CS3211 Project Presentation
TLS Protocol

Group Members: Sai Ganesh Suresh, Jefferson Chu, Mohamed Riyas

Tasks

- Please construct a CSP concurrent model regarding the protocol description and verify the properties.
- After your verify the given protocol with your PAT program, please consider the following questions:

1. If the attacker would like to compromise the protocol, what other capability does he or she need?
2. Given existing protocol, what is the most damage can the attacker cause?

Please create new assertions and verify them.

Note that, it is a simplified version of the protocol. Interested students can implement and verify a model in a more sophisticated design, which can be referred in [5] for more details.

Verification properties

We are going to model both protocols and verify the following security properties.

1. Client and server should establish the same session key after key exchange.

2. If the client/server believes it has established a session key with an authenticated peer, then the attacker does not know the session key when it is being used.

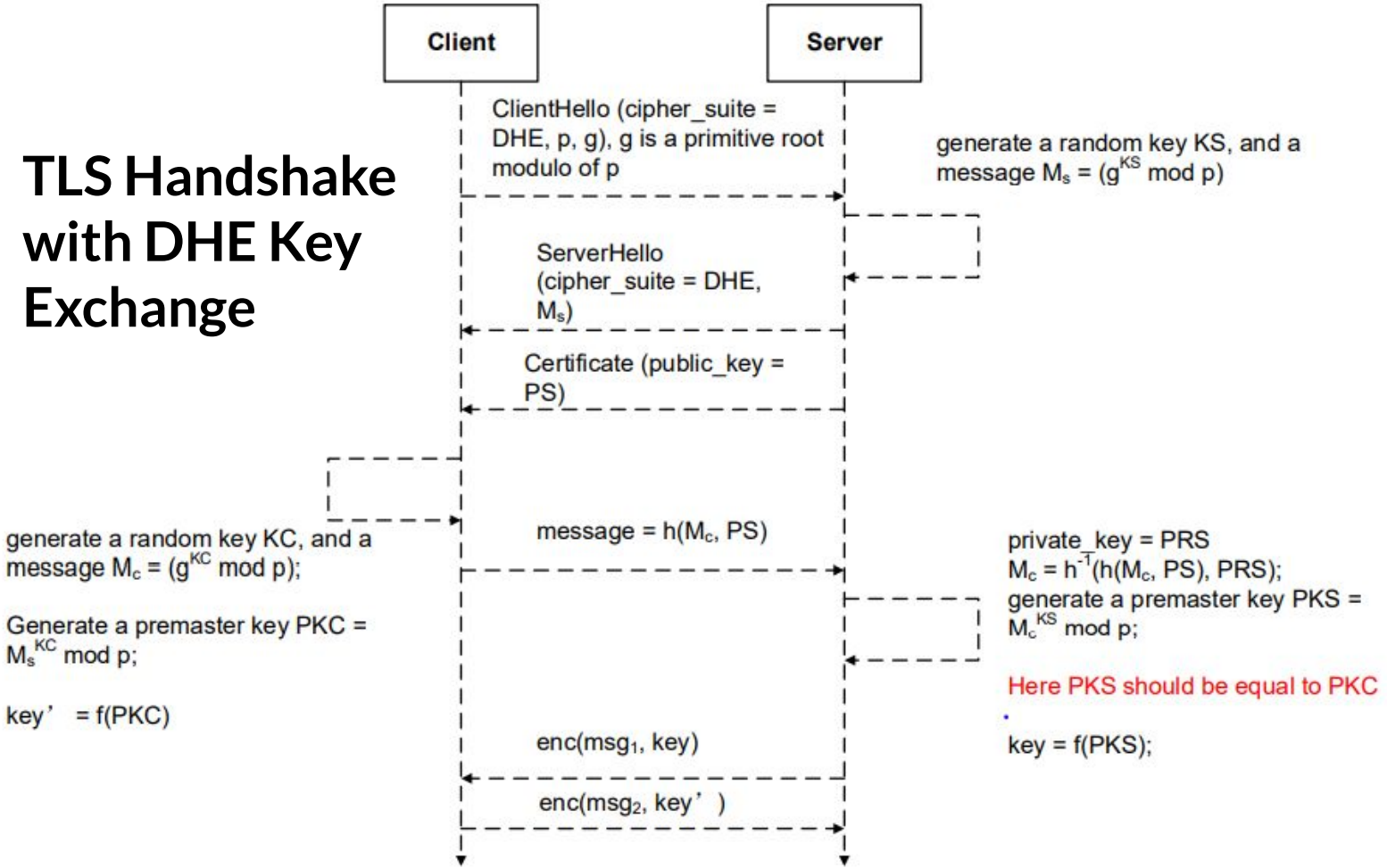
3. Even if the private key of the server (of the time being) is stolen, the attacker cannot compromise the communication in the next phase, i.e., perfect forward secrecy (PFS).

Contents

1. Ephemeral Diffie-Hellman (DHE in the context of TLS)
    a. TLS Model
    b. Attacker Capability
    c. Attacker Model
2. Verification properties
    a. Same Session Key
    b. Establishing Session Key without Attacker knowing
    c. Perfect forward Secrecy
3. Other considerations

Ephemeral Diffie-Hellman (DHE in the context of TLS)

TLS Handshake with DHE Key Exchange



Client Process

```
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;
```

Client Process

```
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 ->
    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);} ->
    storePreviousSessionKey{previousClientSessionKey = clientSessionKey;} ->
    network!Client.SendCipherText ->
    ClientConnected(x);

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

Server Process

```
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;
```

myTLS library functions

```
public static int randomG() {
        return 2;
}

public static int randomP() {
        return 9;
}

public static int random() {
        Random rnd = new Random();
        return rnd.Next(20);
}

public static int createMsg(int key, int G, int P) {
        // return G^key mod P
        return (int)(Math.Pow(G, key) % P);
}
```

```
public static int createPreMasterSecret(int key, int message, int P) {
        // return message^key mod P

        return (int)(Math.Pow(message, key) % P);
}

public static int transferFunct(int seed) {
        Random rnd = new Random(seed);
        return rnd.Next();
}

public static int encrypt(int plainText, int publicKey) {
        // do something
        return plainText + publicKey;
}

public static int decrypt(int cipherText) {
        return cipherText - 5;
}
```

Server Process

```
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);} ->
    chooseCipherSuite ->
    network!Server.Hello ->
    network!Server.SendCertificate ->
    network?x.SendCipherText ->
    decryptCipherText{plainText = call(decrypt, cipherText);} ->
    createPreMasterSecret{serverPreMasterSecret = call(createPreMasterSecret, KS, plainText);} ->
    generateKeyFromPreMasterSecret{serverSessionKey = call(transferFunct, serverPreMasterSecret);} ->
    ServerConnected(x);

ServerConnected(x) = serverconnected{serverConnectedTo = x;} -> ServerConnected(x);
```

Attacker Models

Attacker 1 Model

Attacker 1: Attacker with following capabilities:

- Dolev-Yao (DY) attacker
- 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 creating 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.

Attacker 2 & 3 Model

Attacker 2: Attacker with following capabilities:

- Attacker 1 (Dolev-Yao (DY) attacker) + can compromise the private key of protocol participants,

Attacker 3: Attacker with following capabilities:

- Attacker 1 (Dolev-Yao (DY) attacker) + can compromise the random keys

Attacker 1 Process

```
//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;
```

Attacker 1 Process

```
//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;
```

Attacker 2 Process

```
//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;
```

Attacker 2 Process

```
//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;
```

Attacker 3 Process

```
//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;
```

Attacker 3 Process

```
//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;
```

Verification & Other considerations

Verification Properties

Same Session Key

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

Session Key not stolen

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

Connected

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

Verification Properties

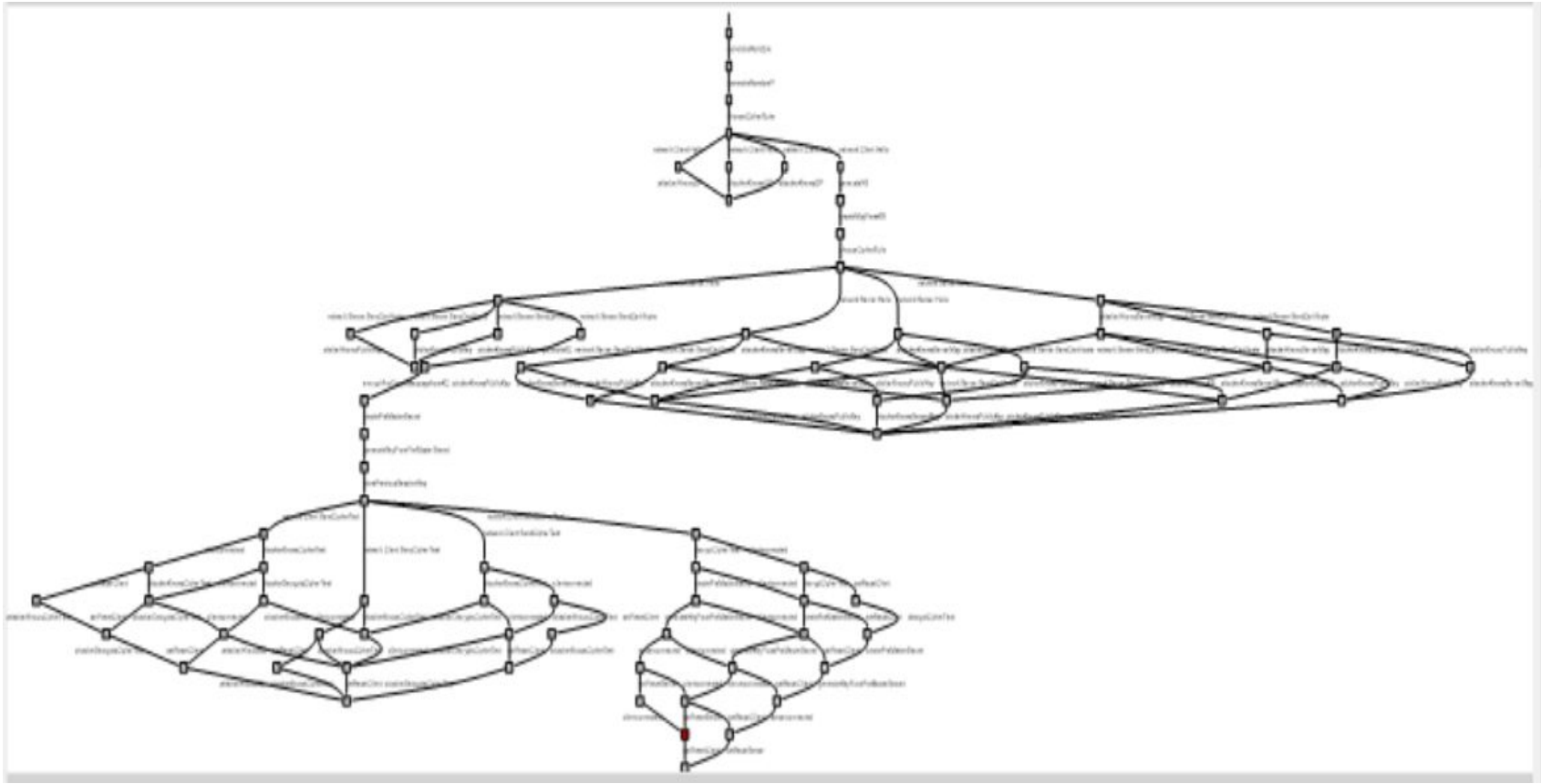
Perfect Forward Secrecy

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


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


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

Implementation Graph - Verification Method



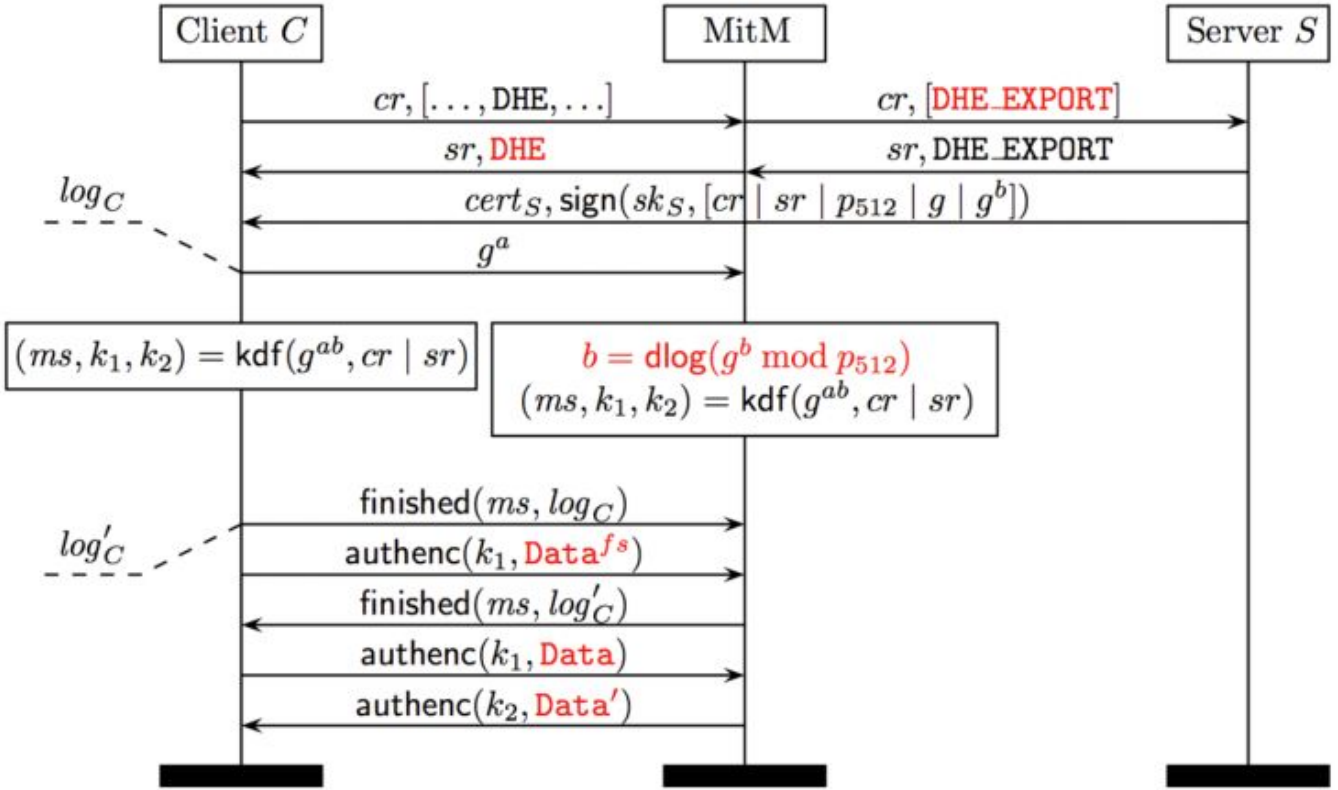
Demonstration

Limitations & Considerations

Other considerations: Logjam Attack

This is giving an attacker the ability to create a rainbow table to crack the discrete log problem. Given a list of possible g & p, generate all possible values of m such that: m = g ^ a mod p and stores it in a rainbow table. When the attacker receives m from client, looks up the table to figure out the a.

Other considerations: Logjam Attack



CSP library does not allow strings

- We cannot model the RSA (server certificate) and DHE properly to the 2048 bits implementation as we are limited by the CSP custom library capabilities which only allows return of integer (it would be computationally expensive and inefficient to return a bit a time).

- After taking into account 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

Forward Secrecy

- As we attempted to prove that our implementation was indeed resistant to attacks which 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 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.

Probabilistic 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.

Thank You!