scenarios in which the attacker has various different abilities:

1. **Attacker doesn't know any random or private keys but can snoop on the network connection.**

```
// - Attacker will know P, G, ClientMessage (Hashed with server's public key) and
ServerMessage
// EXPECT ATTACK TO FAIL (But will succeed since we brute force)
AttackerNetworkSnoopOnly(clientID) =
	GenerateAttackerRandomKey{attackerRandomKey[clientID] = call(randomNumber,
randomKeyLowerBound, randomKeyUpperBound)} -> // Attacker has to brute force random key. We
assume this is improbable so give some random number.
```

```
       // Attacker can't get clientMessage as it is hashed with server's public key. It can
only get serverMessage unencrypted.
       GenerateAttackerPreMasterKey{attackerPreMasterKey[clientID] = call(Pow,
serverMessage[clientID], attackerRandomKey[clientID]) % p[clientID]} -> // Attacker tries to
generate same premasterkey as client
       Skip;
```

Here, the attacker knows P, G and can sniff the client and server message. However, since it doesn't know the random key, it will be unable to decrypt the messages.

2. **Attacker knows the server's private keys only AND can snoop on the network connection. Attacker doesn't know the random keys.**

```
// - Attacker will know P, G, ClientMessage (Can be decrypted), ServerMessage and
ServerPrivateKey
// EXPECT ATTACK TO FAIL (But will succeed since we brute force)
AttackerKnowsServerPrivateKey(clientID) =
       GenerateAttackerRandomKey{attackerRandomKey[clientID] = call(randomNumber,
randomKeyLowerBound, randomKeyUpperBound)} -> // Attacker has to brute force random key. We
assume this is improbable so give some random number.
       // Attacker can get both ClientMessage (after decryption) and ServerMessage. Here, we
try using ClientMessage since we did ServerMessage earlier.
       DecryptClientMessage{attackerClientMessage[clientID] = call(decrypt,
clientMessage[clientID], serverPrivateKey)} -> // Receive the hashed clientMessage and
decrypt using server's private key
       GenerateAttackerPreMasterKey{attackerPreMasterKey[clientID] = call(Pow,
attackerClientMessage[clientID], attackerRandomKey[clientID]) % p[clientID]} -> // Attacker
tries to generate same premasterkey as server
       Skip;
```

While the attacker knows the private keys of the server and can decrypt the client message, it does not know the random key of the server and thus is still unable to generate the shared premaster key.

3. **Attacker knows the random keys (of client only) but doesn't know server's private key, but can snoop on the network connection.**

```
AttackerKnowsClientRandomKeyOnly(clientID) =
       // Since we know the client random key, we just try using serverMessage since it's
unhashed. We directly use clientRandomKey since we know it.
       GenerateAttackerPreMasterKey{attackerPreMasterKey[clientID] = call(Pow,
serverMessage[clientID], clientRandomKey[clientID]) % p[clientID]} -> // Attacker tries to
generate same premasterkey as client
       Skip;
```

Since the attacker knows the random key of the client and the server message is unencrypted, it can simply generate the shared premaster key for this round and the attack succeeds.

4. **Attacker knows the random keys (of server only) but doesn't know server's private key, but can snoop on the network connection.**

```
// - Attacker will know P, G, ClientMessage (Hashed with server's public key), ServerMessage
and ServerRandomKey
// EXPECT ATTACK TO FAIL (But will succeed since we brute force)
```

```
AttackerKnowsServerRandomKeyOnly(clientID) =
       // We can't do anything even when knowing server's random key since we are unable to
get ClientMessage to generate the PreMasterKey (ClientMessage is hashed with
ServerPrivateKey).
       // Therefore, we have to do the same thing as when we know nothing - generate random
key and hope to get lucky with the unhashed ServerMessage.
       GenerateAttackerRandomKey{attackerRandomKey[clientID] = call(randomNumber,
randomKeyLowerBound, randomKeyUpperBound)} -> // Attacker has to brute force random key. We
assume this is improbable so give some random number.
       // Attacker can't get clientMessage as it is hashed with server's public key. It can
only get serverMessage unencrypted.
       GenerateAttackerPreMasterKey{attackerPreMasterKey[clientID] = call(Pow,
serverMessage[clientID], attackerRandomKey[clientID]) % p[clientID]} -> // Attacker tries to
generate same premasterkey as client
       Skip;
```

While the attacker knows the random key of the server, the client message is encrypted and thus the attacker is unable to generate the shared premaster key without knowing the client message.

5. **Attacker knows the random keys (of both client and server), knows the server's private key, and can snoop on the network connection.**

```
// - Attacker will know P, G, ClientMessage (Can be decrypted), ServerMessage,
ClientRandomKey and ServerRandomKey
// EXPECT ATTACK TO SUCCEED
AttackerKnowsEverything(clientID) =
       // In this case, attacker is all powerful and so we can try anything. Let's try it
with ClientMessage.
       DecryptClientMessage{attackerClientMessage[clientID] = call(decrypt,
clientMessage[clientID], serverPrivateKey)} -> // Receive the hashed clientMessage and
decrypt using server's private key
       GenerateAttackerPreMasterKey{attackerPreMasterKey[clientID] = call(Pow,
attackerClientMessage[clientID], serverRandomKey[clientID]) % p[clientID]} -> // Attacker
tries to generate same premasterkey as server
       Skip;
```

Since the attacker knows the random key of the client and the server, it can simply do the same thing as scenario 3 where it generates the premaster key from the server message and the client's random key. However, since it also knows the server's private key, it can also generate the premaster key from the decrypted client's message and the server's random key, presenting two attack vectors.

Modelling perfect forward secrecy

A key part of TLS is the Diffie-Hellman Ephemeral (DHE) protocol, which regenerates the random key for the client and server after a set amount of time or messages. This in turn results in a new shared premasterkey, resulting in a bad actor having to re-guess the new random keys.

Perfect forward secrecy maintains that an attacker that knows the current shared premaster key should not know the next round's premaster key, thus enabling perfect forward secrecy. This is important so that while the messages for a particular round can be read, all other messages in the past and future rounds stay safe and encrypted.

To model this, we keep track of each process' loop state, or each round.

```
// Track the loop state (cycle)
var globalLoop = 0;
var attackerLoop = 0;
var clientLoop = 0;
var serverLoop = 0;
```

Essentially, we want to make sure that the premasterkey that the attacker holds is only for one round, and will not be kept the same after a round passes for the client and the server. We increment each loop round after a process has finished all its serial executions.

Naturally, if a loop round has been incremented by another process, then this process will not increment it further. For example, if the client process has already advanced the loop from 0 to 1, then the server process will not advance it from 1 to 2 when it finishes its first serial execution. Instead, it will restart its loop in round 1 to match the client process.

```
CheckClientLoop(loop, clientID) =
       if (loop >= maxLoops) { Skip }
       else {UpdateClientLoop(loop, clientID)};

UpdateClientLoop(loop, clientID) =
       if (loop != globalLoop) {Client(globalLoop, clientID)}
       else {UpdateLoop{globalLoop++} -> Client(globalLoop, clientID)};
```

We will look into verifying perfect forward secrecy later on.

Simulating processes

We generate various simulations to model the client, attacker and server in parallel.

```
// ====================================
// Simulations
// ====================================
// 0. Normal TLS process without any attackers (i.e. client and server only)
TLS() = (||| clientID:{0..NoOfClients-1} @ Client(clientID)) ||| Server();

// 1. Attacker doesn't know any random or private keys but can snoop on the network
connection
AttackerNetworkSnoopOnly1() = (||| clientID:{0..NoOfClients-1} @ Client(clientID)) |||
Server() ||| (||| clientID:{0..NoOfClients-1} @ AttackerNetworkSnoopOnly(clientID));

// 2. Attacker knows the server's private keys only AND can snoop on the network connection.
Attacker doesn't know the random keys.
AttackerKnowsServerPrivateKey2() = (||| clientID:{0..NoOfClients-1} @ Client(clientID)) |||
Server() ||| (||| clientID:{0..NoOfClients-1} @ AttackerKnowsServerPrivateKey(clientID));

// 3. Attacker knows the random keys (of client only) but doesn't know server's private key,
but can snoop on the network connection
AttackerKnowsClientRandomKeyOnly3() = (||| clientID:{0..NoOfClients-1} @ Client(clientID))
||| Server() ||| (||| clientID:{0..NoOfClients-1} @
AttackerKnowsClientRandomKeyOnly(clientID));

// 4. Attacker knows the random keys (of server only) but doesn't know server's private key,
but can snoop on the network connection
```

```
AttackerKnowsServerRandomKeyOnly4() = (||| clientID:{0..NoOfClients-1} @ Client(clientID))
||| Server() ||| (||| clientID:{0..NoOfClients-1} @
AttackerKnowsServerRandomKeyOnly(clientID));

// 5. Attacker knows the random keys (of both client and server), knows the server's private
key, and can snoop on the network connection.
AttackerKnowsEverything5() = (||| clientID:{0..NoOfClients-1} @ Client(clientID)) |||
Server() ||| (||| clientID:{0..NoOfClients-1} @ AttackerKnowsEverything(clientID));
```

Using the PAT software, we are able to consider the different execution sequences of multiple processes allowing us to verify the following TLS security properties over many various scenarios and possible iterations via depth-first search or breadth-first search. A possible trace of a Client-Server connection generated by PAT is depicted below:

Verification goals

After simulating the various processes running in parallel, we have to have some verifications in order to check the accuracy and security of the program (in this case, TLS). PAT works by having goals and assertions.

A goal is simply a state in a trace that we wish to achieve, such as being deadlock free, or if the generated client premaster key is the same as the server generated premaster key to check that the algorithm works correctly.

Our various goals to verify are as follows:

1. Same premaster key generated on client and server
2. Message sent by client to server, encrypted using the shared premasterkey is correctly encrypted and decrypted and vice-versa
3. Whether the attacker is able to get the same premaster key as the client or the server

4. Check that the attacker is able to get the same premaster key as the client or the server AND the attacker is in a different loop cycle as the client or the server, meaning that perfect forward secrecy is broken (Since the attacker is able to get the preamster key of another round)

```
// ====================================
// Goals
// ====================================
// Normal TLS process goals
#define sameClientServerKeyGoal serverPreMasterKey[0] == clientPreMasterKey[0];
#define correctClientMessageReceivedByServerGoal serverMessagesReceived[0] ==
clientMessagesToSend[0];
#define correctServerMessageReceivedByClientGoal clientMessagesReceived[0] ==
serverMessagesToSend[0];

// Attack goals (as long as one succeeds, attack is a success)
#define attackerSamePreMasterKeyAsServerClient attackerPreMasterKey[0] ==
serverPreMasterKey[0] && sameClientServerKeyGoal;

// Check for perfect forward secrecy
#define sameLoop serverLoop == clientLoop == attackerLoop; // Ensure all are on the same
loop
#define perfectForwardSecrecyBroken attackerSamePreMasterKeyAsServerClient && attackerLoop <
clientLoop && attackerLoop < serverLoop;
```

Assertions

Now that we have our goals, we can then assert them in PAT to link the processes running in parallel to a goal. Effectively, this means that we are check whether a process running in parallel, in this case, TLS() reaches any of the goals.

```
// ====================================
// Verifications
// ====================================

// 0. Normal TLS process without any attackers (i.e. client and server only)
#assert TLS() deadlockfree; // Check for deadlock free
#assert TLS() reaches sameClientServerKeyGoal; // Check that generated premaster keys are
the same
#assert TLS() reaches correctClientMessageReceivedByServerGoal; // Check that server
receives client message properly (encrypted decrypted using premaster key)
#assert TLS() reaches correctServerMessageReceivedByClientGoal; // Check that client
receives server message properly (encrypted decrypted using premaster key)
#assert TLS() reaches perfectForwardSecrecyBroken; // Check for perfect forward secrecy

// 2. Attacker knows the server's private keys only AND can snoop on the network connection.
Attacker doesn't know the random keys.
#assert AttackerKnowsServerPrivateKey2() deadlockfree; // Check for deadlock free
#assert AttackerKnowsServerPrivateKey2() reaches sameClientServerKeyGoal; // Check that
generated premaster keys are the same
#assert AttackerKnowsServerPrivateKey2() reaches correctClientMessageReceivedByServerGoal;
// Check that server receives client message properly (encrypted decrypted using premaster
key)
#assert AttackerKnowsServerPrivateKey2() reaches correctServerMessageReceivedByClientGoal;
// Check that client receives server message properly (encrypted decrypted using premaster
key)
```

```
#assert AttackerKnowsServerPrivateKey2() reaches attackerSamePreMasterKeyAsServerClient; //
Check if attacker manages to generate same preMasterKey as server and client
#assert AttackerKnowsServerPrivateKey2() reaches perfectForwardSecrecyBroken; // Check for
perfect forward secrecy

// 3. Attacker knows the random keys (of client only) but doesn't know server's private key,
but can snoop on the network connection
#assert AttackerKnowsClientRandomKeyOnly3() deadlockfree; // Check for deadlock free
#assert AttackerKnowsClientRandomKeyOnly3() reaches sameClientServerKeyGoal; // Check that
generated premaster keys are the same
#assert AttackerKnowsClientRandomKeyOnly3() reaches
correctClientMessageReceivedByServerGoal; // Check that server receives client message
properly (encrypted decrypted using premaster key)
#assert AttackerKnowsClientRandomKeyOnly3() reaches
correctServerMessageReceivedByClientGoal; // Check that client receives server message
properly (encrypted decrypted using premaster key)
#assert AttackerKnowsClientRandomKeyOnly3() reaches attackerSamePreMasterKeyAsServerClient;
// Check if attacker manages to generate same preMasterKey as server and client
#assert AttackerKnowsClientRandomKeyOnly3() reaches perfectForwardSecrecyBroken; // Check
for perfect forward secrecy

// 4. Attacker knows the random keys (of server only) but doesn't know server's private key,
but can snoop on the network connection
#assert AttackerKnowsServerRandomKeyOnly4() deadlockfree; // Check for deadlock free
#assert AttackerKnowsServerRandomKeyOnly4() reaches sameClientServerKeyGoal; // Check that
generated premaster keys are the same
#assert AttackerKnowsServerRandomKeyOnly4() reaches
correctClientMessageReceivedByServerGoal; // Check that server receives client message
properly (encrypted decrypted using premaster key)
#assert AttackerKnowsServerRandomKeyOnly4() reaches
correctServerMessageReceivedByClientGoal; // Check that client receives server message
properly (encrypted decrypted using premaster key)
#assert AttackerKnowsServerRandomKeyOnly4() reaches attackerSamePreMasterKeyAsServerClient;
// Check if attacker manages to generate same preMasterKey as server and client
#assert AttackerKnowsServerRandomKeyOnly4() reaches perfectForwardSecrecyBroken; // Check
for perfect forward secrecy

// 5. Attacker knows the random keys (of both client and server), knows the server's private
key, and can snoop on the network connection.
#assert AttackerKnowsEverything5() deadlockfree; // Check for deadlock free
#assert AttackerKnowsEverything5() reaches sameClientServerKeyGoal; // Check that generated
premaster keys are the same
#assert AttackerKnowsEverything5() reaches correctClientMessageReceivedByServerGoal; //
Check that server receives client message properly (encrypted decrypted using premaster key)
#assert AttackerKnowsEverything5() reaches correctServerMessageReceivedByClientGoal; //
Check that client receives server message properly (encrypted decrypted using premaster key)
#assert AttackerKnowsEverything5() reaches attackerSamePreMasterKeyAsServerClient; // Check
if attacker manages to generate same preMasterKey as server and client
#assert AttackerKnowsEverything5() reaches perfectForwardSecrecyBroken; // Check for perfect
forward secrecy
```

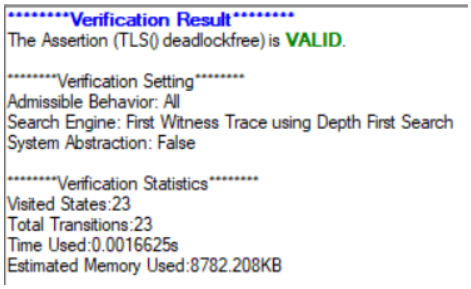
We will run these verification assertions for each of our attacker processes.

Results & Evaluation

Verification results

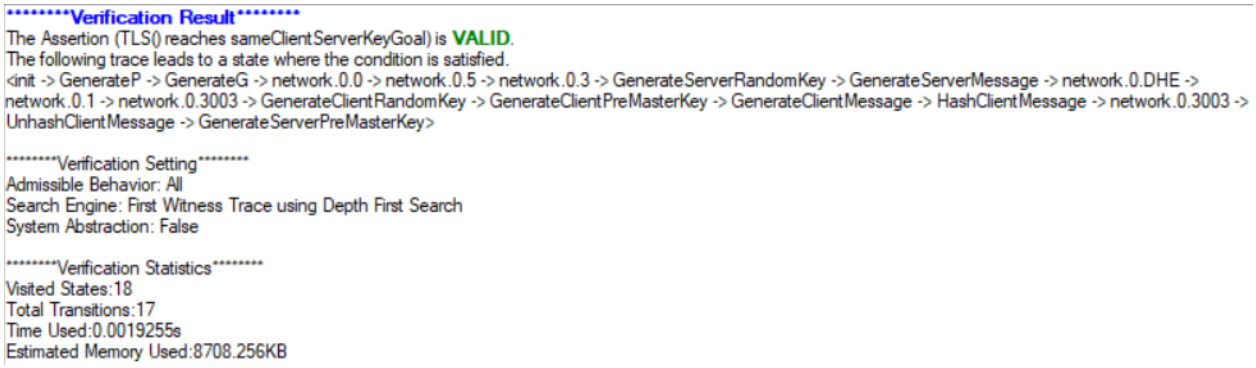
Process terminates with no deadlocks

One of the requirements for the TLS protocol is that it should be deadlock free. This means that regardless of the sequence order of the processes running, the process should eventually terminate.



Same session key after key exchange

Another requirement that we check for is that in a typical setting of the TLS protocol, the client and server should obtain the same preMasterKey value when the TLS process is completed.



From these assertions, we can conclude that our program has been implemented correctly as it is both deadlock free and the client and server are able to arrive at the same premaster key.

Attacker 1-5 verifications

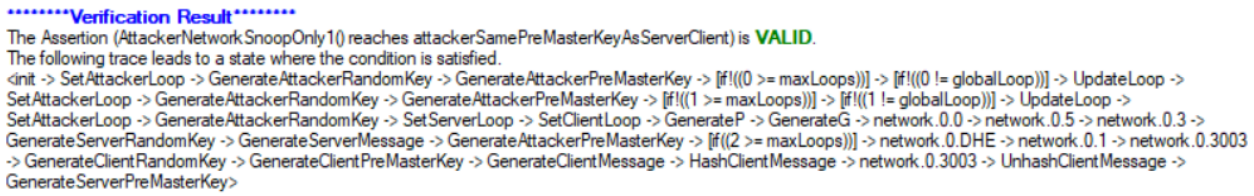
We attempt verifications for the five attacker process we mentioned earlier. Each attacker's individual verifications look like such below:

```
#assert AttackerNetworkSnoopOnly1() deadlockfree; // Check for deadlock free
#assert AttackerNetworkSnoopOnly1() reaches sameClientServerKeyGoal; // Check that generated
premaster keys are the same
#assert AttackerNetworkSnoopOnly1() reaches correctClientMessageReceivedByServerGoal; //
Check that server receives client message properly (encrypted decrypted using premaster key)
#assert AttackerNetworkSnoopOnly1() reaches correctServerMessageReceivedByClientGoal; //
Check that client receives server message properly (encrypted decrypted using premaster key)
```

```
#assert AttackerNetworkSnoopOnly1() reaches attackerSamePreMasterKeyAsServerClient; // Check
if attacker manages to generate same preMasterKey as server and client
#assert AttackerNetworkSnoopOnly1() reaches perfectForwardSecrecyBroken; // Check for
perfect forward secrecy
```

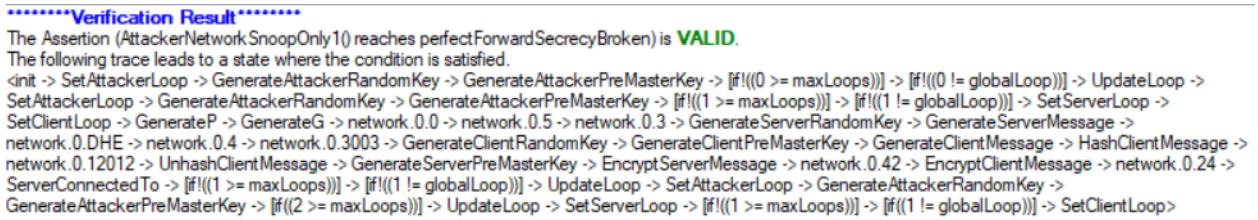
Our results for all attackers were similar, due in no small part to the simplicity of the generated random key being constrained to a small lower and upper bound. This results in PAT being able to brute force by constantly generating random keys, such that all attackers are ultimately able to succeed.

Attacker reaches same premaster key as Client/Server



Attacker cannot guess next session key in spite of knowing current premaster key

Similarly, we consider perfect forward secrecy for verification:



Accordingly, perfect secrecy was also broken by all attackers since it is trivial to brute force the random key since we limited it to a small range (e.g. 3-10).

Evaluation of results

In theory, our program works as expected and replicates the state and processes of a TLS exchange with Diffie Hellman Ephemeral protocol. However, due to oversimplified assumptions about the generation of the random keys, this results in the attacker being able to guess the random key each time, causing us to be unable to correctly ascertain if the processes have perfect forward secrecy and prevents the attacker from guessing the premaster key.

Limitations & Improvements

As mentioned, our assumptions have caused this experiment to generate results that are unable to verify our initial goals. The small range of random numbers used to generate the random key for the client and the server results in it being easy to brute force, and PAT finds a possible trace upon trying random numbers randomly true depth or breadth first search.

A possible improvement would be to implement probabilistic model checking in PAT instead, which will allow us to simulate the probability of the attacker being able to guess the random key, rather than an absolute value.

From there, we can use the percentage to make certain deductions on the feasibility of the model in a real world scenario - in which attacks are not impossible, but improbable.

Conclusion

To conclude, through PAT modelling, we are able to verify that the TLS protocol together with the Diffie-Helman ephemeral key exchange is theoretically able to achieve perfect forward secrecy as well as having no deadlocks. We are also able to verify that the shared key is successfully generated and is the same for the client and the server.

However, we were unable to verify if the attacker is unable to compromise the next session even if the attacker has the server's private key.

TLS itself is not an extremely complex system, but we can already notice many various traces and ways in which an attacker can exploit the system, or ways in which the system may fail to achieve its original purpose or specifications.

As such, PAT is a great tool to verify the integrity of systems, and almost imperative for more complex ones, in order to ensure the safety of complex, perhaps asynchronous systems.

Appendix

1. Cas Cremers, Marko Horvat, Sam Scott, and Thyla van der Merwe. 2016. Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication. In Security and Privacy (SP), 2016 IEEE Symposium on. IEEE