NUS
National University of Singapore

Formal Verification of TLS 1.3 (Key Exchange) Protocol

CS3211 Parallel & Concurrent Programming
AY20/21 Semester 2

Team Moo
Sai Ganesh Suresh (A0184103R)
Jefferson Chu (A0182590B)
Mohamed Riyas (A0194608W)

Table of Contents

Background

Transport Layer Security, TLS is a protocol designed to build a secure communication channel between the Client and the Server. The communication is secure because symmetric cryptography is used to encrypt the data transmitted. The keys are uniquely generated for each connection and are based on a shared secret, negotiated at the beginning of the session, also known as a TLS handshake.

Overall, the Client and Server shall use an asymmetric encryption technique (e.g., RSA and DH key exchange protocol) to create a shared secret during each session. With the shared secret, both peers use a symmetric encryption technique (e.g., AES) to build the encrypted communication channel. In this project, our group modeled a key exchange protocol based on the Diffie-Hellman Ephemeral key exchange protocol.

Diffie-Hellman Key Exchange

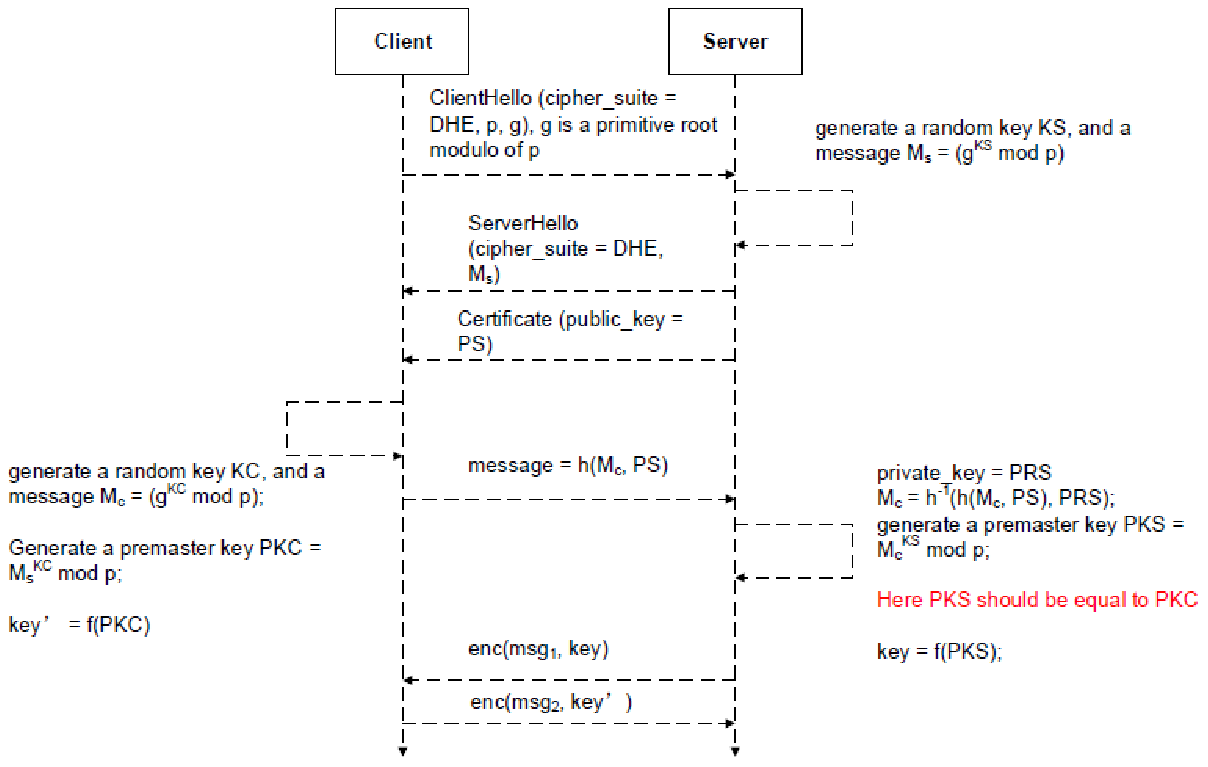

Client
Server
ClientHello (cipher_suite = DHE, p, g), g is a primitive root modulo of p
generate a random key KS, and a message $M_s = (g^{KS} \bmod p)$
ServerHello (cipher_suite = DHE, M_s)
Certificate (public_key = PS)
generate a random key KC, and a message $M_c = (g^{KC} \bmod p)$;
Generate a premaster key PKC = $M_s^{KC} \bmod p$;
key' = f(PKC)
message = h(M_c, PS)
private_key = PRS
$M_c = h^{-1}(h(M_c, PS), PRS)$;
generate a premaster key PKS = $M_c^{KS} \bmod p$;
Here PKS should be equal to PKC
key = f(PKS);
enc(msg_1, key)
enc(msg_2, key')

Figure 1. TLS Handshake with DHE Key Exchange

Figure 1 above shows a pictorial view of the DHE key exchange between the Client and the Server. The various stages that make up the DHE key exchange are described below:

1. The Client initiates the TLS connection with a ClientHello message, indicating possible options for the cipher suites. In this course project, there is only one, stipulated way of key exchange: DHE. Moreover, DHE has two parameters, a primitive p and a base g, where g is a primitive root modulo of p.

2. Then, the Server generates a random key KS and use it to derive a message M_s with the formula shown in Figure 1. Then, it transmits a Server Hello message back to the Client, informing the Client that (1) it agrees to use DHE as a key exchange method, and (2) it can derive the message M_s from the given parameters. In addition, the Server will send its public key for encrypting all ensuing communication.

3. After the Client receives the message, it performs the same procedure, i.e., generate a random key KC and a message M_c. Moreover, it generates a pre-master key $PKC = M_s^{kc} mod\, p$. Then, it encrypts M_c with the public key PS back to the Server.

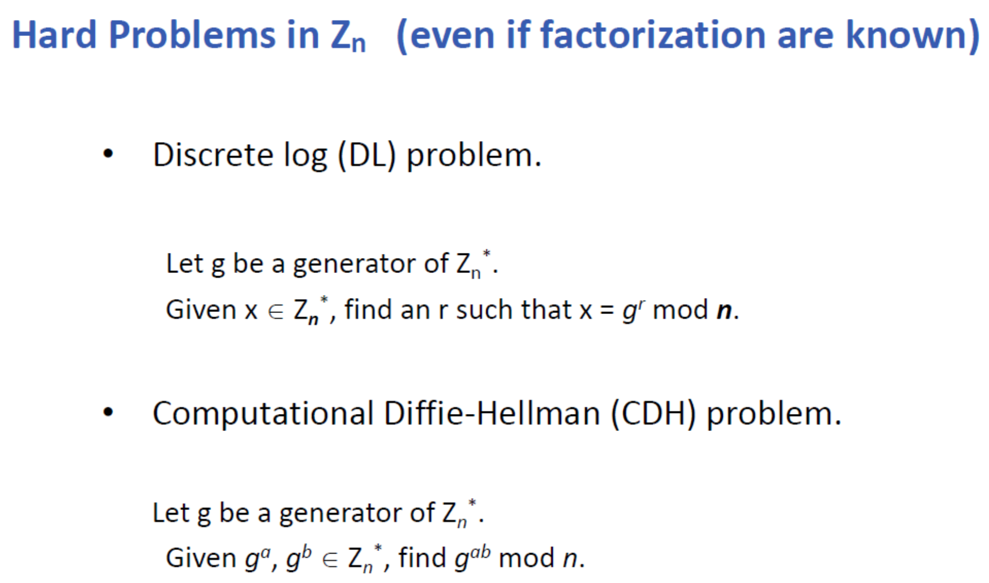
4. After the Server receives the encrypted message, it decrypts it into M_c with its private key PRS. Then, it generate a pre-master key $PKS = M_c^{ks} mod\, p$.*

   * Note here: Given the primitive p and its primitive root modulo g, regardless of the value of the random key KS and KC, the PKS in the Server should be of the same value as the PKC of the Client. By this means, the Client and the Server reach a shared secret even while the the attacker who could have observed all the transmitted values, e.g., p, g, M_s, etc cannot get the same shared secret.

5. Finally, using the same transferring function f, the Client and the Server can generate the same session key (i.e., key in the Server and key' in the Client, while key is equal to key') to enable symmetric encryption during communication.

   The above procedure is called the DH key exchange. However, in order to avoid the attacker from successfully applying the replay attack, i.e., record everything during the communication and spend more time on guessing the random key KS and KC, the Client and Server will regenerate their random keys after a short period, which makes the key exchange a DHE (Diffie-Hellman Ephemeral).

Why does DHE work?



Hard Problems in Z_n (even if factorization are known)

- Discrete log (DL) problem.

  Let g be a generator of Z_n^*.
  Given $x \in Z_n^*$, find an r such that $x = g^r \bmod \boldsymbol{n}$.

- Computational Diffie-Hellman (CDH) problem.

  Let g be a generator of Z_n^*.
  Given $g^a, g^b \in Z_n^*$, find $g^{ab} \bmod n$.

Figure 2. Discrete log and the computational Diffie-Hellman problem

As shown in Figure 2 above, DHE works because of the Discrete log problem and the Computational Diffie-Hellman problem. As yet, there does not exist a polynomial algorithm to resolve the two problems and they are believed to be NP-hard (non-deterministic polynomial-time hardness). Therefore, even if the attacker is able to eavesdrop the entire DHE key exchange protocol, the attacker will not be able to derive the pre-master and session keys.

CSP modelling goals

Through the CSP modelling of this problem, we hope to achieve the following goals and prove the security standard of the current TLS protocol.

1. Client and Server should establish the same session key after key exchange.
2. If the Client or Server believes that it has established a session key with an authenticated peer, then the attacker does not know the session key when it is being used (in-session).
3. Even if the private key of the Server is stolen, the attacker cannot compromise the communication in the next phase. This is also known as Perfect Forward Secrecy (PFS).

CSP model

Our group chose to use the CSP model in PAT to model our project due to the nature of the TLS protocol. Although there is a time element with respect to the ephemeral portion of DHE (which regenerates random keys), we opted not to choose other models in PAT as it may result in a state explosion. Furthermore, we can easily mimic the ephemeral property in our model by simply calling the DH key exchange process again.

Our overall CSP model is defined by having all main processes run interleaved shown in Figure 3 below, since each main process should be running their events independently and synchronization is achieved throughout the network.

```
DHE() = ServerProcess() ||| ClientProcess() ||| AttackerM1Process() ||| AttackerM2Process() |||
AttackerM3Process() ||| ResetConnection();
```

Figure 3. Overall System Implementation

In the below sections, we will explain more about the respective processes, starting with the Client and Server process.

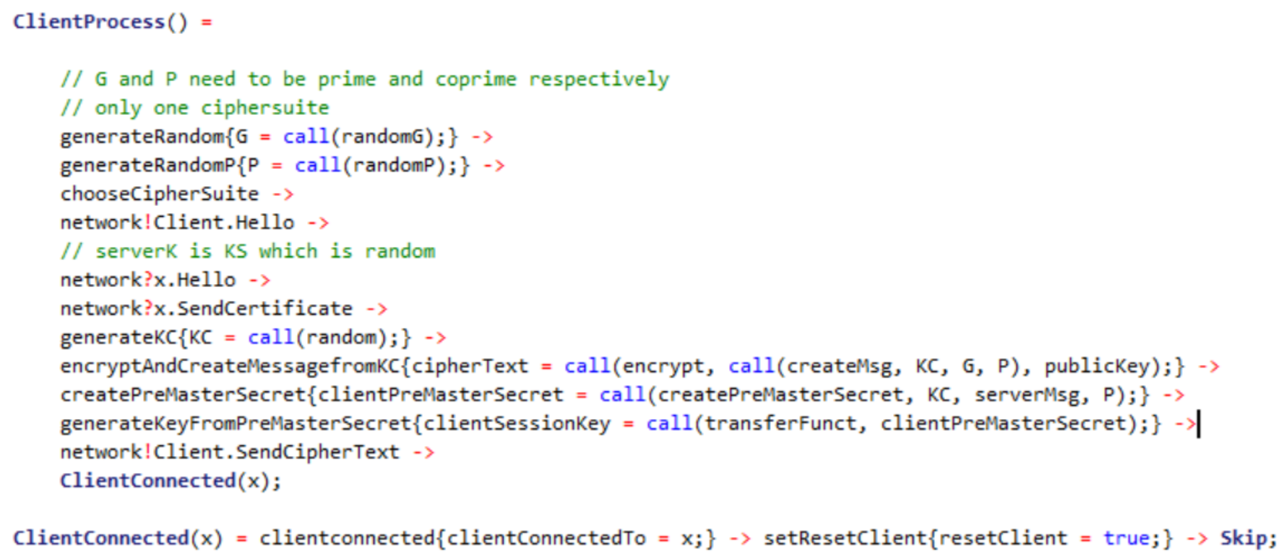
Client & Server model

We first begin by modelling the Client and Server in this key exchange protocol. We assume that the Client and Server communicate through a public channel known as the network.

Client process

The stages that make up the Client process are shown below:

1. Generate random g
2. Generate random p
3. Choose cipher suite
4. Send Client Hello through the network
5. Waits for Hello reply from the network
6. Waits for certificate from the network
7. Generate KC
8. Generate message from KC and encrypt with public key and send through the network
9. Generate pre-master secret and session key using Hello in step 5 and KC
10. Establish connection



```
ClientProcess() =

    // G and P need to be prime and coprime respectively
    // only one ciphersuite
    generateRandom{G = call(randomG);} ->
    generateRandomP{P = call(randomP);} ->
    chooseCipherSuite ->
    network!Client.Hello ->
    // serverK is KS which is random
    network?x.Hello ->
    network?x.SendCertificate ->
    generateKC{KC = call(random);} ->
    encryptAndCreateMessagefromKC{cipherText = call(encrypt, call(createMsg, KC, G, P), publicKey);} ->
    createPreMasterSecret{clientPreMasterSecret = call(createPreMasterSecret, KC, serverMsg, P);} ->
    generateKeyFromPreMasterSecret{clientSessionKey = call(transferFunct, clientPreMasterSecret);} ->
    network!Client.SendCipherText ->
    ClientConnected(x);

ClientConnected(x) = clientconnected{clientConnectedTo = x;} -> setResetClient{resetClient = true;} -> Skip;
```

Figure 4. Implementation of the Client Process

Figure 4 above shows our code implementation of the Client process. Now, we will explain the different functions in the code and their functionalities.

The first step in our Client process is where the Client initiates the TLS connection with a ClientHello message, indicating possible cipher suite options which in our case is simply DHE (due to the project requirements). Also, DHE has two parameters namely, a primitive p and a base g where g is the primitive root modulo of p. All of these are clearly shown on the top part of the code where we first have two processes namely, **generateRandom** and **generateRandomP** to generate the respective g and p, followed by the next process **chooseCipherSuite** to indicate the cipher suite and lastly **network!Client.Hello** which will send all these parameters to the Server as part of the hello message. Before we proceed to the next part of the code, there are two things to note here. The first is that **network** is used as the channel as mentioned earlier. Secondly, we will be using a custom PAT library (our own C# file) to generate the numbers.

Moving on, the next step involves the Server receiving the message from the Client and then sending its hello message to the Client which will consist of a derived message M_s and DHE as the key exchange method together with its own public key for encrypting all the follow up communication. We will explain about it in more detail under the Server process.

Back on the Client side, in the next step, the Client will wait for the hello message and the public key from the Server side. This is reflected by the **network?x.Hello** and **network?x.SendCertificate** processes. Once it receives these, it will conduct the same procedure as the Server. First, it will generate a random key, KC reflected by the **generateKC** process. It will also generate the message M_C using the given formula as reflected by **encryptAndCreateMessageFromKC** process. By using all these parameters, it will generate a pre-master key PKC using the given formula as reflected by the **createPreMasterSecret** process.

The Client will then also generate the key from the premaster secret under the **generateKeyFromPreMasterSecret** process.

This step concludes with the Client encrypting M_C with the public key PS back to the Server. This is reflected by the **network!Client.SendCipherText** process. On the Server side, we will be performing a similar task. By using the same transferring function f, the Client and the Server will have the same session key.

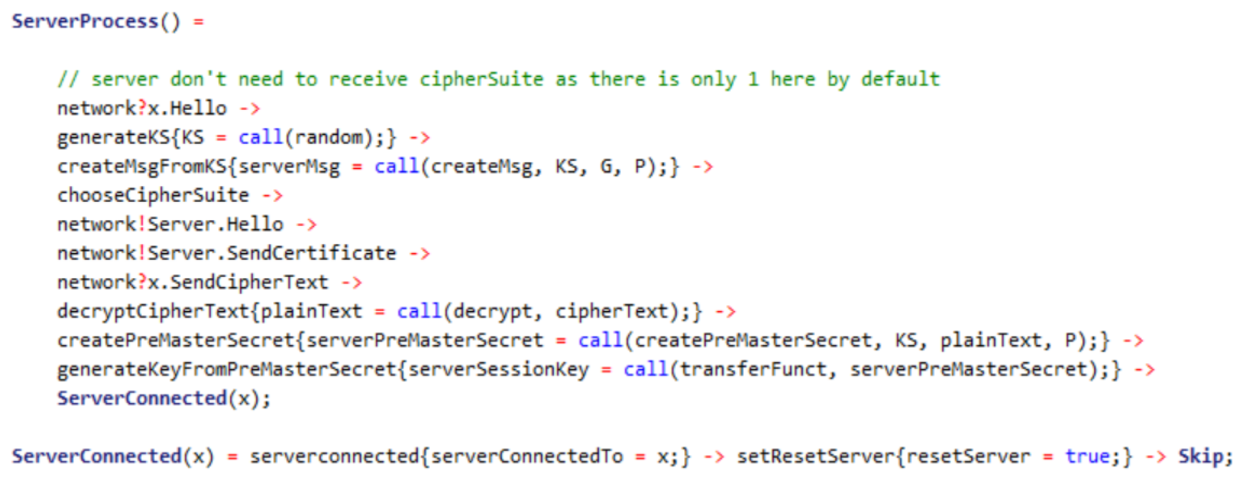
Lastly, to establish connection with the Server, we call another separate subprocess **ClientConnected**. This subprocess will simply define the identity of which the Client is connected to, which should be the Server. This subprocess will also help in preventing periodic replay attacks. This sums up the Client side process.

As mentioned earlier, in order to replicate the creation of a final session key, we implemented the generation of KC, g, p etc by creating a custom PAT library and invoking function calls which we will be explained in detail later.

Server process

The various stages that make up the server process are as follows:

1. Waits for Hello message from the network
2. Generate random KS and generate message based on KS
3. Send Server Hello (contains message) through the network
4. Send Server certificate though the network
5. Waits for ciphertext from the network
6. Decrypts ciphertext using private key of certificate
7. Generate premaster secret and session key using plaintext from step 6 and KS
8. Established connection



```
ServerProcess() =

    // server don't need to receive cipherSuite as there is only 1 here by default
    network?x.Hello ->
    generateKS{KS = call(random);} ->
    createMsgFromKS{serverMsg = call(createMsg, KS, G, P);} ->
    chooseCipherSuite ->
    network!Server.Hello ->
    network!Server.SendCertificate ->
    network?x.SendCipherText ->
    decryptCipherText{plainText = call(decrypt, cipherText);} ->
    createPreMasterSecret{serverPreMasterSecret = call(createPreMasterSecret, KS, plainText, P);} ->
    generateKeyFromPreMasterSecret{serverSessionKey = call(transferFunct, serverPreMasterSecret);} ->
    ServerConnected(x);

ServerConnected(x) = serverconnected{serverConnectedTo = x;} -> setResetServer{resetServer = true;} -> Skip;
```

Figure 5. Implementation of the Server Process

Figure 5 above shows our code implementation of the Server process. Now, we will explain the different functions in the code and their functionalities.

On the Server side, the Server will be first waiting for the Client hello message. This is reflected by the **network?x.Hello** process. Upon receiving it, the Server will generate a random key KS and use it to derive a message M_s with the given formula as reflected by **generateKS** and **createMsgFromKS** processes. Then, the Server will send a Server Hello message back to the Client, telling the Client that (1) it agrees to use DHE as the key exchange method, and (2) it can derive a message M_s from the given parameters. This is reflected by the **chooseCipherSuite** and **network!Server.Hello** processes. Moreover, the Server will send its public key for encrypting all of the follow-up communication as indicated by the **network!Server.SendCertificate** process.

Then as we saw earlier, the Client will then encrypt its message, M_C with the public key PS back to the Server. On the Server's side, after the Server receives the encrypted message as indicated by the **network?x.SendCipherText** process, it decrypts it into M_C with its private key PRS under the **decryptCipherText** process. Then, it generates a pre-master key PKS using the given formula under the **createPreMasterSecret** process. It will also generate the key from the premaster secret under the **generateKeyFromPreMasterSecret** process.

At this stage, as mentioned earlier, irrespective of the KS and KC values, the PKS in the Server will be of the same value of the PKC in the Client. The final process is **ServerConnected** which is to establish connection with the Client and define the identity of the Client the Server is connected to. This also helps to prevent any periodic replay attacks.

That concludes the Server side and in the next section, we will talk about the custom PAT library that invokes the function calls which were used in our processes.

Custom TLS Library

To supplement the aforementioned process, we created a custom library to invoke pure function calls. These function calls are not dependent on any process and produce no side effects[1] thus can be abstracted into the library without affecting the functionality of the CSP model.

We made use of the PAT functionality which allows the user to create a custom C# library for the TLS implementation.

RandomG()

Input type: *void*
Return type : *int*

[1] Side effects are functions which can modify the environment variables

This function does not take in any input and returns a number g. g will be a primitive root modulo of p that is generated from **RandomP()**. This is a necessary condition for the Diffie-Hellman Key Exchange to work.

RandomP()

Input type: *void*
Return type : *int*

This function does not take in any input and returns a number p. g that is generated from **RandomG()** will be a primitive root modulo of p. This is a necessary condition for the Diffie-Hellman Key Exchange to work.

Random()

Input type: *void*
Return type : *int*

This function does not take in any input and returns a random number.

CreateMsg()

Input type: *int, int, int*
Return type : *int*

This function takes in 3 integers representing the original message, g and p respectively. It returns the derived message given by the mathematical formula $g^x mod\, p$

Encrypt()

Input type: *int, int*
Return type : *int*
This function takes in 2 integers representing the plaintext and public key respectively. It returns the ciphertext.

Decrypt()

Input type: *int*
Return type : *int*

This function takes in the ciphertext. It returns the plaintext, decrypted using the private key. As the private key is private, only the Server is able to call this function. (the only exception is when the attacker has the capability to compromise the private key)

Adversarial model

We consider an extension of the Dolev-Yao (DY) attacker as our threat model. The DY attacker has complete control of the network, and can intercept, send, replay, and delete any message. To construct a new message, the attacker can combine any information previously learnt, e.g., decrypting messages for which it knows the key, or create its own encrypted messages. We assume perfect cryptography, which implies that the attacker cannot encrypt, decrypt or sign messages without knowledge of the appropriate keys. We additionally allow the attacker to do the following:

1. Compromise the private key of protocol participants.
2. Compromise the random keys.

This is because we should always assume Kerckchoff's principle: *"A cryptosystem should be secure even if everything about the system, except the key, is public knowledge"* where by simulating the strongest possible attacker, we can formally prove the security of DHE.

As a result, we will have to model 3 different attackers:

1. Dolev-Yao attacker
2. Dolev-Yao attacker + compromise private key
3. Dolev-Yao attacker + compromise random keys

Attacker 1

The attacker can interact with both the Server and Client through the network. As such, as shown in Figure 6 below, we split the interactions between the Client and Server into 2 subprocesses, in which they can interleave with one another just like how a man-in-the-middle-attack can be conducted in real life.

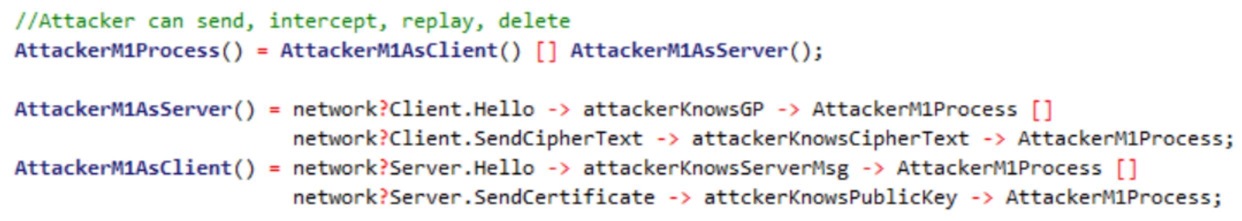

```
//Attacker can send, intercept, replay, delete
AttackerM1Process() = AttackerM1AsClient() [] AttackerM1AsServer();

AttackerM1AsServer() = network?Client.Hello -> attackerKnowsGP -> AttackerM1Process []
                       network?Client.SendCipherText -> attackerKnowsCipherText -> AttackerM1Process;
AttackerM1AsClient() = network?Server.Hello -> attackerKnowsServerMsg -> AttackerM1Process []
                       network?Server.SendCertificate -> attckerKnowsPublicKey -> AttackerM1Process;
```

Figure 6. Code Implementation that mimics Attacker 1

The subprocess is as follows:

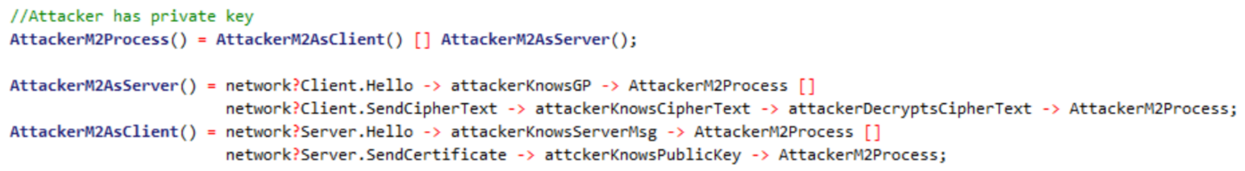
1. Attacker picks up Client hello
2. Attacker will know g and p as it is part of the Client hello message and not encrypted.
3. Attacker can choose to wait and eavesdrop for more information from either the Client or Server
4. Attacker picks up ciphertext from Client

Similarly, the attacker can also pick up Server hello and certificate by eavesdropping from the Server. We make use of external choice over here to account for the possibility in which the attacker can actually eavesdrop on all pieces of information which is very practical in real life.

However, we need to recall that the attacker needs to know the random keys of the **current session** in order to consider an attack successful. Otherwise, because of the property of DHE, just from this information alone, the attacker will not be able to extract the random keys. This is verified when we run the CSP model on PAT and check the corresponding assertions of the TLS model.

Attacker 2

To put it simply, attacker 2 is attacker 1 with the additional capability of compromising the private key. The private key here refers to the private key of the RSA encryption which the Server possesses. This means that the attacker is able to decrypt all ciphertext encrypted using the public key.



```
//Attacker has private key
AttackerM2Process() = AttackerM2AsClient() [] AttackerM2AsServer();

AttackerM2AsServer() = network?Client.Hello -> attackerKnowsGP -> AttackerM2Process []
                       network?Client.SendCipherText -> attackerKnowsCipherText -> attackerDecryptsCipherText -> AttackerM2Process;
AttackerM2AsClient() = network?Server.Hello -> attackerKnowsServerMsg -> AttackerM2Process []
                       network?Server.SendCertificate -> attckerKnowsPublicKey -> AttackerM2Process;
```

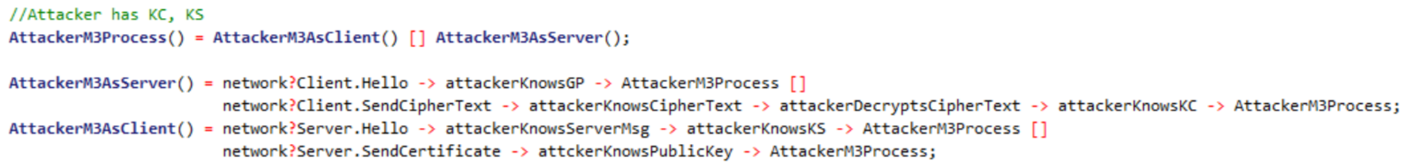
Figure 7. Code Implementation that mimics Attacker 2

Thus, as shown in Figure 7 above, the modelling of the process is extremely similar to attacker 1 apart from an extra step in which the attacker decrypts the ciphertext. As throughout the TLS handshake, there is only 1 message that is encrypted using the public key, the only piece of extra information gain by the attacker will simply be $g^{kc} mod\, p$ which corresponds to the plaintext of the ciphertext.

Again, PAT verifies that the attacker is still not able to derive the session keys and hence the TLS protocol is not susceptible to the attacker.

Attacker 3

Attacker 3 has the ability to compromise the random keys, however, the random keys would be of the previous session. As if the current session random keys are compromised, the attacker would be able to obtain the session keys and the attacker is capable of performing any attack. Similar to the previous attackers, Attacker 3 has an additional capability of gaining access of KC and KS which are the random keys respectively, as shown in Figure 8.



```
//Attacker has KC, KS
AttackerM3Process() = AttackerM3AsClient() [] AttackerM3AsServer();

AttackerM3AsServer() = network?Client.Hello -> attackerKnowsGP -> AttackerM3Process []
                       network?Client.SendCipherText -> attackerKnowsCipherText -> attackerDecryptsCipherText -> attackerKnowsKC -> AttackerM3Process;
AttackerM3AsClient() = network?Server.Hello -> attackerKnowsServerMsg -> attackerKnowsKS -> AttackerM3Process []
                       network?Server.SendCertificate -> attckerKnowsPublicKey -> AttackerM3Process;
```

Figure 8. Code Implementation that mimics Attacker 3

In this scenario, PAT will assert that the attacker can compromise the protocol if and only if the KC and KS obtained by the attacker is the same as the KC and KS of the current session. This is expected, and for that reason, we need to model the ephemeral aspect of the DHE key exchange. This is done through a process call **resetConnection** which regenerates a new KC and $KS$by repeating the DH key exchange process again.

Other Capabilities the Attacker needs

The attacker would need to get access to the random keys of both the Server and the Client in **real-time** to generate the current session key which would enable the attacker to compromise the system in ways such as modifying the messages, perform a man-in-the-middle attack and eavesdrop on the messages between the Client and Server. This would be the final hurdle that the attacker needs to cross to be able to have any meaningful access to the messages.

The Extent of Damage to Current Existing Protocol

Lastly, the attackers that have been included in the current existing protocol, may gain access to only the previous messages that were communicated by getting the previous session keys. However, gaining access to the previous session keys is computationally expensive even for Attacker 3. Hence, this does not cause immense damage to the system. Furthermore, a system that incorporates TLS is usually built on the fact that the messages communicated are time-relevant. Hence, even if past messages are compromised, the attacker will rarely be able to gain access to the relevant information.

Verification properties

In order to check if we have met the 3 requirements of the CSP modelling goals, we have established following assertions for the goals:

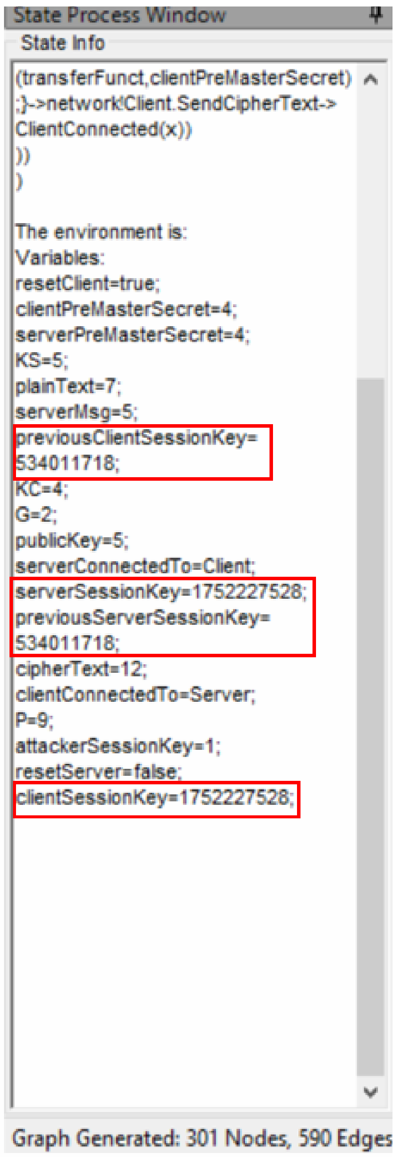
1. Both Client & Server have the same session keys
2. Session key not stolen
3. Perfect Forward Secrecy

```
ResetConnection() = [resetClient && resetServer]reset{previousClientSessionKey = clientSessionKey;
previousServerSessionKey = serverSessionKey; resetClient = false; resetServer = false; } ->
                    ServerProcess() ||| ClientProcess();

DHE() = ServerProcess() ||| ClientProcess() ||| AttackerM1Process() ||| AttackerM2Process() |||
AttackerM3Process() ||| ResetConnection();


#define sameKey serverSessionKey == clientSessionKey;
#assert DHE reaches sameKey;

#define sessionKeyNotStolen attackerSessionKey != serverSessionKey && attackerSessionKey != clientSessionKey;
#assert DHE |= []<> sessionKeyNotStolen;

#define connected serverConnectedTo == Client && clientConnectedTo == Server;
#assert DHE reaches connected;

#define perfectForwardSecrecy resetClient && resetServer;
#assert DHE reaches perfectForwardSecrecy;
```

Figure 9. Verification Properties

To check whether the Client and Server have the same session key, we simply asserted an equality to both keys. If both keys are equal, it means that they have the same session key.

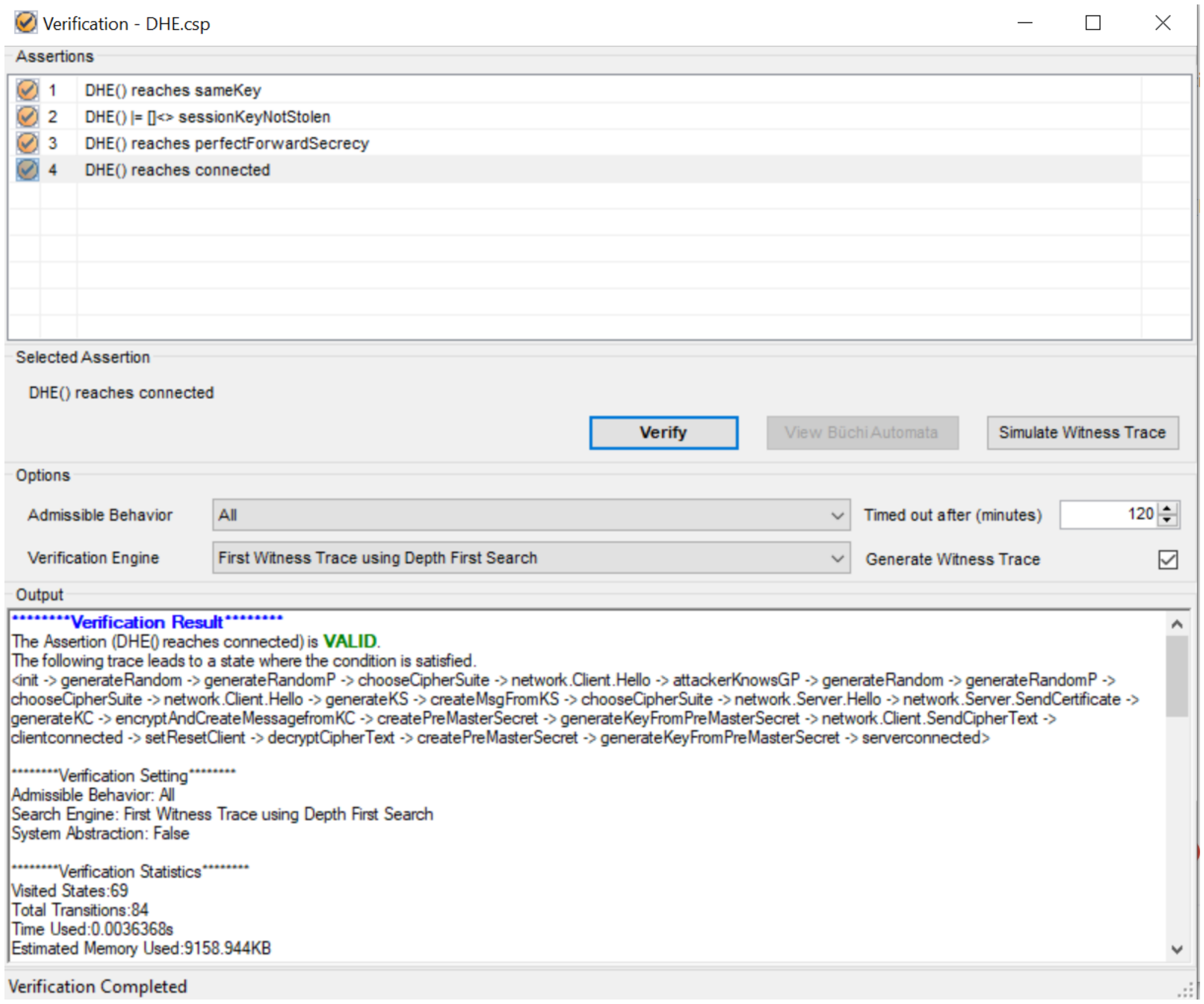
To check whether the session key has been stolen would be more tricky and is therefore split into 2 conditions. The first condition is that if the attacker does not launch a man-in-the-middle-attack, then the attacker must not have the session keys of either the Client or Server. The second condition is that if the attacker does launch a man-in-the-middle-attack, the attacker will share two different sets of session keys, one to pair with the Client and another to pair with the Server. In this case, both the Server and Client must be aware that they are not connected to each other, and the identity which they are connected to must be the attacker instead. If these aforementioned conditions hold, then we can safely assert that the session keys were not stolen, as can be seen in Figure 10.



```
State Process Window
State Info
(transferFunct,clientPreMasterSecret)
;}->network!Client.SendCipherText->
ClientConnected(x))
))
)

The environment is:
Variables:
resetClient=true;
clientPreMasterSecret=4;
serverPreMasterSecret=4;
KS=5;
plainText=7;
serverMsg=5;
previousClientSessionKey=
534011718;
KC=4;
G=2;
publicKey=5;
serverConnectedTo=Client;
serverSessionKey=1752227528;
previousServerSessionKey=
534011718;
cipherText=12;
clientConnectedTo=Server;
P=9;
attackerSessionKey=1;
resetServer=false;
clientSessionKey=1752227528;

Graph Generated: 301 Nodes, 590 Edges
```

Figure 10: Verification of Different Keys

To check for perfect forward secrecy, we initially verified that different keys were generated between sessions by storing the previous session keys and making a comparison against the current session keys. In the real TLS protocol, the previous session keys are simply discarded and even the Client and Server are not aware of the keys and do not store these values . But in our model, we keep track of the previous session keys in order to model and validate Attacker 3, who has access to the random keys. To launch a new DH key exchange, we make use of the reset process to act as a trigger to repeat the process (the significance of this is elaborated in the perfect forward secrecy portion of the report under other considerations).



```
Verification - DHE.csp
Assertions
1  DHE() reaches sameKey
2  DHE() |= []<> sessionKeyNotStolen
3  DHE() reaches perfectForwardSecrecy
4  DHE() reaches connected

Selected Assertion
DHE() reaches connected

Options
Admissible Behavior   All          Timed out after (minutes)   120
Verification Engine   First Witness Trace using Depth First Search     Generate Witness Trace ☑

Output
********Verification Result********
The Assertion (DHE() reaches connected) is VALID.
The following trace leads to a state where the condition is satisfied.
<init -> generateRandom -> generateRandomP -> chooseCipherSuite -> network.Client.Hello -> attackerKnowsGP -> generateRandom -> generateRandomP ->
chooseCipherSuite -> network.Client.Hello -> generateKS -> createMsgFromKS -> chooseCipherSuite -> network.Server.Hello -> network.Server.SendCertificate ->
generateKC -> encryptAndCreateMessagefromKC -> createPreMasterSecret -> generateKeyFromPreMasterSecret -> network.Client.SendCipherText ->
clientconnected -> setResetClient -> decryptCipherText -> createPreMasterSecret -> generateKeyFromPreMasterSecret -> serverconnected>

********Verification Setting********
Admissible Behavior: All
Search Engine: First Witness Trace using Depth First Search
System Abstraction: False

********Verification Statistics********
Visited States:69
Total Transitions:84
Time Used:0.0036368s
Estimated Memory Used:9158.944KB

Verification Completed
```

Figure 11. Screenshot of Verification Properties in PAT

Figure 11 above shows the verification on PAT. We chose the ALL option under admissible behaviour, 120 minute timeout and used the First Witness Trace Using Depth First Search verification engine. These are the default options in PAT and are sufficient for our model.

Last but not least, we also verified the security properties by manually inspecting the state diagrams that PAT can generate. This was done by manually iterating through each state and inspecting the variables' value at each and every state. As the entire state diagram was very large, we did not screenshot the entire diagram and only included a smaller portion which is shown in Figure 12. Manually inspecting the state diagrams allowed us to easily debug and check our implementation as well as uncover issues with our model when we began designing the different aspects of the project.

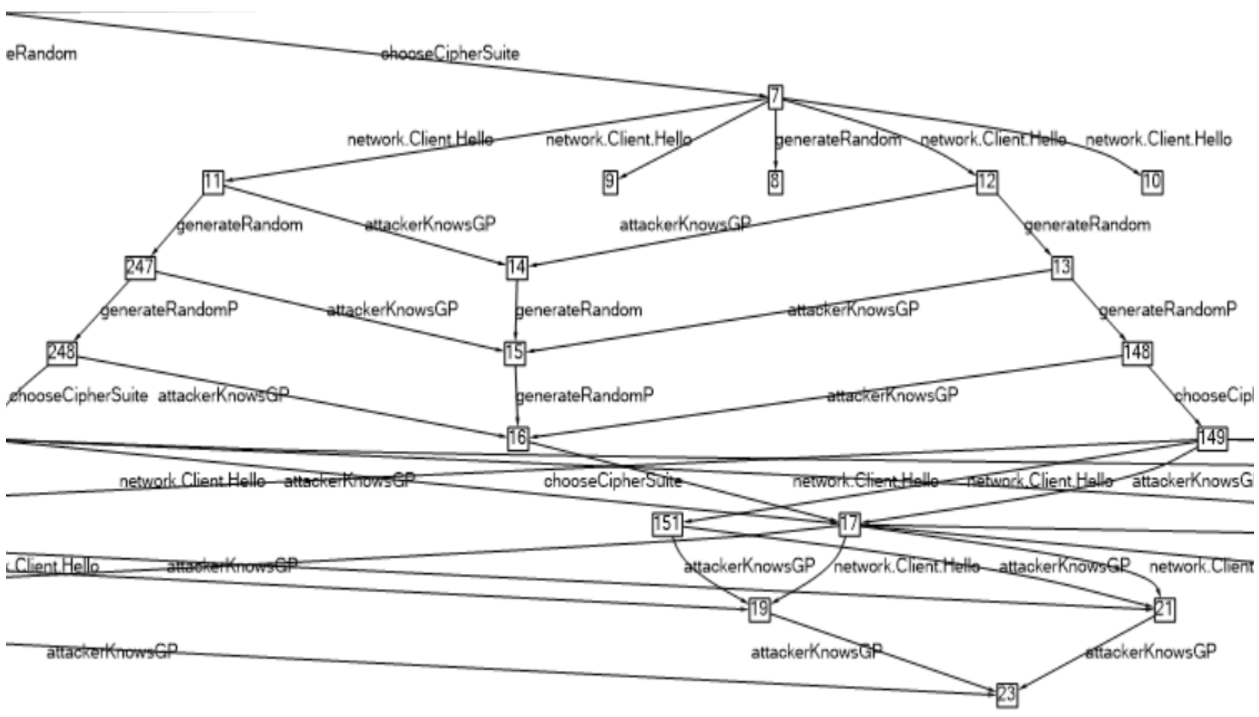


Figure 12. Screenshot of complete state diagrams

Limitations

PAT is the best system out there that offers general purpose model checking compared to other model checkers such as SPIN and PRISM. However, there are still inherent limitations that we faced while running PAT to perform model checking on the TLS protocol.

PAT limited computational capabilities

PAT does not have infinite computational capabilities. Therefore, when we simulated a large number of states, PAT can easily run out of memory. To deal with this issue, we have used a variety of optimization techniques we have learnt from CS3211.

First, we used the hidden variables that PAT offers. This solves the issue by reducing the number of possible states that PAT has to deal with. However, this is at the cost of the users; requiring a formal proof that such a reduction of a normal variable to hidden variable will not affect the security of the TLS protocol. This means that we needed to ensure that there are no possible viable attacks for the attackers in the states that were pruned away to validate the use of hidden variables.

Secondly, we reduced the number of bits involved in the DHE key exchange protocol. In TLS 1.3, 2048 bits keys are often used, however doing arithmetic operations on large numbers can be very expensive. This is especially the case if we are running on PAT and simulating it manually ourselves. Real world applications make use of advanced algorithms and hardware acceleration to perform these calculations. On the other hand, our goal is **to prove the security of the TLS protocol rather than implementing the TLS protocol**. Therefore, we choose to reduce the number of bits to reduce the computational power required of the PAT model checker.

Lastly, we also implemented our own TLS library to support the function calls. This reduced the number of states as calculations done within the functions do not count towards the total number of states computed by the PAT model checker.

CSP library does not allow strings

We cannot model the RSA (Server certificate) and DHE comprehensively to the 2048 bits implementation as we are limited by the CSP custom library capabilities which only allow the return of an integer (it would be computationally expensive and inefficient to return a bit at a time). After taking into account the possibility of overflow after multiplication, we came up with a mock-up version of DHE which is able to replicate the properties, although with lesser bits due to the constraints mentioned. This ensured that our model served as a reduced version of the actual real-world implementation.

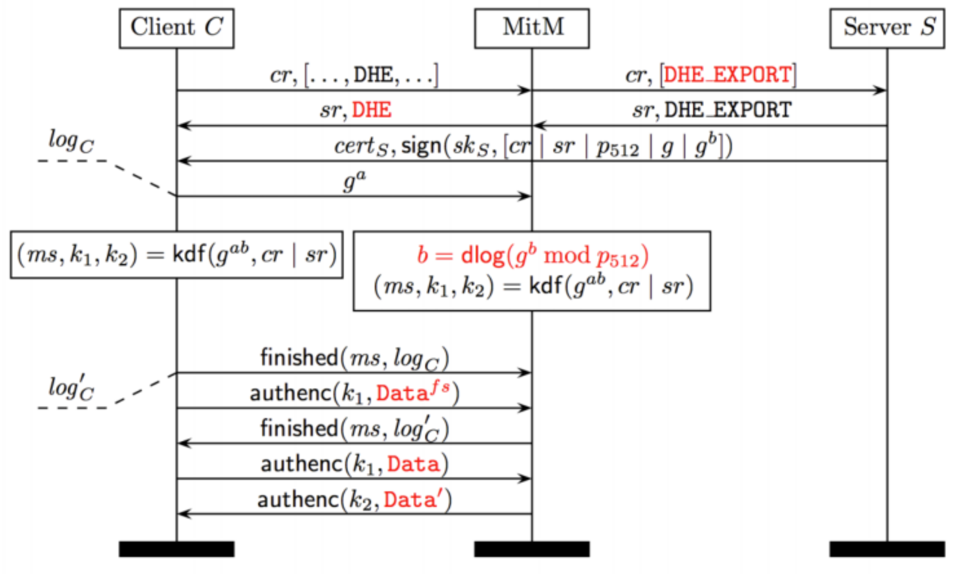
Other considerations

Through the use of a Dolev-Yao attacker model, we were able to abstract the TLS protocol and model the DHE key exchange. However, in reality, there are still attacks that can happen due to the rapid advancement in the computational capabilities of computers today which have been linearly improving as per Moore's law.

Logjam attack

Logjam attack is one of the well known attacks that targets the DHE key exchange protocol in older versions of TLS. It exploits the fact that 512-bits keys are allowed to be used in the DHE key exchange in older versions of TLS. With the number of bits reduced, it is computational feasible for today computers to precompute all possible $g^x mod\, p$. In order to store this huge amount of numbers, a rainbow table is typically used. A rainbow table is a space-time tradeoff for hash tables and makes use of hash chains to store more numbers. Therefore, this would allow the attacker to figure out KS and KC in real-time if they were to launch a man-in-the-middle-attack. Moreover, at that time of discovery, many of the computers actually used the same g and p making the attack even more feasible.

The logjam attack described above is summarised and shown in Figure 13:



Client C
MitM
Server S
$cr, [\ldots, \text{DHE}, \ldots]$
$cr, [\text{DHE\_EXPORT}]$
sr, DHE
$sr, \text{DHE\_EXPORT}$
log_C
$cert_S, \mathsf{sign}(sk_S, [cr \mid sr \mid p_{512} \mid g \mid g^b])$
g^a
$(ms, k_1, k_2) = \mathsf{kdf}(g^{ab}, cr \mid sr)$
$b = \mathsf{dlog}(g^b \bmod p_{512})$
$(ms, k_1, k_2) = \mathsf{kdf}(g^{ab}, cr \mid sr)$
$\mathsf{finished}(ms, log_C)$
log'_C
$\mathsf{authenc}(k_1, \mathtt{Data}^{fs})$
$\mathsf{finished}(ms, log'_C)$
$\mathsf{authenc}(k_1, \mathtt{Data})$
$\mathsf{authenc}(k_2, \mathtt{Data}')$

Figure 13. Logjam Attack

However, in today's TLS 1.3 protocol, this attack is no longer plausible. First and foremost, an attempt to use 512 bits keys would be rejected. Secondly, it is not feasible to precompute and store all possible $g^x mod\, p$ for 2048 bits as it is exponentially larger. Lastly, computers today (assuming good practise), do not use the same g and p. This makes logjam attack ineffective on TLS 1.3.

Perfect Forward Secrecy

For the scope of the project, it was important to verify that the Server and Client session keys generated during each session were indeed different such that perfect forward secrecy was enforced. As we attempted to prove that our implementation was indeed resistant to attacks we tried multiple methods such as running two instances to show that the session keys generated were indeed different. This would work - but it simply relies on the probability of the random number generators and random coprime pair generators. Hence this method of proving forward secrecy is invalid.

Therefore the true measure to ensure forward secrecy would be to reset the connection as even if the attacker is able to decrypt a previous ServerSessionKey it would not be able to compromise the current messages and due to the time-relevancy of the data exchanged, the attacker may not be able to obtain any useful information.

Probability Real-Time System Model

We considered utilising this model to show the time-based resets of the connections and regenerations Server keys but felt that it would be more accurate to utilise a non-time based implementation due to the inherent issues of a time-based implementation. If the system relies on time to be the seed of the random number generated, the attacker will be able to obtain the real time data being transmitted once the attacker determines that the model is based on time. Moreover, a time-based reset may prove to be significantly inefficient as these resets should occur based on the amount of data exchanged rather than a fixed time period.

Probability Communicating Sequential Process Model

We also considered using the probability based CSP model for our implementation but realised that there were no interdependencies or effective use cases to be exemplified such as the attackers having varying probabilities of compromising session keys or even attackers having varying probabilities of breaking any of the implemented protocols.

Hence, as the requirements stated that the scope of the project was to model based on different abilities rather than both probability and ability, we opted for the non-probability approach. Moreover, probability CSP requires more computing resources to compute (due to the probabilities) compared to CSP, allowing us to explore more options while using CSP.

Conclusion

In conclusion, our model has verified the DHE key exchange protocol in TLS. We were able to create a model that represented the protocol fairly, and run verifications that investigated various properties. With PAT, we have successfully verified the 3 essential requirements - Client and Server should establish the same session key after key exchange, attacker unable to steal session key if the Server and Client is connected, and perfect forward secrecy. We can see that these traits are upheld and TLS is secure with our model under the assumptions. Looking ahead, our implementation could potentially be improved to model more elements that would be present in a real-life implementation of the protocol in order to uncover other security aspects of the TLS protocol.

Extensions

In this section we will explore possible extensions in the implementations, to verify the protocol in a wider range of scenarios as well as consider the other dimensions of the project that were stipulated as sufficient to verify our implementation.

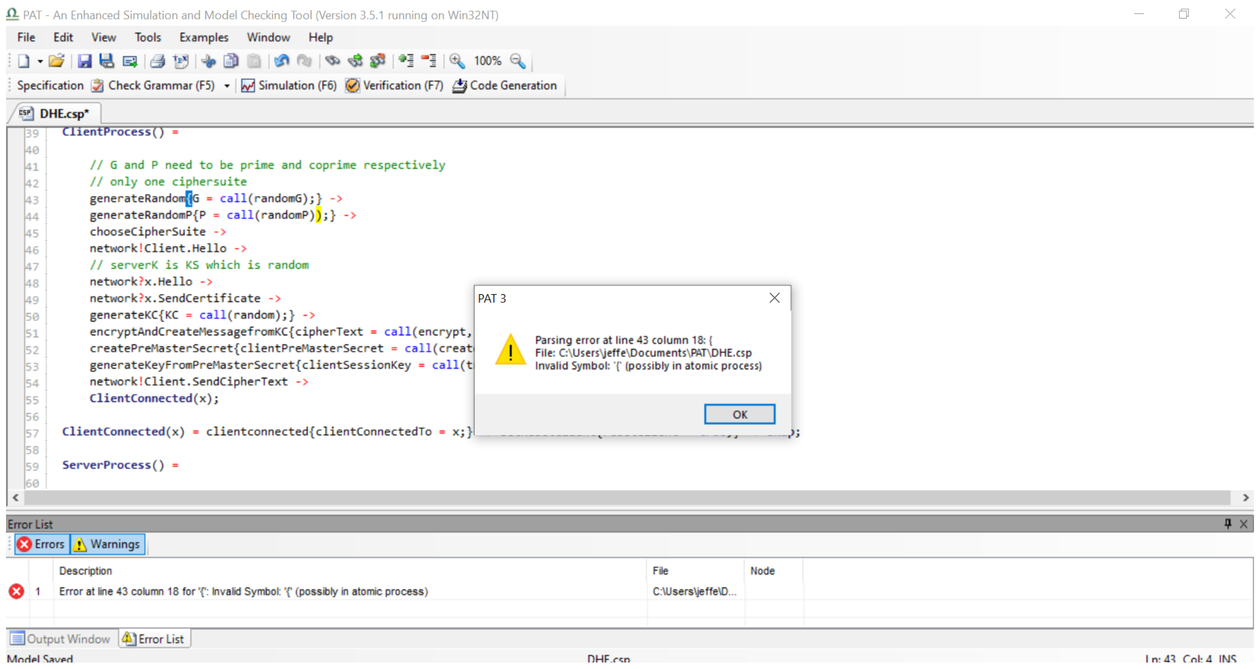
Certificate Verification Implementation

In the TLS handshake, certificate verification is a very important aspect during the handshake to avoid a man-in-the-middle-attack. In our model, we simply abstract this process of certificate verification. In reality, the certificate is verified by looking at the Certificate Authority and seeing if it is trusted by the Client, checking if the certificate indeed belongs to the Server and checking if the certificate hasn't expired yet.

To model this, we would need to introduce a trusted 3rd party, Charlie, who can act as a Certificate Authority. The attacker will not be able to get this trusted 3rd party to sign the certificate to spoof the Client. Next, the Client would verify the authenticity of the certificate upon receiving.

By modelling it this way, we can show that if the trusted 3rd party is compromised, the attacker can actually act as a Server and trick the Client into thinking that it is connected to the Server, while in reality, it is connected to the attacker. When the trusted 3rd party is not compromised, the attacker would not be able to do so.

Suggestions for PAT


PAT - An Enhanced Simulation and Model Checking Tool (Version 3.5.1 running on Win32NT)
File Edit View Tools Examples Window Help
100%
Specification Check Grammar (F5) Simulation (F6) Verification (F7) Code Generation
DHE.csp*

```
ClientProcess() =

    // G and P need to be prime and coprime respectively
    // only one ciphersuite
    generateRandom{G = call(randomG);} ->
    generateRandomP{P = call(randomP));} ->
    chooseCipherSuite ->
    network!Client.Hello ->
    // serverK is KS which is random
    network?x.Hello ->
    network?x.SendCertificate ->
    generateKC{KC = call(random);} ->
    encryptAndCreateMessagefromKC{cipherText = call(encrypt,
    createPreMasterSecret{clientPreMasterSecret = call(creat
    generateKeyFromPreMasterSecret{clientSessionKey = call(t
    network!Client.SendCipherText ->
    ClientConnected(x);

ClientConnected(x) = clientconnected{clientConnectedTo = x;}

ServerProcess() =
```

PAT 3

Parsing error at line 43 column 18: {
File: C:\Users\jeffe\Documents\PAT\DHE.csp
Invalid Symbol: '{' (possibly in atomic process)

OK

Error List

Errors | Warnings

	Description	File	Node
1	Error at line 43 column 18 for '{': Invalid Symbol: '{' (possibly in atomic process)	C:\Users\jeffe\D...	

Figure 14: Screenshot of Error Message

While we utilised PAT extensively for the project, we did encounter issues during the debugging of our model. Firstly, there was a generalised error message (possibly in atomic process) even though sometimes the error was indeed just an extra bracket as shown above which we feel did not help the debugging process and might confuse new users. As seen above in Figure 14, the actual error is highlighted in yellow while the error identified by the PAT IDE is highlighted in blue which is incorrect. In addition, as the error message states possibly in atomic process, the user could be misled further away from the actual issue.