NTNU
Norwegian University of
Science and Technology

Analysis of the Transport Layer Security protocol

Tia Helene Firing

Master of Science in Physics and Mathematics
Submission date: June 2010
Supervisor: Kristian Gjøsteen, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

In this master thesis, the student is supposed to analyse the TLS protocol by use of the UC (Universal Composability) security framework. The analysis should focus on the Handshake protocol, with particular emphasis on the recently discovered renegotiation attack and the fix proposed by IETF.

Assignment given: 14. January 2010
Supervisor: Kristian Gjøsteen, MATH

Abstract

In this master thesis we have presented a security analysis of the TLS protocol with particular emphasis on the recently discovered renegotiation attack. From our security proof we get that the Handshake protocol with renegotiation, including the fix from IETF, is secure, and hence not vulnerable to the renegotiation attack anymore.

We have also analysed the Handshake protocol with session resumption, and the Application data protocol together with the Record protocol. Both of these protocols were deemed secure as well.

All the security proofs are based on the UC (Universal Composability) security framework.

Preface

When choosing a subject for my master thesis, it was important to me to write about something practical, something that is used in more applied information security, so when my supervisor suggested that I analysed TLS based on the UC security framework, I said yes right away. Not only is TLS one of the most popular protocols within web security, someone had also discovered a serious security flaw in one of its subprotocols just a few months earlier. IETF published a fix for this new attack against TLS in January. However, when I started working on this thesis, no other UC theory based security analyses of TLS including this fix had been published yet, as far as I know. This gave me the opportunity to analyse a protocol that had not been analysed before (again, as far as I know), and it has made working on this master thesis even more motivating, exciting and fun.

For me, writing this master thesis has been challenging, fun, frustrating and rewarding, all at the same time. I have learned a lot about UC theory and how to use it to prove security in cryptographic protocols. I have also learned a lot about TLS and how and why it works. In addition, I now know the entire Handshake and Record protocols by heart. This past semester truly seems like the perfect way to end my five years at NTNU.

Of course, I could not have done this without the help and support from others. I would sincerely like to thank my supervisor, associate professor Kristian Gjøsteen. Thank you for all the help, answers and support you have given me, for sharing your knowledge of LaTeX, and for all the motivating and interesting conversations about cryptography and security in general. I would also like to thank Cato, my better half, for your sincere interest in my work, and for supporting me and being there for me.

Contents

Chapter 1

Introduction

This master thesis provides a security analysis of the Handshake protocol from TLS, including the renegotiation and session resumption features. We will also look at TLS's Application data and Record protocols. The analysis will be based on the UC security framework developed by Canetti[1].

TLS is a very popular and commonly used protocol in web security, and it has already been analysed several times. However, in September 2009 a serious security flaw was discovered in the renegotiation feature in the Handshake protocol. The Handshake protocol allows the communicating parties to renegotiate a session, but it does not tie these two sessions together cryptographically. This is something an attacker can take advantage of to perform a man-in-the-middle attack, where a harmful request from the attacker will be treated as a prefix to a legitimate client's request. We will describe the attack in more detail in Chapter 4.3. The designers of the TLS standard, the IETF (Internet Engineering Task Force), has proposed a fix to prevent this vulnerability[7] that we present in Chapter 4.4.

One of our main goals in this master thesis is to analyse the Handshake protocol including this fix when we allow renegotiation. If we can give a successful security proof based on the UC theory of this protocol, it will be a good indication that the fix will solve the problem at hand, and that including it does not create any new security issues. Of course, since this is a theoretical proof we can not guarantee that the protocol is secure against all possible attacks, but it will be secure against many attacks.

To make the analysis more complete, we will also do similar security proofs for the Handshake protocol with session resumption and for the Application data protocol. In our analysis of the Application data protocol we will also include the Record protocol, which we don't use in any of the handshake proofs.

The outline of the rest of this paper is as follows: We begin with presenting the basics of the UC theory, and we design ideal functionalities for authentication (which is what the Handshake protocol does when we don't use the Record protocol) and for sending

and receiving messages in a secure manner. Then we will take a closer look at TLS and its subprotocols, and we will describe the attack and the fix. After that we are ready to dive into the most interesting part: The security proofs.

Chapter 2

UC theory

There are many different standards and suggestions when it comes to evaluating the security of a cryptographic protocol. We will use the UC (Universal Composability) security framework developed by Ran Canetti [1]. The basic idea of the UC security framework is to compare the actual protocol to the ideal case. The ideal case would be to use a trusted third party (TTP) instead of the protocol. The TTP would serve the same purpose as the protocol, but if we use a TTP we can control what kind of influence the attacker is allowed to have. In this scenario, the attacker can communicate with the TTP, but we decide what kind of information the attacker gets access to. A model of this situation is what we call an *ideal functionality* for the protocol. If we look at a key exchange protocol, the ideal case would be a TTP that simply gave both parties the same key and never shared the key with anyone else. We will present some examples of ideal functionalities in Chapter 2.3.

However, if we want to prove that the protocol is secure, it should not be possible to distinguish between a situation where the actual protocol is being used and one where the participants use a TTP. This means that any effect that you can see when running the protocol is supposed to be visible when using the TTP as well, so it will not be realistic to deny the attacker all knowledge. At least he should be notified that something is happening, because an attacker who is monitoring the network traffic will usually be able to tell if, for instance, two parties are performing a key exchange.

We will now define some important concepts used in this framework. See Figure 2.1 to understand how they are connected. P_X and P_Y are protocol machines. They are both running one or more instances of the protocol we are interested in (or some subprotocol of it). We can have many such protocol machines. $\mathcal{Z}$ is called the environment. $\mathcal{Z}$ represents everything that is outside the protocol machines, for instance other programs or even human users. $\mathcal{A}$ is the attacker. P_X, P_Y, $\mathcal{Z}$ and $\mathcal{A}$ are modelled as interactive Turing machines. This means that they take some input and produce some output, in addition to being able to communicate with other such machines.

The protocol machines receive instructions from the environment, and give the results back to $\mathcal{Z}$. P_X and P_Y communicate via the attacker. $\mathcal{Z}$ can give the attacker information, and the attacker can try to influence the environment in some way. The situation in Figure 2.1 can be seen as an abstract version of how the protocol works in reality.

We don't know exactly what $\mathcal{Z}$ and $\mathcal{A}$ do. To make it easier to work with this model without losing security, we can define a dummy attacker, $\mathcal{D}$. The dummy attacker can only forward messages. It is always possible to replace the (modelled) real attacker $\mathcal{A}$ with the dummy attacker by redefining the environment. Now there is only one unknown in the scheme: $\mathcal{Z}$.

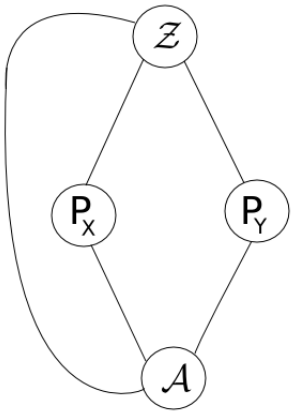

Figure 2.1: Illustration of how the protocol machines P_X and P_Y, the environment $\mathcal{Z}$ and the attacker $\mathcal{A}$ interact.

To model the ideal situation, we design ideal functionalities. An ideal functionality ($\mathcal{F}$) is also modelled as an interactive Turing machine. As illustrated in Figure 2.2, it will receive some input from a protocol machine (or several protocol machines), and produce some output to this or another protocol machine. It is also able to communicate with both the protocol machines and the attacker. In the ideal situation we use $\mathcal{S}$ to represent the attacker. $\mathcal{S}$ is called the simulating attacker. The protocol machines in Figure 2.2 are dummy protocol machines. This means that they can only forward messages. It is modelled this way because there are no protocols, only the trusted third party (represented by the ideal functionality), but the environment expects to use protocols.

We let the environment output some values, 1 or 0. It is usually best not to put any meaning into it. We use this output to compare protocols when proving security: We study the output from the environment when using the actual protocol and the output in the ideal case (using the same input), and see if there are any statistical relationships (for instance, the output could be more likely to be 1 when using the protocol than when using the ideal functionality). If it is impossible to determine if we are using the actual protocol (Figure 2.1) or the ideal functionality (Figure 2.2) given only the output from $\mathcal{Z}$, we say that the actual protocol UC-realizes the ideal functionality. This means that

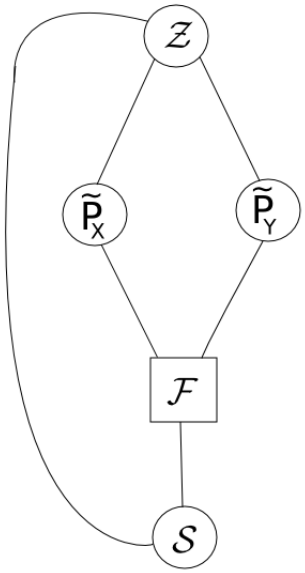

Figure 2.2: Illustration of how the ideal functionality $\mathcal{F}$, the dummy protocol machines $\tilde{P_X}$ and $\tilde{P_Y}$, the environment $\mathcal{Z}$ and the simulating attacker $\mathcal{S}$ are connected.

the protocol is secure.

So, to summarize: To prove security in a protocol, we must first make an abstract version of the protocol. Then we have to define an ideal functionality for the protocol using a trusted third party and the concepts from UC theory. Finally we must prove that the protocol realizes the ideal functionality.

Sometimes it is useful to let the parties use ideal functionalities in addition to regular protocols. Such a protocol is called a *hybrid protocol*. The ideal functionality will act as a separate subprotocol, and solve one particular challenge that is a part of the main protocol.

We let π and ρ be two protocols. $\pi[\rho]$ means that π uses ρ as a subprotocol. The composition theorem from [1] says that if we have two protocols (π, ρ), and an ideal functionality $\mathcal{F}$ where ρ UC-realizes $\mathcal{F}$, then $\pi[\rho]$ UC-realizes $\pi[\mathcal{F}]$. We will not prove this, see [1] for the complete proof. We can use this result to simplify the proof of a complicated protocol by separating the subprotocols and create ideal functionalities for them, and prove each subprotocol separately before combining them.

2.1 Games

A *game* consists of some interactive, probabilistic machines and the probability that some security related event E will occur [8]. The game will give 0 or 1 as output, and can be

seen as a probability space over $\{0,1\}$. These interactive, probabilistic machines will in this context usually be the environment $\mathcal{Z}$, the dummy adversary $\mathcal{D}$ and the protocol machines. The output comes from the environment.

We can prove security in a cryptosystem by making a sequence of games, where each game is only slightly different from the previous game (the difference in the probability of the output from $\mathcal{Z}$ is negligible)[8]. If we have n games where Game 1 is the actual cryptoprotocol and Game n is the ideal functionality for this cryptoprotocol, we can define E_i to be the event that $\mathcal{Z}$ outputs 1 in Game i. If $|\Pr[E_{i+1}] - \Pr[E_i]|$ is very small for each $i \in \{1, ..., n-1\}$, then the difference between the output from $\mathcal{Z}$ when using the protocol is only negligibly different from the output from $\mathcal{Z}$ when using the ideal functionality. If this is the case, then the protocol UC-realizes the ideal functionality (the two situations are indistinguishable). Hence the protocol is secure.

2.2 Useful definitions from probability theory

To complete our analysis of TLS we will need to use some concepts from probability theory. From the previous section it is clear that *distinguishing problems* are important here. A distinguishing problem is a triple (S, X_0, X_1) where S is a set of instances and X_0 and X_1 are probability spaces over S, and given some data the task is to decide which probability space the data most likely came from. A distinguishing problem is a special case of a *problem*, which can be defined as a triple (S_1, S_2, X) where S_1 is a set of instances, S_2 is a set of solutions, and X is a probability space over $S_1 \times S_2$.

We can define an algorithm A that solves the (distinguishing) problem. A has solved the distinguishing problem if it guesses correctly which distribution some data came from. The definition of A's success probability for solving a general problem is

$$\mathrm{Succ}_P(A) = \Pr[z = s_2 | (s_1, s_2) \xleftarrow{r} X, z \leftarrow A(s_1)]$$

Since we want A to distinguish between two probability spaces, it would be right half of the time just by guessing. We define A's *advantage* against the distinguishing problem to be

$$\begin{aligned}\mathrm{Adv}_P(A) &= |\mathrm{Succ}_P(A) - \frac{1}{2}| \\ &= \frac{1}{2}|\Pr[A(x) = 1 | x \xleftarrow{r} X_1] - \Pr[A(x) = 1 | x \xleftarrow{r} X_0]|\end{aligned}$$

We are usually interested in finding the advantage the attacker $\mathcal{A}$ has against some protocol. This can be defined as

$$\mathrm{Adv}_{\mathcal{A}} = |\Pr[E_1] - \Pr[E_n]|$$

where E_i is the event that $\mathcal{Z}$ outputs 1 in Game i ($i = 1$ is the actual protocol and $i = n$ is the ideal functionality), as defined in the previous section.

2.3 Examples of ideal functionalities

We will now present some examples of ideal functionalities. We will look at ideal functionalities for signatures, $\mathcal{F}_{\text{SIG}}$, and public-key encryption (RSA), $\mathcal{F}_{\text{CPKE}}$. We have borrowed $\mathcal{F}_{\text{SIG}}$ from Canetti [1] and $\mathcal{F}_{\text{CPKE}}$ from Herzog and Canetti[2]. We will also define an ideal functionality for the TCP network, $\mathcal{F}_{\text{NET}}$. This functionality does not have any cryptographic content, it is more like an abstract version of the TCP network, but where we have the ability to control the attacker's influence and behaviour. Later, we will use these ideal functionalities to simplify the security proofs of TLS.

2.3.1 Ideal functionality for signatures

A signature is the output of a function that takes a message and a private signing key as input[10]. Since the signing key is private, only the owner of the key should be able to produce the signature, and it should not be possible to change the message after it has been signed (then the signature check should fail). Hence, the signature ties the message to the participant who signed it. Anyone can verify the signature using a verification algorithm that takes the message, the signature and the sender's public verification key as input, and gives "true" or "false" as output.

Below we have the ideal functionality for signatures, $\mathcal{F}_{\text{SIG}}$, that we have borrowed from Canetti [1]:

$\mathcal{F}_{\text{SIG}}$:

Key Generation: Upon receiving a value $(\texttt{keygen}, sid)$ from some party P, verify that $sid = (P, sid')$ for some sid'. If not, then ignore the request. Else, hand $(\texttt{keygen}, sid)$ to the adversary. Upon receiving $(\texttt{algs}, sid, s, v)$ from the adversary, where s is a description of a PPT ITM, and v is a description of a *deterministic* polytime ITM, output $(\texttt{ver} - \texttt{alg}, sid, v)$ to P.

Signature Generation: Upon receiving a value $(\texttt{sign}, sid, m)$ from P, let $\sigma = s(m)$, and verify that $v(m, s) = 1$. If so, then output $(\texttt{signature}, sid, m, \sigma)$ to P and record the entry (m, σ). Else, output an error message to P and halt.

Signature Verification: Upon receiving a value $(\texttt{verify}, sid, m, \sigma, v')$ from some party V, do: If $v' = v$, the signer is not corrupted, $v(m, \sigma) = 1$, and no entry (m, σ') for any σ' is recorded, then output an error message to P and halt. Else, output $(\texttt{verified}, sid, m, v'(m, \sigma))$ to V.

ITM stands for "interactive Turing machine", and PPT is short for "probabilistic polynomial time".

This ideal functionality is only designed for one session. If we want to use it for multiple sessions and multiple parties, we let each party use its own independent instance of $\mathcal{F}_{\text{SIG}}$

to sign messages. We can add one instance of $\mathcal{F}_{\text{SIG}}$ for each party without losing security by using the UC theorem repeatedly (once for each time we add an instance of $\mathcal{F}_{\text{SIG}}$).

If some party P_1 signed a message with its $\mathcal{F}_{\text{SIG}}^{(1)}$, then the receiving party P_2 has to use $\mathcal{F}_{\text{SIG}}^{(1)}$ to verify the signature. If P_2 tried to use its own $\mathcal{F}_{\text{SIG}}^{(2)}$ to verify P_1's signature, it would fail since $\mathcal{F}_{\text{SIG}}^{(1)}$ and $\mathcal{F}_{\text{SIG}}^{(2)}$ are independent of each other, and they don't know what the other has signed.

Canetti's session ID, *sid*, consists of the identity of the party and something more (sid') that can be anything. Throughout this analysis we will use the identity of the party as the input corresponding to *sid*, but we will not specify what we use as sid'. It does not really matter what sid' is, so we will just let it be something that is included in the identity of the party.

2.3.2 Ideal functionality for public-key encryption

Public-key encryption (like RSA) allows two parties to communicate securely without exchanging keys first[10]. Both parties, let's call them P_X and P_Y, have a public encryption key that is used to encrypt messages, and a private decryption key to decrypt received messages. When P_X wants to send a message to P_Y, P_X encrypts the message using P_Y's public encryption key. When P_Y receives the encrypted message, it will decrypt it using its private decryption key. As long as the private key is kept secret, P_Y is the only one that is able to decrypt the message. To not be forced to exchange keys before communicating can be very useful in many situations. However, public-key encryption and decryption operations are usually resource consuming and slow compared to the operations in traditional symmetric cryptosystems, so it is often used just to establish a symmetric key.

The public key of a participant is usually a part of that participant's certificate. Since it can be quite cumbersome to model certificates and certificate authorities, we can say that the participant sends its ID instead of its certificate. The ID itself does not contain the public key, but we can use it to ask for the public key. Canetti and Herzog [2] has defined an ideal functionality for certified public-key encryption that allows us to encrypt a message only knowing the ID of the receiver.

$\mathcal{F}_{\text{CPKE}}$:

$\mathcal{F}_{\text{CPKE}}$ proceeds as follows, when parameterized by message domain M, a formal encryption function E with domain M and range $\{0,1\}^*$, and a formal decryption function D of domain $\{0,1\}^*$ and range $M \cup \texttt{error}$. The SID is assumed to consist of a pair $SID = (PID_{\text{owner}}, SID')$, where PID_{owner} is the identity of a special party, called the owner of this instance.

Encryption: Upon receiving a value $(\texttt{Encrypt}, SID, m)$ from a party P, where $SID = (PID_{\text{owner}}, SID')$ proceed as follows:

1. If this is the first encryption request made by P then notify the adversary that P made an encryption request.
2. If $m \notin M$ then return an error message to P
3. If $m \in M$ then:
   - If PID_{owner} is corrupted, then let $ciphertext \leftarrow \text{E}_k(m)$
   - Otherwise, let $ciphertext \leftarrow \text{E}_k(1^{|m|})$

   Record the pair (m, c) and return c.

Decryption: Upon receiving a value $(\texttt{Decrypt}, SID, c)$, with $SID = (PID_{\text{owner}}, SID')$, from PID_{owner}, proceed as follows. (If the input is received from another party then ignore).

1. If this is the first decryption request made then notify the adversary that a decryption request was made.
2. If there is a recorded pair (c, m), then hand m to PID_{owner}. (If there is more than one value m that corresponds to c then output an error message to PID_{owner}.)
3. Otherwise, compute $m = \text{D}(c)$, and hand m to PID_{owner}.

This ideal functionality is also just designed for one session. If we want to use it for multiple sessions and multiple parties, we let each party use its own independent instance of $\mathcal{F}_{\text{CPKE}}$, just like we did for $\mathcal{F}_{\text{SIG}}$.

2.3.3 Ideal functionality for the TCP network

TCP (Transmission Control Protocol)[11] is a network protocol that operates on top of the IP layer, and we use it to send and receive messages. TCP guarantees that all messages will be delivered, and that they are delivered in the correct order. The sender will be notified whenever a message is received. However, this will only hold as long as there is no active attacker. None of these features are protected cryptographically, so if an attacker wants to modify the delivery order and send fake message receipts, he can.

TCP allows us to create a direct "tunnel" between two instances of two protocol machines. This means that each time the protocol machine P_X sends a message to P_Y using the

same TCP connection identifier *coID*, the ideal functionality will ask $\mathcal{S}$ to deliver the message from $P_{X,i}$ to $P_{Y,j}$ (the same instances are used every time). Whether the attacker actually chooses to do this is a different question.

The connection ID, *coID*, is chosen by the attacker and has to be unique.

$\mathcal{F}_{\text{NET}}$:

On input $(\texttt{connect}, P_Y)$ from $P_{X,i}$:

1. Send $(\texttt{connect}, P_{X,i}, P_Y)$ to $\mathcal{S}$

On input $(\texttt{connect}, coID, P_{X,i}, P_Y)$ from $\mathcal{S}$:

1. Choose available instance $P_{Y,j}$
2. Record $(coID, P_{X,i}, P_{Y,j})$
3. Send delayed output $(\texttt{connect}, coID, P_X, P_Y)$ to $P_{X,i}$ and $P_{Y,j}$

On input $(\texttt{send}, coID, P_Y, m)$ from P_X:

1. If recorded $(coID, P_{Y,j}, P_{X,i})$ or $(coID, P_{X,i}, P_{Y,j})$, send $(\texttt{send}, coID, P_{Y,j}, P_{X,i}, m)$ to $\mathcal{S}$

On input $(\texttt{receive}, coID, P_X, P_Y, m)$ from $\mathcal{S}$:

1. If recorded $(coID, P_{X,i}, P_{Y,j})$ or $(coID, P_{Y,j}, P_{X,i})$, send $(\texttt{receive}, coID, P_X, m)$ to $P_{Y,j}$

In $\mathcal{F}_{\text{NET}}$ we let the attacker be in complete control of the network. Two parties can not begin a connection without the attacker's approval. The attacker is also in charge of delivering the messages, so he can do pretty much whatever he wants: Delay messages, change the contents, replay messages, send messages to other parties than the intended receiver or not deliver the message at all.

Chapter 3

Ideal functionalities

As we will see in Chapter 4, the main tasks of TLS is to establish a secure session between the participants with optional authentication, and sending and receiving messages in a secure way. In this chapter we will define general ideal functionalities for different levels of authentication and for sending and receiving messages. We will need these ideal functionalities in the security proofs of TLS.

3.1 Ideal functionality for authentication

When designing an ideal functionality for authentication, what we need is just a trusted third party that can verify that the participants are who they claim to be. To make the ideal functionality realistic, however, we need to allow the attacker the same knowledge and influence as if the participants used an actual authentication protocol. If the parties were using an actual authentication protocol, the attacker would usually know which parties were trying to authenticate themselves. Since we use the TCP network (modelled as $\mathcal{F}_{\text{NET}}$), the attacker can delay messages, so he would also be able to choose when the two parties should finish the protocol.

We let the authentication happen between two protocol machines, P_C and P_S. We design the ideal functionality $\mathcal{F}_{\text{TLS}-\text{AUT}}$ for three different kinds of authentication. When the functionality receives a message marked `establish`, we let the attacker pick a unique session ID (*SID*), and we authenticate P_S. P_C's identity will not be verified, which we illustrate like this: $\hat{P_C}$. The *tsid*, a temporary session ID chosen by $\mathcal{F}_{\text{TLS}-\text{AUT}}$, ties the parties and the session together with *SID*, in case the environment wants to run the establish part more than once between P_C and P_S at the same time. *tsid* is only used in the communication between the ideal functionality and the attacker.

If the functionality receives a message of the type `resume` or `reneg`, the parties must have finished the establish part successfully first. If the functionality receives a message

marked `resume`, we want it to resume an earlier connection that had this session ID. P_S will still be authenticated, and P_C is still anonymous. If the functionality receives a `reneg` message, we want to begin a new session where P_C is authenticated as well. Since we already have a unique session ID, *SID*, we don't need a *tsid* in the reneg and resume parts.

$\tilde{P_C}$ and $\tilde{P_S}$ are dummy protocol machines for P_C and P_S that only forward what they receive.

$\mathcal{F}_{\text{TLS–AUT}}$:
On input $(\texttt{establish}, \hat{P_C}, P_S)$ from $\tilde{P_C}/\mathcal{Z}$:

1. Choose unique, random *tsid*
2. Send $(\texttt{establish}, P_C, P_S, tsid)$ to $\mathcal{S}$
3. On $(\texttt{ok}, tsid, SID, P_S)$ from $\mathcal{S}$, *SID* is new, send $(\texttt{ok}, SID, \hat{P_C}, P_S)$ to $\tilde{P_S}$
4. On $(\texttt{ok}, tsid, SID, P_C)$ from $\mathcal{S}$, send $(\texttt{ok}, SID, P_C, P_S)$ to $\tilde{P_C}$
5. Store $(P_C, P_S, SID, 0)$

On input $(\texttt{reneg}, SID, \hat{P_C}, P_S)$ from $\tilde{P_S}/\mathcal{Z}$:

1. If $(P_C, P_S, SID, 0)$ not stored, stop.
2. Send $(\texttt{reneg}, SID, P_C, P_S)$ to $\mathcal{S}$
3. On $(\texttt{reneg} - \texttt{ok}, SID, SID_r, P_S)$ from $\mathcal{S}$, send $(\texttt{reneg} - \texttt{ok}, SID_r, P_C, P_S)$ to $\tilde{P_S}$
4. On $(\texttt{reneg} - \texttt{ok}, SID, SID_r, P_C)$ from $\mathcal{S}$, send $(\texttt{reneg} - \texttt{ok}, SID_r, P_C, P_S)$ to $\tilde{P_C}$
5. Store $(P_C, P_S, SID_r, 1)$, discard $(P_C, P_S, SID, 0)$

On input $(\texttt{resume}, SID, \hat{P_C}, P_S)$ from $\tilde{P_C}/\mathcal{Z}$:

1. If $(P_C, P_S, SID, 0)$ not stored, stop.
2. Send $(\texttt{resume}, SID, P_C, P_S)$ to $\mathcal{S}$
3. On $(\texttt{resume} - \texttt{ok}, SID, P_S)$ from $\mathcal{S}$, send $(\texttt{resume} - \texttt{ok}, SID, \hat{P_C}, P_S)$ to $\tilde{P_S}$
4. On $(\texttt{resume} - \texttt{ok}, SID, P_C)$ from $\mathcal{S}$, send $(\texttt{resume} - \texttt{ok}, SID, P_C, P_S)$ to $\tilde{P_C}$

The 0 that is stored together with P_C, P_S and *SID* in the establish part is set to 1 after the reneg part has been executed to mark whether this has happened or not.

3.2 Ideal functionality for sending/receiving messages

If we use a trusted third party to send and receive messages, we can demand that the messages are delivered in the correct order, and that the attacker does not know the content of the messages. For the ideal functionality to resemble the reality, we must let the attacker decide when the messages should arrive since we use the TCP network (but they must still be in the correct order, and they can only be received once). The attacker should also know the length of the message.

Before sending or receiving anything, we need to establish a secure connection where (at least) one of the parties are authenticated, so we include the `establish` part from the

ideal functionality for authentication. This guarantees that at least P_C knows that it is receiving messages from P_S, and not some other party.

$\mathcal{F}_{\text{TLS}}$:
On input $(\texttt{establish}, \hat{P_C}, P_S)$ from $\tilde{P_C}/\mathcal{Z}$:

1. Choose unique, random *tsid*
2. Send $(\texttt{establish}, P_C, P_S, tsid)$ to $\mathcal{S}$
3. On $(\texttt{ok}, tsid, SID, P_S)$ from $\mathcal{S}$, *SID* is new, send $(\texttt{ok}, SID, \hat{P_C}, P_S)$ to $\tilde{P_S}$
4. On $(\texttt{ok}, tsid, SID, P_C)$ from $\mathcal{S}$, send $(\texttt{ok}, SID, P_C, P_S)$ to $\tilde{P_C}$
5. Store $(P_C, P_S, SID, 0)$

For $(X, Y) \in \{(C, S), (S, C)\}$:

On input $(\texttt{send}, SID, P_Y, m)$ from $\tilde{P_X}/\mathcal{Z}$:

1. If (P_X, P_Y, SID) or (P_Y, P_X, SID) not recorded, stop.
2. Record (P_X, P_Y, m) in a queue for *SID*
3. Send $(\texttt{send}, SID, P_X, P_Y, |m|)$ to $\mathcal{S}$

On input $(\texttt{deliver}, P_X, P_Y, SID)$ from $\mathcal{S}$:

1. If (P_X, P_Y, SID) not recorded, stop.
2. Choose next (P_X, P_Y, m) in the queue for *SID*
3. Send $(\texttt{receive}, SID, P_X, m)$ to $\tilde{P_Y}$

We use P_X and P_Y ($(X, Y) \in \{(C, S), (S, C)\}$) instead of P_C and P_S in the send and receive parts to signal that both parties can send and receive messages.

Chapter 4

TLS

TLS (Transport Layer Security)[3] is a widely used protocol for secure connections between a server and a client[1]. By client we mean anyone who uses the server's services, like displaying a webpage, downloading a file or purchasing a product online. TLS provides confidentiality, data integrity and authentication, and can be used to protect all kinds of traffic between a client and a server. For example, TLS is often used to protect payment information, such as the credit card number, when a customer purchases something from an Internet shop[10].

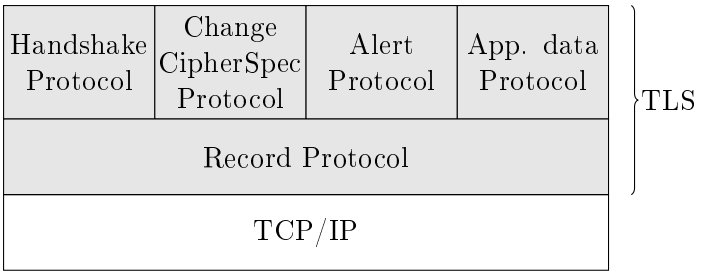

Figure 4.1: Illustration of layering of TLS subprotocols and TCP/IP

TLS communicates downwards with the TCP network and upwards with other higher level applications or protocols, for example HTTP, FTP or SMTP[9]. When TLS communicates with the TCP network, it uses the Record protocol. This protocol receives the message that some protocol wants to send, and fragments, compresses, encrypts and sends it to the receiver via the TCP network. When the Record protocol receives an encrypted message from the TCP network, it decrypts, decompresses and reassembles the text before sending the result to the appropriate protocol. We will look at the Record protocol in more detail in Chapter 4.2.

Above the Record protocol we have the Alert protocol, the ChangeCipherSpec protocol,

[1] We will only look at TLS version 1.2

the Handshake protocol and the Application data protocol[3], as illustrated in Figure 4.1. The Alert protocol is a list of different alert messages that can be sent during the communication. Some of them signals a fatal alert (like `handshake_failure`) which causes the protocol to abort. Others are just a warning (like `no_renegotiation`). We will not go into details about this protocol. If we say that we stop the protocol, we mean that a fatal error is sent. The ChangeCipherSpec protocol is used in the Handshake protocol, and consists of a single message with just one byte: 1. This message is sent to signal that from now on the Record protocol will use the newly negotiated algorithms and ciphersuites to protect the traffic. We will not use this protocol explicitly in our analysis. The Application data protocol is responsible for sending and receiving application data from the higher level applications like HTTP or FTP. By application data we mean the actual messages for instance HTTP wants to send, and not messages that comes from any of the other TLS subprotocols. We will study the Application data protocol more closely together with the Record protocol in Chapter 4.2. The Handshake protocol is used to set up a secure session between the client and the server. In the next section we will take a closer look at the Handshake protocol, which is what we will focus on in most of this analysis.

We represent the server as P_S and the client as P_C. We allow multiple parties and multiple sessions in all the abstract versions of the TLS subprotocols we will present. Multiple sessions means that we can have several instances (or local copies) of P_C and P_S (remember, the protocol that P_C and P_S are running can be run many times, creating different security parameters each time). For example, $P_{C_{a,i}}$ and $P_{C_{a,k}}$ can be two instances of P_{C_a}. When we have multiple parties, there are several protocol machines $P_{C_a}, a > 1$ and $P_{S_b}, b > 1$, that can communicate with each other.

Since TLS uses TCP, we let $\mathcal{F}_{\text{NET}}$ choose which instance of the receiving party should be used. When instances of P_{C_a} and P_{S_b} have been chosen, these instances will communicate with each other directly. Keep in mind that since the attacker is responsible for delivering the messages, he can send messages to other instances if he wants to.

4.1 The Handshake protocol

The purpose of the Handshake protocol is to establish a secure connection between a server, P_S, and a client, P_C. Through this protocol, the parties will agree upon which algorithms the Record protocol should use for symmetric encryption, compression, MAC, and signatures. We will not include this in our model. They will also negotiate secret, shared keys for MACs and symmetric encryption. The computation of encryption and MAC keys is really done by the Record protocol, not the Handshake protocol. The Record protocol will receive the necessary parameters from the Handshake protocol, and the Handshake protocol will never see the computed keys. However, in our analysis, we will not use the Record protocol in the handshake, so we let the Handshake protocol compute the keys in the same way as it is done in the Record protocol.

The parties doing the handshake can choose if one or both of them should be authenticated. Either both server and client are authenticated, or the server is authenticated and the client is anonymous. An anonymous server is not allowed to ask the client to authenticate itself. It is possible for both client and server to be anonymous, but then the session will be vulnerable to a man-in-the-middle attack[3]. We will throughout this analysis assume that the server should be authenticated.

We will now describe what happens in a handshake, cryptographically speaking. Have a look at [3] for more technical details. A handshake between two parties usually begins with the client (P_{C_a}) sending a random nonce, $n^C_{a,i}$, to the server (P_{S_b}). Sometimes the server initiates the connection by sending a `HelloRequest` (an empty message) first, to which the client answers with its random nonce. When the server has received the client's nonce, it will choose a unique session ID (SID). Then P_{S_b} will send this session ID and its random nonce ($n^S_{b,j}$) to the client, together with the server's ID (ID_{S_b}) and a flag that indicates whether the client must authenticate itself (by presenting a valid certificate, represented by the client's ID) or not. The server's ID is used to look up the server's certificate, and hence its public encryption key (or verification algorithm for signatures).

Now the server and client must find a way to share a premaster secret, pm. This premaster secret will be used to compute the master secret, ms. There are mainly two ways of doing this. Given that the server is not anonymous (it has presented a valid certificate), the client can choose a random pm and encrypt it using RSA with the server's public key from the certificate. They can also use the Diffie-Hellman key exchange. If the parties should be authenticated, their Diffie-Hellman parameters must be a part of their certificates. If both parties are anonymous, they must use Diffie-Hellman since the client doesn't know the server's public key. As mentioned above, unauthenticated Diffie-Hellman is vulnerable to a man-in-the-middle attack. In this analysis, we will use RSA to exchange the premaster secret.

If the client is supposed to authenticate itself, it will now send its own ID, ID_{C_a}, and σ in addition to the encrypted premaster secret. σ is the client's signature on all messages that has been sent this far so that the server can verify the identity of the client and that they agree upon what has been sent.

Now both server and client can compute the master secret, which they use to compute the encryption and MAC keys. For the symmetric encryption we will use the same key as we would if we used the Record protocol in the handshake. The encryption keys (and MAC keys) are produced by a pseudorandom function (PRF) that uses the master secret as key: $keyblock = \mathrm{PRF}(ms, ''\mathrm{keyexp}'', n^C_{a,i} + n^S_{b,j})$. This $keyblock$ is usually divided into four different keys k_1, k_2, k_3 and k_4 (two for MACs and two for encryption). We have one upstream key, $rs_U = (k_1, k_2)$, that is used in communication from client to server and one downstream key, $rs_D = (k_3, k_4)$, that is used in the opposite direction. To save space in the protocols, we will just say that the encryption key $rs = (rs_U \| rs_D)$ is $\mathrm{PRF}(ms, ''\mathrm{keyexp}'', n^C_{a,i} + n^S_{b,j})$.

To complete the handshake, both parties will send a `finished` message. These messages

are encrypted using the new keys, and contain the hash value of all the messages in this handshake. This lets both parties compare the received `finished` message to the expected one. This marks the end of the handshake, and the parties are ready to exchange application data.

A handshake between an anonymous client and an authenticated server looks like this:

Initial handshake (Protocol 1):

On input $(\texttt{establish}, P_{C_a}, P_{S_b})$ from $\mathcal{Z}$:

1. Choose some instance $P_{C_{a,i}}$ with a unique nonce $n^C_{a,i}$
2. Send $(\texttt{connect}, P_{S_b})$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{connect}, coID, P_{C_a}, P_{S_b})$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{send}, coID, P_{S_b}, n^C_{a,i})$ to $\mathcal{F}_{\text{NET}}$
5. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}))$ from $\mathcal{F}_{\text{NET}}$
6. Choose random pm
7. Send $(\texttt{enc}, pm, \text{ID}_{S_b})$ to $\mathcal{F}_{\text{CPKE}}$
8. Receive $(\texttt{ciphertext}, c_1)$ from $\mathcal{F}_{\text{CPKE}}$
9. Send $(\texttt{send}, coID, P_{S_b}, c_1)$ to $\mathcal{F}_{\text{NET}}$
10. Compute $ms = \text{PRF}(pm, ''\text{ms}'', n^C_{a,i} + n^S_{b,j})$ and $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
11. Compute $f_C = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i}, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, c_1))$
12. Compute $c_2 = \text{E}_{rs_U}(f_C)$ and send $(\texttt{send}, coID, P_{S_b}, c_2)$ to $\mathcal{F}_{\text{NET}}$, and receive $(\texttt{receive}, coID, P_{S_b}, \text{E}_{rs_D}(\tilde{f}_S))$ from $\mathcal{F}_{\text{NET}}$
13. Check that $\tilde{f}_S = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i}, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, c_1))$. If not, stop.
14. Store $(P_{C_a}, P_{S_b}, SID, ms, (rs_U \| rs_D))$ and output $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

On input $(\texttt{receive}, coID, \hat{P_{C_a}}, n^C_{a,i})$ from $\mathcal{F}_{\text{NET}}$:

1. Choose unique SID
2. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert})$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_1)$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{dec}, c_1)$ to $\mathcal{F}_{\text{CPKE}}$
5. Receive $(\texttt{plaintext}, pm)$ from $\mathcal{F}_{\text{CPKE}}$
6. Compute $ms = \text{PRF}(pm, ''\text{ms}'', n^C_{a,i} + n^S_{b,j})$ and $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
7. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, \text{E}_{rs_U}(\tilde{f_C}))$ from $\mathcal{F}_{\text{NET}}$
8. Check that $\tilde{f_C} = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i}, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, c_1))$. If not, stop.
9. Compute $f_S = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i}, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, c_1))$
10. Compute $c_3 = \text{E}_{rs_D}(f_S)$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_3)$ to $\mathcal{F}_{\text{NET}}$
11. Store $(\hat{P_{C_a}}, P_{S_b}, SID, ms, (rs_U \| rs_D))$ and output $(\texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\mathcal{Z}$

$\mathcal{F}_{\text{SIG}}$, $\mathcal{F}_{\text{CPKE}}$ and $\mathcal{F}_{\text{NET}}$ are the ideal functionalities for signatures, public-key encryption

and decryption and the TCP network that we looked at in Chapter 2.3.

4.1.1 Session resumption

The Handshake protocol offers session resumption[3], a feature that enables a client and a server to resume an earlier session. When we say "resume", we mean to use the same master secret and algorithms as in this earlier session. The parties must compute new encryption and MAC keys, but they can use the master secret from the earlier session together with new nonces from this handshake. The authentication level will be the same as well, so if the client was authenticated in the initial session, it will be authenticated when resuming this session. The session resumption is always initiated by the client, and the server is free to ignore it and force the client to do a complete handshake. If a session ended with a fatal alert (which causes both parties to close the connection in an abnormal way), it can not be resumed.

If $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have ended a session and the client wants to resume it, it will be the same two instances that carry out the resumption handshake, since we use TCP. Still, they have to use new nonces, so we assume that all instances have two nonces. Keep in mind that since the attacker is responsible for delivering the messages, he can send messages to other instances than $P_{C_{a,i}}$ and $P_{S_{b,j}}$ if he wants to.

When starting a fresh session, the initial message from the client does not contain any session ID. If the client wants to resume an earlier session with a server, the client can include the *SID* from that session in the initial message. If the server accepts to resume this session, the parties can go straight to the `finished` messages, if not, they have to do a complete handshake.

Below we show the protocol for an initial handshake as described in Protocol 1 followed by a session resumption, where the server is authenticated and the client is anonymous. We assume that the first session was closed normally.

Initial handshake (see Protocol 1)

Resumption:

On input $(\texttt{resume}, SID, P_{C_a}, P_{S_b})$ from $\mathcal{Z}$:

1. Check that $(P_{C_a}, P_{S_b}, SID, ms, (rs_U \| rs_D))$ is stored. If not, stop.
2. Send $(\texttt{send}, coID, P_{S_b}, (n^C_{a,i,r}, SID))$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j,r}, SID))$ from $\mathcal{F}_{\text{NET}}$
4. Compute $(rs_{U,r} \| rs_{D,r}) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i,r} + n^S_{b,j,r})$
5. Compute $f_{C,r} = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i,r}, SID, n^S_{b,j,r}, SID))$
6. Compute $c_{2,r} = \text{E}_{rs_{U,r}}(f_{C,r})$ and send $(\texttt{send}, coID, P_{S_b}, c_{2,r})$ to $\mathcal{F}_{\text{NET}}$, and receive $(\texttt{receive}, coID, P_{S_b}, \text{E}_{rs_{D,r}}(\tilde{f_{S,r}}))$ from $\mathcal{F}_{\text{NET}}$
7. Check that $\tilde{f_{S,r}} = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i,r}, SID, n^S_{b,j,r}, SID))$. If not, stop.
8. Store $(P_{C_a}, P_{S_b}, SID, ms, (rs_{U,r} \| rs_{D,r}))$ and output $(\texttt{resume} - \texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

On input $(\texttt{receive}, coID, \hat{P_{C_a}}, (n^C_{a,i,r}, SID))$ from $\mathcal{F}_{\text{NET}}$:

1. Check that $(\hat{P_{C_a}}, P_{S_b}, SID, ms, (rs_U \| rs_D))$ is stored. If not, stop.
2. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j,r}, SID))$ to $\mathcal{F}_{\text{NET}}$
3. Compute $(rs_{U,r} \| rs_{D,r}) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i,r} + n^S_{b,j,r})$
4. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, \text{E}_{rs_{U,r}}(\tilde{f_{C,r}}))$ from $\mathcal{F}_{\text{NET}}$
5. Check that $\tilde{f_{C,r}} = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i,r}, SID, n^S_{b,j,r}, SID))$. If not, stop.
6. Compute $f_{S,r} = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i,r}, SID, n^S_{b,j,r}, SID))$
7. Compute $c_{3,r} = \text{E}_{rs_{D,r}}(f_{S,r})$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_{3,r})$ to $\mathcal{F}_{\text{NET}}$
8. Store $(\hat{P_{C_a}}, P_{S_b}, SID, ms, (rs_{U,r} \| rs_{D,r}))$ and output $(\texttt{resume} - \texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\mathcal{Z}$

4.1.2 Renegotiation

It is usually possible for a client and a server to renegotiate the cryptographic parameters for an existing session[3]. This can happen if a server and a client are using an established session, but need to change the algorithms, keys or permissions. One such situation could be when the client is anonymous and tries to access a part of the server where authentication is necessary. In the following we will assume that an authenticated server has negotiated a session with an anonymous client. Then, when the initial handshake has finished and the parties have perhaps sent some application data back and forth, the server sends a `HelloRequest` to the client to begin a renegotiation.

If $P_{C_{a,i}}$ has completed a handshake with $P_{S_{b,j}}$ and the server wants a renegotiation, it will be the same instances that carry out the renegotiation since we use TCP, but the instances have to use different nonces. So we still assume that all instances have two

nonces: One for the initial handshake and one for renegotiation (or session resumption). Since the attacker is responsible for delivering the messages, he is free to send messages to other instances than $P_{C_{a,i}}$ and $P_{S_{b,j}}$.

Then the handshake continues similar to a normal handshake, except that all messages up until the `finished` messages are encrypted using the keys from the initial session. The `finished` messages from this handshake will be encrypted using the keys computed from the new master secret.

The renegotiation described above can be illustrated like this:

Initial handshake (see Protocol 1)

Renegotiation:

On input $(\texttt{reneg}, SID, \hat{P_{C_a}}, P_{S_b})$ from $\mathcal{Z}$:

1. Check that $(\hat{P_{C_a}}, P_{S_b}, SID, ms, (rs_U \| rs_D))$ is stored. If not, stop.
2. Send $(\texttt{send}, coID, \hat{P_{C_a}}, \mathrm{E}_{rs_D}(\texttt{reneg}))$ to $\mathcal{F}_{\mathrm{NET}}$
3. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, \mathrm{E}_{rs_U}(n^C_{a,i,r}))$ from $\mathcal{F}_{\mathrm{NET}}$
4. Choose unique SID_r
5. Send $(\texttt{send}, coID, \hat{P_{C_a}}, \mathrm{E}_{rs_D}(n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}))$ to $\mathcal{F}_{\mathrm{NET}}$
6. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, (\mathrm{ID}_{C_a}, c_{1,r}, \sigma))$ from $\mathcal{F}_{\mathrm{NET}}$
7. Send $(\texttt{ver}, (n^C_{a,i,r}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, \mathrm{ID}_{C_a}, c_{1,r}), \sigma)$ to $\mathcal{F}_{\mathrm{SIG}}$
8. If signature ok, send $(\texttt{dec}, c_{1,r})$ to $\mathcal{F}_{\mathrm{CPKE}}$. If not, stop.
9. Receive $(\texttt{plaintext}, pm_r)$ from $\mathcal{F}_{\mathrm{CPKE}}$
10. Compute $ms_r = \mathrm{PRF}(pm_r, ''\mathrm{ms}'', n^C_{a,i,r} + n^S_{b,j,r})$ and $(rs_{U,r} \| rs_{D,r}) = \mathrm{PRF}(ms_r, ''\mathrm{keyexp}'', n^C_{a,i,r} + n^S_{b,j,r})$
11. Receive $(\texttt{receive}, coID, P_{C_a}, \mathrm{E}_{rs_{U,r}}(\tilde{f_{C,r}}))$ from $\mathcal{F}_{\mathrm{NET}}$
12. Check that $\tilde{f_{C,r}} = \mathrm{PRF}(ms_r, ''\mathrm{C}'', \texttt{hash}(n^C_{a,i,r}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, \mathrm{ID}_{C_a}, c_{1,r}, \sigma))$. If not, stop.
13. Compute $f_{S,r} = \mathrm{PRF}(ms_r, ''\mathrm{S}'', \texttt{hash}(n^C_{a,i,r}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, \mathrm{ID}_{C_a}, c_{1,r}, \sigma))$
14. Compute $c_{3,r} = \mathrm{E}_{rs_{D,r}}(f_{S,r})$ and send $(\texttt{send}, coID, P_{C_a}, c_{3,r})$ to $\mathcal{F}_{\mathrm{NET}}$
15. Store $(P_{C_a}, P_{S_b}, SID_r, ms_r, (rs_{U,r} \| rs_{D,r}))$ and output $(\texttt{reneg} - \texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

On input $(\texttt{receive}, coID, P_{S_b}, \mathrm{E}_{rs_D}(\texttt{reneg}))$ from $\mathcal{F}_{\mathrm{NET}}$:

1. Check that $(P_{C_a}, P_{S_b}, SID, ms, (rs_U \| rs_D))$ is stored. If not, stop.
2. Send $(\texttt{send}, coID, P_{S_b}, \mathrm{E}_{rs_U}(n^C_{a,i,r}))$ to $\mathcal{F}_{\mathrm{NET}}$
3. Receive $(\texttt{receive}, coID, P_{S_b}, \mathrm{E}_{rs_D}(n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}))$ from $\mathcal{F}_{\mathrm{NET}}$
4. Choose random pm_r, send $(\texttt{enc}, pm_r, \mathrm{ID}_{S_b})$ to $\mathcal{F}_{\mathrm{CPKE}}$
5. Receive $(\texttt{ciphertext}, c_{1,r})$ from $\mathcal{F}_{\mathrm{CPKE}}$
6. Send $(\texttt{sign}, (n^C_{a,i,r}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, \mathrm{ID}_{C_a}, c_{1,r}))$ to $\mathcal{F}_{\mathrm{SIG}}$
7. Receive $(\texttt{signature}, \sigma)$ from $\mathcal{F}_{\mathrm{SIG}}$
8. Send $(\texttt{send}, coID, P_{S_b}, (\mathrm{ID}_{C_a}, c_{1,r}, \sigma))$ to $\mathcal{F}_{\mathrm{NET}}$
9. Compute $ms_r = \mathrm{PRF}(pm_r, ''\mathrm{ms}'', n^C_{a,i,r} + n^S_{b,j,r})$ and $f_{C,r} = \mathrm{PRF}(ms_r, ''\mathrm{C}'', \texttt{hash}(n^C_{a,i,r}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, \mathrm{ID}_{C_a}, c_{1,r}, \sigma))$ and $(rs_{U,r} \| rs_{D,r}) = \mathrm{PRF}(ms_r, ''\mathrm{keyexp}'', n^C_{a,i,r} + n^S_{b,j,r})$
10. Compute $c_{2,r} = \mathrm{E}_{rs_{U,r}}(f_{C,r})$ and send $(\texttt{send}, coID, P_{S_b}, c_{2,r})$ to $\mathcal{F}_{\mathrm{NET}}$, and receive $(\texttt{receive}, coID, P_{S_b}, \mathrm{E}_{rs_{D,r}}(\tilde{f_{S,r}}))$ from $\mathcal{F}_{\mathrm{NET}}$
11. Check that $\tilde{f_{S,r}} = \mathrm{PRF}(ms_r, ''\mathrm{S}'', \texttt{hash}(n^C_{a,i,r}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, \mathrm{ID}_{C_a}, c_{1,r}, \sigma))$. If not, stop.
12. Store $(P_{C_a}, P_{S_b}, SID_r, ms_r, (rs_{U,r} \| rs_{D,r}))$ and output $(\texttt{reneg} - \texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

In the above protocol we use `reneg` instead of `HelloRequest`, because in TLS the `HelloRequest` message does not necessarily mean "renegotiation", it can also be used any time the server wants to begin a new session.

In Figure 4.2 we show the message flow in first an initial handshake where the server is authenticated and the client is anonymous, and then this session is closed in the proper way. Then we show the message flow when the client wants to resume that session. The last part of the illustration shows a renegotiation of this session.

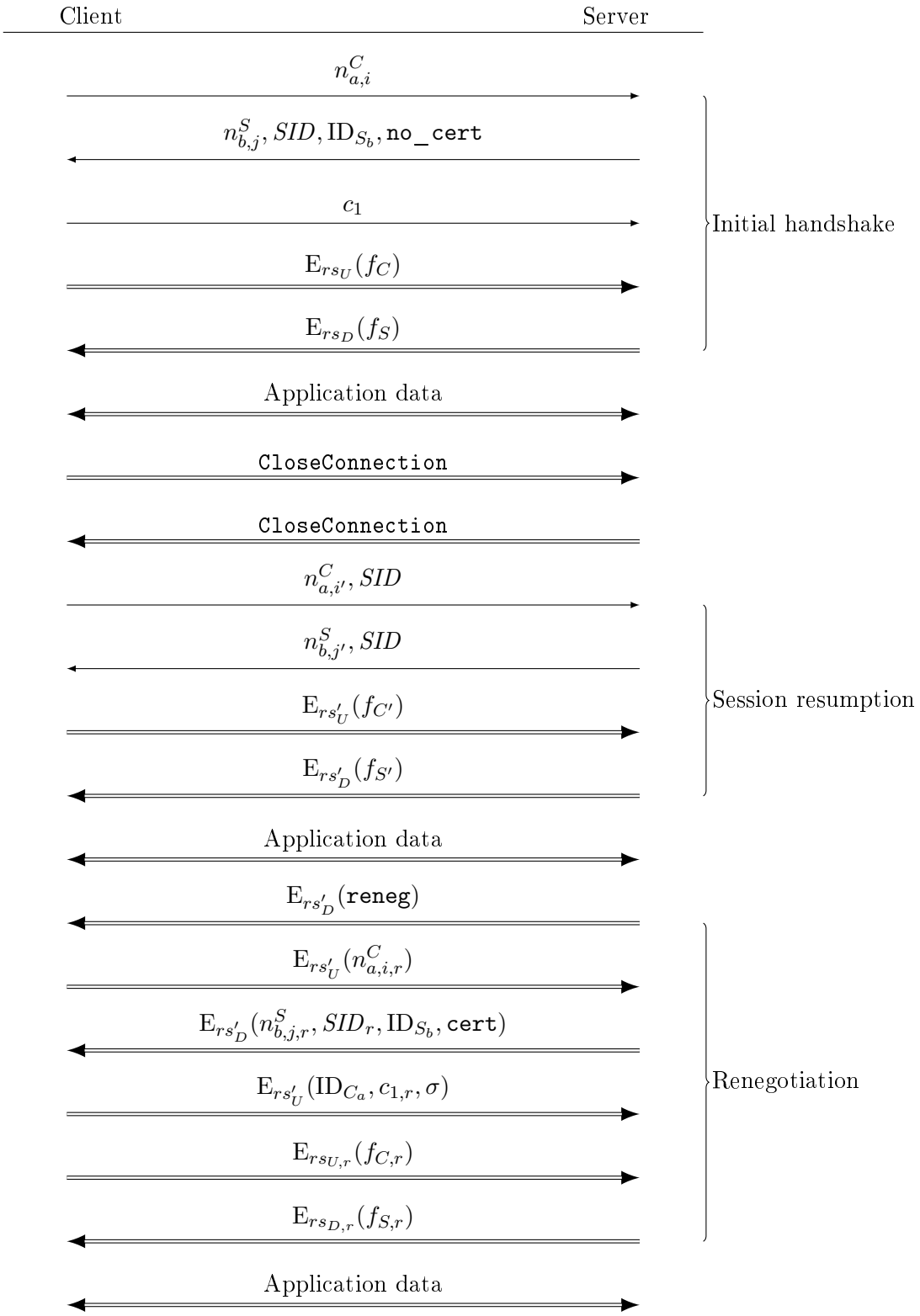

Figure 4.2: Illustration of initial handshake, resumption of this session and renegotiation of the same session. Encrypted messages are marked with double arrows.

4.2 The Application data protocol/Record protocol

The Application data protocol is used to send and receive application data to and from higher-level applications, like HTTP[3]. When it receives a message m from for instance HTTP, it will forward the message and what kind of message it is to the Record protocol like this: (`Application_data`, m).

When a message is sent from one party to another using TLS, it is the Record protocol's responsibility to divide the message up into fragments of the appropriate size, to compress it, and to encrypt it symmetrically with the key that was computed in this session's handshake[3]. It will also apply a MAC on the message to provide integrity of the message. If there has not been an initial handshake and there are no shared keys, no encryption or MAC is used. Since we only use the Record protocol on application data (which is usually sent after the handshake has finished), we will just stop the protocol if there has not been an initial handshake between the parties. The Record protocol also adds padding if the plaintext is not a multiple of the block size for block ciphers (such as AES). We will not look at fragmentation, compression or padding in this analysis.

The Record protocol can receive messages that it is supposed to send from all the other TLS subprotocols. All messages are marked with what type of message it is (`Handshake`, `Alert`, `ChangeCipherSpec` and `Application_data`), like we showed above. This information is not protected by encryption or MAC, and since we only look at the Record protocol when it receives application data messages, we will not include this feature in our model.

The Record protocol also includes a sequence number, *seq*. Each session has a sequence number that increases with 1 each time a message is sent using this session ID. This number makes sure that the packets are delivered in the correct order, and that a replay will be caught. The sequence number can therefore only be used once. The sequence number can not be larger than $2^{64} - 1$, so if the parties want to send more messages, they must renegotiate the connection[3].

When sending a message, the sender first computes a MAC over both message (in plaintext) and the sequence number so that the receiver can verify that none of them have been changed along the way. Then both MAC value and plaintext message are encrypted using the key from the handshake, and sent. How the encryption and MAC are used on the message can vary depending on which cipher is used, but what we just described applies to many of the most common ciphers. After this we increase the sequence number with 1 (because we just sent a message), and store it together with the *SID*.

When the receiver gets the ciphertext from the TCP network, it will first decrypt it using the key from the handshake. Then it will check that the MAC value is correct (and hence that the sequence number is as expected). If it is correct, we add 1 to the sequence number before we store it, and then we send the message (in plaintext) to the environment. If, for some reason, the MAC value is not what the receiver expects, the

receiver will discard the message and close the connection by sending a fatal alert. If the parties want to send more messages, they must do a complete handshake to set up a new session.

Below we have showed an abstract version of sending and receiving (application data) messages. We assume that the handshake has already finished successfully before we start sending and receiving messages using the Application data protocol/Record protocol.

Handshake (see Protocol 1)

Application data protocol/Record protocol:

For $(X, Y) \in \{(C_a, S_b), (S_b, C_a)\}$:

On input $(\texttt{send}, SID, P_Y, m)$ from $\mathcal{Z}$:

1. If $(P_X, P_Y, SID, (rs_U \| rs_D), seq)$ or $(P_Y, P_X, SID, (rs_U \| rs_D), seq)$ not stored, stop.
2. Compute $t = \text{MAC}_{rs_{MAC}}(seq \| m)$
3. Compute $c = \text{E}_{rs}(m \| t)$
4. Send $(\texttt{send}, coID, P_Y, c)$ to $\mathcal{F}_{\text{NET}}$
5. Store $seq = seq + 1$

On input $(\texttt{receive}, coID, P_X, c')$ from $\mathcal{F}_{\text{NET}}$:

1. If $(P_X, P_Y, SID, (rs_U \| rs_D), seq)$ not stored, stop.
2. Compute $(m' \| t') = \text{D}_{rs}(c')$
3. Check if $t' = \text{MAC}_{rs_{MAC}}(seq \| m')$. If not, send out a fatal alert and close the connection.
4. Store $seq = seq + 1$
5. Output m' to $\mathcal{Z}$

We use P_X and P_Y instead of P_{C_a} and P_{S_b} in the send and receive parts to mark that both server and client can send and receive messages. We do not specify in the protocol if the upstream or downstream keys should be used. Instead we just say rs, and remember that if P_X is the client, we use rs_U (the upstream key) in the send and receive parts, and the downstream key, rs_D, if P_X is the server.

We assume that the TCP connection from the handshake is remembered and used for sending and receiving application data as well.

4.3 The renegotiation attack against TLS

The renegotiation attack against the Handshake protocol in TLS was discovered in September 2009 by Marsh Ray[6]. This is how it works[7]:

A legitimate client tries to start a regular handshake with a server. This initial message is caught by the attacker. The attacker then begins an anonymous session with the server. Since the attacker is not authenticated, he will only get access to the publicly available information on the server (which does not require authentication and special privileges). The attacker will now send a request to the server that requires special privileges. The

server will respond with a `HelloRequest` to begin a renegotiation so that the attacker can identify himself. Then the attacker encrypts and sends the initial message from the honest client to the server. The server responds with a new random nonce, its ID and a flag that indicates that the client should authenticate itself, all encrypted with the key the server shares with the attacker. The attacker can now decrypt this message and forward it to the client, who sees this as the response to the nonce it sent to the server earlier. The client will not pay any attention to the fact that it is asked to authenticate itself even if it didn't try to access anything that required authentication.

The server can not tell the difference between the attacker and the client (because the attacker is anonymous), so it treats the response from the client as a renegotiation of the session it had with the attacker since the messages are encrypted using their shared key. In some protocols, for instance HTTPS, the negotiating parties can not distinguish between events that occured before and after the parties were authenticated[7]. This means that when the client and the server are finished negotiating a session where the client was successfully authenticated, the illegal command from the attacker is remembered by the server and used as a prefix to the client's request.

We show the attack in Figure 4.3.

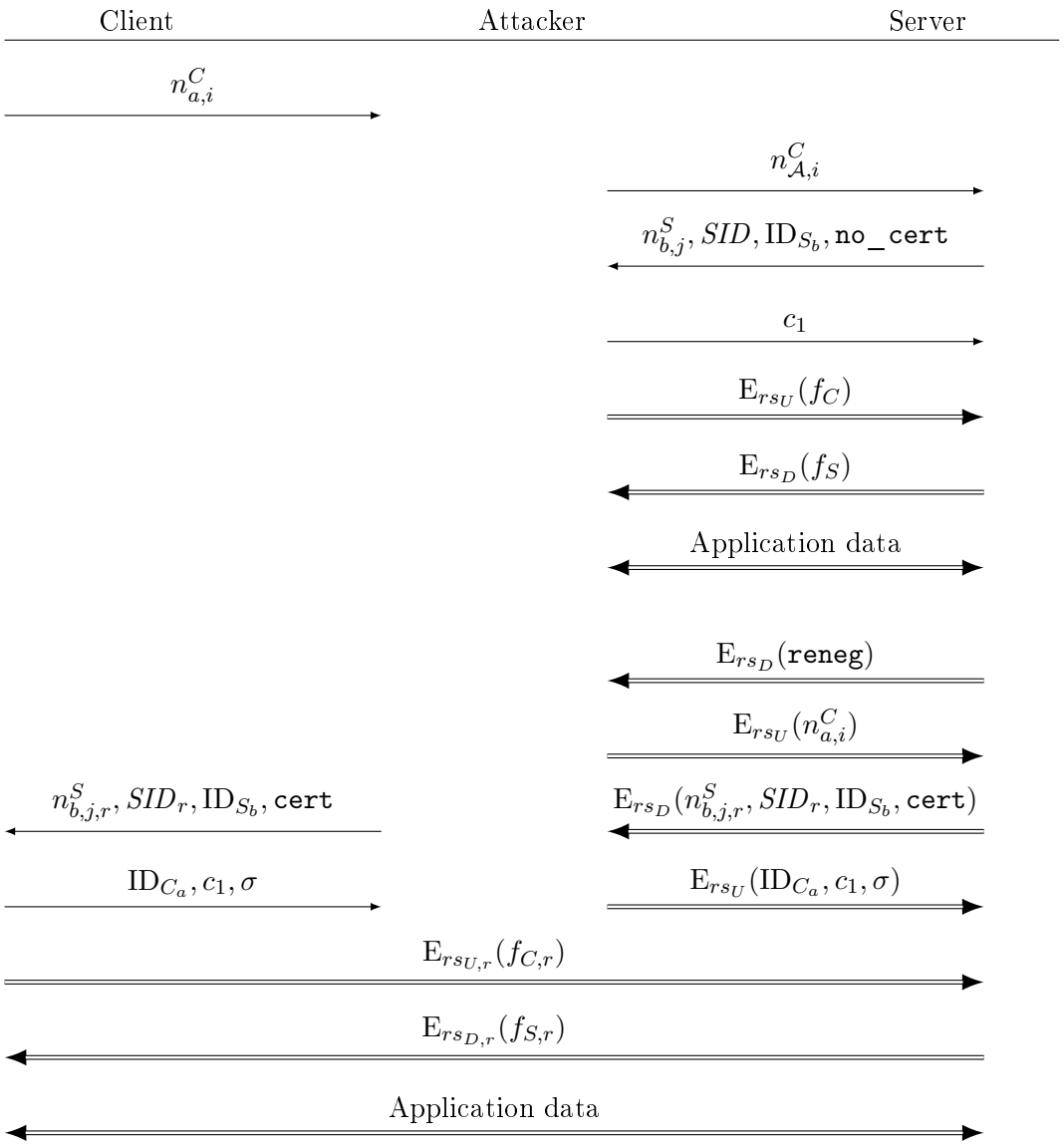

Figure 4.3: Message flow in the renegotiation attack. Encrypted messages are marked with double arrows.

4.4 The proposed fix

IETF has proposed a solution to prevent the renegotiation attack[7]. We will now take a look at it, and include it in our abstract version of TLS with one renegotiation.

The fix is added to the Handshake protocol as an extension called `renegotiation_info`, which is included in the first message from the client and in the first response from the server. The extension is supposed to tie the renegotiated session to the previous session so that both parties agree upon what the previous session was, and if there was any. It requires both client and server to store the `finished` messages from the previous session.

If this is the initial handshake, we add a 0 to the first message from the client and in the first message from the server. If this is a renegotiation of a previous session, then the first message from the client should contain the client's `finished` message from the previous handshake, and the first message from the server should contain both server and client's `finished` messages from the previous handshake. These values will be checked when received to see if they match the stored `finished` messages from the previous handshake.

If we try the renegotiation attack described in the previous section against this protocol, it will obviously not work anymore. The first handshake from the (anonymous) attacker will succeed, as it should. When the actual client sends its initial message, it will have an empty renegotiation extension (0) since this is the initial handshake for this client. This message is not encrypted, so the attacker, who is in the middle, can change the extension to contain his own `finished` message from the first handshake, before encrypting it and sending it to the server.

So far the attack is still working, and it continues to work until the client produces σ: The client's signature on all the handshake messages, including the renegotiation extension. This message will reveal to the server that the renegotiation extension in the client's first message actually was empty (hence this was an initial handshake for the client). Since we use an ideal functionality for signatures, the attacker can not change or forge the client's signature. Hence this attack falls apart. However, the protocol can still be vulnerable to many other attacks, so we will do a security proof of it (including the fix) in Chapter 5.

Chapter 5

Proof of the Handshake protocol with renegotiation

We will now carry out a security proof of the Handshake protocol (including the new fix) with renegotiation that we described in Chapters 4.1.2 and 4.4. We will only do the proof with one renegotiation. The situation we are considering is one initial handshake which is initiated by the client, and one renegotiation. The renegotiation starts when the server sends a `HelloRequest` to the client. The proof still holds if the renegotiation was initiated by the client (then we just skip the `HelloRequest`). We still use `reneg` in the protocol instead of `HelloRequest`.

We must be aware that there also exists a protocol P'_C that allows the client to authenticate itself in the initial handshake. We will not analyse this protocol here, but it is important to know that it is a possibility that a client will authenticate itself in the initial handshake (if not, the renegotiation attack would never have worked).

The proof will consist of a series of games. In the first eight games we will only look at the initial handshake. First we have to prove that all nonces and premaster secrets are unique. If several instances could have the same nonce, we could no longer use the nonce to identify an instance uniquely. Since all instances of the server can decrypt the premaster secret if they receive it, we must show that if two instances share premaster secret, but not nonces, they can not compute the same master secret. We will also prove that if some other party does not share neither premaster secret nor nonces with the server and client that are doing the handshake, this other party can not guess the correct master secret.

When we have proved all the above, we can connect the client instance and server instance to their nonces, the premaster secret and the master secret. The master secret will be unique, and only these two instances will know it.

Then we will show that it does not matter if the master secret is computed by the

instances or chosen randomly outside the protocol and then given to the instances. We do this because the master secret is used as key in the PRF (pseudorandom function) when we compute the encryption keys, $(rs_U \| rs_D)$, and we know that the key used in a PRF has to be random for the PRF to be random. After that we must prove that it is not necessary to decrypt the `finished` messages as long as the correct ciphertext is received. Then we prove that we can send encrypted random noise instead of the actual `finished` messages. Then the `finished` messages never leave the instances, so no other party (or the attacker) will know what they are.

At this point we know that if $P_{C_{a,i}}$ outputs $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$, then $P_{S_{b,j}}$ must have outputted $(\texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\mathcal{Z}$ first. Then both instances must have agreed upon the nonces, pm, ms, $(rs_U \| rs_D)$ and the `finished` messages. They are also both aware of the fact that this is their initial handshake.

Then we can look at the renegotiation part. We must first show that we don't have to decrypt the first messages that contain the `finished` messages from the initial handshake as long as we check that the ciphertext is correct, and then that we can send encrypted random noise instead. We do this to make sure that the `finished` messages are secret (which they must be now since they never leave the instances, just like we did for the initial handshake).

After this, we must prove the same things as we did for the initial handshake so that we can tie the new nonces, premaster secret and master secret together. Then we will know that the master secret is unique, both instances have computed the same master secret, and only these two instances know what it is. Then we have to prove that we can send encrypted random noise instead of the new `finished` messages. Now, if the client outputs $(\texttt{reneg} - \texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$, we know that both parties agree upon the new nonces, premaster secret, master secret, encryption keys and the `finished` messages from the initial handshake.

5.1 Initial handshake

Game 1 Game 1 is simply multiple instances of the Handshake protocol with the new fix running together with the environment $\mathcal{Z}$, the dummy attacker $\mathcal{D}$ and the ideal functionalities for signatures, public key encryption and the TCP network.

Game 1, part 1 (Handshake, $P^{(1)}$):

On input $(\texttt{establish}, P_{C_a}, P_{S_b})$ from $\mathcal{Z}$:

1. Choose some instance $P_{C_{a,i}}$ with a unique nonce $n^C_{a,i}$
2. Send $(\texttt{connect}, P_{S_b})$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{connect}, coID, P_{C_a}, P_{S_b})$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{send}, coID, P_{S_b}, (n^C_{a,i}, 0))$ to $\mathcal{F}_{\text{NET}}$
5. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ from $\mathcal{F}_{\text{NET}}$
6. Choose random pm
7. Send $(\texttt{enc}, pm, \text{ID}_{S_b})$ to $\mathcal{F}_{\text{CPKE}}$
8. Receive $(\texttt{ciphertext}, c_1)$ from $\mathcal{F}_{\text{CPKE}}$
9. Send $(\texttt{send}, coID, P_{S_b}, c_1)$ to $\mathcal{F}_{\text{NET}}$
10. Compute $ms = \text{PRF}(pm, ''\text{ms}'', n^C_{a,i} + n^S_{b,j})$ and $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
11. Compute $f_C = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$
12. Compute $c_2 = \text{E}_{rs_U}(f_C)$ and send $(\texttt{send}, coID, P_{S_b}, c_2)$ to $\mathcal{F}_{\text{NET}}$, and receive $(\texttt{receive}, coID, P_{S_b}, \text{E}_{rs_D}(\tilde{f_S}))$ from $\mathcal{F}_{\text{NET}}$
13. Check that $\tilde{f_S} = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$. If not, stop.
14. Store $(P_{C_a}, P_{S_b}, SID, (rs_U \| rs_D), f_C, f_S)$ and output $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

On input $(\texttt{receive}, coID, \hat{P_{C_a}}, (n^C_{a,i}, 0))$ from $\mathcal{F}_{\text{NET}}$:

1. Choose unique SID
2. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_1)$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{dec}, c_1)$ to $\mathcal{F}_{\text{CPKE}}$
5. Receive $(\texttt{plaintext}, pm)$ from $\mathcal{F}_{\text{CPKE}}$
6. Compute $ms = \text{PRF}(pm, ''\text{ms}'', n^C_{a,i} + n^S_{b,j})$ and $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
7. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, \text{E}_{rs_U}(\tilde{f_C}))$ from $\mathcal{F}_{\text{NET}}$
8. Check that $\tilde{f_C} = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$. If not, stop.
9. Compute $f_S = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$
10. Compute $c_3 = \text{E}_{rs_D}(f_S)$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_3)$ to $\mathcal{F}_{\text{NET}}$
11. Store $(\hat{P_{C_a}}, P_{S_b}, SID, (rs_U \| rs_D), f_C, f_S)$ and output $(\texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\mathcal{Z}$

Game 1, part 2:

On input $(\texttt{reneg}, SID, \hat{P_{C_a}}, P_{S_b})$ from $\mathcal{Z}$:

1. Check that $(\hat{P_{C_a}}, P_{S_b}, SID, (rs_U \| rs_D), f_C, f_S)$ is stored. If not, stop.
2. Send $(\texttt{send}, coID, \hat{P_{C_a}}, \mathrm{E}_{rs_D}(\texttt{reneg}))$ to $\mathcal{F}_{\mathrm{NET}}$
3. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, \mathrm{E}_{rs_U}(n^C_{a,i,r}, \tilde{f_C}))$ from $\mathcal{F}_{\mathrm{NET}}$
4. Check that $\tilde{f_C} = f_C$. If not, stop.
5. Choose unique SID_r
6. Send $(\texttt{send}, coID, \hat{P_{C_a}}, \mathrm{E}_{rs_D}(n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, f_C \| f_S))$ to $\mathcal{F}_{\mathrm{NET}}$
7. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, (\mathrm{ID}_{C_a}, c_{1,r}, \sigma))$ from $\mathcal{F}_{\mathrm{NET}}$
8. Send $(\texttt{ver}, (n^C_{a,i,r}, \tilde{f_C}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, f_C \| f_S, \mathrm{ID}_{C_a}, c_{1,r}), \sigma)$ to $\mathcal{F}_{\mathrm{SIG}}$
9. If signature ok, send $(\texttt{dec}, c_{1,r})$ to $\mathcal{F}_{\mathrm{CPKE}}$. If not, stop.
10. Receive $(\texttt{plaintext}, pm_r)$ from $\mathcal{F}_{\mathrm{CPKE}}$
11. Compute $ms_r = \mathrm{PRF}(pm_r, ''\mathrm{ms}'', n^C_{a,i,r} + n^S_{b,j,r})$ and $(rs_{U,r} \| rs_{D,r}) = \mathrm{PRF}(ms_r, ''\mathrm{keyexp}'', n^C_{a,i,r} + n^S_{b,j,r})$
12. Receive $(\texttt{receive}, coID, P_{C_a}, \mathrm{E}_{rs_{U,r}}(\tilde{f_{C,r}}))$ from $\mathcal{F}_{\mathrm{NET}}$
13. Check that $\tilde{f_{C,r}} = \mathrm{PRF}(ms_r, ''\mathrm{C}'', \texttt{hash}(n^C_{a,i,r}, \tilde{f_C}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, f_C \| f_S, \mathrm{ID}_{C_a}, c_{1,r}, \sigma))$. If not, stop.
14. Compute $f_{S,r} = \mathrm{PRF}(ms_r, ''\mathrm{S}'', \texttt{hash}(n^C_{a,i,r}, \tilde{f_C}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, f_C \| f_S, \mathrm{ID}_{C_a}, c_{1,r}, \sigma))$
15. Compute $c_{3,r} = \mathrm{E}_{rs_{D,r}}(f_{S,r})$ and send $(\texttt{send}, coID, P_{C_a}, c_{3,r})$ to $\mathcal{F}_{\mathrm{NET}}$
16. Store $(P_{C_a}, P_{S_b}, SID_r, (rs_{U,r} \| rs_{D,r}), f_{C,r}, f_{S,r})$ and output $(\texttt{reneg} - \texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

On input $(\texttt{receive}, coID, P_{S_b}, \mathrm{E}_{rs_D}(\texttt{reneg}))$ from $\mathcal{F}_{\mathrm{NET}}$:

1. Check that $(P_{C_a}, P_{S_b}, SID, (rs_U \| rs_D), f_C, f_S)$ is stored. If not, stop.
2. Send $(\texttt{send}, coID, P_{S_b}, \mathrm{E}_{rs_U}(n^C_{a,i,r}, f_C))$ to $\mathcal{F}_{\mathrm{NET}}$
3. Receive $(\texttt{receive}, coID, P_{S_b}, \mathrm{E}_{rs_D}(n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, \tilde{f_C} \| \tilde{f_S}))$ from $\mathcal{F}_{\mathrm{NET}}$
4. Check that $\tilde{f_C} = f_C$ and $\tilde{f_S} = f_S$. If not, stop.
5. Choose random pm_r, send $(\texttt{enc}, pm_r, \mathrm{ID}_{S_b})$ to $\mathcal{F}_{\mathrm{CPKE}}$
6. Receive $(\texttt{ciphertext}, c_{1,r})$ from $\mathcal{F}_{\mathrm{CPKE}}$
7. Send $(\texttt{sign}, (n^C_{a,i,r}, \tilde{f_C}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, f_C \| f_S, \mathrm{ID}_{C_a}, c_{1,r}))$ to $\mathcal{F}_{\mathrm{SIG}}$
8. Receive $(\texttt{signature}, \sigma)$ from $\mathcal{F}_{\mathrm{SIG}}$
9. Send $(\texttt{send}, coID, P_{S_b}, (\mathrm{ID}_{C_a}, c_{1,r}, \sigma))$ to $\mathcal{F}_{\mathrm{NET}}$
10. Compute $ms_r = \mathrm{PRF}(pm_r, ''\mathrm{ms}'', n^C_{a,i,r} + n^S_{b,j,r})$ and $f_{C,r} = \mathrm{PRF}(ms_r, ''\mathrm{C}'', \texttt{hash}(n^C_{a,i,r}, f_C, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, \tilde{f_C} \| \tilde{f_S}, \mathrm{ID}_{C_a}, c_{1,r}, \sigma))$ and $(rs_{U,r} \| rs_{D,r}) = \mathrm{PRF}(ms_r, ''\mathrm{keyexp}'', n^C_{a,i,r} + n^S_{b,j,r})$
11. Compute $c_{2,r} = \mathrm{E}_{rs_{U,r}}(f_{C,r})$ and send $(\texttt{send}, coID, P_{S_b}, c_{2,r})$ to $\mathcal{F}_{\mathrm{NET}}$, and receive $(\texttt{receive}, coID, P_{S_b}, \mathrm{E}_{rs_{D,r}}(\tilde{f_{S,r}}))$ from $\mathcal{F}_{\mathrm{NET}}$
12. Check that $\tilde{f_{S,r}} = \mathrm{PRF}(ms_r, ''\mathrm{S}'', \texttt{hash}(n^C_{a,i,r}, f_C, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, \tilde{f_C} \| \tilde{f_S}, \mathrm{ID}_{C_a}, c_{1,r}, \sigma))$. If not, stop.
13. Store $(P_{C_a}, P_{S_b}, SID_r, (rs_{U,r} \| rs_{D,r}), f_{C,r}, f_{S,r})$ and output $(\texttt{reneg} - \texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

Game 2 In Game 2 we put all the protocol instances into one simulator, $P^{(2)}$, see Figure 5.1. The protocol instances are still doing the same as in Game 1, we have just put them together. Since there are no actual change, $\Pr[\mathrm{E}_2] = \Pr[\mathrm{E}_1]$.

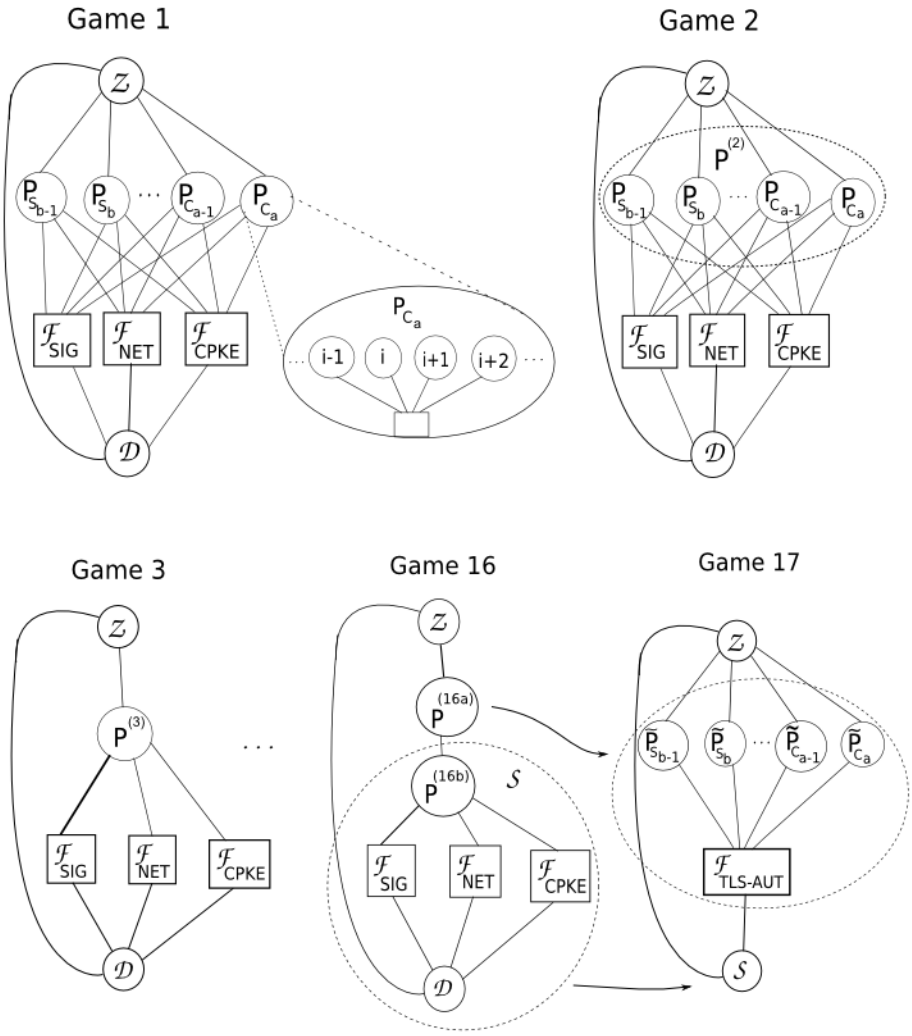

Figure 5.1: The different stages of the simulator through the games

Game 3 We define Game 3 to be the same as Game 2, except that in the simulator in Game 3, $P^{(3)}$, we check that the nonces and premaster secret are unique. If they are not, we stop the game. We use the nonces to identify the protocol instances in a unique way, and to tie one instance of P_{C_a} to one instance of P_{S_b} to make sure that they end up with the same premaster secret and master secret.

The nonces and premaster secret are chosen randomly from a large set of random values.

The birthday paradox says that the probability of a collision (that two instances choose the same nonce or pm) is approximately $\frac{k^2}{n}$, where k is the number of used nonces and premaster secrets (or sessions) and n is the number of all possible nonces and premaster secrets [4]. In TLS the nonces are 32 bytes (256 bits)[3], so we have $n = 2^{256}$ possible nonces (since the premaster secrets are longer (48 bytes or 384 bits), they have a smaller probability of a collision than the nonces). If we have $k = 2^{40}$ sessions (which is not an unrealistically large number), the probability of a collision will be approximately 2^{-176}. This probability is negligible.

We define the event F to be that two instances choose the same nonce or premaster secret. Because of the birthday paradox we know that this probability will be small, so we can say that $\Pr[F] \leq \varepsilon$, where ε is a small number. Given that F does not happen, Game 2 and Game 3 are equal, or $\Pr[E_3|\neg F] = \Pr[E_2|\neg F]$. Then the Difference Lemma from [8] says that $|\Pr[E_3] - \Pr[E_2]| \leq \Pr[F] \leq \varepsilon$. This is what [8] calls a transition based on *failure events*.

We observe that the only way any instance of the server, say $P_{S_{b,j}}$, could know the nonce belonging to the instance of the client ($P_{C_{a,i}}$), is if it received the initial message from $P_{C_{a,i}}$. The corresponding is true for the nonce of the instance of the server.

We also know that the only ones that know the premaster secret are $P_{C_{a,i}}$ who chose it, and any instance of P_{S_b} that receives it (the attacker can replay the message to several instances). An instance of any other server that receives c_1 from the client has no way of finding pm, because pm is not really sent. $P_{C_{a,i}}$ sends $(\texttt{enc}, pm, \mathrm{ID}_{S_b})$ to $\mathcal{F}_{\mathrm{CPKE}}$ and gets $(\texttt{ciphertext}, c_1)$ in return. Then c_1 is sent to $P_{S_{b,j}}$. Keep in mind that $\mathcal{F}_{\mathrm{CPKE}}$ acts as a trusted third party, so c_1 will not actually be the output of an encryption algorithm, it will just be random noise. When $P_{S_{b,j}}$ (or any instance of P_{S_b}) sends $(\texttt{dec}, c_1)$ to $\mathcal{F}_{\mathrm{CPKE}}$, it will get pm in return, because $\mathcal{F}_{\mathrm{CPKE}}$ knows that this message was intended for P_{S_b}. Any other party would just get an error if it tried to decrypt the same message. Hence, since the premaster secret never leaves $P_{C_{a,i}}$ and P_{S_b}, only these instances know pm.

Game 4 We know from the discussion above that only $P_{C_{a,i}}$ and instances of P_{S_b} will share pm. If any other instances of P_{S_b} (not $P_{S_{b,j}}$) were to compute the same ms, there has to be a collision in the pseudorandom function (we assume that the nonces are unique). A collision means in this situation that $\mathrm{PRF}(pm, ''\mathrm{ms}'', n^C_{a,i} + n^S_{b,j}) = \mathrm{PRF}(pm, ''\mathrm{ms}'', n^C_{a,i} + n^S_{b,l})$, where $n^S_{b,j} \neq n^S_{b,l}$.

We now define the simulator $P^{(4)}$ from Game 4 to be the same as in Game 3, except that we stop the game if there is a collision in the PRF. We have a PRF that is used with the same key (pm) by all instances that has received pm. We can say that a PRF with a key is the same as a set of random functions, where the key decides which random function should be used[5]. Since all instances use pm as the key, using the PRF to compute the master secret will be equivalent to choosing it randomly.

If we use the birthday paradox again, we get that the probability that two instances

choose the same ms is approximately $\frac{k^2}{n}$, where k is the number of used master secrets and n is the number of all possible master secrets [4]. The master secret is 48 bytes (384 bits)[3], so we have $n = 2^{384}$ possible master secrets. If we still assume that we have $k = 2^{40}$ sessions, the probability of a collision will be approximately 2^{-304}. This probability is negligible.

We do the same as in Game 3, and define the event F' to be that two instances choose the same master secret. From the previous discussion we know that this probability will be very small, so we say that $\Pr[F'] \leq \delta$, where δ is a small number. Given that F' does not happen, Game 3 and Game 4 are equal, or $\Pr[E_4|\neg F'] = \Pr[E_3|\neg F']$. We use the Difference Lemma from [8] again, so we get that $|\Pr[E_4] - \Pr[E_3]| \leq \Pr[F'] \leq \delta$.

We can conclude that since there are no collisions in the PRF, $\text{PRF}(pm, "\text{ms}", n^C_{a,i} + n^S_{b,j}) \neq \text{PRF}(pm, "\text{ms}", n^C_{a,i} + n^S_{b,l})$, for $n^S_{b,j} \neq n^S_{b,l}$. Hence, the instance of the client and the instance of the server must have the same two nonces to compute the same master secret.

Game 5 We have showed that only one instance of the server and one instance of the client can compute a shared master secret. Since this master secret never leaves these two instances, no other instance or party (and particularily not the attacker) will know what it is. However, if some other party happens to end up with the same master secret as $P_{C_{a,i}}$ and $P_{S_{b,j}}$, it is a potential problem.

In the simulator $P^{(5)}$ in Game 5, we add an additional check to make sure that the master secret is unique. If it is not, we stop the game. If we use a PRF with two different keys (pm_1 and pm_2), it is equivalent to using two different random functions. However, we can still use the birthday paradox to compute the probability of a collision since both functions are random.

So we use the birthday paradox as we did in the previous game to show that the probability of a collision will be approximately 2^{-304}, which is negligible. We define the event F'' to be that an instance (of any party) chooses the same master secret as $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have computed. We know that this probability will be very small, so we can say that $\Pr[F''] \leq \lambda$, where λ is a small number. Given that F'' does not happen, Game 4 and Game 5 are equal, or $\Pr[E_5|\neg F''] = \Pr[E_4|\neg F'']$. By the Difference Lemma from [8] we get that $|\Pr[E_5] - \Pr[E_4]| \leq \Pr[F''] \leq \lambda$.

We observe that as soon as $P_{C_{a,i}}$ and $P_{S_{b,j}}$ share nonces and premaster secret, they are coupled (paired) together as the only two instances that will know the master secret, because it is not possible for instances that don't share nonces and premaster secrets to compute or guess the correct master secret.

Game 6 In Game 6 we let a unique ms be chosen randomly (independent of pm) outside the protocol instances, and then given to the instances when they need it.

In the previous games we have proved that $P_{C_{a,i}}$ and $P_{S_{b,j}}$ (and only these two instances) will compute a unique master secret. As we pointed out in Game 4, using a PRF with a key is the same as using a random function, so if we compute the master secret from pm or choose it randomly does not make any difference. Since we only give it to $P_{C_{a,i}}$ and $P_{S_{b,j}}$, it will still be secret. Hence $\Pr[E_6] = \Pr[E_5]$.

The key that the instances use for the symmetric encryption, $(rs_U \| rs_D)$, is defined like this: $(rs_U \| rs_D) = \text{PRF}(ms, \text{"keyexp"}, n^C_{a,i} + n^S_{b,j})$, which is 128 bytes (1024 bits)[3]. We know from the previous games that ms is unique and secret, and that the PRF is collision-free. If we use the same reasoning as we did for the master secret in Game 5, we get that the probability that some other party happens to end up with the same $(rs_U \| rs_D)$ is approximately 2^{-944}, which is negligible.

Game 7 In Game 7 we will no longer decrypt the `finished` messages, we will just check that the correct ciphertext is received. We know what the contents should be, so it is not necessary to decrypt the ciphertext if it came from the right sender. We will also stop those instances that don't share nonces and premaster secret with anyone (the client instances will be stopped after they have sent their `finished` message, and the server instances when they have received the premaster secret).

According to the Handshake protocol, the server should first receive a `finished` message from the client that must be encrypted using the upstream encryption key rs_U. This will be the first message that is encrypted with this key, and hence the only valid ciphertext that has been produced this far, so it can not be a replay. Because of the integrity of ciphertexts (INT-CTXT) you can not produce a valid ciphertext without the key. We have proved that only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have rs_U, and since the server never encrypts with the upstream key, the message must have come from $P_{C_{a,i}}$. Then the `finished` message (the plaintext) will be correct, and hence the ciphertext must be correct as well.

The reasoning above is true when the client receives the `finished` message from the server as well. Then we can conclude that it is enough to check that the correct ciphertext was received.

The instances that don't share nonces and premaster secret with anyone will compute their own master secret, but no other instance will know it. Hence they will not share encryption keys $(rs'_U \| rs'_D)$ with anyone. It follows from INT-CTXT that these instances will never receive any `finished` message which is encrypted with the correct key from another instance. Then we know that they will never finish, so it doesn't matter if we stop them.

Then we get that $|\Pr[E_7] - \Pr[E_6]|$ is small, because of INT-CTXT.

Game 8 In Game 8 we will no longer send the actual `finished` messages. Instead we will send some (encrypted) random noise of the same length. We still check that the correct ciphertext is received, from the previous game.

We know from the previous game that we don't have to decrypt the `finished` messages. Because of the indistinguishability under chosen-ciphertext attack (IND-CCA)[5], the attacker can not tell the difference between random noise and an actual message when they are encrypted, so it doesn't matter what we encrypt as long as it has the right length. Then we have that $|\Pr[E_8] - \Pr[E_7]|$ is small because of IND-CCA.

We know that if $P_{C_{a,i}}$ outputs $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$, then $P_{S_{b,j}}$ must have outputted $(\texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\mathcal{Z}$ first. Then both instances must have agreed upon everything, in particular the nonces, pm, ms, $(rs_U \| rs_D)$ and the `finished` messages. They are also both aware of the fact that this is their initial handshake.

5.2 Renegotiation part

Until now we have not changed anything in the renegotiation part of the simulator. We need to prove that the instances agree upon the new nonces, premaster secret, the server's identity, master secret, shared encryption key and `finished` messages for this part of the simulator as well. In addition, we have to show that both instances know that this is a renegotiation, and that the identity of the client is verified. We observe that since several of the messages in the renegotiation part are encrypted using the secret keys that $P_{C_{a,i}}$ and $P_{S_{b,j}}$ agreed upon in the initial handshake, $(rs_U \| rs_D)$, both instances can be sure that they are doing the renegotiation with each other, not any other (honest) party. Of course, this is only the case when both $P_{C_{a,i}}$ and $P_{S_{b,j}}$ are honest.

Game 9 In Game 9 the instances no longer decrypt the first messages of the renegotiation, $\mathrm{E}_{rs_U}(n^C_{a,i,r}, f_C)$ and $\mathrm{E}_{rs_D}(n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, f_C \| f_S)$. They will just check that the correct ciphertext is received. All necessary information from these messages (the nonces and SID_r) will be given to the instances when needed.

First, we observe that although these two messages are encrypted using the same keys as the `finished` messages from the previous handshake, f_C and f_S from the previous handshake are of different form and length than these messages from the renegotiation part, so the ciphertexts would also be of different form and length.

$(n^C_{a,i,r}, f_C)$ is encrypted using rs_U, which we have proved in the previous games that only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ know. Since it is not possible to produce a valid ciphertext without knowing the key, the ciphertext must have been produced by $P_{C_{a,i}}$ or $P_{S_{b,j}}$. We also know that only the client uses rs_U for encryption, so the message must have come from $P_{C_{a,i}}$. This means that the content must be correct, and then the ciphertext will be correct as well. The same is true for the message from the server. Hence, $|\Pr[E_9] - \Pr[E_8]|$ is small because of INT-CTXT.

Game 10 We let the simulator $P^{(10)}$ in Game 10 be the same as $P^{(9)}$ from Game 9, except that the instances now send encrypted random noise of the same length instead of $\mathrm{E}_{rs_U}(n^C_{a,i,r}, f_C)$ and $\mathrm{E}_{rs_D}(n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, f_C \| f_S)$.

We use the same reasoning as we did in Game 8: It follows from IND-CCA that the attacker can not tell the difference between random noise and an actual message of the same length when they are encrypted, so it does not matter what we encrypt since the messages will not be decrypted anyway. Hence $|\Pr[E_{10}] - \Pr[E_9]|$ is small, because of IND-CCA.

Game 11 We know from Game 3 and Game 4 that only $P_{C_{a,i}}$ and instances of P_{S_b} will share pm_r, and that if any other instances of P_{S_b} (not $P_{S_{b,j}}$) were to compute the same ms_r, there has to be a collision in the pseudorandom function (we still assume that the nonces are unique).

We let the simulator in Game 11 be the same as in Game 10, except that we stop the game if there is a collision in the PRF, just like we did in Game 4. As in Game 4, we have a PRF that is used with the same key (pm_r) by all instances that has received pm_r. So using the PRF to compute the new master secret will be equivalent to choosing it randomly.

We use the birthday paradox again, and get that the probability that two instances choose the same ms_r is the same as in Game 4, approximately 2^{-304}. This probability is negligible.

We let the event $F^{(3)}$ be that two instances choose the same master secret. This probability will be the same as in Game 4, so we get that $|\Pr[E_{11}] - \Pr[E_{10}]| \leq \Pr[F^{(3)}] \leq \delta$, where δ is a very small number.

Game 12 We still have that since the master secret never leaves the instances, only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ will know it, but we also need to make sure that no other party can simply "guess" the correct master secret. This is exactly the same as what we did in Game 5.

In the simulator $P^{(12)}$ in Game 12, we add an additional check to make sure that the renegotiated master secret is unique. If it is not, we stop the game. As we discussed in Game 5, we can still use the birthday paradox to compute the probability of a collision when we pick two random numbers using two different random functions since both functions are random.

When we use the birthday paradox again as we did in the Game 5, we get that the probability of a collision will be approximately 2^{-304}, which is negligible. We can define the event $F^{(4)}$ to be that two instances (of any party) choose the same (renegotiated) master secret. This probability will be the same as in Game 5, so we get that $|\Pr[E_{12}] - \Pr[E_{11}]| \leq \Pr[F^{(4)}] \leq \lambda$, where λ is a very small number.

Game 13 Corresponding to what we did for the initial handshake, we now let a unique ms_r be chosen randomly (independent of pm_r) outside the protocol instances, and then given to the parties when they need it.

In the previous games we have proved that only two instances will share a master secret, and that it is unique. We have also showed that there are no collisions in the PRF. Then we know that $P_{C_{a,i}}$ and $P_{S_{b,j}}$ (and only these two instances) will compute a unique new master secret. From Game 4 we have that using a PRF with a key is the same as using a random function, so computing ms_r from pm_r and choosing ms_r randomly are equivalent. Since we only give ms_r to $P_{C_{a,i}}$ and $P_{S_{b,j}}$, it will still be secret. Hence $\Pr[E_{13}] = \Pr[E_{12}]$.

It follows from the previous games that ms_r is unique and secret, and that the PRF is collision-free. We use the same reasoning as we have done for the master secret and encryption keys in Game 5 and Game 12. Then we get that the probability that some other party happens to end up with the same $(rs_{U,r} \| rs_{D,r})$ is approximately 2^{-944}, which is negligible. Hence $(rs_{U,r} \| rs_{D,r})$ is unique and secret as well.

Game 14 In Game 14 we will no longer decrypt the `finished` messages, we will just check that the correct ciphertext is received. We will also stop those instances that don't share nonces and premaster secret with any other instance (we stop the client instances after they have sent their `finished` message, and the server instances when they have received the premaster secret). This is the same as what we did in Game 7. We know what the contents of the `finished` messages should be, so it is not necessary to decrypt the ciphertext as long as it came from the right sender.

The reasoning here is exactly the same as in Game 7: According to the protocol, the server should first receive a `finished` message from the client that is encrypted using the upstream encryption key $rs_{U,r}$. This will be the first message that is encrypted with this key, and hence the only valid ciphertext that has been produced this far, so it can not be a replay. Because of the integrity of ciphertexts (INT-CTXT) you can not produce a valid ciphertext without the key. We have proved that only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have $rs_{U,r}$, and since the server never encrypts with the upstream key, the message must have come from $P_{C_{a,i}}$. Then the `finished` message (the plaintext) will be correct, and hence the ciphertext must be correct as well. The same is true for the `finished` message that the client receives.

The instances that don't share nonces and premaster secret with anyone will compute their own master secret, but no other instance will know it. Hence, these instances will not share encryption keys $(rs'_{U,r} \| rs'_{D,r})$ with anyone. It follows from INT-CTXT that these instances will never receive any `finished` message which is encrypted with the correct key from another instance. Then we know that they will never finish, so it doesn't matter if we stop them. Hence, $|\Pr[E_{14}] - \Pr[E_{13}]|$ is small because of INT-CTXT.

Game 15 We keep following the proof for the initial handshake, so in Game 15 we will no longer send the actual `finished` messages. Instead we will send some (encrypted) random noise of the same length. We still check that the correct ciphertext is received, from the previous game.

We have from Game 14 that we don't have to decrypt the `finished` messages as long as they came from the right sender. Since the attacker can not tell the difference between random noise and an actual message when they are encrypted because of IND-CCA, it does not matter what we encrypt as long as it has the right length. Then it follows from IND-CCA that $|\Pr[E_{15}] - \Pr[E_{14}]|$ is small.

We know that if $P_{C_{a,i}}$ outputs $(\texttt{reneg} - \texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$, then $P_{S_{b,j}}$ must have outputted $(\texttt{reneg} - \texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$ first. Then both instances must have agreed upon the new nonces, pm_r, ms_r, $(rs_{U,r} \| rs_{D,r})$ and the `finished` messages. They are also both aware of the fact that this is a renegotiation, and they agree upon the `finished` messages from the previous handshake.

Game 16 In Game 16 we will divide the simulator into $P^{(16a)}$ and $P^{(16b)}$, see Figure 5.1. We let $P^{(16a)}$ take care of all communication between the simulator and the environment, and let $P^{(16b)}$ be responsible for all communication between the simulator and the ideal functionalities; $\mathcal{F}_{\text{CPKE}}$, $\mathcal{F}_{\text{NET}}$ and $\mathcal{F}_{\text{SIG}}$. $\tilde{P_{C_a}}$ and $\tilde{P_{S_b}}$ are dummy protocol machines for P_{C_a} and P_{S_b} that simply forward what they receive.

Game 16, $P^{(16a)}$:

On input $(\texttt{establish}, \hat{P_{C_a}}, P_{S_b})$ from $\tilde{P_{C_a}}/\mathcal{Z}$

1. Choose unique, random *tsid*
2. Send $(\texttt{establish}, P_{C_a}, P_{S_b}, tsid)$ to $P^{(16b)}$
3. On $(\texttt{ok}, tsid, SID, P_{S_b})$ from $P^{(16b)}$, SID is new, send $(\texttt{ok}, sid, \hat{P_{C_a}}, P_{S_b})$ to $\tilde{P_{S_b}}$
4. On $(\texttt{ok}, tsid, SID, P_{C_a})$ from $P^{(16b)}$, send $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\tilde{P_{C_a}}$
5. Store $(P_{C_a}, P_{S_b}, SID, 0)$

On input $(\texttt{reneg}, SID, \hat{P_{C_a}}, P_{S_b})$ from $\tilde{P_{S_b}}/\mathcal{Z}$:

1. If $(P_{C_a}, P_{S_b}, SID, 0)$ not stored, stop.
2. Send $(\texttt{reneg}, SID, \hat{P_{C_a}}, P_{S_b})$ to $P^{(16b)}$
3. On $(\texttt{reneg} - \texttt{ok}, SID, SID_r, P_{S_b})$ from $P^{(16b)}$, send $(\texttt{reneg} - \texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ to $\tilde{P_{S_b}}$
4. On $(\texttt{reneg} - \texttt{ok}, SID, SID_r, P_{C_a})$ from $P^{(16b)}$, send $(\texttt{reneg} - \texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ to $\tilde{P_{C_a}}$
5. Store $(P_{C_a}, P_{S_b}, SID_r, 1)$, discard $(P_{C_a}, P_{S_b}, SID, 0)$

Game 16, $P^{(16b)}$, part 1:

On input $(\texttt{establish}, P_{C_a}, P_{S_b}, tsid)$ from $P^{(16a)}$:

1. Choose some instance $P_{C_{a,i}}$ with a unique nonce $n^C_{a,i}$
2. Send $(\texttt{connect}, P_{S_b})$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{connect}, coID, P_{C_a}, P_{S_b})$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{send}, coID, P_{S_b}, (n^C_{a,i}, 0))$ to $\mathcal{F}_{\text{NET}}$
5. Choose unique SID
6. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ to $\mathcal{F}_{\text{NET}}$
7. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ from $\mathcal{F}_{\text{NET}}$
8. Choose unique, random pm
9. Send $(\texttt{enc}, pm, \text{ID}_{S_b})$ to $\mathcal{F}_{\text{CPKE}}$
10. Receive $(\texttt{ciphertext}, c_1)$ from $\mathcal{F}_{\text{CPKE}}$
11. Send $(\texttt{send}, coID, P_{S_b}, c_1)$ to $\mathcal{F}_{\text{NET}}$
12. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_1)$ from $\mathcal{F}_{\text{NET}}$
13. Send $(\texttt{dec}, c_1)$ to $\mathcal{F}_{\text{CPKE}}$
14. Receive $(\texttt{plaintext}, pm)$ from $\mathcal{F}_{\text{CPKE}}$
15. Choose unique, random ms and compute $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
16. Choose unique, random f_C
17. Compute $c_2 = \text{E}_{rs_U}(f_C)$ and send $(\texttt{send}, coID, P_{S_b}, c_2)$ to $\mathcal{F}_{\text{NET}}$
18. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_2')$ from $\mathcal{F}_{\text{NET}}$
19. Check that $c_2' = c_2$. If not, stop.
20. Choose unique, random f_S
21. Compute $c_3 = \text{E}_{rs_D}(f_S)$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_3)$ to $\mathcal{F}_{\text{NET}}$
22. Output $(\texttt{ok}, tsid, SID, P_{S_b})$ to $P^{(16a)}$
23. Receive $(\texttt{receive}, coID, P_{S_b}, c_3')$ from $\mathcal{F}_{\text{NET}}$
24. Check that $c_3' = c_3$. If not, stop.
25. Output $(\texttt{ok}, tsid, SID, P_{C_a})$ to $P^{(16a)}$

Game 16, $P^{(16b)}$, part 2:

On input $(\texttt{reneg}, SID, \hat{P_{C_a}}, P_{S_b})$ from $P^{(16a)}$:

1. Send $(\texttt{send}, coID, \hat{P_{C_a}}, \mathrm{E}_{rs'_D}(\texttt{reneg}))$ to $\mathcal{F}_{\mathrm{NET}}$
2. Choose unique, random m_4
3. Compute $c_4 = \mathrm{E}_{rs_U}(m_4)$
4. Send $(\texttt{send}, coID, P_{S_b}, c_4)$ to $\mathcal{F}_{\mathrm{NET}}$
5. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c'_4)$ from $\mathcal{F}_{\mathrm{NET}}$
6. Check that $c'_4 = c_4$. If not, stop.
7. Choose unique SID_r
8. Choose unique, random m_5
9. Compute $c_5 = \mathrm{E}_{rs_U}(m_5)$
10. Send $(\texttt{send}, coID, \hat{P_{C_a}}, c_5)$ to $\mathcal{F}_{\mathrm{NET}}$
11. Receive $(\texttt{receive}, coID, P_{S_b}, c'_5)$ from $\mathcal{F}_{\mathrm{NET}}$
12. Check that $c'_5 = c_5$. If not, stop.
13. Choose unique, random pm_r, send $(\texttt{enc}, pm_r, \mathrm{ID}_{S_b})$ to $\mathcal{F}_{\mathrm{CPKE}}$
14. Receive $(\texttt{ciphertext}, c_{1,r})$ from $\mathcal{F}_{\mathrm{CPKE}}$
15. Send $(\texttt{sign}, (n^C_{a,i,r}, \tilde{f_C}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, f_C \| f_S, \mathrm{ID}_{C_a}, c_{1,r}))$ to $\mathcal{F}_{\mathrm{SIG}}$
16. Receive $(\texttt{signature}, \sigma)$ from $\mathcal{F}_{\mathrm{SIG}}$
17. Send $(\texttt{send}, coID, P_{S_b}, (\mathrm{ID}_{C_a}, c_{1,r}, \sigma))$ to $\mathcal{F}_{\mathrm{NET}}$
18. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, (\mathrm{ID}_{C_a}, c_{1,r}, \sigma))$ from $\mathcal{F}_{\mathrm{NET}}$
19. Send $(\texttt{ver}, (n^C_{a,i,r}, \tilde{f_C}, n^S_{b,j,r}, SID_r, \mathrm{ID}_{S_b}, \texttt{cert}, f_C \| f_S, \mathrm{ID}_{C_a}, c_{1,r}), \sigma)$ to $\mathcal{F}_{\mathrm{SIG}}$
20. If signature ok, send $(\texttt{dec}, c_{1,r})$ to $\mathcal{F}_{\mathrm{CPKE}}$. If not, stop.
21. Receive $(\texttt{plaintext}, pm_r)$ from $\mathcal{F}_{\mathrm{CPKE}}$
22. Choose unique, random ms_r and compute $(rs_{U,r} \| rs_{D,r}) = \mathrm{PRF}(ms_r, ''\mathrm{keyexp}'', n^C_{a,i,r} + n^S_{b,j,r})$
23. Choose unique, random $f_{C,r}$
24. Compute $c_{2,r} = \mathrm{E}_{rs_{U,r}}(f_{C,r})$ and send $(\texttt{send}, coID, P_{S_b}, c_{2,r})$ to $\mathcal{F}_{\mathrm{NET}}$
25. Receive $(\texttt{receive}, coID, P_{C_a}, c'_{2,r})$ from $\mathcal{F}_{\mathrm{NET}}$
26. Check that $c'_{2,r} = c_{2,r}$. If not, stop.
27. Choose unique, random $f_{S,r}$
28. Compute $c_{3,r} = \mathrm{E}_{rs_{D,r}}(f_{S,r})$ and send $(\texttt{send}, coID, P_{C_a}, c_{3,r})$ to $\mathcal{F}_{\mathrm{NET}}$
29. Output $(\texttt{reneg} - \texttt{ok}, SID, SID_r, P_{S_b})$ to $P^{(16a)}$
30. Receive $(\texttt{receive}, coID, P_{S_b}, c'_{3,r})$ from $\mathcal{F}_{\mathrm{NET}}$
31. Check that $c'_{3,r} = c_{3,r}$. If not, stop.
32. Output $(\texttt{reneg} - \texttt{ok}, SID, SID_r, P_{C_a})$ to $P^{(16a)}$

We observe that $P^{(16a)}$ is the ideal functionality for the handshake protocol, $\mathcal{F}_{\mathrm{TLS-AUT}}$ (without the resumption option). $P^{(16b)}$ does both sides of the Handshake protocol with renegotiation. Since we have only split the simulator into two parts and not added or removed anything that changes the behaviour, there is no change from the previous game, so $\Pr[E_{16}] = \Pr[E_{15}]$.

Let us assume that an honest server finishes the renegotiation successfully, and it believes it is communicating with an honest client, P_{C_a}. This means that the server outputs $(\texttt{reneg}-\texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ after the renegotiation, where P_{C_a} is the identity of an honest client. Then the server and the client must have the same premaster secret. This premaster secret is chosen by the client, encrypted using the server's public key and then signed, together with the `finished` messages from the initial handshake. Since we use an ideal functionality for the signatures, it is not possible to forge a signature. Hence the premaster secret must come from P_{C_a} and not any other party (or the attacker).

Since the `finished` messages from the previous handshake are also signed by P_{C_a}, both parties must agree upon the initial handshake. Then it must have been P_{C_a} that the server did the initial handshake with as well, not any other party (or the attacker).

We will now assume that an honest client thinks it is communicating with an honest server, and that the client outputs $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ after the initial handshake, where P_{S_b} is the identity of an honest server. Then the client and the server must have the same premaster secret. The premaster secret is, as we know, chosen by the client and encrypted with P_{S_b}'s public key. Since we use an ideal functionality for public key encryption, only P_{S_b} can decrypt the premaster secret. Hence the client is in fact communicating with P_{S_b}, and not any other party (or the attacker).

Now we assume that the honest client thinks it is communicating with an honest server, and that the client outputs $(\texttt{reneg}-\texttt{ok}, SID_r, P_{C_a}, P_{S_b})$ after the renegotiation, where P_{S_b} is the identity of an honest server. This implies that the client and the server must have the same premaster secret, which is chosen by the client and encrypted with P_{S_b}'s public key. Since we use an ideal functionality for public key encryption, only P_{S_b} can decrypt the premaster secret. Hence the client is in fact communicating with P_{S_b}, and not any other party (or the attacker). We also know that the `finished` messages from the initial handshake is a part of the `finished` messages in this handshake, so both parties must agree upon the initial handshake. Hence the client must have done the initial handshake with P_{S_b}, not some other party (or the attacker).

We can now define the simulating attacker $\mathcal{S}$ to be $P^{(16b)}$, $\mathcal{F}_{\text{CPKE}}$, $\mathcal{F}_{\text{NET}}$, $\mathcal{F}_{\text{SIG}}$ and the dummy attacker $\mathcal{D}$, as illustrated in Figure 5.1. The protocol machines are now a part of $\mathcal{S}$. $\mathcal{S}$ can use the secret keys to sign messages using the program from $\mathcal{F}_{\text{SIG}}$ in the simulation. However, since $\mathcal{F}_{\text{SIG}}$ acts as a trusted third party it will remember that it did not sign the messages that came from $\mathcal{S}$, and so the verification of the signatures will only succeed when $\mathcal{S}$ tries to verify them. If the messages were sent to some other protocol machine that is not in $\mathcal{S}$, the verification would fail. $\mathcal{F}_{\text{CPKE}}$ will behave the same way.

Game 17 In Game 17 we let the simulator be the ideal functionality, where $\mathcal{S}$ is as defined above. We have only renamed $P^{(16a)}$ (it is the same as $\mathcal{F}_{\text{TLS}-\text{AUT}}$), and defined $\mathcal{S}$ to be $\mathcal{D}$, the ideal functionalities and $P^{(16b)}$. We have not changed how the simulator behaves in any way, so $\Pr[E_{17}] = \Pr[E_{16}]$.

Using the definition from Chapter 2.2, we get that the adversary's advantage against the Handshake protocol with the new fix can be described by this telescoping sum:

$$\begin{aligned}\mathrm{Adv}_{\mathcal{A}} &= |\Pr[E_1] - \Pr[E_{17}]| \\ &= |\sum_{i=1}^{16}(\Pr[E_i] - \Pr[E_{i+1}])| \\ &\leq \sum_{i=1}^{16}|\Pr[E_i] - \Pr[E_{i+1}]|\end{aligned}$$

We know that most of these differences are very small, approximately 2^{-304} and 2^{-944}, which means they are negligible. The most significant contribution comes from $|\Pr[E_3] - \Pr[E_2]| \leq \varepsilon$, where ε is approximately 2^{-176}, which is also negligible. We can conclude that the attacker's advantage, and the difference in probability between the ideal functionality and the Handshake protocol with fix and one renegotiation, is at most 2^{-176}. Then we can say that the Handshake protocol (including the fix) with renegotiation UC-realizes $\mathcal{F}_{\mathrm{TLS-AUT}}$ (without the resumption part). Hence the protocol is secure according to the UC theory.

Chapter 6

Proof of the Handshake protocol with session resumption

We will now try to prove that the session resumption feature in the Handshake protocol is secure. The situation we are looking at is an anonymous client doing the protocol with an authenticated server. They will first do a complete initial handshake, close the connection in the proper way perhaps after sending some messages back and forth, and then the client will try to resume that session. We will include the fix for the renegotiation attack in the Handshake protocol in this proof as well. When the client tries to resume an earlier session, this request is treated as a new handshake, so it does not include any `finished` message.

6.1 Initial handshake

Since the initial handshake is exactly the same as it was in the renegotiation proof from Chapter 5.1, we will reuse most of those games in this part of the proof.

Game 1 Game 1 is just multiple instances of the Handshake protocol running together with the environment $\mathcal{Z}$, the dummy attacker $\mathcal{D}$ and the ideal functionalities for signatures, public key encryption and the TCP network.

Game 1, part 1 (Handshake, $P^{(1)}$):

On input $(\texttt{establish}, P_{C_a}, P_{S_b})$ from $\mathcal{Z}$:

1. Choose some instance $P_{C_{a,i}}$ with a unique nonce $n^C_{a,i}$
2. Send $(\texttt{connect}, P_{S_b})$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{connect}, coID, P_{C_a}, P_{S_b})$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{send}, coID, P_{S_b}, (n^C_{a,i}, 0))$ to $\mathcal{F}_{\text{NET}}$
5. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ from $\mathcal{F}_{\text{NET}}$
6. Choose random pm
7. Send $(\texttt{enc}, pm, \text{ID}_{S_b})$ to $\mathcal{F}_{\text{CPKE}}$
8. Receive $(\texttt{ciphertext}, c_1)$ from $\mathcal{F}_{\text{CPKE}}$
9. Send $(\texttt{send}, coID, P_{S_b}, c_1)$ to $\mathcal{F}_{\text{NET}}$
10. Compute $ms = \text{PRF}(pm, ''\text{ms}'', n^C_{a,i} + n^S_{b,j})$ and $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
11. Compute $f_C = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$
12. Compute $c_2 = \text{E}_{rs_U}(f_C)$ and send $(\texttt{send}, coID, P_{S_b}, c_2)$ to $\mathcal{F}_{\text{NET}}$, and receive $(\texttt{receive}, coID, P_{S_b}, \text{E}_{rs_D}(\tilde{f_S}))$ from $\mathcal{F}_{\text{NET}}$
13. Check that $\tilde{f_S} = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$. If not, stop.
14. Store $(P_{C_a}, P_{S_b}, SID, ms, f_C, f_S)$ and output $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

On input $(\texttt{receive}, coID, \hat{P_{C_a}}, (n^C_{a,i}, 0))$ from $\mathcal{F}_{\text{NET}}$:

1. Choose unique SID
2. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_1)$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{dec}, c_1)$ to $\mathcal{F}_{\text{CPKE}}$
5. Receive $(\texttt{plaintext}, pm)$ from $\mathcal{F}_{\text{CPKE}}$
6. Compute $ms = \text{PRF}(pm, ''\text{ms}'', n^C_{a,i} + n^S_{b,j})$ and $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
7. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, \text{E}_{rs_U}(\tilde{f_C}))$ from $\mathcal{F}_{\text{NET}}$
8. Check that $\tilde{f_C} = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$. If not, stop.
9. Compute $f_S = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$
10. Compute $c_3 = \text{E}_{rs_D}(f_S)$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_3)$ to $\mathcal{F}_{\text{NET}}$
11. Store $(\hat{P_{C_a}}, P_{S_b}, SID, ms, f_C, f_S)$ and output $(\texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\mathcal{Z}$

Game 1, part 2 (resumption):

On input $(\texttt{resume}, SID, P_{C_a}, P_{S_b})$ from $\mathcal{Z}$:

1. Check that $(P_{C_a}, P_{S_b}, SID, ms, f_C, f_S)$ is stored. If not, stop.
2. Send $(\texttt{send}, coID, P_{S_b}, (n^C_{a,i,r}, SID, 0))$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j,r}, SID, 0))$ from $\mathcal{F}_{\text{NET}}$
4. Compute $(rs_{U,r} \| rs_{D,r}) = \text{PRF}(ms, \text{"keyexp"}, n^C_{a,i,r} + n^S_{b,j,r})$
5. Compute $f_{C,r} = \text{PRF}(ms, \text{"C"}, \texttt{hash}(n^C_{a,i,r}, SID, 0, n^S_{b,j,r}, SID, 0))$
6. Compute $c_{2,r} = \text{E}_{rs_{U,r}}(f_{C,r})$ and send $(\texttt{send}, coID, P_{S_b}, c_{2,r})$ to $\mathcal{F}_{\text{NET}}$, and receive $(\texttt{receive}, coID, P_{S_b}, \text{E}_{rs_{D,r}}(\tilde{f_{S,r}}))$ from $\mathcal{F}_{\text{NET}}$
7. Check that $\tilde{f_{S,r}} = \text{PRF}(ms, \text{"S"}, \texttt{hash}(n^C_{a,i,r}, SID, 0, n^S_{b,j,r}, SID, 0))$. If not, stop.
8. Store $(P_{C_a}, P_{S_b}, SID, ms, f_{C,r}, f_{S,r})$ and output $(\texttt{resume} - \texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

On input $(\texttt{receive}, coID, \hat{P_{C_a}}, (n^C_{a,i,r}, SID, 0))$ from $\mathcal{F}_{\text{NET}}$:

1. Check that $(\hat{P_{C_a}}, P_{S_b}, SID, ms, f_C, f_S)$ is stored. If not, stop.
2. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j,r}, SID, 0))$ to $\mathcal{F}_{\text{NET}}$
3. Compute $(rs_{U,r} \| rs_{D,r}) = \text{PRF}(ms, \text{"keyexp"}, n^C_{a,i,r} + n^S_{b,j,r})$
4. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, \text{E}_{rs_{U,r}}(\tilde{f_{C,r}}))$ from $\mathcal{F}_{\text{NET}}$
5. Check that $\tilde{f_{C,r}} = \text{PRF}(ms, \text{"C"}, \texttt{hash}(n^C_{a,i,r}, SID, 0, n^S_{b,j,r}, SID, 0))$. If not, stop.
6. Compute $f_{S,r} = \text{PRF}(ms, \text{"S"}, \texttt{hash}(n^C_{a,i,r}, SID, 0, n^S_{b,j,r}, SID, 0))$
7. Compute $c_{3,r} = \text{E}_{rs_{D,r}}(f_{S,r})$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_{3,r})$ to $\mathcal{F}_{\text{NET}}$
8. Store $(\hat{P_{C_a}}, P_{S_b}, SID, ms, f_{C,r}, f_{S,r})$ and output $(\texttt{resume} - \texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\mathcal{Z}$

Games 2-8 are exactly the same as they were in the renegotiation proof, so we will not do them again. We can just summarize that after Game 8 we know that only one client instance ($P_{C_{a,i}}$) and one server instance ($P_{S_{b,j}}$) will be paired, and that only these two instances will share nonces and premaster secret. Hence they are the only two instances that will know the correct master secret and encryption keys, and that can compute the correct `finished` messages. Given that the initial handshake finishes successfully, the client will also know that it is talking to the correct server since the client encrypted the premaster secret using the server's public key.

6.2 Resumption part

We will now look at the resumption part of the protocol. We need to tie both the initial handshake and the resumption handshake to the same client instance and server instance, and show that only these two instances can successfully resume the session, similarily to what we did for the initial handshake.

Game 9 In Game 9 we will no longer decrypt the `finished` messages in the resumption part, we will just check that the correct ciphertext is received. If the message came from the right sender we know what the contents should be, so it is not necessary to decrypt the ciphertext.

The server should first receive a `finished` message from the client that is encrypted using the upstream encryption key $rs_{U,r}$. This will be the first message that is encrypted with this key, and hence the only valid ciphertext that has been produced this far, so it can not be a replay. Because of the integrity of ciphertexts (INT-CTXT) you can not produce a valid ciphertext without the key. We have showed in the previous proof that only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have $rs_{U,r}$ (because only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have ms), and since the server never encrypts with the upstream key, the message must have come from $P_{C_{a,i}}$. Then the `finished` message (the plaintext) will be correct, and hence the ciphertext must be correct as well.

The reasoning above is true when the client receives the `finished` message from the server as well. Then we can conclude that it is enough to check that the correct ciphertext was received. Hence $|\Pr[E_9] - \Pr[E_8]|$ is small, because of INT-CTXT.

Game 10 In Game 10 we will no longer send the actual `finished` messages. Instead we will send some (encrypted) random noise of the same length. We still check that the correct ciphertext is received, from the previous game.

We know from the previous game that we don't have to decrypt the `finished` messages. Since it follows from IND-CCA that the attacker can not tell the difference between random noise and an actual message when they are encrypted, we can encrypt whatever we want as long as it has the right length. Hence, $|\Pr[E_{10}] - \Pr[E_9]|$ is small because of IND-CCA.

Game 11 In Game 11 we divide the simulator into $P^{(11a)}$ and $P^{(11b)}$, just like we did in Game 16 in the renegotiation proof. We let $P^{(11a)}$ take care of all communication downwards (between the simulator and the environment), and let $P^{(11b)}$ be responsible for all communication upwards (between the simulator and $\mathcal{F}_{\text{CPKE}}$, $\mathcal{F}_{\text{NET}}$ and $\mathcal{F}_{\text{SIG}}$). $\tilde{P_{C_a}}$ and $\tilde{P_{S_b}}$ are dummy protocol machines for P_{C_a} and P_{S_b} that just forward what they receive.

Game 11, $P^{(11a)}$:

On input $(\texttt{establish}, \hat{P_{C_a}}, P_{S_b})$ from $\tilde{P_{C_a}}/\mathcal{Z}$

1. Choose unique, random *tsid*
2. Send $(\texttt{establish}, P_{C_a}, P_{S_b}, tsid)$ to $P^{(11b)}$
3. On $(\texttt{ok}, tsid, SID, P_{S_b})$ from $P^{(11b)}$, *SID* is new, send $(\texttt{ok}, sid, \hat{P_{C_a}}, P_{S_b})$ to $\tilde{P_{S_b}}$
4. On $(\texttt{ok}, tsid, SID, P_{C_a})$ from $P^{(11b)}$, send $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\tilde{P_{C_a}}$
5. Store $(P_{C_a}, P_{S_b}, SID, 0)$

On input $(\texttt{resume}, SID, P_{C_a}, P_{S_b})$ from $\tilde{P_{C_a}}/\mathcal{Z}$:

1. If $(P_{C_a}, P_{S_b}, SID, 0)$ not stored, stop.
2. Send $(\texttt{resume}, SID, \hat{P_{C_a}}, P_{S_b})$ to $P^{(11b)}$
3. On $(\texttt{resume} - \texttt{ok}, SID, P_{S_b})$ from $P^{(11b)}$, send $(\texttt{resume} - \texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\tilde{P_{S_b}}$
4. On $(\texttt{resume} - \texttt{ok}, SID, \hat{P_{C_a}})$ from $P^{(11b)}$, send $(\texttt{resume} - \texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\tilde{P_{C_a}}$

Game 11, $P^{(11b)}$, part 1:

On input $(\texttt{establish}, P_{C_a}, P_{S_b}, tsid)$ from $P^{(11a)}$:

1. Choose some instance $P_{C_{a,i}}$ with a unique nonce $n^C_{a,i}$
2. Send $(\texttt{connect}, P_{S_b})$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{connect}, coID, P_{C_a}, P_{S_b})$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{send}, coID, P_{S_b}, (n^C_{a,i}, 0))$ to $\mathcal{F}_{\text{NET}}$
5. Choose unique SID
6. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ to $\mathcal{F}_{\text{NET}}$
7. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ from $\mathcal{F}_{\text{NET}}$
8. Choose unique, random pm
9. Send $(\texttt{enc}, pm, \text{ID}_{S_b})$ to $\mathcal{F}_{\text{CPKE}}$
10. Receive $(\texttt{ciphertext}, c_1)$ from $\mathcal{F}_{\text{CPKE}}$
11. Send $(\texttt{send}, coID, P_{S_b}, c_1)$ to $\mathcal{F}_{\text{NET}}$
12. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_1)$ from $\mathcal{F}_{\text{NET}}$
13. Send $(\texttt{dec}, c_1)$ to $\mathcal{F}_{\text{CPKE}}$
14. Receive $(\texttt{plaintext}, pm)$ from $\mathcal{F}_{\text{CPKE}}$
15. Choose unique, random ms and compute $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
16. Choose unique, random f_C
17. Compute $c_2 = \text{E}_{rs_U}(f_C)$ and send $(\texttt{send}, coID, P_{S_b}, c_2)$ to $\mathcal{F}_{\text{NET}}$
18. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_2')$ from $\mathcal{F}_{\text{NET}}$
19. Check that $c_2' = c_2$. If not, stop.
20. Choose unique, random f_S
21. Compute $c_3 = \text{E}_{rs_D}(f_S)$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_3)$ to $\mathcal{F}_{\text{NET}}$
22. Output $(\texttt{ok}, tsid, SID, P_{S_b})$ to $P^{(11a)}$
23. Receive $(\texttt{receive}, coID, P_{S_b}, c_3')$ from $\mathcal{F}_{\text{NET}}$
24. Check that $c_3' = c_3$. If not, stop.
25. Output $(\texttt{ok}, tsid, SID, P_{C_a})$ to $P^{(11a)}$

Game 11, $P^{(11b)}$, part 2:

On input $(\texttt{resume}, SID, \hat{P_{C_a}}, P_{S_b})$ from $P^{(11a)}$:

1. Check that $(P_{C_a}, P_{S_b}, SID, ms, f_C, f_S)$ is stored. If not, stop.
2. Send $(\texttt{send}, coID, P_{S_b}, (n^C_{a,i,r}, SID, 0))$ to $\mathcal{F}_{\text{NET}}$
3. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j,r}, SID, 0))$ to $\mathcal{F}_{\text{NET}}$
4. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j,r}, SID, 0))$ from $\mathcal{F}_{\text{NET}}$
5. Compute $(rs_{U,r} \| rs_{D,r}) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i,r} + n^S_{b,j,r})$
6. Choose unique, random $f_{C,r}$
7. Compute $c_{2,r} = \text{E}_{rs_{U,r}}(f_{C,r})$ and send $(\texttt{send}, coID, P_{S_b}, c_{2,r})$ to $\mathcal{F}_{\text{NET}}$
8. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c'_{2,r})$ from $\mathcal{F}_{\text{NET}}$
9. Check that $c'_{2,r} = c_{2,r}$. If not, stop.
10. Choose unique, random $f_{S,r}$
11. Compute $c_{3,r} = \text{E}_{rs_{D,r}}(f_{S,r})$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_{3,r})$ to $\mathcal{F}_{\text{NET}}$
12. Output $(\texttt{resume} - \texttt{ok}, SID, P_{S_b})$ to $P^{(11a)}$
13. Receive $(\texttt{receive}, coID, P_{S_b}, c'_{3,r})$ from $\mathcal{F}_{\text{NET}}$
14. Check that $c'_{3,r} = c_{3,r}$. If not, stop.
15. Output $(\texttt{resume} - \texttt{ok}, SID, \hat{P_{C_a}})$ to $P^{(11a)}$

We can observe that $P^{(11a)}$ is the ideal functionality for the Handshake protocol, $\mathcal{F}_{\text{TLS}-\text{AUT}}$ (without renegotiation). $P^{(11b)}$ does both sides of the Handshake protocol with one session resumption. Since we have only divided the simulator into two parts without adding or removing anything, the simulator will behave exactly as it did in the previous game, so $\Pr[E_{11}] = \Pr[E_{10}]$.

Let us assume that an honest client thinks it is communicating with an honest server, and that the client outputs $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ after the initial handshake, where P_{S_b} is the identity of an honest server. Then the client and the server must have the same premaster secret. The premaster secret has been chosen by the client and encrypted with P_{S_b}'s public key. Since we use an ideal functionality for public key encryption, only P_{S_b} can decrypt the premaster secret. Hence the client is in fact communicating with P_{S_b}, and not any other party (or the attacker). This is the same as in the renegotiation proof.

Now we assume that the honest client thinks it is communicating with an honest server, and that the client outputs $(\texttt{resume} - \texttt{ok}, SID, P_{C_a}, P_{S_b})$ after the session resumption, where P_{S_b} is the identity of an honest server. This implies that the client and the server must have the same master secret, which was computed in the initial handshake. We have proved that only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ know ms after the initial handshake, and since ms never leaves these two instances, no other party (or the attacker) knows it. Hence the client must have done the initial handshake with P_{S_b}, not some other party (or the attacker).

Just like in the renegotiation proof, we can now define the simulating attacker $\mathcal{S}$ to be $P^{(11b)}$, $\mathcal{F}_{\text{CPKE}}$, $\mathcal{F}_{\text{NET}}$, $\mathcal{F}_{\text{SIG}}$ and the dummy attacker $\mathcal{D}$. The protocol machines are now

a part of $\mathcal{S}$.

Game 12 In Game 12 we let the simulator be the ideal functionality, where $\mathcal{S}$ is as defined above. We have just renamed $P^{(11a)}$ (it is clearly the same as $\mathcal{F}_{\text{TLS}-\text{AUT}}$, without the renegotiation option), and defined $\mathcal{S}$ to be $\mathcal{D}$, the ideal functionalities and $P^{(11b)}$. Since we have not changed the simulator's behaviour in any way, we must have that $\Pr[E_{12}] = \Pr[E_{11}]$.

If we compute the attacker's advantage, we get the same telescoping sum as in the renegotiation proof, where 2^{-176} is an upper bound for the difference in probability between the ideal functionality and the actual protocol. This is negligible, so we can say that the Handshake protocol with session resumption UC-realizes $\mathcal{F}_{\text{TLS}-\text{AUT}}$ (without the renegotiation part). Hence it is secure according to the UC theory.

Chapter 7

Proof of the Application data protocol/Record protocol

We will now present a security proof of the Application data protocol together with the Record protocol. The situation we are considering is first a handshake between an anonymous client and an authenticated server. We will not use the Record protocol in the handshake, so this part of the proof will be exactly the same as in Chapter 5.1. Then we let the client and server use the Application data protocol to send each other messages. In this part we will use the Record protocol to make sure that the messages are encrypted with some symmetric algorithm and the key from the handshake. We include the fix against the renegotiation attack in the Handshake protocol in this proof as well.

7.1 Handshake

Game 1 The first game consists of multiple instances of the Handshake protocol and the Application data protocol/Record protocol running together with the environment $\mathcal{Z}$, the dummy attacker $\mathcal{D}$ and the ideal functionalities for signatures, public key encryption and the TCP network.

Game 1, part 1 (Handshake, $P^{(1)}$):

On input $(\texttt{establish}, P_{C_a}, P_{S_b})$ from $\mathcal{Z}$:

1. Choose some instance $P_{C_{a,i}}$ with a unique nonce $n^C_{a,i}$
2. Send $(\texttt{connect}, P_{S_b})$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{connect}, coID, P_{C_a}, P_{S_b})$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{send}, coID, P_{S_b}, (n^C_{a,i}, 0))$ to $\mathcal{F}_{\text{NET}}$
5. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ from $\mathcal{F}_{\text{NET}}$
6. Choose random pm
7. Send $(\texttt{enc}, pm, \text{ID}_{S_b})$ to $\mathcal{F}_{\text{CPKE}}$
8. Receive $(\texttt{ciphertext}, c_1)$ from $\mathcal{F}_{\text{CPKE}}$
9. Send $(\texttt{send}, coID, P_{S_b}, c_1)$ to $\mathcal{F}_{\text{NET}}$
10. Compute $ms = \text{PRF}(pm, ''\text{ms}'', n^C_{a,i} + n^S_{b,j})$ and $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
11. Compute $f_C = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$
12. Compute $c_2 = \text{E}_{rs_U}(f_C)$ and send $(\texttt{send}, coID, P_{S_b}, c_2)$ to $\mathcal{F}_{\text{NET}}$, and receive $(\texttt{receive}, coID, P_{S_b}, \text{E}_{rs_D}(\tilde{f_S}))$ from $\mathcal{F}_{\text{NET}}$
13. Check that $\tilde{f_S} = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$. If not, stop.
14. Store $(P_{C_a}, P_{S_b}, SID, (rs_U \| rs_D), f_C, f_S)$ and output $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\mathcal{Z}$

On input $(\texttt{receive}, coID, \hat{P_{C_a}}, (n^C_{a,i}, 0))$ from $\mathcal{F}_{\text{NET}}$:

1. Choose unique SID
2. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_1)$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{dec}, c_1)$ to $\mathcal{F}_{\text{CPKE}}$
5. Receive $(\texttt{plaintext}, pm)$ from $\mathcal{F}_{\text{CPKE}}$
6. Compute $ms = \text{PRF}(pm, ''\text{ms}'', n^C_{a,i} + n^S_{b,j})$ and $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
7. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, \text{E}_{rs_U}(\tilde{f_C}))$ from $\mathcal{F}_{\text{NET}}$
8. Check that $\tilde{f_C} = \text{PRF}(ms, ''\text{C}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$. If not, stop.
9. Compute $f_S = \text{PRF}(ms, ''\text{S}'', \texttt{hash}(n^C_{a,i}, 0, n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0, c_1))$
10. Compute $c_3 = \text{E}_{rs_D}(f_S)$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_3)$ to $\mathcal{F}_{\text{NET}}$
11. Store $(\hat{P_{C_a}}, P_{S_b}, SID, (rs_U \| rs_D), f_C, f_S)$ and output $(\texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\mathcal{Z}$

Game 1, part 2 ($P^{(1)}$, Application data protocol/Record protocol):

For $(X, Y) \in \{(C_a, S_b), (S_b, C_a)\}$:

On input $(\texttt{send}, SID, P_Y, m)$ from $\mathcal{Z}$:

1. If $(P_X, P_Y, SID, (rs_U \| rs_D), seq)$ or $(P_Y, P_X, SID, (rs_U \| rs_D), seq)$ not stored, stop.
2. Compute $t = \text{MAC}_{rs_{MAC}}(seq \| m)$
3. Compute $c = \text{E}_{rs}(m \| t)$
4. Send $(\texttt{send}, coID, P_Y, c)$ to $\mathcal{F}_{\text{NET}}$
5. Store $seq = seq + 1$

On input $(\texttt{receive}, coID, P_X, P_Y, c')$ from $\mathcal{F}_{\text{NET}}$:

1. If $(P_X, P_Y, SID, (rs_U \| rs_D), seq)$ not stored, stop.
2. Compute $(m' \| t') = \text{D}_{rs}(c')$
3. Check if $t' = \text{MAC}_{rs_{MAC}}(seq \| m')$. If not, send out a fatal alert and close the connection.
4. Store $seq = seq + 1$
5. Output m' to $\mathcal{Z}$

Here P_X and P_Y can be either P_{C_a} or P_{S_b}. As we explained in Chapter 4.2, we will use rs in the send and receive parts of the protocol, and we just remember that if P_X is the client, we use rs_U (the upstream key), and rs_D (the downstream key) if P_X is the server.

Games 2-8 are the same as in Chapter 5.1. From this we know that if we assume that an honest client thinks it is communicating with an honest server, and that the client outputs $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ after the handshake, then the client is in fact communicating with P_{S_b}, and not some other party (or the attacker). We also know that both parties agree upon the nonces, master secret, encryption and MAC keys and the `finished` messages.

7.2 Application data protocol/Record protocol

We must now prove that no other party (or the attacker) will know the contents of the messages that P_{C_a} and P_{S_b} are sending to each other. We must also show that the messages can not be changed along the way.

Game 9 In Game 9 we will no longer check that the MAC is correct. Instead, we will check that the message came from the right sender. If we receive a message from some other instance (not $P_{C_{a,i}}$ or $P_{S_{b,j}}$), we will discard it.

Since both the message and the MAC are encrypted using rs_U or rs_D, which only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ share, we know that the message must have come from $P_{C_{a,i}}$ or $P_{S_{b,j}}$ because it follows from INT-CTXT that you can not produce a valid ciphertext without the key. If the message came from $P_{C_{a,i}}$ or $P_{S_{b,j}}$, we know which message m and which sequence number seq should be received. Since both $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have the correct MAC key, the MAC must be correct. Then we get that the difference $|\Pr[E_9] - \Pr[E_8]|$ is small, because of INT-CTXT.

We know that only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have rs_U and rs_D. Then, if $P_{S_{b,j}}$ receives a message from some other instance (not $P_{C_{a,i}}$), it will be encrypted with some other key that $P_{S_{b,j}}$ does not have, so $P_{S_{b,j}}$ would not be able to decrypt the message. The same is true if $P_{C_{a,i}}$ receives a message from any other instance than $P_{S_{b,j}}$. Then we know that the message will never be received by $\mathcal{Z}$, so it doesn't matter if we discard these messages.

Game 10 The simulator in Game 10 does the same as the simulator from the previous game, except that we will no longer decrypt the ciphertext we receive, we will just check that the correct ciphertext was received. Since we know what the contents should be, it is not necessary to decrypt the ciphertext as long as it is the expected ciphertext (it came from the right sender, and has not been changed along the way).

Since the encrypted message contains the sequence number, which is only used once, it can not be a replay because the ciphertext would not be correct if the sequence number was wrong. Because of INT-CTXT you can not produce a valid ciphertext without the key. We have proved that only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have rs_U and rs_D. If the message was encrypted with rs_U, we know that since the server never encrypts with the upstream key, the message must have come from $P_{C_{a,i}}$. The corresponding is true for rs_D and $P_{S_{b,j}}$. Then the plaintext will be what we expect, and hence the ciphertext must be correct as well.

We get that $|\Pr[E_{10}] - \Pr[E_9]|$ is small, because of INT-CTXT.

Game 11 In Game 11 we will no longer send the actual message, but encrypted random noise of the same length.

Since the attacker can not tell the difference between random noise and an actual message when they are encrypted because of IND-CCA, it doesn't matter what we encrypt as long as it has the right length. Then we have that $|\Pr[E_{11}] - \Pr[E_{10}]|$ is small, because of IND-CCA.

Game 12 In Game 12 we divide the simulator vertically into $P^{(12a)}$ and $P^{(12b)}$. We let $P^{(12a)}$ take care of all communication between the simulator and the environment, and let $P^{(12b)}$ be responsible for all communication between the simulator and the ideal functionalities, just like we have done in the previous proofs.

Game 12, $P^{(12a)}$:

On input $(\texttt{establish}, \hat{P_{C_a}}, P_{S_b})$ from $\tilde{P_{C_a}}/\mathcal{Z}$

1. Choose unique, random *tsid*
2. Send $(\texttt{establish}, P_{C_a}, P_{S_b}, tsid)$ to $P^{(12b)}$
3. On $(\texttt{ok}, tsid, SID, P_{S_b})$ from $P^{(12b)}$, *SID* is new, send $(\texttt{ok}, SID, \hat{P_{C_a}}, P_{S_b})$ to $\tilde{P_{S_b}}$
4. On $(\texttt{ok}, tsid, SID, P_{C_a})$ from $P^{(12b)}$, send $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ to $\tilde{P_{C_a}}$
5. Store $(P_{C_a}, P_{S_b}, SID, 0)$

For $(X, Y) \in \{(C_a, S_b), (S_b, C_a)\}$:

On input $(\texttt{send}, SID, P_Y, m)$ from $\tilde{P_X}/\mathcal{Z}$:

1. If (P_X, P_Y, SID) or (P_Y, P_X, SID) not recorded, stop.
2. Record (P_X, P_Y, m) in a queue for *SID*
3. Send $(\texttt{send}, SID, P_X, P_Y, |m|)$ to $P^{(12b)}$

On input $(\texttt{deliver}, P_X, P_Y, SID)$ from $P^{(12b)}$:

1. If (P_X, P_Y, SID) not recorded, stop.
2. Choose next (P_X, P_Y, m) in the queue for *SID*
3. Send $(\texttt{receive}, SID, P_X, m)$ to $\tilde{P_Y}$

Game 12, $P^{(12b)}$:

On input $(\texttt{establish}, P_{C_a}, P_{S_b}, tsid)$ from $P^{(12a)}$:

1. Choose some instance $P_{C_{a,i}}$ with a unique nonce $n^C_{a,i}$
2. Send $(\texttt{connect}, P_{S_b})$ to $\mathcal{F}_{\text{NET}}$
3. Receive $(\texttt{connect}, coID, P_{C_a}, P_{S_b})$ from $\mathcal{F}_{\text{NET}}$
4. Send $(\texttt{send}, coID, P_{S_b}, (n^C_{a,i}, 0))$ to $\mathcal{F}_{\text{NET}}$
5. Choose unique SID
6. Send $(\texttt{send}, coID, \hat{P_{C_a}}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ to $\mathcal{F}_{\text{NET}}$
7. Receive $(\texttt{receive}, coID, P_{S_b}, (n^S_{b,j}, SID, \text{ID}_{S_b}, \texttt{no\_cert}, 0))$ from $\mathcal{F}_{\text{NET}}$
8. Choose unique, random pm
9. Send $(\texttt{enc}, pm, \text{ID}_{S_b})$ to $\mathcal{F}_{\text{CPKE}}$
10. Receive $(\texttt{ciphertext}, c_1)$ from $\mathcal{F}_{\text{CPKE}}$
11. Send $(\texttt{send}, coID, P_{S_b}, c_1)$ to $\mathcal{F}_{\text{NET}}$
12. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_1)$ from $\mathcal{F}_{\text{NET}}$
13. Send $(\texttt{dec}, c_1)$ to $\mathcal{F}_{\text{CPKE}}$
14. Receive $(\texttt{plaintext}, pm)$ from $\mathcal{F}_{\text{CPKE}}$
15. Choose unique, random ms and compute $(rs_U \| rs_D) = \text{PRF}(ms, ''\text{keyexp}'', n^C_{a,i} + n^S_{b,j})$
16. Choose unique, random f_C
17. Compute $c_2 = \text{E}_{rs_U}(f_C)$ and send $(\texttt{send}, coID, P_{S_b}, c_2)$ to $\mathcal{F}_{\text{NET}}$
18. Receive $(\texttt{receive}, coID, \hat{P_{C_a}}, c_2')$ from $\mathcal{F}_{\text{NET}}$
19. Check that $c_2' = c_2$. If not, stop.
20. Choose unique, random f_S
21. Compute $c_3 = \text{E}_{rs_D}(f_S)$ and send $(\texttt{send}, coID, \hat{P_{C_a}}, c_3)$ to $\mathcal{F}_{\text{NET}}$
22. Output $(\texttt{ok}, tsid, SID, P_{S_b})$ to $P^{(12a)}$
23. Receive $(\texttt{receive}, coID, P_{S_b}, c_3')$ from $\mathcal{F}_{\text{NET}}$
24. Check that $c_3' = c_3$. If not, stop.
25. Output $(\texttt{ok}, tsid, SID, P_{C_a})$ to $P^{(12a)}$

For $(X, Y) \in \{(C_a, S_b), (S_b, C_a)\}$:

On input $(\texttt{send}, SID, P_X, P_Y, |m|)$ from $P^{(12a)}$:

1. Choose random noise n of length $|m|$
2. Compute $t = \text{MAC}_{rs_{MAC}}(seq \| n)$
3. Compute $c = \text{E}_{rs}(n \| t)$
4. Send $(\texttt{send}, coID, P_Y, c)$ to $\mathcal{F}_{\text{NET}}$
5. Store $seq = seq + 1$

On input $(\texttt{receive}, coID, P_X, c')$ from $\mathcal{F}_{\text{NET}}$:

1. Check that $c' = c$, if not, stop.
2. Store $seq = seq + 1$
3. Output $(\texttt{deliver}, P_X, P_Y, SID)$ to $P^{(12a)}$

We see that $P^{(12a)}$ is the ideal functionality for the Application data protocol/Record protocol, $\mathcal{F}_{\text{TLS-REC}}$. $P^{(12b)}$ does both sides of the Handshake protocol and the Application data protocol/Record protocol. Since we have only split the simulator into two

parts, and not added or removed anything that affects its behaviour, there is no change from the previous game, so $\Pr[E_{12}] = \Pr[E_{11}]$.

We know from the previous proofs that if the client outputs $(\texttt{ok}, SID, P_{C_a}, P_{S_b})$ after the handshake, where P_{S_b} is the identity of an honest server, then the client is in fact communicating with P_{S_b}, and not any other party (or the attacker). Since only $P_{C_{a,i}}$ and $P_{S_{b,j}}$ have the necessary encryption keys, it follows from INT-CTXT that a ciphertext produced with any of these keys must come from one of these two instances (which of the upstream or downstream key was used decides which party the message came from), and not any other party, or the attacker.

Since the message m never leaves P_{C_a} and P_{S_b}, no other party (or the attacker) will ever see the content. This also implies that it can not be changed along the way.

We let the simulating attacker, $\mathcal{S}$, be defined as $P^{(12b)}$, $\mathcal{F}_{\text{CPKE}}$, $\mathcal{F}_{\text{NET}}$, $\mathcal{F}_{\text{SIG}}$ and the dummy attacker, $\mathcal{D}$, just like in the previous proofs.

Game 13 In Game 13 we let the simulator be the ideal functionality, where $\mathcal{S}$ is as we defined above. Since we have only renamed $P^{(12a)}$ (it is exactly the same as $\mathcal{F}_{\text{TLS-REC}}$), and defined $\mathcal{S}$ to be $\mathcal{D}$, the ideal functionalities and $P^{(12b)}$, and not changed how the simulator behaves in any way, we have that $\Pr[E_{13}] = \Pr[E_{12}]$.

We can compute the attacker's advantage like we did in Chapter 5.2. Then we get the same telescoping sum as in the renegotiation proof, so 2^{-176} is an upper bound for the attacker's advantage against the Handshake protocol and the Application data protocol together with the Record protocol. Since this is negligible, we have that the protocol UC-realizes the ideal functionality. Hence it is secure according to the UC theory.

Chapter 8

Conclusion

We have now carried out three security proofs of different TLS subprotocols. In the first proof, in Chapter 5, we analysed the Handshake protocol with renegotiation, including the new fix. This proof was perhaps the most interesting one, since the protocol including the fix had not been analysed using the UC security framework before when this master thesis was produced. In this proof we got the result that the attacker's advantage against the protocol was negligible. This means that the fix does not only stop the renegotiation attack (like we showed in Chapter 4.4), but also that it does not create any new security vulnerabilites, so that the protocol can be considered secure.

In Chapter 6 we did a similar proof of the Handshake protocol with one session resumption, and in Chapter 7 we studied the Application data protocol together with the Record protocol. The result in both of these proofs was that the attacker's advantage against the protocols is at most 2^{-176}, which is negligible. This means that it is a 2^{-176} probability that the attacker can tell the difference between the actual protocols and the ideal functionalities. Hence, we can conclude that both protocols are secure, according to the UC theory.

However, it is important to be aware of the fact that this kind of proof can not necessarily guarantee that the protocols are secure against all possible attacks. These proofs are theoretical exercises, and if we prove that the protocol is "secure", meaning that it UC-realizes the ideal functionality we have defined for it, the protocol will only be as secure as the ideal functionality we are comparing it to. In addition, how TLS is implemented on a particular system can also affect the security, which we do not consider at all.

Still, this kind of proof will provide us with valuable knowledge about the security in the protocol. We can conclude that the protocol is resistant to many types of attacks. This proof technique can also be quite useful if we, for instance, have designed a new cryptographic protocol, and want to check if we have made any apparent mistakes.

Bibliography

[1] Ran Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Cryptology ePrint Archive, Report 2000/067, 2000. Updated version from 2005.

[2] Ran Canetti and Jonathan Herzog. Universally Composable Symbolic Security Analysis. Cryptology ePrint Archive, Report 2004/334, 2004. Updated version from 2009.

[3] T. Dierks and E. Rescorla. *RFC5246: The Transport Layer Security (TLS) Protocol Version 1.2.* IETF, 2008.

[4] O. Goldreich, M. Bellare, and H. Krawczyk. Stateless evaluation of pseudorandom functions: Security beyond the birthday barrier. `www.wisdom.weizmann.ac.il/~oded/PS/bgk.ps`, 1999. Downloaded November 21 2009.

[5] Shafi Goldwasser and Mihir Bellare. Lecture Notes on Cryptography. `http://cseweb.ucsd.edu/~mihir/papers/gb.html`, 2008. Downloaded March 24 2010.

[6] Marsh Ray. Authentication Gap in TLS Renegotiation. `http://extendedsubset.com/?p=8`, 2009.

[7] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. *Transport Layer Security (TLS) Renegotiation Indication Extension.* IETF, 2010.

[8] Victor Shoup. Sequences of Games: A Tool for Taming Complexity in Security Proofs. Cryptology ePrint Archive, Report 2004/332, 2004. Updated version from 2006.

[9] William Stallings. *Cryptography and network security.* Pearson Prentice Hall, 2006.

[10] Douglas Stinson. *Cryptography, theory and practice.* Chapman and Hall/CRC, 2006.

[11] Frode Sørensen. *Innføring i nettverk.* IDG Norge Books, 2004.