

**CS5232 – Formal Specification and Design Techniques**

**Project Report**

**Formal Verification of Transport Layer Security Protocol**

**Team Members – Group 5**

Vipul Sharma (A0178385M)

Sonal Devadas Shenoy (A0174450J)

Maddi Kamal Divya (A0178511E)

Ramya Gurunathan (A0174410R)

Contents

[1. Introduction 4](#_Toc513209029)

[1.1 Problem Description 4](#_Toc513209030)

[1.2 Analysis and Results: 4](#_Toc513209031)

[1.3 Structure 5](#_Toc513209032)

[2. TLS 1.3 5](#_Toc513209033)

[2.1 Introduction 5](#_Toc513209034)

[2.2 Handshake Protocol 6](#_Toc513209035)

[2.2.1 Notation 6](#_Toc513209036)

[2.2.1 Overview 7](#_Toc513209037)

[2.2.2 Pure Diffie-Hellman Mode 8](#_Toc513209038)

[2.2.3 Pure Pre-Shared Key Mode 9](#_Toc513209039)

[2.2.4 Diffie-Hellman Pre-Shared Key Mode 9](#_Toc513209040)

[2.2.5 Hello Retry Request 10](#_Toc513209041)

[2.2.6 Zero Round-Trip-Time 10](#_Toc513209042)

[3. Modelling 10](#_Toc513209043)

[3.1 Overview 10](#_Toc513209044)

[3.2 Global Variables 10](#_Toc513209045)

[3.3 Constants 11](#_Toc513209046)

[3.4 Variables 12](#_Toc513209047)

[3.5 Channels 12](#_Toc513209048)

[3.6 Events 12](#_Toc513209049)

[3.6.1 Client Certificate Request by Server 12](#_Toc513209050)

[3.6.2 Client handling the Certificate Request by Server 13](#_Toc513209051)

[3.6.3 Calculation of Secret Key 13](#_Toc513209052)

[3.6.4 PAT code Library to Calculate Secret Key 14](#_Toc513209053)

[3.6.5 Selection of Exchange Mode 14](#_Toc513209054)

[3.7 Processes in TLS 1.3 15](#_Toc513209055)

[3.7.1 Client and Server Negotiation 15](#_Toc513209056)

[3.7.3 0 RTT 16](#_Toc513209057)

[4. Attacker Model 17](#_Toc513209058)

[4.1 DY Attack: 17](#_Toc513209059)

[4.2 0RTT Replay Attack: 18](#_Toc513209060)

[5. Security Properties 19](#_Toc513209061)

[5.1 Establishing the same session keys 19](#_Toc513209062)

[5.2 Peer Authentication 19](#_Toc513209063)

[5.3 Secrecy of session keys 20](#_Toc513209064)

[5.4 Prevent Eavesdropping in DY Attack Model 21](#_Toc513209065)

[5.5 Deadlock 21](#_Toc513209066)

[5.6 Prevent Man in the middle attack 22](#_Toc513209067)

[5.6 0RTT replay attack: 23](#_Toc513209068)

[6. Results: 24](#_Toc513209069)

[7. Traffic 26](#_Toc513209070)

[7.1 Data collection and graph/model creation 27](#_Toc513209071)

[7.2 Analysis 31](#_Toc513209072)

[8. Z Modelling 32](#_Toc513209073)

[9. Conclusion 33](#_Toc513209074)

[10. References: 34](#_Toc513209075)

1. Introduction

1.1 Problem Description

In the fast-paced development of the modern technology, internet has become primary mode of communication. However, the security aspect of Internet always remains as a concern leading to the development of many security protocols that protect the data’s confidentiality, integrity and authenticity. Protocols use cryptographic methods like encryption and digital signatures to secure the data. Transport Layer Security (TLS) is one such protocol that was developed in the mid 90s and it soon became the most used security protocol. It was first designed and developed by Netscape as Secure Socket Layer (SSL) and has been improved and updated ever since. Today, TLS 1.2 is a proposed Internet standard and it is used to protect the sensitive data of many applications and services. TLS is most predominantly used in HyperText Transfer Protocol Secure (HTTPS), ensuring security for pivotal parts of the world-wide-web. TLS has evolved over the days, overcoming many vulnerabilities and the Internet Engineering Task Force (IETF) is at present designing the latest version, TLS 1.3. Development and design of security protocols is not an easy task and is often complicated and not intuitive. Even the most straightforward protocols, that has only three messages, can contain flaws that might allow intruders to gain knowledge and extract sensitive data. Several automated protocol verification tools were developed owing to the numerous attacks on older versions of TLS/SSL and other protocols. When protocol execution involves a substantial number of participants, executing the protocol in parallel and exposing it to the malicious actions of an adversary with multiple other peers, is not “manually” possible. Tools like Process Analysis Toolkit (PAT) are specifically built for this purpose to reason about the numerous possible implementations to prove or disprove the correctness of models automatically. These tools model possible executions by defining an internal protocol specification language. The correctness of a protocol is expressed by satisfaction of certain properties and is reasoned using formal methods. These automated verifications can evaluate the security of protocols that are still in development to find and fix faults before it is implemented and used in practice. In this report, we use PAT to automatically verify and formally examine the latest revision of the TLS 1.3 draft.

1.2 Analysis and Results:

We used PAT to verify the handshakes and flow of traffic in TLS 1.3. We worked on modelling the key components such as DHE, PSK, new session ticket, 0-RTT. We demonstrated the generation and calculation of secret key by using C# library for Mathematical operations. We verified few security properties and demonstrated that the TLS1.3 can prevent DY attack. We also verified that 0RTT replay attack can be successful with few prerequisites which are explained in detail in the later sections. In the second part of the project, we compared the real TLS traffic with the specification by developing a fully automatic framework that can output TLS handshake pattern in DTMC for a given web application. We tested our framework on popular sites such as Facebook, Twitter, Skype Desktop and PayPal and captured the significant pattern changes of the traffic flow.

1.3 Structure

In this report, we first present an outline of the TLS 1.3 protocol specification. We conceptually categorize the handshake into three main modes of execution. The initial handshake mode where a Diffie-Hellman key exchange is used without pre-shared keys, the pure pre-shared key handshake where only the value of the pre-shared key is used, and the pre-shared key handshake where an additional Diffie-Hellman secret is established for perfect forward secrecy. We then discuss our TLS 1.3 handshake modelling, ultimately summarizing the analysis and results of our model by concluding the security guarantees of TLS 1.3.

2. TLS 1.3

In this section we present the proposal of Transport Layer Security 1.3. It presently is an Internet draft preserved by the Internet Engineering Task Force (IETF) and it aims to substitute its predecessor, TLS 1.2.

2.1 Introduction

Transport Layer Security (TLS) is an Internet Engineering Task Force (IETF) standard and is perhaps the most significant security protocol in the Internet. It operates over some reliable transport layer protocol such as TCP, and intends to deliver confidentiality, integrity and authentication to various applications. A well-known example of an application layer protocol utilizing TLS, designed to provide secure communication over the Internet is the HTTPS (HTTP over TLS). The security of this protocol is very crucial, as numerous widely used, and security critical applications depend on it. Possibly the most renowned example is e-banking, whose acceptance has seen a swift growth in the past few years. The antecedent of TLS, Secure Socket Layer (SSL), was first developed for secure web transactions by Netscape. Then in 1999, as an upgrade to the latest SSL 3.0, the first version of TLS, TLS 1.0. However, these two protocols are not interoperable, even though the design is similar. Therefore, we cannot downgrade a TLS 1.0 connection to a SSL 3.0. Building on top of TLS 1.0, TLS 1.2 was developed in August 2008 and it is de facto standard over the Internet for secure communications. Since its inception in 1999, the TLS specification went through multiple modifications to include things like, the support for new cryptographic methods such as Advanced Encryption Standard (AES) to adapt to various newly found attacks (e.g., cipher-block chaining attacks). TLS was backward compatible among its own versions and TLS 1.2, also supports the use of insecure cryptographic methods (e.g., MD5) that were used in TLS 1.0. This backward compatibility added a great deal of complexity to the specification and TLS 1.3 aims to refactor the specification in such a way that it will reduce the complexity. Unused features such as static RSA, Non-AEAD ciphers, renegotiation and custom ECDHE groups are removed in this update. TLS 1.3 differs from its previous versions in the sense that all ciphers are executed in Authenticated Encryption with Associated Data (AEAD) block cipher mode of operation. With the more constrained design of this version, another objective of TLS 1.3 is to have better performance and it manages to reduce the number of round trips involved in the handshakes. Since the number of options for cryptographic parameters for negotiation are less, the client can most likely predict the server’s preferences. In TLS 1.3, the server can start sending encrypted application data in its first flight, as it is possible for the cipher suite to be negotiated after only two messages. It uses an optimistic approach, and only when the client cannot “guess” the preferences of the servers, additional negotiation messages are sent. It is also possible to have zero round-trip messages (0-RTT), enabling the client to transfer application data in its first flight and it gives us a considerable performance gain. However, the protection of this data is weaker than sending the data after handshake. Particularly it is not protected against replay attacks die to the lack of perfect forward secrecy (PFS). TLS 1.3 is divided into three protocols namely, the handshake protocol, the record protocol, and the alert protocol. As the record and alert protocols are not important in the symbolic model, we are not considering them in this work.

2.2 Handshake Protocol

In this section, we provide an overview of the TLS handshake. We define a notation for the TLS messages initially and then we explain the three distinct phases of a TLS handshake: Key Exchange, Server Parameters, and Authentication Phase.

2.2.1 Notation

The TLS 1.3 handshake involves distinct types of messages being exchanged during the protocol and we refer to these messages with the following syntax. Each message is further explained in the remaining subsections:

|  |  |
| --- | --- |
| Message Type | Abbreviation |
| ClientHello | CH |
| ServerHello | SH |
| HelloRetryRequest | HRR |
| EarlyFinished | EF |
| EncryptedExtensions | EE |
| CertificateRequest | CR |
| Server CertificateVeriy | CTS |
| Server Certificate | CVS |
| Server Finished | FIS |
| Client Certificate | CTC |
| Client CertificateVerify | CVC |
| Client Finished | FIC |
| NewSessionTicket | NST |

2.2.2 Overview

The goal of the handshake protocol is the authentication of its peers and the establishment of the required security parameters. It negotiates a session that is defined by cipher specifications, peer certificates, and a resumption secret. The future handshakes can then use this resumption secret, to reuse the authentication of the initial handshake, in a process called session resumption. This is done by deriving a resumption psk (pre-shared key) and a resumption context. Therefore, it is possible to have multiple connections with the same session and with only one resumption secret. The identities are authenticated using either a pre-shared symmetric key or asymmetric cryptography using certificates in case the pre-shared key is not available. The handshake protocol supports three key exchange modes –

DH Mode – Uses Diffie-Hellman combined with asymmetric cryptography for authentication.

PSKONLY Mode – Uses a pre-shared key and no Diffie-Hellman at the cost of perfect forward secrecy.

PSKDH Mode – Uses both Diffie-Hellman and a pre-shared key.

Each of these handshakes can be partitioned into the following three phases –

**Key Exchange Phase**

The handshake starts with the client sending a ClientHello, containing a newly generated nonce, extensions, and a list of cipher suites offered. The extensions may contain one or more DiffieHellman key shares, or one or more pre-shared key identities, or both. The server chooses one cipher suite from the offered list and responds with a ServerHello. The ServerHello contains a nonce, the selected cipher suite, and the extensions. Again, the extensions sent by the server depends on the contents of the ClientHello extensions. In case, the server does not find any appropriate Diffie-Hellman share, then it will send a HelloRetryRequest, which leads to another ClientHello from the client. Once this exchange phase is done, both Client and the Server presumably share a secret key, which could have been derived from either a Diffie-Hellman secret and/or the pre-shared key. Also, if the client uses a pre-shared key, it may include an appropriate extension in the ClientHello to indicate the presence of early application data. This is called as zero round-trip-time (0-RTT) data and it is sent on the client’s first flight. All the subsequent messages are encrypted using a key derived from the input keying material.

**Server Parameters**

This phase comprises of two messages sent by the server, which are EncryptedExtensions and CertificateRequest. The first message EncryptedExtensions, is the response to the extension of the ClientHello. The CertificateRequest specifies that the client must authenticate itself, and all these messages, are encrypted using the keys derived from the input secrets.

**Authentication**

The authentication phase contains two sets of authentication messages shared by the client and the server. Each set comprises of three messages, namely Certificate, CertificateVerify and Finished. The Certificate message, as the name suggests, consists of a certificate that binds the peer’s name to a public key. The CertificateVerify message is the signature of the hash of the resumption context and the hash of all the handshake messages sent till now. The Finished message is like CertificateVerify and is the hash of resumption context and the MAC of the hash of all handshake messages until now. The Finished message is always sent, while the Certificate and CertificateVerify messages are only sent if authentication is required in the handshake mode. For example, in the handshake that use only Diffie-Hellman, the server sends all the authentication messages, but the client only sends Client Finished. In the end, the client and the server establish a resumption secret and subsequently the client uses this resumption secret to indicate a session resumption with only its ClientHello.

2.2.3 Pure Diffie-Hellman Mode

Once the handshake is done, the server might send a NewSessionTicket message. The server has an opaque ticket (or psk identity) which it sends with this message and bounds it to the resumption psk and resumption context. The client may use this ticket in ClientHello as a psk identity extension. This message must be encrypted using the traffic key (trKey) as it is a part of the post handshake phase.

Diffie-Hellman Half-Keys

The Diffie-Hellman key exchange is a method of securely establishing a shared secret over an insecure channel. The half-keys, which are used to compute the shared Diffie Hellman secret, is calculated by using a generating element ‘g’ of an appropriate cyclic group raised to the power of a randomly generated number. The client adds a list of all potential Diffie-Hellman half-keys which may belong to different groups to the ClientHello. The server may accept one of them and add its half-key belonging the chosen group to the ServerHello. As there is no key available at the early state of the handshake, the client cannot send application data in its first flight in this mode. The server then specifies whether it entails the client to authenticate itself, by sending the EncryptedExtensions message. If authentication is required, then a CertificateRequest message is sent. The server must send its full authentication block containing, Server Certificate, Server CertificateVerify and Server Finished, in this instance as there is not pre-shared key. The client decides whether to send the full authentication block consisting of Client Certificate, Client CertificateVerify and Client Finish or to just send Client Finished depending on the CertificateRequest message. If the handshake fails to authenticate the client, then the authentication will still take place either in the application layer, or in a future PSK handshake or using post-handshake authentication features. The server can send the application data only after the Server Finished message.

2.2.4 Pure Pre-Shared Key Mode

TLS 1.3 offers the possibility, if the parties initially share a secret key (PSK), to run the handshake without using Diffie-Hellman. In this case, there are no Diffie-Hellman half-keys included in the ClientHello and ServerHello. The ClientHello contains the so-called PSK identity (psk id), uniquely identifying the PSK. If the client and the server share multiple keys, then the ClientHello may contain a list of PSK identities wherefrom the server may choose one. The ServerHello then contains the selected PSK identity. If a previous handshake is to be resumed, then the value of the PSK identity corresponds to the ticket of the last NewSessionTicket message sent by the server. As the server is already authenticated by the pre-shared key, it is not necessary to send Server Certificate and Server CertificateVerify. This is because the Server Finished, using a MAC key derived from the PSK, also authenticates the handshake. This is because the preshared key is transitively derived from, and thus bound to, the hash of a DHmode handshake that involved the Server Certificate. Since, however, client authentication is not mandatory in the DH mode, it is possible to use delayed client authentication in PSK modes. The disadvantage of this mode is the lack of perfect forward secrecy: Since there is no Diffie-Hellman key involved in this mode, the adversary may learn the contents of the handshake, as well as the keys used to protect application data, if he manages to obtain the pre-shared key later in time.

2.2.5 Diffie-Hellman Pre-Shared Key Mode

This mode differs from the pure PSK mode, where it requires both the parties to initially share a Diffie-Hellman key, to provide perfect forward secrecy to the handshake and the traffic keys. The client adds a list of Diffie-Hellman half-keys along with the psk id to the ClientHello. The ServerHello then consists of one selected PSK identity and a Diffie-Hellman half-key belonging to the same group as the ClientHello. All the keys are then derived using these PSK and the Diffie-Hellman key. However, the keys used to protect 0-RTT data are an exception as they are only derived from the pre-shared key.

2.2.6 Hello Retry Request

TLS 1.3 uses an optimistic approach where only two messages are used to agree on the key and cryptographic algorithms. However, if the ClientHello only consist of Diffie-Hellman keys that are not accepted by the server, then the server will send a HelloRetryRequest, instead of a ServerHello indicating the client of the groups that must be used. The client will then send another ClientHello, incorporating the Diffie-Hellman share of the supported group. Note that no more than one HelloRetryRequest will be sent but still the overall process adds an additional round-trip-time to the handshake.

2.2.7 Zero Round-Trip-Time

To avoid the network latency and to cut the number of round trips, TLS 1.3 allows the client to send application data in its first flight. A well-known use case of this is the HTTP get request, where the server is able send the content of a web page directly after the Server Finished message. At this early point, since the client is yet to receive any message from the server, the server will not have any freshness guarantee to the data and the data during this phase will not be replay protected. It is possible to send early data only if a pre-shared key is available as the key used to protect early data is only derived it. In this case if the client wishes to send early data, it includes an appropriate extension to the ClientHello, along with Early EncryptedExtensions, EarlyFinished, a MAC over the handshake hash of ClientHello, followed by the application data and the early data alert. The handshake is continued normally after all these messages and a no 0-RTT message is included in the handshake hash.

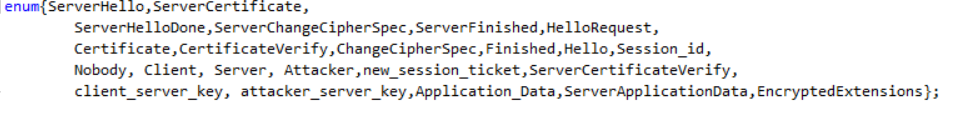
3. Modelling

3.1 Overview

Our model has different systems to explain the detailed properties of TLS 1.3. Each system explains the handshake and traffic flow between Client, Server and Attacker wherever it is applicable. To build a strong Client – Server Handshake model, we have taken considerations of all the important properties of TLS 1.3 such as calculation of secret key, choice of Server to request Client certificates and the options to select the mode of Secret Key operations. We also created a Math library to calculate the secret key on the both Client and Server end. We have considered scenarios where the attacker can act as middle man and try to attack the Handshake. The attacker stores Client/Server information in an attacker buffer to perform the attack in subsequent handshakes.

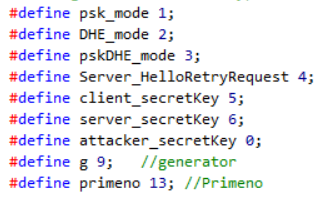
3.2 Global Variables

Global variables are defined in enum structure and the value for these variables is constant. All the variables added here are self – descriptive.



*Figure:Global Variables used in PAT code*

3.3 Constants

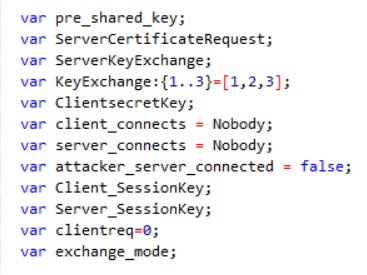


*Figure: Constants used in PAT code*

|  |  |  |
| --- | --- | --- |
| Name of the Constant | Value of the constant | Description |
| psk\_mode | 1 | If the value is 1, the mode of operation is selected as PSK to authenticate the server. |
| DHE\_mode | 2 | If the value is 2, the mode of operation is selected as DHE which is the default mode of TLS 1.3 |
| pskDHE\_mode | 3 | If the value is 3, the mode of operation is selected as DHE and PSK which limit the number of expensive public key  operations that the server needs to perform |
| Server\_HelloRetryRequest | 4 | If the value is set to 4, it indicates the client with the groups the server will accept. |
| client\_secretKey | 5 | Client generates secret key at client end based on the cryptographic parameters that are exchanged. |
| server\_secretKey | 6 | Server generates secret key at Server end based on the cryptographic parameters that are exchanged. |
| attacker\_secretKey | 0 | Attacker tries to generate secret key at Attacker end based on the sensitive information captured during the handshake between client and server. |
| g | 9 | ‘g’ is generator used to calculate the secret key. We gave value = 9 for testing purpose. |
| primeno | 13 | ‘p’ is prime number used in calculation of the secret key. We gave value = 13 for testing purpose. |

*Table: Description of the Constants used in PAT code*

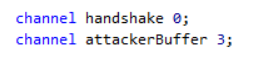
3.4 Variables



*Figure: Variables used in PAT code*

3.5 Channels

We used two channels in our model. The Client Server communication uses handshake channel. attackerBuffer stores the information that is captured during the information.

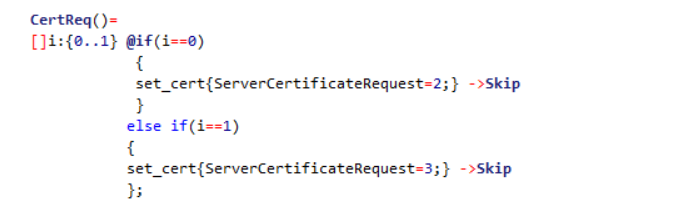


*Figure: Channels used in PAT code*

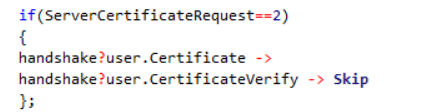
3.6 Events

3.6.1 Client Certificate Request by Server

As server-side Authentication is mandatory and Client-side authentication is optional, Server can choose to request or not request the Client Certificate.



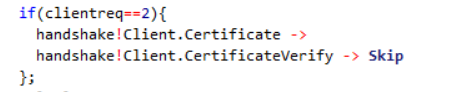
*Figure: Server Certificate Request*



*Figure: Server Requesting the Client Certificate*

3.6.2 Client handling the Certificate Request by Server

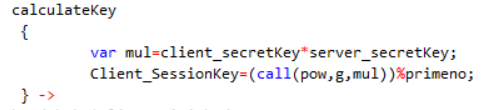
The below event describes the event where the Server requests the Client to share its certificate. The Client will send the Client Certificate along with the ClientVerify. If the Client Certificate is not requested, it will skip the event and continue with the handshake.



*Figure: Client responding with its certificate*

3.6.3 Calculation of Secret Key

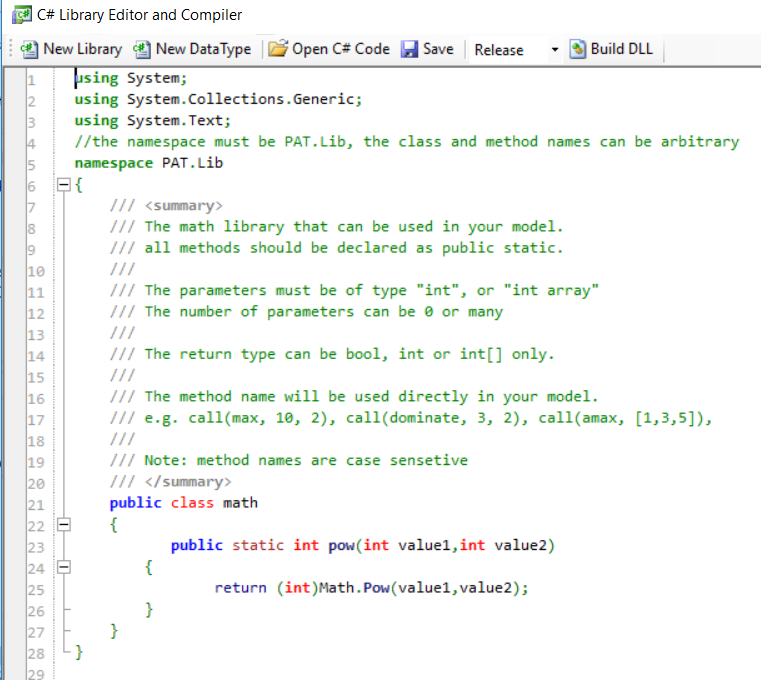
The information exchanged between the client and Server is used to calculate the secret key on both Client and Server end. The secret keying material generated will be unique and is known only to the Client and the Server.



*Figure: Calculation of Secret key on both Client and Server end.*

3.6.4 PAT code Library to Calculate Secret Key

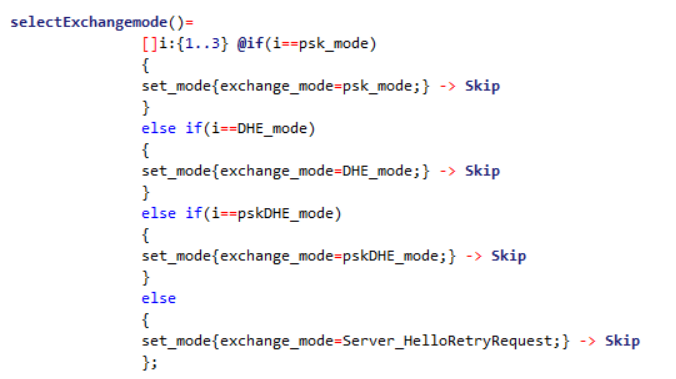
We had written a C# library to calculate the secret key and added it to it the PAT Library.



*Figure: Defined the Mathematical Power function in C# library*

3.6.5 Selection of Exchange Mode

During the handshake the client and Server can negotiate the Exchange Mode among DHE mode, PSK mode, PSK + DHE mode. If the exchange mode doesn’t match, Server will send a HelloRetryRequest giving a chance to the client to renegotiate the mode of operation.

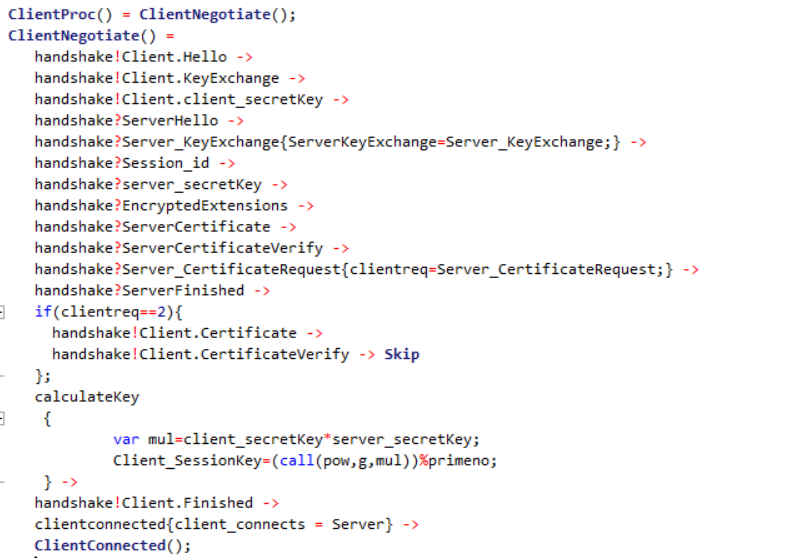


*Figure: Selecting the mode of the operation*

3.7 Processes in TLS 1.3

3.7.1 Client and Server Negotiation

The below handshake describes the Client and Server Process in TLS 1.3. Client initiates a connection with Server by sending Client Hello, Key exchange information and Client Secret Key. If the exchanged key matches on both Client and Server end, it will continue with the subsequent handshakes. It will establish a session between the client and Server and a session id is generated which is unique for each session. Both Client and Server exchange the information such as Encrypted Extensions and the Certificates. Once the Client Sends a Finish Message, it will be connected to the Server.



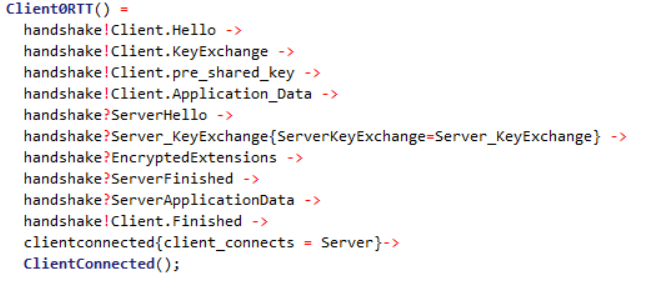
*Figure: Client handshake process*



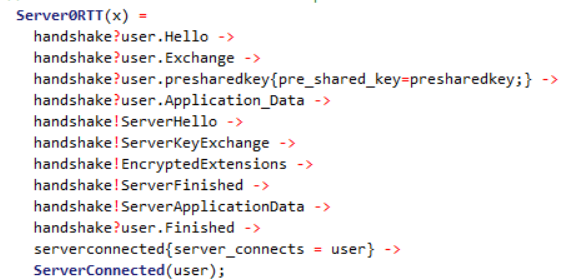
*Figure: Server handshake process*

3.7.3 0 RTT

0 RTT is a feature in TLS 1.3 where it allows a client to use a PSK to send the application data in the first flight of messages. This will reduce the latency in the connection. As the application data is sent without establishing the secure channel, the application data is not protected against replay attacks.



*Figure: Client 0RTT*



*Figure: Server 0RTT*

4. Attacker Model

4.1 DY Attack:

We verified the Dolev Yao Attacker model against our TLS 1.3 model. The attacker has the capability to eavesdrop the sensitive information of Client – Server handshake. Our model is very close to the real time working environment where the secret key is calculated on both Client and Server end. As there is strong cryptographic parameter calculation on both Client and Server end, the attacker may not be able to successfully attack. So, we conclude that TLS 1.3 is secured against DY attack. Please note that we are checking the properties of the attacker model in the properties section.

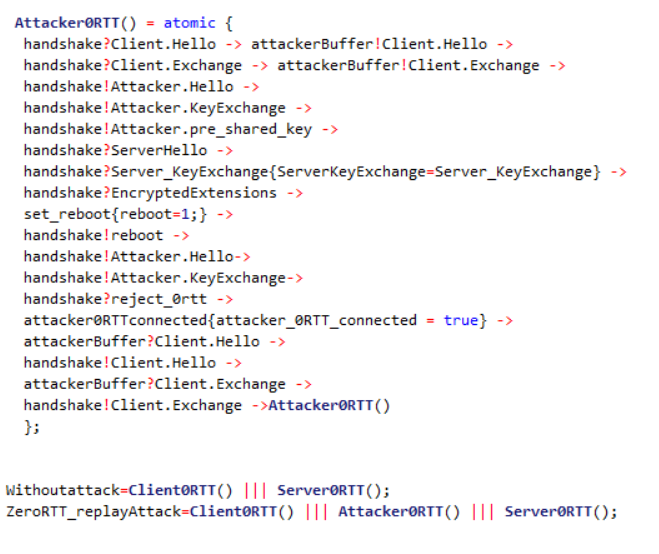


*Figure: DY attacker model*

4.2 0RTT Replay Attack:

We verified 0RTT replay attack against our TLS 1.3 model. We found that the attacker may or may not succeed against TLS 1.3 based on prerequisites. For the successful attack, the attacker has to manipulate the server to reboot itself while the client keeps waiting for response. But, in a real-world scenario, the attacker may not be successful in rebooting the server. Also, considering the scenarios where servers are deployed in distributed environment, the attack may not be successful but there could be slight interruption of the services.

In our model, we designed scenarios where the Server reboot is successful and unsuccessful. Please note that we are checking the properties of the attacker model in the properties section.



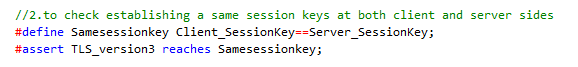
*Figure: 0RTT attack model*

5. Security Properties

Few properties handshake protocol is required to satisfy [3] –

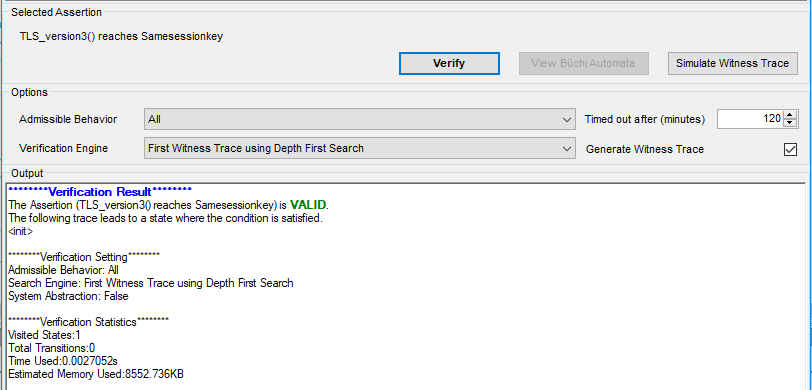
5.1 Establishing the same session keys

Upon completion of the handshake, the client and the server should have established a set of session keys on which they both agree.



*Figure: Establishing Same Session Key Check Code*

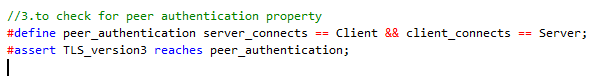
The verification result of establishing same session keys check is given in the Fig xx. The result VALID means that the client and the server established a set of session keys which they both agree.



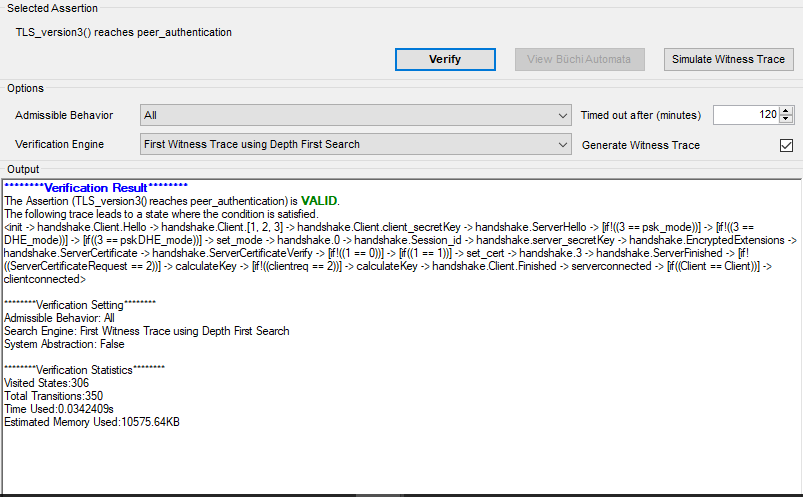
*Figure: Verification result for Establishing same session key.*

5.2 Peer Authentication

This property states that the other side of the TLS 1.3 connection is authenticated based on trusted certificates installed locally.



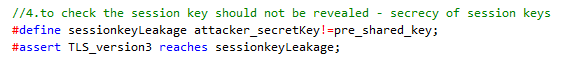
*Figure: Peer Authentication Code Check*



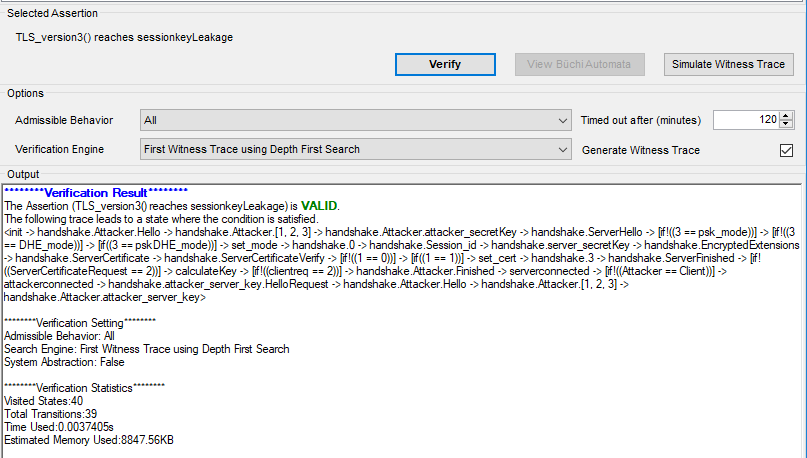
*Figure: Verification result for Peer Authentication*

5.3 Secrecy of session keys

Upon completion of handshake client and server should have established a set of session keys known only to the client and the server.



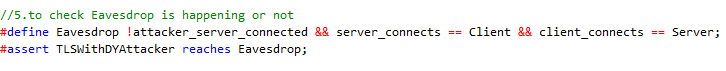
*Figure: Secrecy of Session Key Code Check*



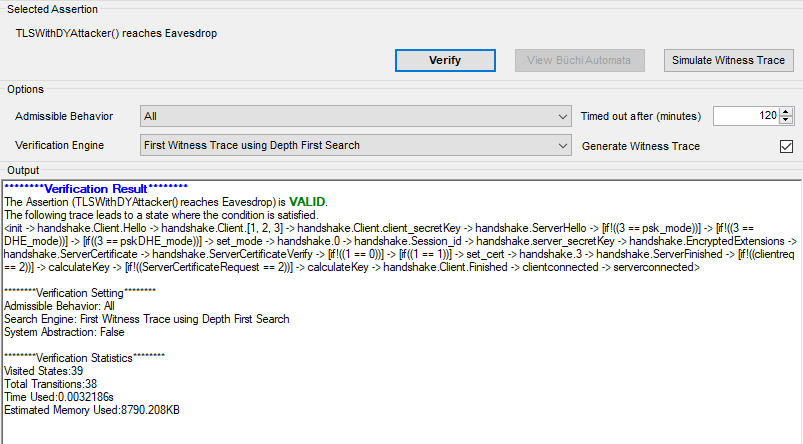
*Figure: Verification Results for Secrecy of Session Key*

5.4 Prevent Eavesdropping in DY Attack Model

Protect the content of the conversation from eavesdropping as we designed our model to check the scenario where attacker is not connected to server when the Client and Server are connected to each other. So, the verification result is shown valid which implies the attacker cannot connect and is not successful.



*Figure: Prevent Eavesdropping Code Check*

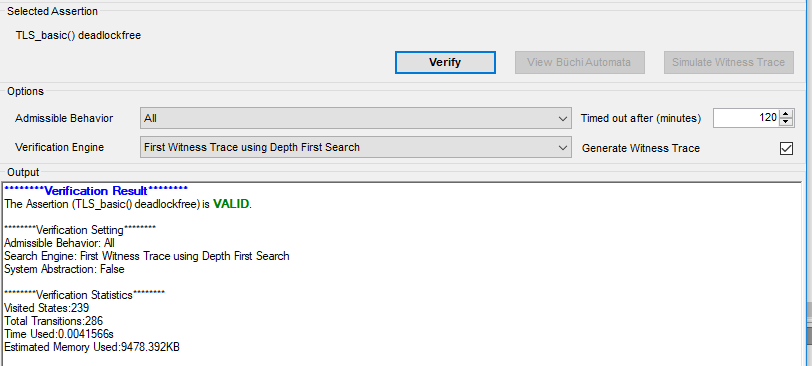


*Figure: Verification result for Prevent Eavesdropping*

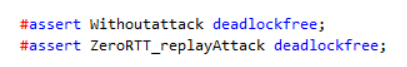
5.5 Deadlock

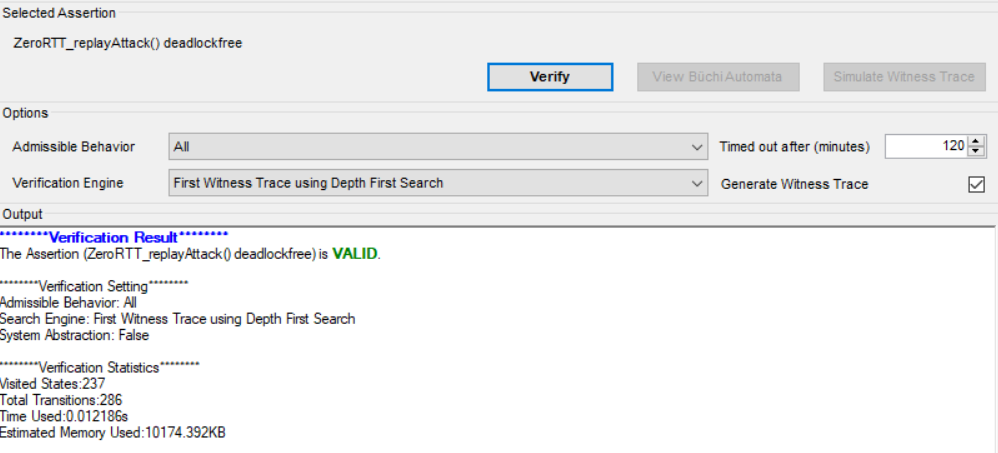
Our model is said to be deadlock free, if the client and the server are finally able to connect in finite states without continuously looping through.





*Figure: Verification result for Deadlock for TLS 1.3*





*Figure: Verification of Deadlock with Attacker*

5.6 Prevent Man in the middle attack

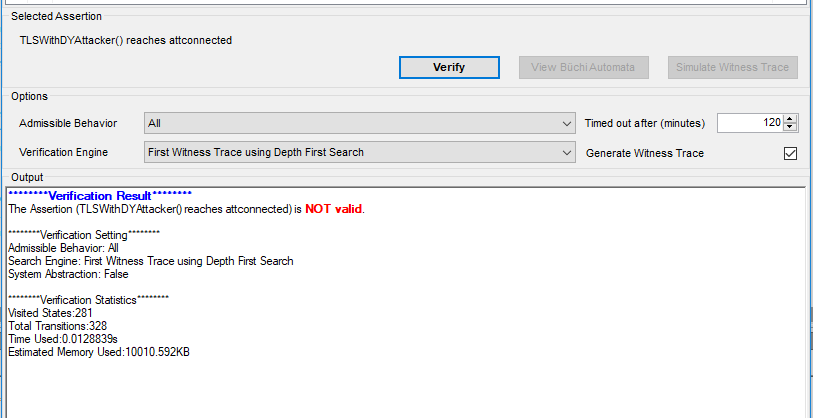
If the client and the server have already established a connection, no third party can join the connecti

on.



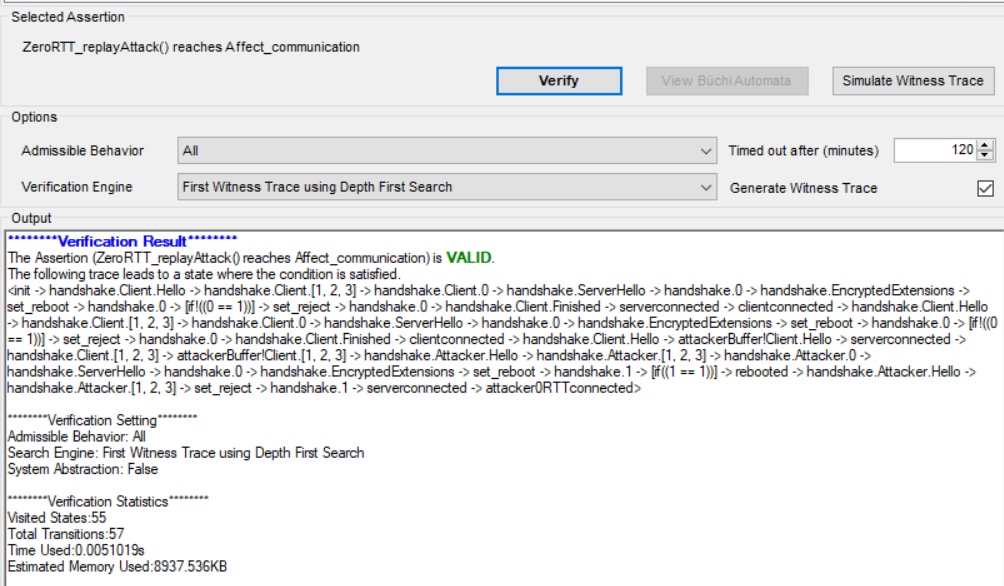
*Figure: Prevent Man in The Middle Attack Code Check*

The verification result shows NOT VALID meaning the assertion attacker is connect is false.



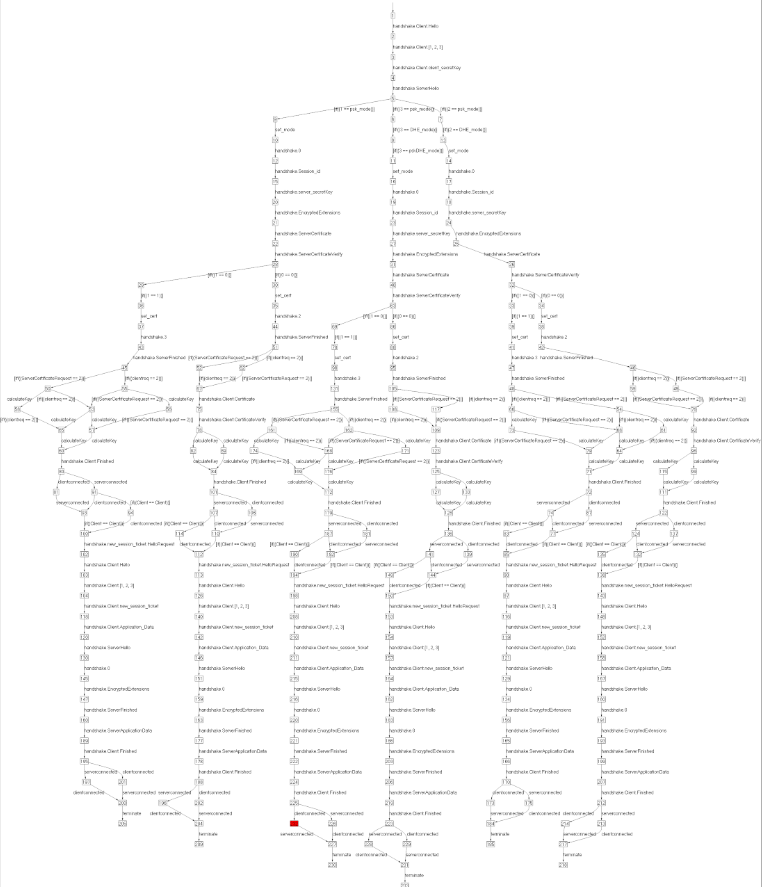
*Figure: Verification result for Man in the middle attack*

5.6 0RTT replay attack: Attacker will be able to successfully interrupt the communication, if the reboot is successful.

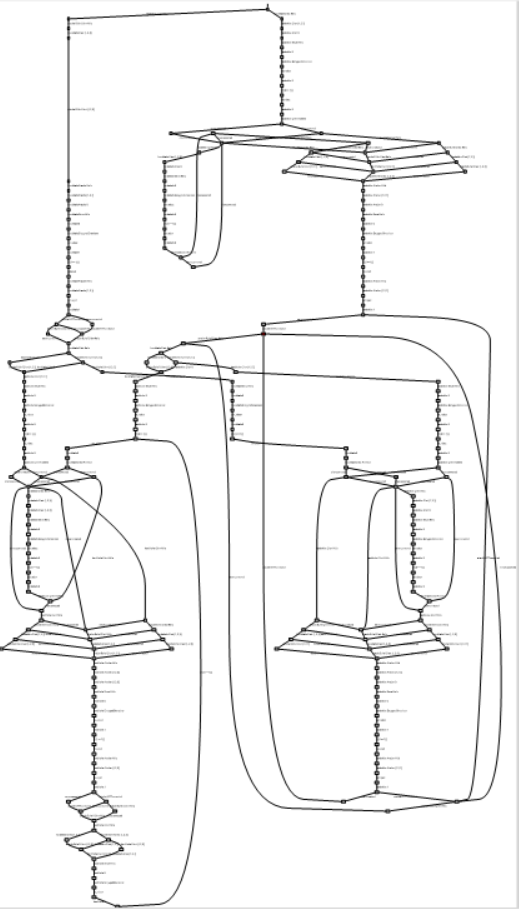


*Figure: 0RTT replay attack success if the Server is rebooted during attack.*

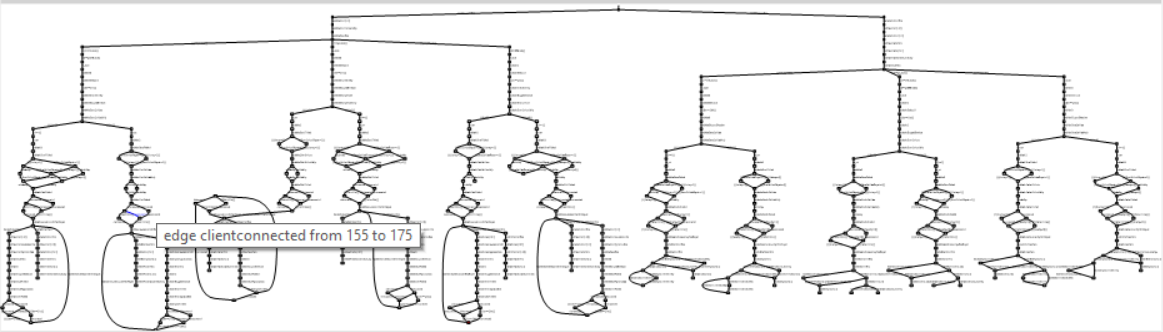
6. Results:



*Figure: Simulation of Client and Server Handshake*



*Figure: Simulation of 0RTT attack*



*Figure: Simulation of DY attack*

7. Traffic

The importance of appropriate traffic classification methods continues to grow. They are essential for effective network planning, policy-based traffic management, application prioritization, and security control. In this work, we propose a payload-based method to identify application flows encrypted with the Secure Socket Layer/Transport Layer Security (SSL/TLS) protocol, which is a fundamental cryptographic protocol suite supporting secure communication over the Internet [10]. Our approach consists of taking advantage of the information embedded in the SSL/TLS header to create statistical fingerprints of sessions to classify application traffic. We call a fingerprint any distinctive feature allowing identification of a given traffic class. In this work, a fingerprint corresponds to a first-order homogeneous Markov chain reflecting the dynamics of an SSL/TLS session. The Markov chain states model a sequence of SSL/TLS message types appearing in a single direction flow of a given application from a server to a client. We have studied the Markov chain fingerprints for four representative applications that make use of SSL/TLS: PayPal (an electronic service allowing online payments and money transfers), Facebook (an online social networking service), Instagram (a photo and video sharing social networking service and Skype (a VoIP service). The resulting models exhibit a specific structure allowing to classify encrypted application flows by comparing its message sequences with fingerprints.

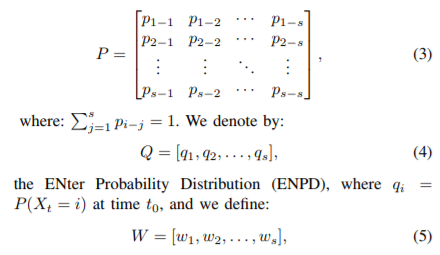
We have taken the approach based on Markov chains to model possible sequences of message types observed in single-directional SSL/TLS sessions. We have chosen a first order homogeneous Markov chain model due to its simplicity. We consider discrete-time random variable Xt for any t = t0, t1, ..., tn ∈ T. It takes values it ∈ {1, ..., s}, where it is an SSL/TLS message type (e.g. client hello). We assume that Xt is a first-order Markov chain:

P(Xt = it|Xt−1 = it−1, Xt−2 = it−2, . . . , X1 = i1) = P(Xt = it|Xt−1 = it−1) (1)

We further assume that the Markov chain is homogeneous, i.e. a state transition from time t−1 to time t is time-invariant:

P(Xt = it|Xt−1 = it−1) = P(Xt = j|Xt−1 = i) = pi−j (2)

with the transition matrix:



as the EXit Probability Distribution (EXPD), where wi represents the probability that the session finishes when it is in state i at time tn. Note that both probability distributions are independent of the Markov chain—they provide the probabilities to enter and quit the Markov chain. In traditional Markov chain models, there is an initial state and one or several absorbing states. In our case, ENPD defines the probability to enter one of the state of the Markov chain and EXPD gives the probability of quitting the Markov chain from any of its states. Based on these definitions, the probability that a sequence of states X1, . . ., XT representing a single SSL/TLS session occurs is as follows:

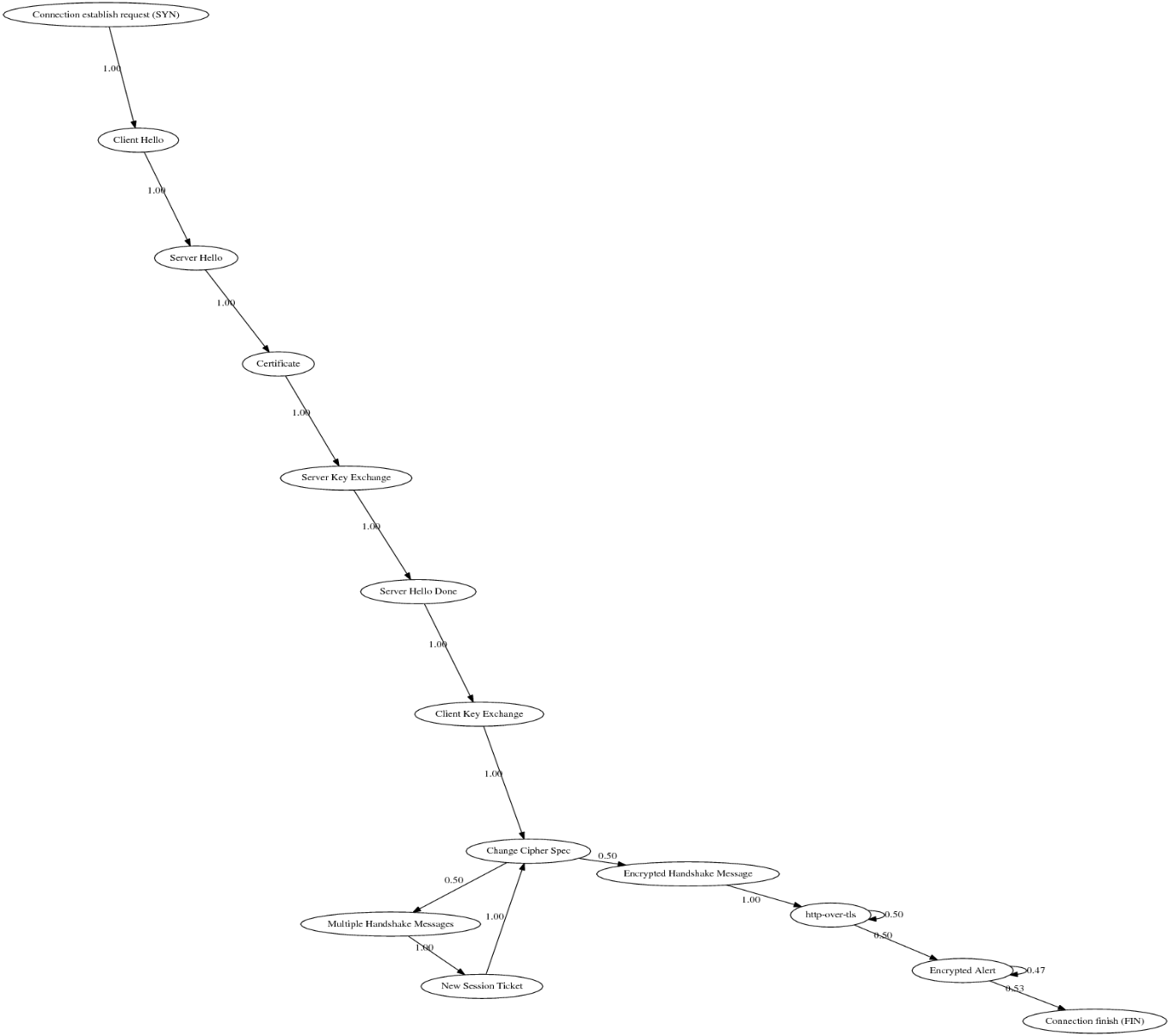


The resulting probability indicates how a given SSL/TLS sequence of message types during a session is close to a model of an application flow: a larger value means that the SSL/TLS session is closer to the model.

7.1 Data collection and graph/model creation

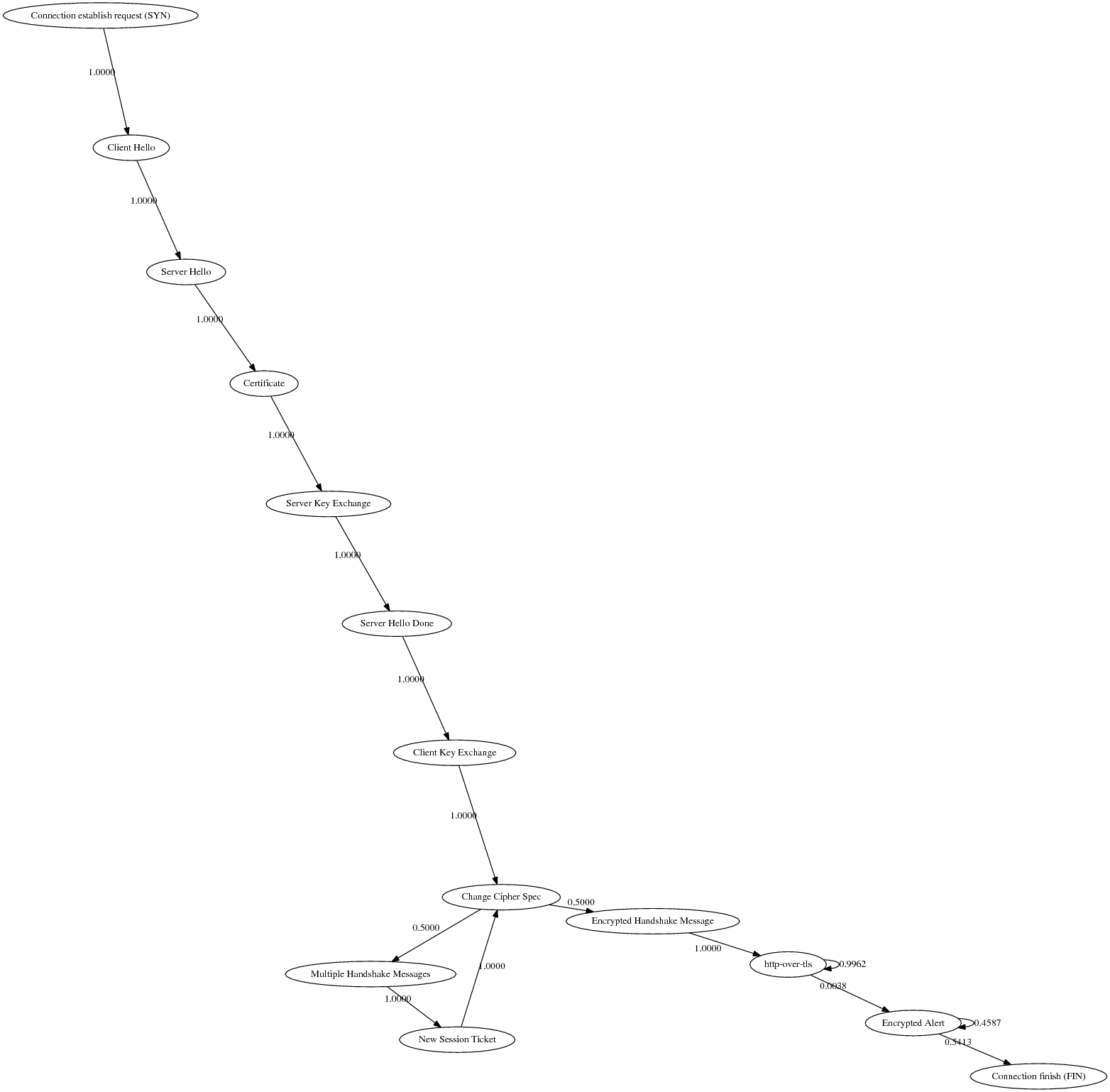
The first and the foremost task was to get the data which we need to analyse. To collect the data packets, we used Wireshark tool, which is a free and open source packet analyser. We wrote a shell script which uses Linux ‘curl’ utility command to create a session with a website for which to collect data packets. The script includes a loop which keeps on opening and closing sessions to collect multiple traces of data packets. Make sure that all other network related things are closed, like internet browsers, to only capture specific and required data packets. This creates a .pcapng packet file in Wireshark which is served as an input to the python script. The python script uses various packages to achieve different goals, namely, pyshark (to read the Wireshark generated pcapng file), JSON (to print the JSON formatted state transition dump), networkx and pydot (to create the .ps and .dot file and finally the transition plot) and pandas (to create the probability matrix). The packets are processed in a loop, and we extract various information from the packet headers like the protocol name, protocol version, layer name, sub protocol, content-type, content length, handshake protocol, session id, cipher suites and lots more. We filter out rest of the data except TLS and then calculate the number of times a state appears after a particular state. This information is stored in a 2D matrix with the rows and columns both being the TLS states which were encountered. Then we process this matrix and calculate probabilities of attaining a state after a particular state. Next, the networkx’s multidigraph() is used to create a directed graph with nodes as the states and edges containing the transition probability. write\_dot() function is used to convert networkx graph to Graphviz dot format file. Finally, we convert Graphviz file to jpeg/png file using neato utility.

These are the screenshots of the final graphs for various applications –



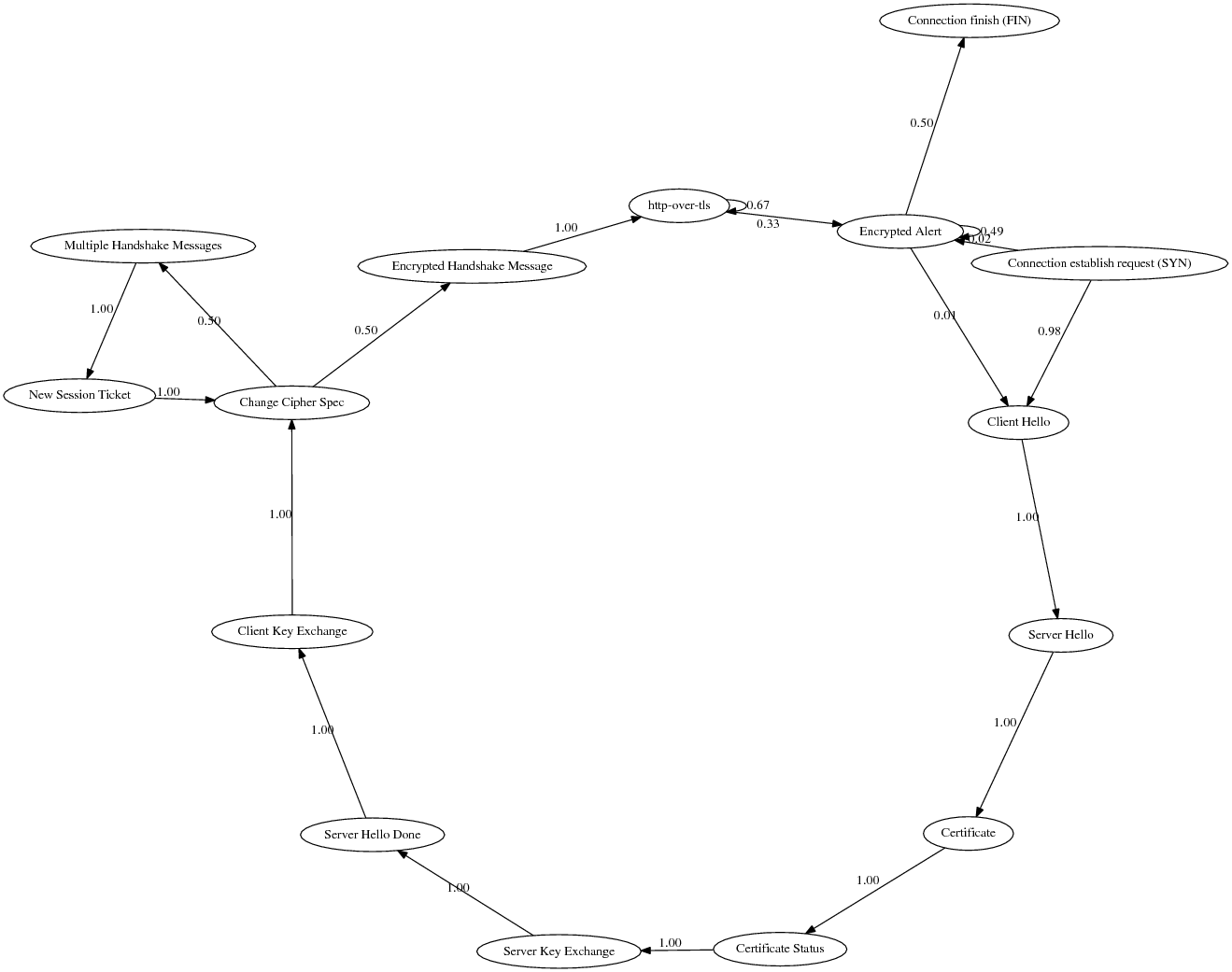
*Figure: Twitter*

For 100% of the times, client hello is followed by a server hello. After the change cipher spec, half of the times it goes to multiple handshake messages and a new session ticket issuance from thereon. The other half of the times it goes to encrypted handshake and then the application data transfer. The pattern is almost similar to the theoretical TLS implementation.



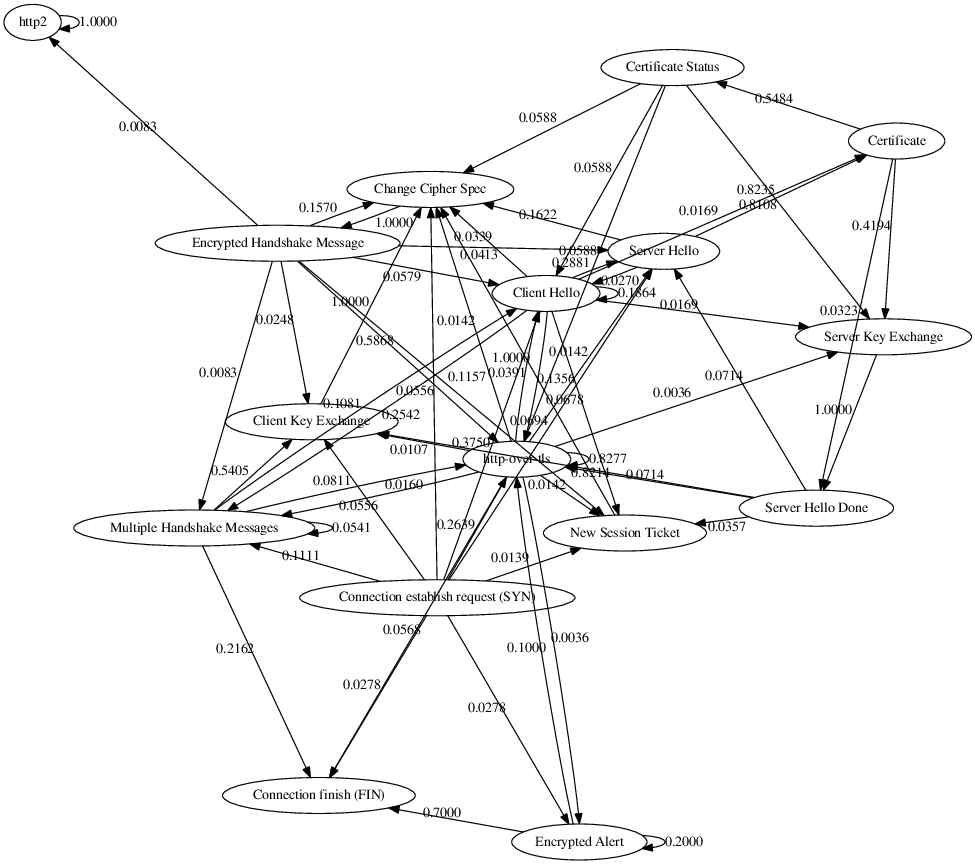
*Figure: Facebook*

The pattern is almost similar to that of twitter, with only difference being less number of encrypted alerts called amidst the application data transfer. The reason for this similarity could be because they both are social networking sites and pretty much have the same connection mechanisms, data flows and security protocols.



*Fig 15: PayPal*

This application has some unique TLS implementation, since it is a finance site and deals in financial transactions where security is of utmost importance. Till the change cipher spec state, it is similar to Facebook/Twitter. But after the encrypted alert, for 1% of the times, it goes to client hello. Also, for 2% of the times, the connection establish request directly goes to encrypted alert.

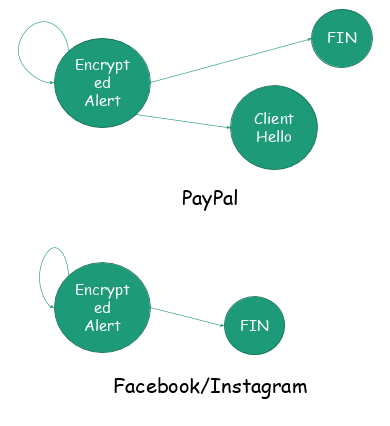
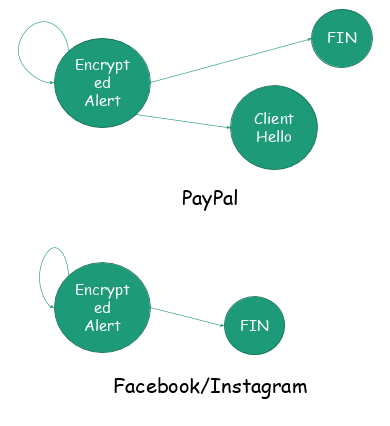


*Fig 16: Skype*

The skype implementation of TLS is completely different from other applications we have encountered so far. The reason could well be because it is a VoIP desktop application and it is a complete bundle in itself. Skype does not have any dependencies on any other libraries for its execution. It uses UDP for voice and video calls. The client hello is called from almost every other state, although for a small fraction of times. Same is the case with change cipher spec. The application data transfer is also not only done after encrypted handshake message but also after many other states in-between.

7.2 Analysis

We find that the two social networking sites Facebook and Twitter have a very similar graph. PayPal has a very peculiar graph in comparison to Facebook and Twitter. This lead to conclusion that PayPal being a Financial site enforces higher security. The PayPal graph sends ClientHello after EncryptedAlert message which is not seen in Facebook/Twitter. According to various online references EncryptedAlert is sent when a client is not able to decrypt the server sent data. Facebook/Twitter when unable to decrypt the data sent by server close the TCP connection with FIN. Where as PayPal tries to re establish the connection again by sending ClientHello. This may be due to the fact to maintain Atomicity of transaction. i.e. to prevent a transaction from aborting in middle.

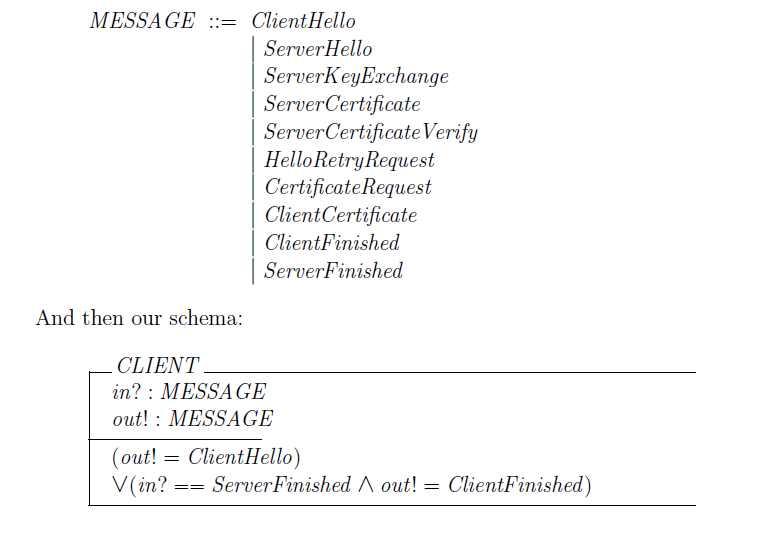


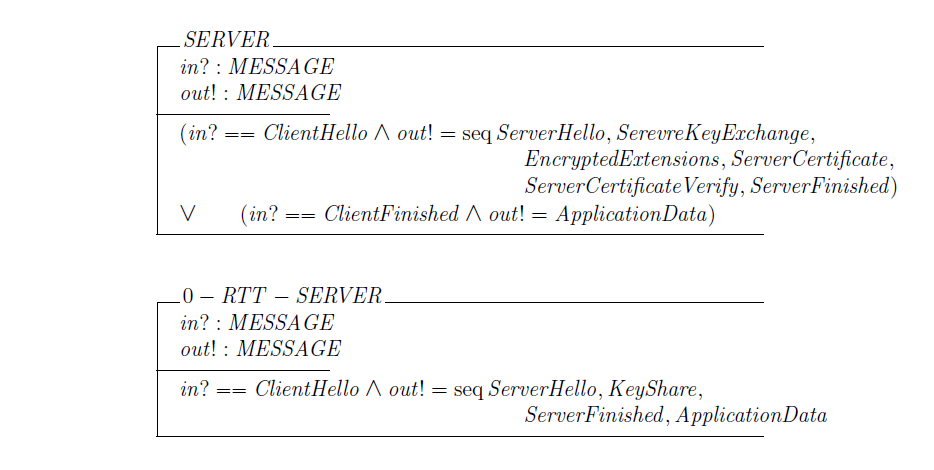
*Fig 17: PayPal vs Facebook/Twitter*

We also see that Skype does not exactly follow the actual TLS sequence, it has its own sequence. This may be due to the fact Skype is securing Voice data. Also, Skype uses its own version of propriety protocols/ TLS version

8. Z Modelling

We have modelled the TLS 1.3 Handshake - Client, Server and 0-RTT – Client and Server.







9. Conclusion

We built a symbolic model by translating the specifications of the TLS 1.3 handshake using PAT language. The protocol design as such is in the later stage of development and the future revisions will not make any major changes to the core design of the handshake. This report analysed the protocol in its current revision and it aimed to further strengthen the confidence of the security guarantees of different handshake modes. Our symbolic model is abstract and the actual implementations of the protocol, might be exposed to a variety of potential flaws. The results of our 0-RTT analysis shows that replay attacks are plausible on the data sent on the client’s first flight. Therefore, it would not be secure to send authentication credentials in that round. Our final model uses pre-shared keys established in former DH handshakes in the current handshake. Our analysis ratifies that session resumption indeed satisfies all its intended security properties and it successfully binds to the context and authentication of the prior handshake to establish a new connection. Our results also establish that it is impossible for a DY attack in the current design, as the client’s signature contains the server’s Finished message. We can also conclude that if the client authenticates itself in at least one handshake, then the secrecy or PFS is preserved. The analysis of the symbolic model also formally proves that each sub-mode satisfies every intended security property. The protocol let’s participants choose among the different modes of execution and applications running on top of TLS must be vary of the different security guarantees, as it could otherwise lead to inadvertent weaknesses. For example, the server can directly send application data after the Finished message, when there is no authentication message of the client yet. In summary, the formal analysis of the TLS 1.3 asserts that the protocol design is secure in the symbolic model and every mode satisfies the properties it aims for.

10. References:

1. <https://tls13tamarin.github.io/TLS13Tamarin/docs/tls13tamarin-draft21.pdf>

2. <https://tlswg.github.io/tls13-spec/draft-ietf-tls-tls13.html>

3. <https://eprint.iacr.org/2017/082.pdf>

4. <https://github.com/tlswg/tls13-spec/issues/1001>

5. <https://en.wikipedia.org/wiki/Dolev%E2%80%93Yao_model>

6. <https://cseweb.ucsd.edu/classes/sp05/cse208/lec-dolevyao.html>