Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

Attacks on re-keying and renegotiation in Key Exchange Protocols

Bachelor Thesis

Rati Gelashvili
Supervision: Cas Cremers

May 31, 2012

Abstract

The TLS protocol has been a subject of studies, analyses and verification attempts over the years, but a recently discovered attack against the key renegotiation in the TLS protocol underlined the need to more thoroughly study the key renegotiation phase and focus on aspects not investigated before.

We study the key renegotiation phase of the TLS protocol and use formal models for automatic verification and detection of potential security flaws. The models we developed capture the vulnerability that led to the recently discovered attack against TLS. We also investigate the key renegotiation aspects of two other protocols, IKEv2 and SSH2.

Contents

1 Introduction

Designing secure cryptographic protocols is a major and a very important challenge. This subject has been studied intensively and numerous advancements have been made. One of the advancements is the automated verification of the cryptographic protocols. Automated verification can help to establish security properties and to provide the desired security proofs. It is also useful to detect potential attacks and vulnerabilities. Automated verification is the most applicable for small and medium-sized security protocols, such as TLS, the Transport Layer Security protocol.

The TLS protocol is the backbone of the security of many Client-Server applications in Internet. Numerous formal analyses of the TLS protocol have been conducted over the years, for example [11], [18]. However, previous analyses failed to detect a recently discovered man-in-the-middle attack [15], [6] on the key renegotiation in TLS.

Key renegotiation, also called re-keying, refers to a process by which a protocol generates new keys and updates the security parameters in use. The re-keying functionality is included in many security protocols for various reasons that will be discussed later.

Consider a simple key exchange protocol between two parties, the initiator I and the responder R. The protocol involves just two messages from I to R. The first message is designated to exchange the initial key K, and along the key contains the identities of the parties, I and R. It is signed by I to provide authenticity and encrypted by the public key of R to provide secrecy. Thus, the key K gets securely shared between I and R. The second message is for renegotiating the key from K to a new key K'. It contains K', the identities I and R and is secured using the key K. As only I and R know K, K' gets securely shared between I and R.

The protocol is depicted in Figure 1. We write $pk(X)$ to denote the public key of X and $sk(X)$ to denote the secret key of X.

1. $I \rightarrow R : \{\{K, I, R\}_{pk(R)}\}_{sk(I)}$
2. $I \rightarrow R : \{I, R, K'\}_K$

Figure 1: Example Key Exchange Protocol

In TLS, the situation is more complex. The key renegotiation is conducted by a sub-protocol of TLS, that renegotiates the security parameters and refreshes the keying material. The same sub-protocol is also responsible for the initial negotiation where the initial keys and security parameters are established. This sub-protocol has been analyzed before, but often the focus was on the initial key exchange phase and possible renegotiations were not considered in enough detail. Therefore, the above mentioned man-in-the-middle attack on the TLS key renegotiation went unnoticed.

The goal of this Bachelor Thesis is to study the construction of formal models where the re-keying and renegotiation behavior of the key exchange protocols is well-analyzed. We build the protocol models in the Scyther framework [3] and use the Scyther tool for automatic verification of security properties of the models.

We conduct formal analysis of the TLS protocol and build models that capture the mentioned TLS key renegotiation attack. There is a trade-off between usefulness and preciseness of the model: we would like to have a reasonably abstract model to work with, but at the same time the model should stay very close to the original protocol description when it comes to analyzing security properties relevant to the key renegotiation. Therefore, we carefully design our own model with emphasis on the key negotiation and renegotiation, instead of enhancing existing models from the literature. In addition, we model, analyze and discuss potential ways of avoiding the attack against TLS. We also analyze the re-keying behavior in the Internet Key Exchange

protocol IKEv2 and the Secure Shell protocol SSH2 and check whether these protocols could experience similar problems as TLS.

In Part I of the thesis the TLS key renegotiation attack [15] is studied in detail: in Section 2 we present the TLS protocol. In Section 3 we explore the key renegotiation weakness of the TLS protocol, associated problems and potential fixes. In Section 4 we explain the process of building the models of TLS in the Scyther framework and automatic detection of the attack.

Part II of the thesis is dedicated to the analysis of the key renegotiation in IKEv2 and SSH2: in Section 5 we analyze IKEv2 and give informal explanation of why it is not vulnerable to a similar renegotiation attack as TLS. In Section 6 we analyze the key renegotiation in SSH2, re-discover a weakness and explore it.

In Section 7 of the thesis the work is summed up. Appendix A contains the important parts of the Scyther models. Appendix B contains the figure of an attack, generated by Scyther. Appendix C contains a lookup table for the symbols and abbreviations used throughout the thesis.

Part I
TLS and key renegotiation attack

2 The TLS protocol

We start the section by explaining the high level structure of the TLS protocol in Section 2.1. After defining the necessary notations in Section 2.2, we cover the two most important sub-protocols of TLS in detail, the TLS Record protocol in Section 2.3 and the TLS Handshake protocol in Section 2.4. Finally, we present the message sequence charts of the TLS protocol in Section 2.5.

2.1 General Overview of the TLS protocol

TLS stands for the "Transport Layer Security". As the name suggests, the TLS protocol secures the communication between the client and the server above the transport layer. Many important higher level protocols, such as the HTTPS protocol, rely on the TLS protocol. The TLS protocol consists of several sub-protocols, most important sub-protocols being the TLS Handshake protocol and the TLS Record protocol.

The TLS Record protocol is the lowest layer of the TLS protocol. It is responsible for the fragmentation, compression, application of MAC and encryption and transmission of the messages. The precise functionality of the TLS Record protocol is defined by the values of its parameters. The MAC and encryption keys are examples of the parameters of the TLS Record protocol. The parameters are initilized with default values. The default values for the keys are $NULL$, meaning that initially the TLS Record protocol does not use MAC or encryption. The TLS Record protocol with fixed parameter values defines the precise functionality that we will call a TLS Record layer session.

The values of the parameters of the TLS Record protocol, including the keys, are updated using the TLS Handshake protocol. The TLS Handshake protocol, on its behalf, transmits messages over the existing TLS Record layer session. After a successful handshake - a successful invocation of the TLS Handshake protocol, the TLS Record protocol updates the parameter values. Before the first handshake the TLS Record protocol uses the default parameter values,

so it does not use MAC or encryption and therefore the messages of the first handshake are neither secured by a MAC nor encrypted.

The first handshake is called an initial negotiation and all the subsequent handshakes are called the renegotiation. The historical purposes of the renegotiation functionality in TLS were:

- Allowing re-keying of long-lived sessions. Using the same cryptographic keys for too long is considered to be a bad practice. Cryptoanalysis becomes easier if there are more packets encrypted by the same key. Using the short-term keys also reduces the consequences if a key is compromised.
- Resetting the sequence numbers. In TLS, the client and the server keep sequence numbers to detect extra, missing or duplicate messages. The sequence number is included in the MAC. Sequence numbers are not allowed to wrap, thus once they overflow the session should be renegotiated.
- Changing the set of cryptographic parameters, for instance, to change the cipher suite in use.

Nowadays, the main use of the TLS Renegotiation feature is very different from the historical purposes. It is used to protect the identity of the client, when the optional Client Certificate Authentication, explained in Section 2.4, is enabled. If the client would have to send the public key certificate in the initial unencrypted handshake, the identity of the client would be exposed in cleartext. To protect the identity, the client often authenticates using the certificate in the second handshake immediately following the initial unauthenticated handshake.

We will now present the notation that will be used throughout the thesis. In particular, this notation will be used to explain the TLS Record protocol and the TLS Handshake protocol in more detail.

2.2 Notation

- We denote concatenation by a comma. The comma is also used for delimiting the arguments of a function and the exact meaning will be clear from the context. For example, $f((x, y), z)$ denotes the application of a function f to two arguments: the first argument is x concatenated with y and the second argument is z.
- We use the notation $RANDOM_n$ to denote the procedure that returns n uniformly random bytes.
- TLS protocol uses a pseudo-random function that we call PRF. We write PRF_n(secret, label, seed) to denote n bytes produced by the pseudo-random function with 3 arguments: secret, label and the seed.
- PRF uses a construction known as $HMAC$. $HMAC$ is based on a hash function that we denote by $HASH$. $HASH(s)$ computes the hash of s. s could for example be $(m_1, m_2, \ldots, m_n)$, a concatenation of n messages.
- We write $\{m\}_k$ to denote a message m encrypted by the encryption key k. If k is a pair of MAC and encryption keys, then the message is encrypted by the encryption key of k.
- We write $MAC_k(m)$ to denote a MAC of the message m generated by the MAC key k. If k is a pair of MAC and encryption keys, then the MAC key of k is used.

Using this notation, we can now explain the TLS Record and Handshake sub-protocols in more detail.

2.3 TLS Record protocol

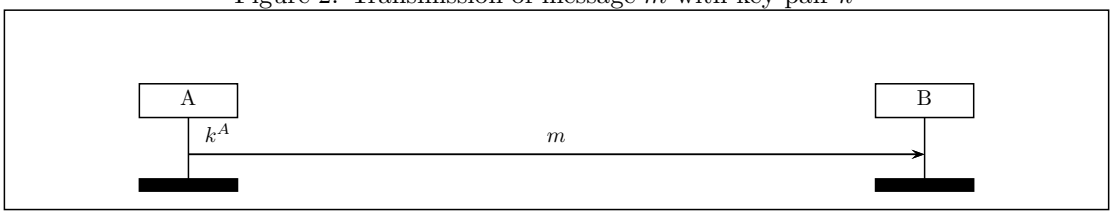
TLS Record protocol is the lowest layer of the TLS protocol, responsible for the fragmentation, compression, application of MAC, encryption and transmission of the messages. It is also used by other TLS sub-protocols, example being the TLS Handshake protocol. The most important parameters of the TLS Record protocol are the client/server write MAC keys and the client/server write encryption keys. The values of these parameters are exactly the keys that will be used for applying MAC and encryption. The write encryption key of a party P is used by P to encrypt transmitted messages and the read encryption key of P is used by P to decrypt received messages. The write and read MAC keys are used analogously.

k^C denotes the pair of the client C write keys in use: the client's write MAC key and the client's write encryption key. The server holds a copy of these keys referred to as the server read keys.

k^S denotes the pair of the server S write keys is use: the server's write MAC key and the server's write encryption key. The client holds a copy of these keys referred to as the client read keys.

We often use message sequence charts to depict protocols. In such charts, we write $\xrightarrow{k^A\ m}$ to denote the transmission of the message m with k^A being the pair of MAC write and encryption write keys of the transmitting party. An example is the message sequence chart in Figure 2. The message m is preprocessed, secured by a MAC and encrypted before transmission by the TLS Record protocol, so the actual transmitted message is $\{m, MAC_{k^A}(m)\}_{k^A}$.

Figure 2: Transmission of message m with key pair k^A



As already mentioned in Section 2.1, no MAC and encryption is used by the TLS Record protocol for the first TLS handshake. For this stage, we denote the pair of write keys to be $NULL$, $k^C = NULL; k^S = NULL$, that is, no MAC and no encryption. After the successful initial TLS handshake, the client gets the pair of write keys $k^C \neq NULL$ and the server gets the pair of write keys $k^S \neq NULL$. After every successful renegotiation the key pairs k^C and k^S are updated and stay not equal to $NULL$.

2.4 TLS Handshake protocol

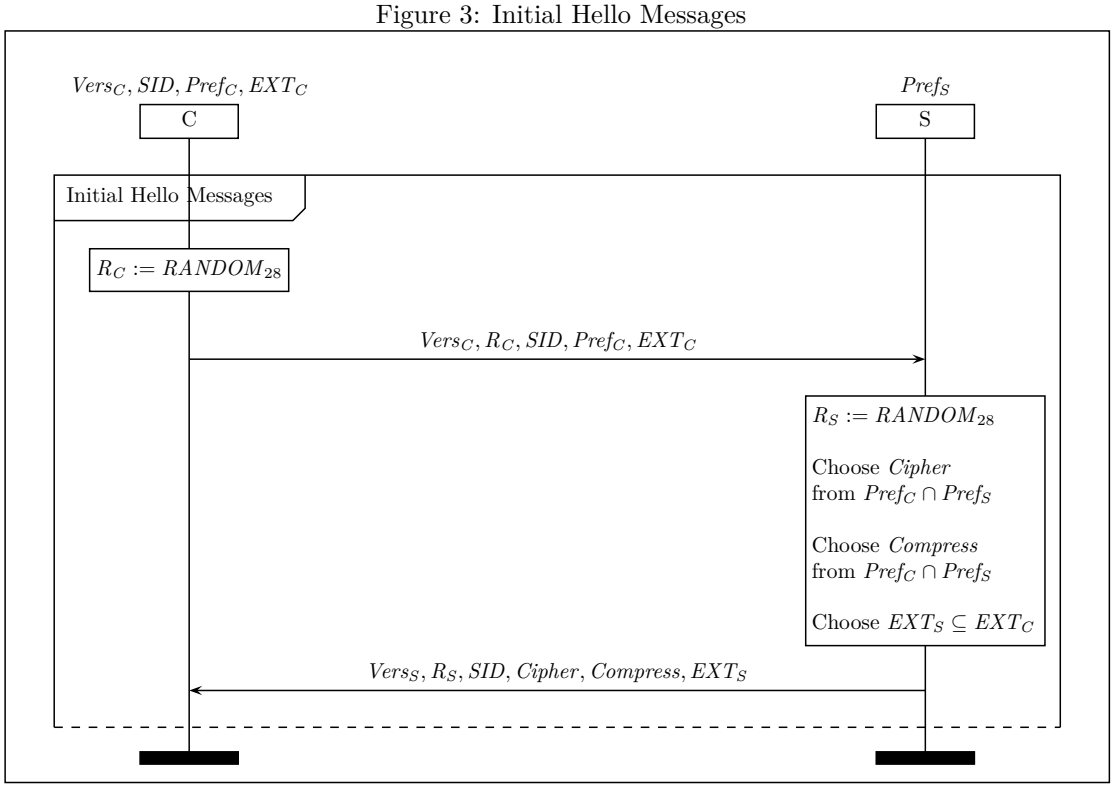
For presentation, we break the TLS Handshake protocol down to three consecutive phases and call them Setup Phase, Negotiation Phase and Finishing Phase. In the following sections we explain these phases in order.

2.4.1 Setup Phase

The TLS Handshake protocol starts with the Setup Phase. The client transmits the *ClientHello* Message using the TLS Record protocol. The server receives the *ClientHello* Message and replies with the *ServerHello* Message. The initial Hello messages are described in the message sequence chart in Figure 3.

The server could have sent a *HelloRequest* message before the *ClientHello* message. *HelloRequest* is an optional constant message which does not even force the client to start the handshake, it is just an offer to do so.

Figure 3: Initial Hello Messages



Here are the meanings of the terms used in Figure 3:

- $Vers_C$ is the TLS version suggested by the client and should be the latest version supported by the client. $Vers_S$ is the minimum of $Vers_C$ and the highest TLS version supported by the server. For TLS v1.2 the version is $\{3, 3\}$ and for TLS v1.0 the version is $\{3, 1\}$.
- R_C and R_S are the random values generated by the client and the server.
- *SID* is the session ID. The client sends $SID = NULL$ to generate new security parameters, or a *SID* of an old session to ask the server for resumption. In the last case, the server tries to find a match for the *SID* and if it is found parties proceed directly to exchange the Finish messages, explained in Section 2.4.3. Otherwise, the server generates a fresh *SID* and sends it. The protocol description does not otherwise specify how the *SID* is generated. The server may also send $SID = NULL$ to indicate that the session will not be cached.
- Using EXT_C, the client requests additional functionality. $EXT_S \subseteq EXT_C$. For instance, the server Name Indication SNI is an extension that specifies the hostname that the client wants to connect to. The SNI extension then allows the server to present different certificates for the same IP address and port number depending on the hostname. This is needed when several HTTPS websites share the same IP address, because they should have different certificates.

- $Pref_C$ contains the preferences of the client for the cipher and the compression algorithms. $Pref_S$ contains the preferences of the server. If $Pref_C \cap Pref_S = \emptyset$, then the server responds with a Handshake Failure alert, otherwise it selects the cipher and the compression algorithm from the intersection. Each *Cipher* defines a key exchange algorithm, a bulk encryption algorithm, MAC algorithm and a pseudo-random function PRF. The client and the server can change their preferences from a session to a session.

Key exchange algorithm used in the TLS Handshake protocol can be RSA, RSA_PSK, DH_DSS, DH_RSA, DHE_DSS, DHE_RSA or DH_ANON. The exact characteristics of these algorithms are described later.

2.4.2 Negotiation Phase

The precise structure of the negotiation phase of the TLS protocol is defined by the exact key exchange algorithm selected in *Cipher*. If the key exchange algorithm is not DH_ANON, then the negotiation phase starts with the Server Certificate Authentication. If the selected key exchange algorithm is DH_ANON, then the server does not authenticate itself to the client. For some key exchange algorithms, like DHE_DSS, the server is required to provide additional information to the client. This is done using a special message called *ServerKeyExchange*.

The next step is optional Client Certificate Authentication. If enabled, it includes the certificate request from the server, reply message from the client with the certificate of the client and the certification verification message.

After the server is done sending the certificate and/or the *ServerKeyExchange* messages to support the key exchange, it sends the *ServerHelloDone* message. The client upon receiving the *ServerHelloDone* message should verify the parameters provided by the server and only proceed if the verification succeeds. The exact type of the parameters depends on the selected key exchange algorithm. For instance, if the algorithm is not DH_ANON, then the client should have received a certificate from the server and has to verify it.

The client will then use the parameters provided by the server to securely share a value of pre-master-secret PMS with the server. The exact method for securely sharing the value with the server depends on the type of parameters provided by the server, thus it depends on the selected key exchange algorithm. For example, if the selected key exchange algorithm is RSA, then the server provides the client with its public encryption key and the client uses this key to encrypt PMS, securely sharing the value with the server.

In the end, regardless of the choice of key-exchange algorithm, the client and the server will securely share a value PMS. From the PMS the parties obtain the Master Secret $MS := PRF_{48}(PMS, \text{"master secret"}, (R_C, R_S))$. The exact keys used for MAC and encryption by the client and the server, k^C and k^S are then derived from the MS. Let l be the total length of those 4 keys. First $key\_block := PRF_l(MS, \text{"key expansion"}, (R_C, R_S))$ is computed. Then, key_block is fragmented in four consecutive pieces to receive $k^C(MAC), k^S(MAC), k^C(\text{encrypt})$ and $k^S(\text{encrypt})$. k_C and k_S will be the next key pairs to use.

Once the key pairs that will be used next are computed, they are stored, but not yet set in use. We refer to these key pairs as the pending key pairs. For each party, there will be one pending key pair for writing and one pending key pair for reading. Pending keys are set in use during the finishing phase.

2.4.3 Finishing Phase

Finishing Phase consists of *ChangeCipherSpec* and *Finished* Messages. The *ChangeCipherSpec* message is a single byte with a value of 1. Reception of this message causes the receiver to

instruct the record layer to immediately start using the pending key pair for reading. Immediately after sending the *ChangeCipherSpec* message the sender must instruct the record layer to start using the pending key pair for writing. Thus, the later traffic will be protected by the newly negotiated values. The Finished messages are the first messages that are encrypted by the new key pairs. Therefore, for correct communication it is essential that the *ChangeCipherSpec* message is received between other handshake messages and the *Finished* message.

Finished messages contain verification data called the *verify_data* of some defined fixed length. The usual length is 12 bytes for the finish message of the client and 24 bytes for the finish message of the server. Verification data is generated by applying the *PRF* to the following parameters: the secret is equal to *MS*, the label is either "client_finished" or "server_finished" and the seed is equal to *HASH* applied to all the prior messages of the current TLS handshake. The optional constant messages like *HelloRequest* and *ClientCertificateRequest*, as well as the *ChangeCipherSpec* message are not included in the hash computation. *ChangeCipherSpec* is omitted, because its functionality is defined as a standalone sub-protocol of TLS and thus the message is not a part of the Handshake protocol.

The assumption here is that the *Finished* messages provide a mechanism to check the integrity of all previous messages and therefore, a mechanism to check the integrity of all the parameters transmitted, including extensions. Finished messages should be validated first, and only then the transmission of data may start. If validation fails, the protocol should abort.

2.5 Message Sequence Charts of the TLS handshake

We will use the Scyther framework [3] to automatically verify formal models of TLS. But before building the Scyther models, we build the message sequence charts to describe the TLS protocol. Using message sequence charts is a common way of depicting security protocols and is very helpful to better see and understand the instrinsics of the depicted protocol.

We had to make some modeling choices to build the message sequence charts and later, the Scyther models. These choices are listed and motivated in Section 2.5.1.

2.5.1 Modeling Choices

While designing our message sequence charts and Scyther models, we made the following choices concerning the abstraction/precision tradeoff

- We ignored the initialization vectors, that are the parameters of the TLS Record protocol. Initialization vectors are not required by all the ciphers and ignoring them simplifies the models.
- The optional Client Certificate Authentication is omitted in the message flow charts. This helps to simplify the charts and to better focus on the problematic scenario. However, when building the Scyther models, we also consider the case when the Client Certificate Authentication is enabled.
- Every TLS handshake starts with a *ClientHello* message, but starting a handshake can be triggered by server sending a *HelloRequest* message. The *HelloRequest* message from a server serves as an invitation to a handshake and does not oblige the client to send the hello message, it also does not participate in the handshake verification data in the *Finished* messages. Therefore, we choose to ignore the *HelloRequest* message.
- When describing the TLS Record protocol, we omitted the fact that the MAC of the record also includes the sequence number so that missing, extra, or repeated messages are detectable.

- A message m will get fragmented if it is longer than the TLS Record protocol packet size. Each individual fragment would be secured by a MAC and encrypted by the TLS Record layer before the transmission. The message m gets reassembled upon receival before being delivered to higher-level clients. The fundamental security measures are MAC and encryption. We do not consider the potential fragmentation and reassembling of the message and assume that the transmitted message is simply $\{m, MAC_k(m)\}_k$, where k denotes the encryption and MAC key pair in use.
- Depending on the exact key exchange algorithm selected, the TLS handshake is executed differently. We built four charts, given in Figures 4, 5, 6 and 7 to describe these different executions.

Now we present the message flow charts describing the TLS handshakes using different key exchange mechanisms.

2.5.2 TLS with the *RSA*/*RSA_PSK* algorithms

If the *Cipher*'s key exchange algorithm is *RSA* or *RSA_PSK*, then the Certificate Key Type is *RSA* public key, which can also be used for encryption.

The public key is provided in the certificate chain, so we will write $Cert(PK_S)$, where PKE_S is the *RSA* public key of the server, that can be used for encryption.

This case is illustrated in the message flow chart in Figure 4.

2.5.3 TLS with the *DH_DSS*/*DH_RSA* algorithms

In this case the Certificate Key Type is Diffie-Hellman public key, signed with either *RSA* or *DSS* public key, which is provided in the certificate chain, so we can write $Cert((g^a \bmod p, g, p))$, where $(g^a \bmod p, g, p)$ is the Diffie-Hellman public Key. Here and later the client and the server use the value g^{ab} as the pre-master-secret *PMS*.

This case is illustrated in the message flow chart in Figure 5.

2.5.4 TLS with the *DHE_DSS*/*DHE_RSA* algorithms

Here the Certificate Key Type is either *RSA* or *DSS* signing public key PKS_S. Let $DHP_S = (g^a \bmod p, g, p)$ be the server's Diffie-Hellman Parameters, where $g^a \bmod p$ is the server's Diffie-Hellman public value.

The client can not secretly share the pre-master-secret $PMS = g^{ab}$ with the server using just PKS_S. Therefore, the server also sends DHP_S in a separate message, signed by PKS_S.

This case is illustrated in the message flow chart in Figure 6.

2.5.5 TLS with the *DH_anon* algorithm

In this case the server does not authenticate using the certificate at all and it only provides its Diffie-Hellman Parameters ($g^a \ mod\ p, g, p$), obviously unsigned.

This case is illustrated in the message flow chart in Figure 7.

Figure 4: TLS with RSA/RSA_PSK

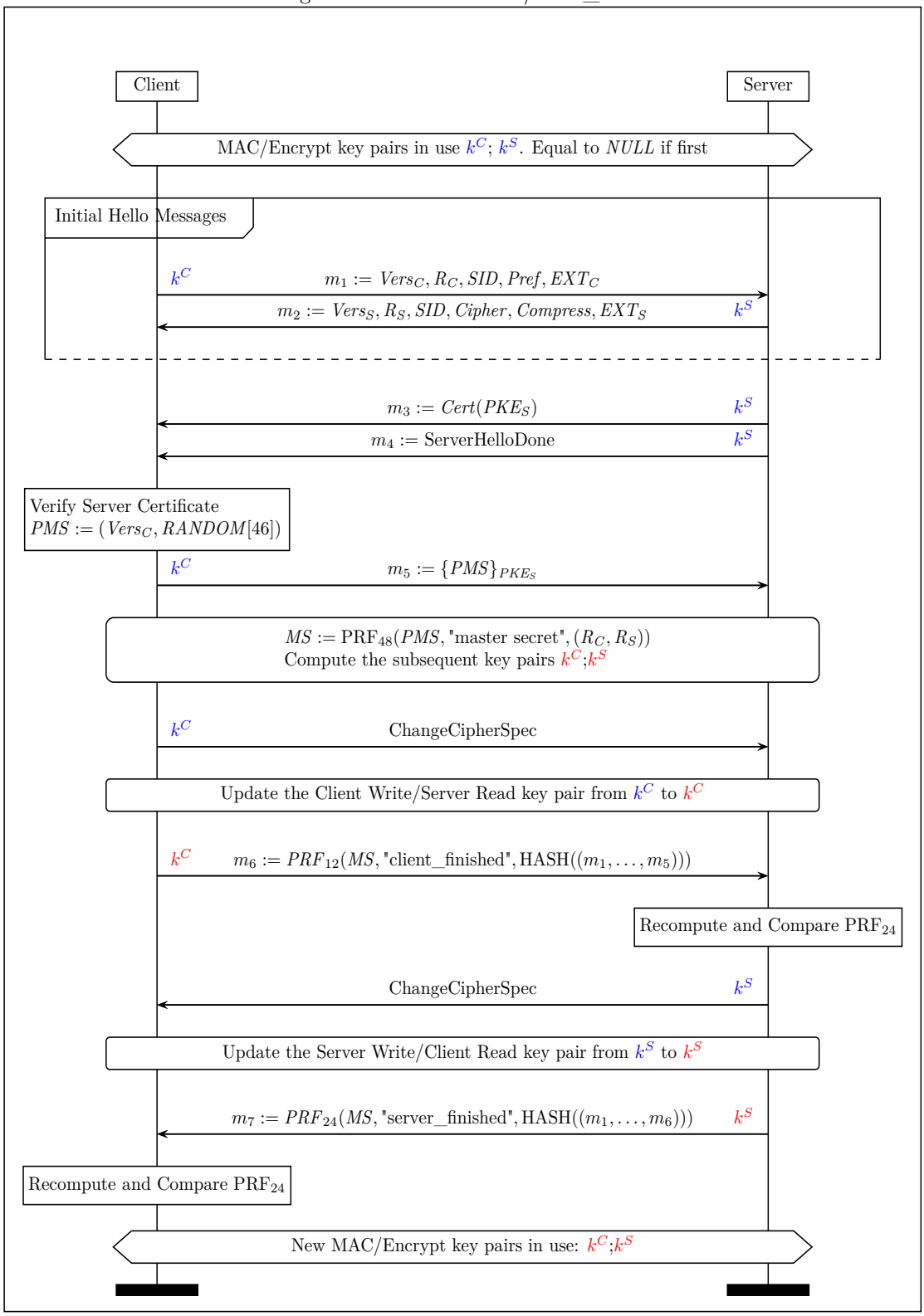


Figure 5: TLS with DH_DSS/DH_RSA

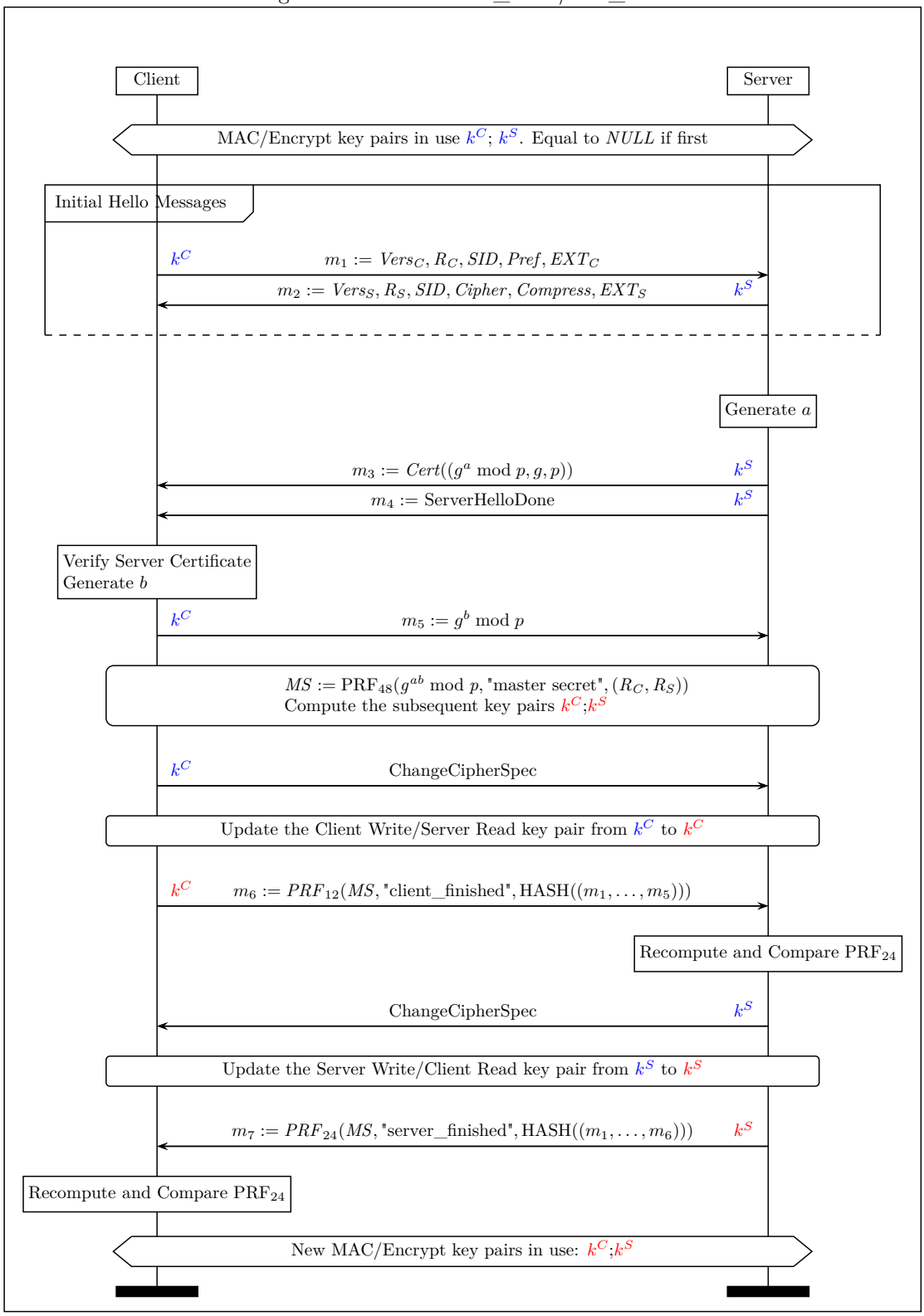


Figure 6: TLS with DHE_DSS/DHE_RSA

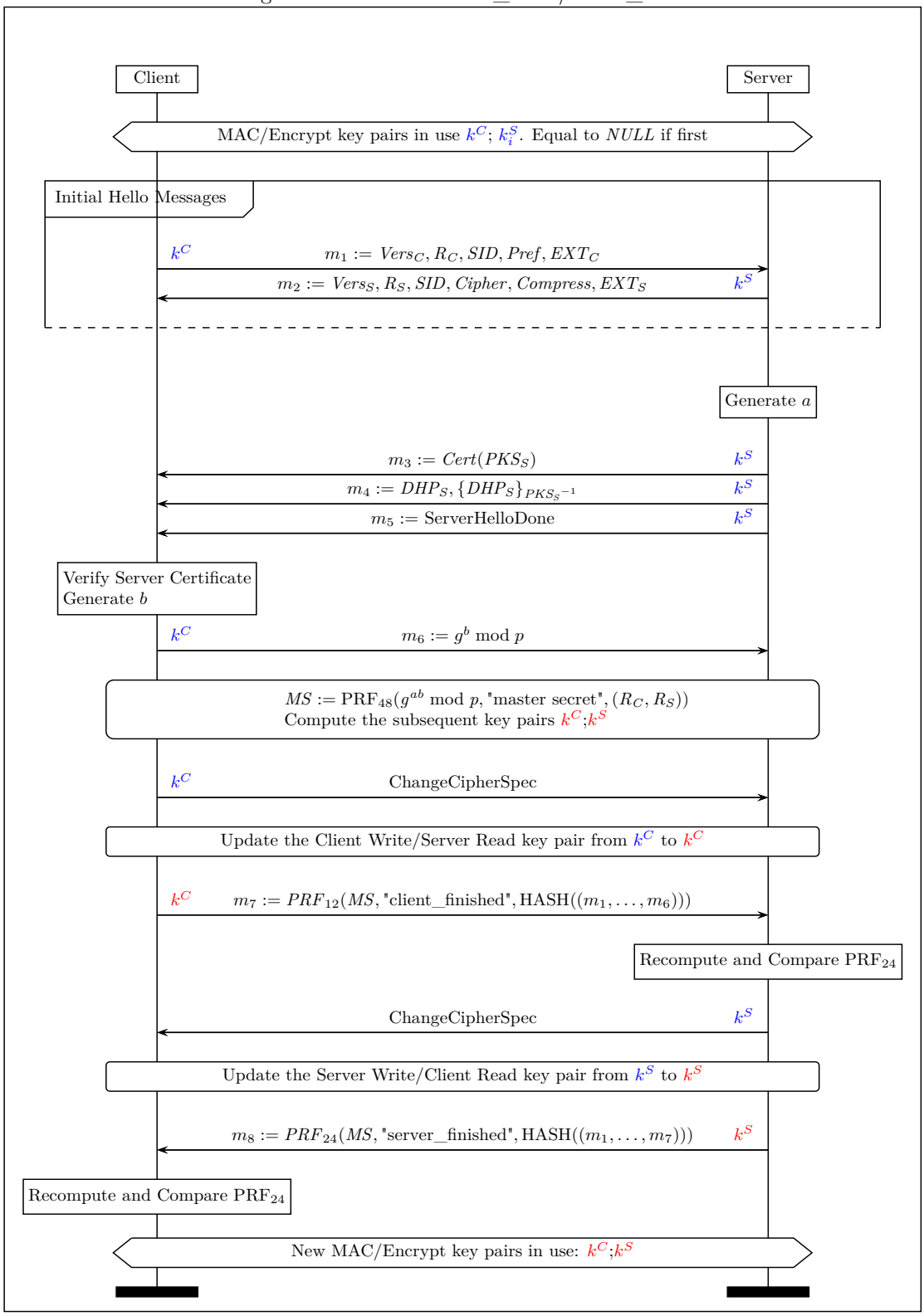
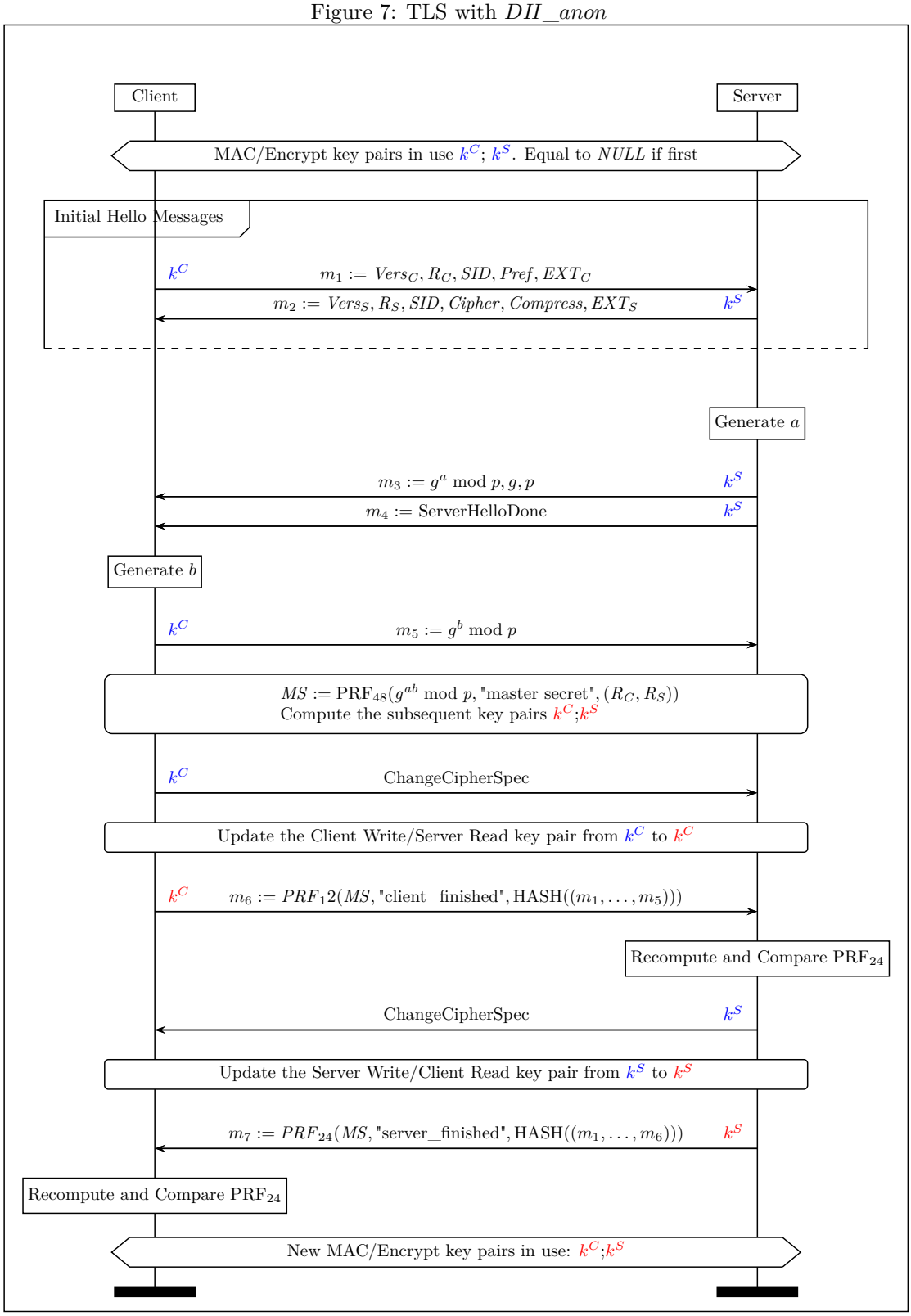


Figure 7: TLS with *DH_anon*



3 TLS key renegotiation vulnerability and the fix

The Man in the Middle attack against the TLS key renegotiation was discovered by Marsh Ray [15]. The exact implications of the attack depend on the application protocol running over TLS, with Marsh Ray presenting an exploit against the HTTPS protocol demonstrating the importance of the problem. Other protocols, including SMTPS and FTPS were also affected by the TLS renegotiation vulnerability [17]. The Internet Engineering Task Force IETF has since suggested a TLS protocol level fix for the vulnerability [10].

We explain the TLS renegotiation attack in Section 3.1. To clarify the implications of the vulnerability, we describe the exploit against the HTTPS protocol in Section 3.3. We proceed by discussing the violated security properties of the TLS protocol in Section 3.4 and the potential solutions to the problem, including the suggested fix by IETF in Section 3.5.

3.1 Overview of the problem

It is important to note two facts about the TLS renegotiation. First, there is no cryptographic binding between the subsequent TLS Record layer sessions. The initial negotiation and the renegotiation of the TLS Record layer session are both an invocation of the same TLS Handshake protocol. The TLS Handshake protocol uses an existing TLS Record layer session to negotiate a new session. But the TLS Handshake protocol is not cryptographically bound to the existing TLS Record layer session - the handshake messages are independent of the ongoing TLS Record layer session that secures and transports them. Therefore, there is no cryptographic binding between the subsequent TLS Record layer sessions.

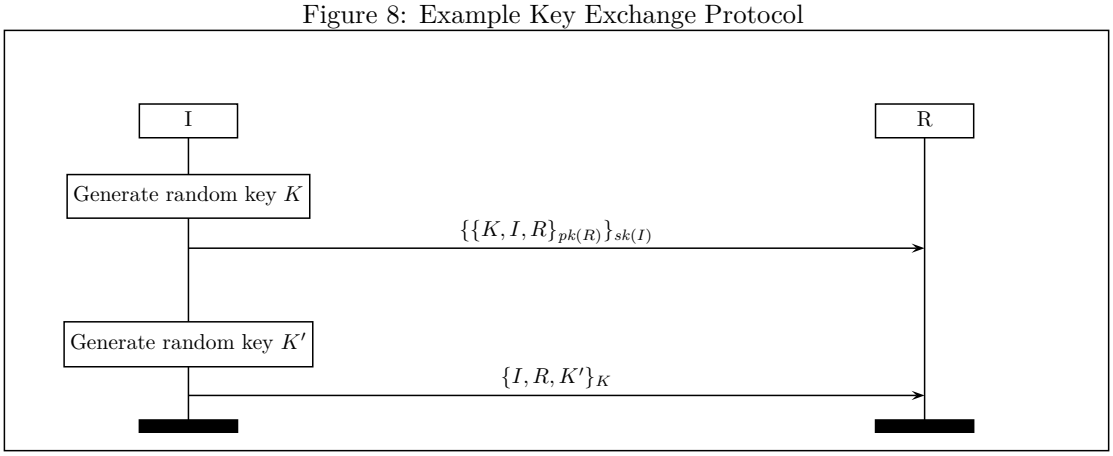
The second fact to note is that the Server does not always know the identity of the client. The Client Certificate Authentication is optional in TLS. In fact, in most cases, even if the Client Certificate Authentication is enabled in TLS, the server will not know the identity of the client after the initial negotiation. The reason is that in order to avoid exposing the identity of the client in cleartext, the Client Authentication is generally avoided in the unencrypted initial negotiation.

So, the server does not explicitly know the identity of the client, neither does the server have any cryptographic guarantees about the subsequent TLS Record layer sessions. In particular, there is no mechanism for the server to check that the client identity stays the same in the renegotiated session. As it turns out, the identity may indeed change. The TLS key renegotiation vulnerability actually bases on this possible identity switch during the TLS renegotiation.

To better understand the concept of an identity switch during key renegotiation, reconsider the simple key exchange protocol given on the message sequence chart in Figure 8. This hypothetical protocol was explained in Introduction and also illustrated in Figure 1.

The identity switch can occur in the simple protocol renegotiation, because there is neither a cryptographic mechanism to guarantee that the identities I and R contained in the renegotiation message match the identities I and R from the initial negotiation message nor do the parties check it. R does not check that the initiator I in the first message is the same as I in the second message and assumes after the second step that the identity of the initiator is I as in the second message. A malicious Initiator I could negotiate a key K with the honest Responder R, conduct illegal activities against R using K and then insert identities I' and R in the renegotiation message. If R believes that the identities can not change through renegotiation, then R will believe that the key K' is shared with I' and even more, R will believe that the key K was shared with I'. The consequences are severe. In general, an attacker could play the role of Initiator and insert any identities in the renegotiation message. Therefore, we can not expect identities to be the same before and after the renegotiation and in order to capture the identity

Figure 8: Example Key Exchange Protocol



switch in Figure 8, the key renegotiation message should have a form $\{I', R', K'\}_K$.

The possible identity switch in TLS renegotiation has a more complex structure. It turns out that the server may not assume that the client does not change through renegotiation. Alas, in reality, correct and secure functionality of many services using TLS rely on this exact implicit assumption. But there have been no critical concerns before Marsh Ray presented a serious exploit against the HTTPS protocol based on using this incorrect assumption about the client identity invariance [15]. Since, other protocols like SMTPS and FTPS have also been affected [17].

It is interesting to know how common it was to make the incorrect assumption about the client identity invariance while designing the services using TLS. To answer this question, we can recall the most common and typical modern use case of the TLS renegotiation discussed in Section 2.1, of not exposing the Client Certificate to cleartext, which also does rely on the assumption that the identity of the client does not change.

Now we can take a closer look the attack scenario and the TLS renegotiation vulnerability.

3.2 The Attack Scenario at TLS level

At TLS level, the Man in the Middle attack works as follows: The attacker forms a TLS connection with the target server. Then, the attacker intercepts an initial handshake from the victim client to the target server. Finally, the attacker tunnels the initial handshake of the client as a renegotiation through its own, that is, attacker's TLS connection with the server. This scenario is illustrated in Figure 9, where different colors denote different key pairs.

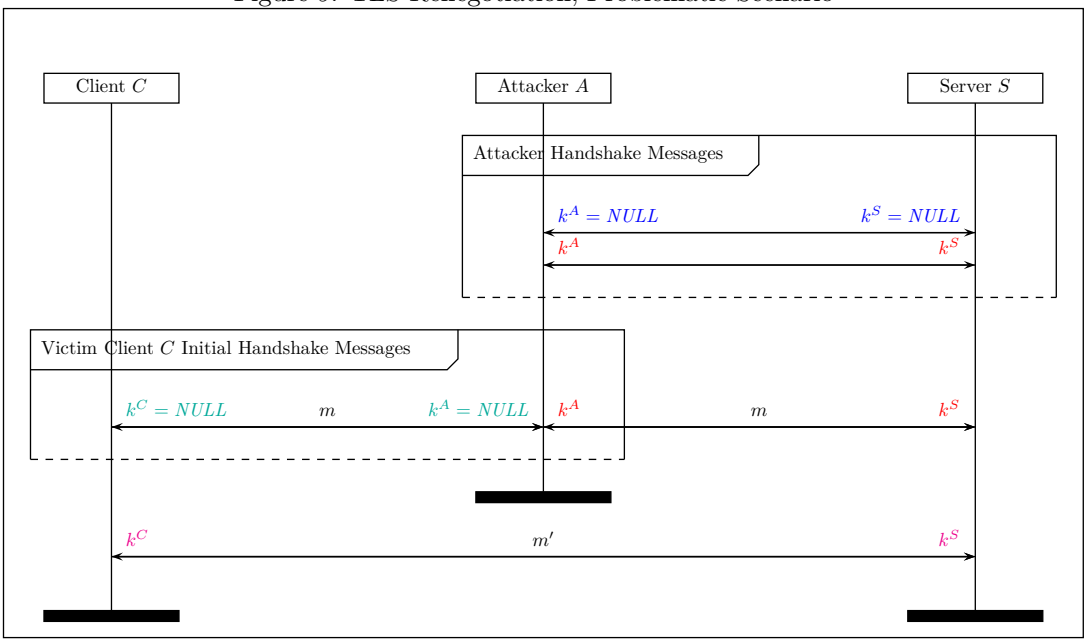
The identity change of the client occurs during the above TLS renegotiation from the attacker to victim. This identity change from the attacker to the victim is exactly what enables the attack against the HTTPS protocol. Next, we present the HTTPS exploit to highlight the practical implications of the vulnerability.

3.3 Understanding the HTTPS exploit

There are several key insights to understanding how exactly the HTTPS exploit works, as described by Marsh Ray in [15]:

- The attack exploits the HTTPS protocol making an incorrect assumption about the underlying TLS protocol, particularly about the client's identity invariance before and after the

Figure 9: TLS Renegotiation, Problematic Scenario



renegotiation. Precisely, according to the HTTPS protocol, given that the renegotiation was successful, the data transmitted before and after the renegotiation is considered to be the same continuous data flow. The net effect is that the attacker can prepend the commands transmitted by the client with his own commands.

- TLS Record layer partitions the data, consisting of HTTPS requests pending transmission, into fragments and does not respect the original message boundaries [8]. Therefore the service provider, that is the HTTPS server has to keep the buffer of received data and execute the commands when they are complete. This allows the attacker to prepend the traffic of the client with an incomplete HTTPS request without the server noticing it.

Now using those insights, consider a scenario of a pizza delivery webpage taking HTTPS requests for pizza delivery. Upon receiving a new HTTPS request for pizza delivery, the server authenticates the client using a cookie, bills the client using the information from the cookie and then pizza is delivered to the address specified in the HTTPS request. Cookie Authentication is used for security, while using cookies simplifies bookkeeping.

The attacker forms a connection with the server and request a pizza delivered to his address, ending the HTTPS request with "X-ignore-what-comes-next: " and no line break. Such a request is not executed by the server because it is considered incomplete and waits in the server buffer for completion, only to be later prepended to the traffic of the client. Then, the attacker asks for renegotiation and splices in the intercepted negotiation from the victim client. If the victim client orders a pizza through his own secure channel to his own address and pays, his HTTPS request for pizza delivery will be ignored as instructed by the prepended attacker's request, pizza will be delivered to the attacker, paid by the victim client. This is obviously a critical security violation.

It needs to be observed that an attacker does not get access to the subsequent traffic of the victim client, but the attacker is able to significantly alter the future behaviour of the system.

This was one example of exploiting HTTPS and Marsh Ray presents several similar exploits in [8].

3.4 Analysis of TLS's security properties

It is never explicitely mentioned in the current TLS specification [8] that the server can safely make the assumption about the other party's identity. Therefore, one may believe that the problem does not lie in the TLS protocol and that it should be fixed in the applications and higher-level protocols rather than in TLS. Another supporting argument is that modifying TLS is a major challenge, i.e. backward-compatibility is one important issue.

One can alternatively consider the cause of the problems to be the imperfections of the TLS protocol, such as that there is no way to tell renegotiation from negotiation and that the renegotiation is not bound to the previous handshake. Even if the way protocol operates does not contradict with its specifications, some undesired behavior definitely manifests itself. If the possibility that an involved party can change the indentity through renegotiation is not a bug of the TLS protocol, it is definitely not a feature and intended property either. The manifested behavior is surely something typically not expected from key renegotiation. Fixing the TLS protocol and denying the identity switch would be an universal fix to every existing and potential exploit of the identity invariance assumption, like the attack against the HTTPS protocol. This motivates us to investigate the associated problems on the underlying TLS protocol level.

After investigating the security properties of the TLS protocol, we observed that the agreement on data property [12] does not hold between the victim client and the server during the problematic scenario in Figure 9. If a server accepts the final verification of a Handshake, the client is expected to have sent all the preceding TLS Handshake messages in a form that they arrived to the server. In particular, the server expects the client to have sent $\{m_1\}_k$, where k denotes the MAC and encryption keys used by the TLS Record layer session and applying k means applying the corresponding MAC and the corresponding encryption. Whereas, in case of an attack the client would have sent the message m_1, not $\{m_1\}_{k_a}$ where k_a denotes the keys used in a session between the attacker and the server. Analogously, if the client accepts the final verification, he expects the server to have sent m_2. Whereas, in case of an attack the server has sent $\{m_2\}_{k_a}$, not m_2.

3.5 The fix to the TLS renegotiation problem

It is very natural to assume that the entity, renegotiating a channel should still be the one connected by it after the channel is renegotiated. We have seen that the applications and higher standing protocols make this assumption and rely on it. Some functionalities of those applications/protocols is crucially dependent on this assumption. If we have a closer look at the reasons why the renegotiation functionality was created, there is the same underlying implicit assumption - if an entity renegotiates to change the cipher suite, update keys or avoid wrapping the sequence number, then it should be the same entity willing to continue the TLS session and not some other entity. There also seems no reason for an entity to start communication with a server by possibly renegotiating through a foreign channel instead of just starting a new TLS session.

These arguments imply that the TLS protocol should be fixed by enforcing the property that the identities of the communicating entities do not change after the TLS renegotitation. Although, it is hard to talk about the identities when the Client Certificate Authentication is optional and even the server's certificate is not always present, as in DH_anon.

One idea to fix the problem is to make the Client Authentication mandatory and check that

the identities are the same after each renegotiation. At the first glance, this approach seems to fix the renegotiation problem of TLS. By having a closer look, it would introduce other major problems:

- The TLS protocol is designed to be abstract and flexible, so that various applications and higher-level protocols can use it. Many of those applications/protocols do not need the mandatory Client Certificate Based authentication, so the change would make TLS useless for them.
- If the Client Certificate Based Authentication was mandatory, the client identity would be exposed to the plaintext during the initial negotiation. The reason is that if the Client Certificate Authentication is enabled, then it is enabled for every handshake including the initial unencrypted negotiation and this policy can not be changed without major modifications of the TLS protocol. Encrypting the client identity in the initial negotiation seems to be an option but it is not trivially possible in every version of the protocol and also requires major changes to the workflow of TLS.
- As our analysis shows in Section 4.5, there exists a dual TLS key renegotiation attack. We would therefore need to enforce that the identity of the server also does not change through renegotiation. To reason about the identity of the server, the server Authentication should also be made mandatory. This would eliminate DH_anon from the possible key exchange algorithms, making TLS even less useful.

To solve the problem of reasoning about identitities nicely, we should focus on what exactly needs to be captured. We do not need to know the identities, we just want to capture that the identity of an entity on the other side of the channel is the same before and after the renegotiation. This means that the entity after the renegotiation should know all the messages of the immediately previous Handshake, equivalently, it should know the *verify_data* of the immediately previous Handshake. This property does not hold for TLS in the problematic scenario in Figure 9, because the victim client does not know the *verify_data* from the attacker's initial Handshake. If the victim would know the *verify_data*, then the victim would not be a different entity from the attacker in our view.

The suggested fix [10] defines a "TLS Renegotiation Extension" to enforce the knowledge about the enclosing channel by the renegotiated party. We can explain the suggested fix using the notation used in message sequence charts of the TLS Handshake protocol. The "Renegotiation Extension" should be supported by all TLS implementations. It includes the data field that is either empty if this is the first negotiation, or contains *verify_data* from the immediately previous handshake, which established the enclosing TLS Record layer session, the session where the renegotiation is ongoing. For instance, the *verify_data* for the client is $\mathrm{PRF}_{12}(MS, \text{"client\_finished"}, \mathrm{HASH}(m_1, \ldots, m_5))$. The clients should generate the Renegotiation Extension with every handshake and include in the demanded extensions EXT_C^{FIX}. The servers should generate the Renegotiation Extension in response to any client which offers it.

To check that the attack is effectively avoided, observe that the attacker needs to modify the EXT_C^{FIX}, because it includes the data describing the first negotiation that the server will not accept as a part of a renegotiation. The extensions are part of the TLS Handshake Hello messages and will be included in *verify_data* messages for the renegotiation instantiated by the attacker. As the messages that were sent and received differ, the verification would fail, the new channel between the victim and the server would not be established and the identity switch and the attack would thus be prevented.

Actually, it would be enough if only the verification data of any TLS protocol renegotiation was cryptographically bound to the previous handshake/session. So, if the renegotiation extension would not be sent, but would just be a part of the verification message, the attack would

still be denied. This is just an observation and has no practical implications for us, because of the way TLS protocol is constructed and the verification data is defined.

Another observation is that we could have also denied the attack by cryptographically binding every renegotiation to the initial negotiation. This could be accomplished by, for instance, letting the renegotiation extension contain the verification data of the initial negotiation instead of the immediately preceding one. As we will see, this is the way sessions are bound together in the Secure Shell SSH2 protocol, and the binding also serves as the unique identifier for all the renegotiated sessions of a single ongoing communication. For SSH2, it is important to have such an unique identifier, but for TLS it is not required. Therefore, it is better to just bind subsequent pairs of renegotiations together, because this can be accomplished by using the existing extension functionality without introducing more major changes into the TLS protocol.

4 Detecting the renegotiation attack using Scyther

Scyther [3] is a framework for automatic verification of security protocols. To be able to use Scyther for the verification of TLS, we first need to build Scyther models of the TLS protocol.

We have seen that the precise execution of the TLS protocol can differ significantly depending on the configuration of TLS. To conduct a complete verification, we build several different Scyther models corresponding to the different message sequence charts in Section 2.5 depicting the different variants of TLS execution.

After building the desired Scyther models we use the Scyther framework to verify TLS and make sure the verification results capture the renegotiation vulnerability. Our next step is to integrate potential fixes into the Scyther models of TLS and verify that the vulnerability is resolved.

In the next section, we draw attention to the process of building models of TLS that would capture the renegotiation problem. We describe important general aspects of building formal models to allow detection of the renegotiation problems. We also explain the process of modeling in Scyther, and the high-level choices made during the process.

4.1 Building a formal model of TLS

To explain the aspects involved in building a model and verifying a key exchange protocol with special focus on key renegotiation recall the simple protocol in Figure 8. In the introduction, we looked at both individual protocol messages shown in Figure 1 in isolation and did not find problems, but in Section 3.1 we revisited the key exchange protocol and identified a potential identity switch.

Informal reasoning about the key negotiation and key renegotiation stages did not suggest that there was a problem with the simple protocol. In order to detect a key renegotiation vulnerability of a protocol, we can not afford to simplify protocol model by separating negotiation and renegotiation stages and making assumptions about their interaction - this type of reasoning about the stages in isolation presumably contributed to not detecting the TLS attack for a long time. It is crucial that the formal model used for verification contains both the key negotation and renegotiation phases together. The key-renegotiation should base upon and directly follow initial negotiation.

Another important aspect is that the model has to capture the connection between the stages as detailed as possible. If we would build a formal model of the simple key exchange protocol in Figure 8 for verification, in that formal model the renegotiation message should have the form similar to $\{I', R', K'\}_K$. This is because the protocol only passes on the key to the renegotiation phase, but not the identities. If the identities from the negotiation and renegotiation are bound

to be the same in the model, the attack may not be detected by verification, as the identity switch is essential for the key-renegotiation attack.

In order to build a TLS model in Scyther we have to define the roles of the TLS protocol. The role in a Scyther model describes the functionality of a participating party of a protocol. As we are modeling TLS, our models will have two roles, the Client and the Server. During verification, an agent assumes a role and acts according to the assumed role definition. The agent assuming the Client role in our model should try to do the initial negotiation and renegotiation with the agent assuming a Server role, while the Server role should provide the functionality to allow Client negotiation and renegotiation. The role descriptions should correspond to the TLS message sequence charts given in Section 2.5.

As we have discussed in Section 3.4, the broken property in the TLS during the key renegotiation attack is agreement on data [12]. The agreement on data property can be easily modeled in the Scyther framework. One possibility is to claim the Niagree property for the Scyther protocol model. Alternatively, the agent taking a certain role should commit to some values it shares with another agent during the protocol execution. For every Commit claim by an $Agent_1$ with an $Agent_2$ on specified values, there should be a Running claimed by the $Agent_2$ with the $Agent_1$ on the same values during the execution. This way of checking the agreement on data is more general, flexible and powerful, therefore we will use it in our Scyther models.

In the next sections, we consider different Scyther models for the important TLS variations.

4.2 Basic Model for TLS with the *RSA*/*RSA_PSK* algorithms

The Scyther model for TLS with *RSA*/*RSA_PSK* is the easiest to construct and explain. The main reason for its simplicity is that it does not involve Diffie-Hellman exponents, because Diffie-Hellman exponentiation must be approximated by additional rules in Scyther. Also, in this basic model, the optional Client Certificate Authentication is omitted. Therefore, the data agreement is only checked for the agent assuming the Client's role.

The exact description of the model is given in Appendix A.1. It closely follows the corresponding message sequence chart from Section 2.5.

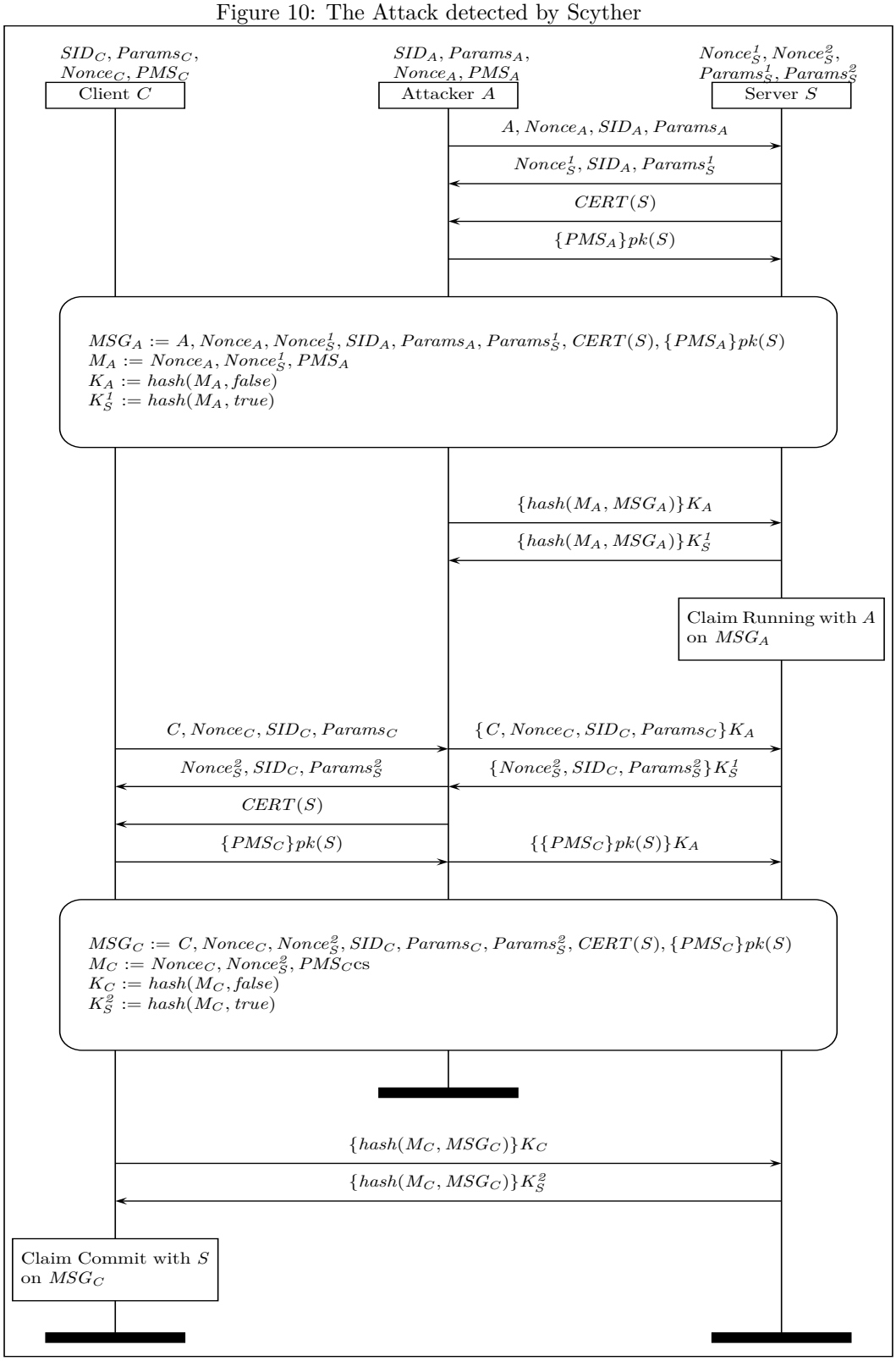
Note that the Client is executing the TLS protocol with the Server, while the Server provides the negotiation and renegotiation functionality for some Client C. As the Client Certificate Authentication is optional and the Server can not know the identity of the client, the client C can provide anything as its name in the renegotiation. This reflects the identity change. As shown in the message sequence chart in Figure 3, the identities of the parties are not part of the initial hello messages of the handshake. They are only included in the Scyther model for better illustrating the attack and the identity switch.

The attack by Scyther is shown in the message sequence chart in Figure 10. The chart is slightly simplified for presentation. The Commit claim without corresponding Running claim corresponds to the violated agreement on the data property. See Appendix B for a figure that more closely resembles the output of Scyther.

4.3 Modeling TLS with Diffie-Hellman exponentiation

Modeling the Diffie-Hellman exponentiation in Scyther is not trivial. The difficulties arise because of the need to model the equational theories, like expressing the knowledge that $g^{ab} = g^{ba}$, using the constructs of the spdl language. But in the spdl language and in Scyther, there is no direct way to express the fact that two expressions that are syntactically different are equal due to an algebraic property.

Figure 10: The Attack detected by Scyther



To work around the problem, we can take the same underapproximation approach as in [4]: to use a one way function p, where $p(g, x)$ corresponds to g^x and we define helper protocols. One helper protocol is needed to simulate the public knowledge of the public keys: given some PK, the protocol returns $p(g, sk(PK))$.

Another helper protocol makes sure that if an agent knows an expression involving g^{ab} then the agent can know the expression where g^{ab} is replaced with g^{ba}. The protocol receives an arbitrary expression of the first form from the environment and returns the algebraically equivalent, but not syntactically equivalent expression of the second form. It is important to include all possible expressions, where such a modification is necessary.

These helper protocols model only a strict subset of the equational properties. It is hard to be sure that every possible execution is captured. Also, because for each expression involving the Diffie-Hellman exponent it is needed to define several roles of a helper protocol, the verification complexity increases drastically. Therefore, it is an appealing possibility to consider modeling the TLS with Diffie-Hellman exponentiation in a different verification framework, with a more natural way to express equational properties of expressions, such as the Tamarin tool [16].

Appendix A.2 contains the changes that need to be made into the basic TLS model with RSA encryption from the previous section in order to get a working Scyther model for TLS with the Diffie-Hellman exponentiation using the above described techniques.

4.4 Model with the *DH_anon* algorithm

In the *DH_anon* setting there are no guarantees on the identities of the client and the server. Therefore the authentication properties will not hold.

On the other hand, if we enable the optional Client Certificate Authentication, we may require the authentication property for the Server role, so the agent assuming the Server role could commit to the values of the protocol run with the honest client. This leads to a dual attack where the server's identity switches during the renegotiation. Note that the switch of server's identity is not possible for any other TLS variant. In the attack scenario, at first, the intruder establishes the connection with the victim client and as the server he would have to authenticate in every setting except *DH_anon*.

TLS with *DH_anon* is not often used in practice. As the server is not authenticated, the client has to have different trust mechanisms. Then again, in order for the dual attack to function, the client should establish the connection with the malicious server, which can still be blamed on the client, because such types of hazard was to be expected without requiring the server authentication. Also, it is not clear whether this dual attack would lead to any exploit similar to the HTTPS exploit. Finally, the suggested fix to the original attack also deals with such a such dual attack, because both the server and the client during the renegotiation are forced to know about the enclosing TLS Record layer session.

We continue the discussion in Section 4.5, where we cover models where the optional Client Certificate Authentication is enabled.

4.5 Models with the enabled Client Certificate Authentication

Even though we decided to build the models and charts of the TLS protocol without considering the Client Certificate Authentication, we can not totally overlook the possible executions of the TLS protocol variants with enabled Client Certificate Authentication. These executions are as relevant as any other possible execution of the TLS protocol, so we have to make sure that even with the enabled Client Certificate Authentication our reasoning about the general TLS protocol execution is correct and the derived observations also hold. The best way to detect any

possible differences or to make sure that the observations hold is to model the Client Certificate Authentication in Scyther too. To do this, we enhance the existing models by adding the Client Certificate Authentication.

First, by testing the enhanced models in the Scyther framework, we establish the fact that enabling the optional Client Certificate Authentication does not eliminate the attack. This is not very surprising if we consider an attack presented in [15], where the TLS renegotiation attack was used to bypass exactly the mandatory Client Certificate Authentication in the renegotiation phase, which is sometimes deployed by the servers for a higher security.

To gain more intuition why the attack is not eliminated, we should recall that the Client Certificate Authentication is optional in the TLS protocol and there exists no TLS level mechanism that would check the invariance of the Client identities established by the Certificate Authentication. The TLS protocol only would check that the Client Certificate and provided parameters were correct, which would also be the case for the attacker and the certificate of the attacker.

By enabling the Client Certificate Authentication in DH_anon, we get a TLS model with the dual TLS renegotiation attack, where the client is authenticated but the identity of the server changes through renegotiation. The model is almost a mirror reflection of the model that we built in Section 4.3 with Diffie-Hellman exponentiation.

4.6 Modeling the Fix

In order to model the suggested fix [10] and check its effectiveness we should consider the existing models and enhance them with the renegotiation extension. Important aspects are that the extensions should be included in the verification messages and that the extension for the renegotiation contain the verification data of the enclosing TLS session. After incorporating the fix into the models, Scyther framework does not detect attacks in any of the above described models, including the model with the dual attack.

For the models with the incorporated fix, the data agreement is checked for both negotiation and renegotiation phases. This is necessary to establish that there are no attacks, previously Scyther was detecting an attack on the data agreement already after the negotiation phase making it unnecessary to check for the agreement after the renegotiation, because it would also be broken.

The models with the renegotiation extension are described in detail in Appendix A.3.

4.7 Accounting for multi-protocol attacks

We have developed several models of the TLS protocol in the Scyther framework and succeeded in automatically finding the TLS renegotiation attack using Scyther. We have also implemented the suggested fix [10] to the TLS renegotiation attack into all our models and checked that the attack was effectively denied.

The testing using the Scyther framework has been done for individual protocol models in isolation. We have not yet accounted for possible multi-protocol attacks [2]. This type of attacks can arise during the interaction of different variants of the protocols.

We have seen that the TLS protocol models without incorporating the fix are vulnerable to the key renegotiation attack. So, if the client and the server do not use the new renegotiation extension, the renegotiation attack is possible. We have also seen that the TLS renegotiation attack is denied if the server and the client both use the designated renegotiation extension. Thus, we have the models where the client and the server either both use the renegotiation extension, or both do not use the renegotiation extension. Unfortunately, all the clients and servers in the world can not start using the renegotiation extension overnight. Therefore, we

should account for the interactions between clients and the servers when one of the parties does not support the renegotiation extension and we should check if such interactions are safe. This can be done by analyzing the possible interactions between our models for the TLS protocol with and without the fix, the models that we have already analyzed in isolation.

We also have models for several other variants of the TLS protocol, such as the models with enabled Client Certificate Authentication, or the models with Diffie-Hellman exponentiation and dual attack. We would like to account for all the possible interactions between all these variants of the TLS protocol.

Scyther provides the functionality to automatically consider possible interactions of different protocol models and check for the potential multi-protocol attacks. A multi-protocol attack would correspond to a claim that holds in the isolation but is violated in the multi-protocol setting. We have tested all our models of the TLS protocol using Scyther and detected no multi-protocol attacks.

4.8 Advanced adversary models

Scyther framework allows to automatically test the protocol models against a family of adversary models [1]. The adversaries can reveal different parts of the security state during the protocol execution. Conducting such advanced analysis of a security protocol helps to understand its security properties in greater detail. In particular, testing against compromising adversaries helps to identify the structure and the limits of the security of the analyzed protocol.

The Scyther generated result of testing against the compromising adversaries has the form of a protocol security hierarchy. A protocol security hierarchy provides the smallest set of non-standard capabilities that the adversary needs to have to be able to attack the given protocols. It also identifies capabilities that do not help the adversary to attack the specified protocols. For our TLS protocol models, attacking the protocol means breaking the agreement on data claims in the models. We say that a model is secure for an adversary with some specified capabilities if that adversary can not make the data agreement claims violated.

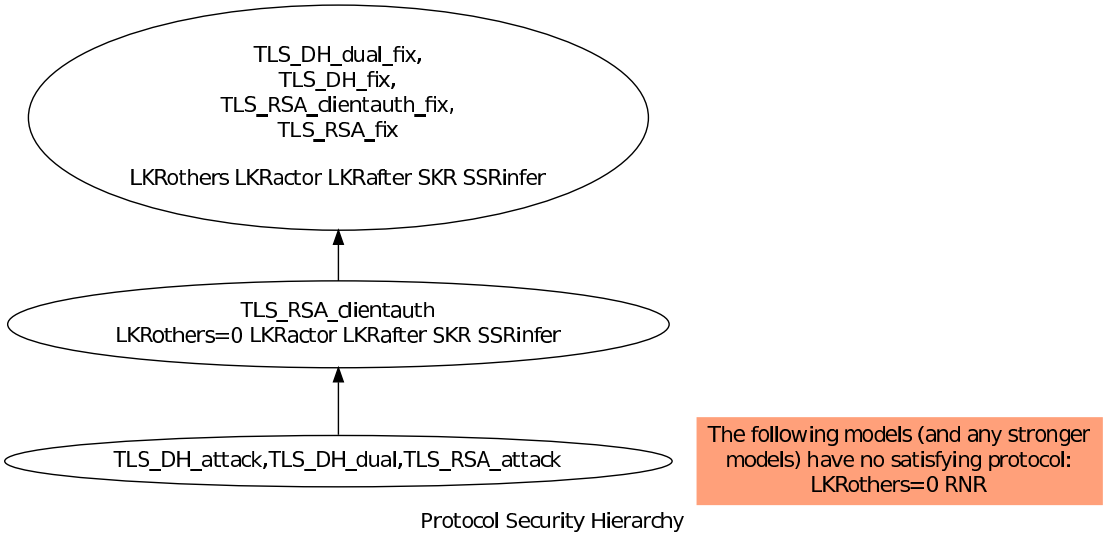
Using Scyther we conducted the automatic analysis of our TLS protocol models against the compromising adversaries. The resulting protocol hierarchy plot is depicted in Figure 11. *RNR*, *LKRothers*, *LKRafter*, *LKRactor*, *SKR*, *SSRinfer* all describe different possible capabilities of the adversary.

RNR capability means that the adversary can learn the random values used during the protocol execution and *LKRothers* allows to compromise long-term keys of a third party not directly involved in the protocol. Standard adversary has the *LKRothers* capability but does not have the *RNR* capability. The results in Figure 11 mean that if the adversary has the *RNR* capability, then the models are not secure. The *RNR* adversary can fake the renegotiation extensionif necessary using the knowledge of the random values and still conduct the renegotiation attack successfully. Such an attack is plausible without being able to compromise long-term keys of any third party, as shown by $LKRothers = 0$.

TLS_DH_attack, *TLS_DH_dual* and *TLS_RSA_attack* are the models of variants of TLS protocols without the incorporated renegotiation fix. Therefore, they are not secure even in the standard environment, as the standard adversary can conduct the TLS key renegotiation attack and violate the data agreement in these models.

TLS_RSA_clientauth is also a model of TLS without the renegotiation fix. However, this model uses the Client Certificate Authentication in both negotiation and renegotiation phases. Thus, in order to conduct the TLS key renegotiation attack, the adversary needs the *LKRothers* capability to compromise the keys of some third party P in order to authenticate as P. If $LKRothers = 0$ is set, the adversary can no longer compromise the keys of a third party, so the

Figure 11: TLS, Compromising adversaries protocol hierarchy



adversary can not authenticate and the model becomes secure.

For the models that implement the fix to the TLS key renegotiation problem and also for the aforementioned $TLS\_RSA\_clientauth$ with $LKRothers = 0$, the following other properties hold:

- No attack against $LKRafter = 1$ means that the models are secure even if the adversary can reveal the long-term keys after the protocol execution.
- No attack against $LKRactor = 1$ means that the models are secure against the Key Compromise Impersonation attacks.
- No attack against $SKR = 1$ means that the session keys are independent.

According to the analysis, the implemented fixes to the renegotiation problem seem to work, but the protocols are always vulnerable to the leakage of the random numbers.

Part II
Related Protocols

Given the existence of the TLS attack, a natural follow-up question is whether the same attack works for other protocols with key renegotiation functionality, such as IKE and SSH.

5 Analyzing IKEv2 protocol

5.1 Overview of the IKEv2 protocol

Internet Key Exchange protocol IKE is a component of the Internet Protocol Security protocol suite IPsec. IPsec is responsible for authenticating and encrypting Internet Protocol packets.

The role of IKE in IPsec is to perform a mutual authentication between the parties and establish and maintain security associations. Security Association, denoted by *SA* [9] is a set of shared security attributes between the parties, such as the encryption keys or the names of cryptographic algorithms is use.

We consider and model IKEv2, the second and the latest version of the Internet Key Exchange protocol.

5.2 Comparison of IKEv2 re-keying and TLS renegotiation

We focus on re-keying aspects of the IKEv2 protocol and check if IKEv2 is vulnerable to a similar key renegotiation attack as the TLS protocol. A good place to start is to compare the re-keying aspects of IKEv2 and TLS, as we have already conducted a comprehensive analysis of TLS and the TLS key renegotiation vulnerability in the first part of the thesis. Inspecting the official description of the IKEv2 protocol [9] and comparing the key-renegotiation related aspects to the TLS protocol, several important differences can be observed.

In IKEv2, the security association *SA* is renegotiated by creating a child security association *CHILD_SA* and then replacing the original *SA* by the *CHILD_SA*. *The identities of the parties sharing the security association do not change.* The only way to change the identities is to use the re-authentication [9] and re-authentication by definition is not vulnerable to a key renegotiation attack.

In IKEv2, a secret value SK_d from the enclosing *SA* is used to create a child security association *CHILD_SA* and its security attributes. The secret value SK_d is computed together with the secret keys of the main security association *SA* and is used solely for the purpose of participating in the creation of child security association. Thus, the child security association is cryptographically bound to the enclosing security association by secret value SK_d, and as the enclosing *SA* is replaced by *CHILD_SA*, we can say that *there is a cryptographic binding the subsequent security associations.*

In a TLS-like key renegotiation attack, when an attacker establishes a session between the parties, the preceding sessions for the parties differ. For instance in the standard scenario in Figure 9, for the Client there is no preceding session while for the Server there is one. Therefore, if a TLS-like key renegotiation attack is attempted against a protocol with a cryptographic binding between subsequent sessions, the parties will hold different cryptographic bindings to the preceding sessions. Thus, the parties would get mismatching values for some security parameters. The protocol is secure if at least one of these mismatching values is verified, ensuring that the protocol terminates with failure if the attack is attempted. If none of the mismatching values are verified and the protocol under the key renegotiation attack does not fail, then the attack succeeds, because the agreement on the data will be trivially violated because of the existing mismatching values.

In the case of IKEv2 and its cryptographic binding between the subsequent IKEv2 sessions - security associations, the new keys would mismatch. The keys are verified when creating the initial security association in IKEv and an attack similar to the TLS key renegotiation attack should not succeed against IKEv2.

To sum up: In the TLS case, there was no cryptographic binding. For IKEv2, there is a cryptographic binding, and its results are verified. Based on it, we can intuitively claim that the Internet Key Exchange protocol IKEv2 should not be vulnerable to the TLS-like key renegotiation attack. To make a more formal statement, we need to formally verify a model of IKEv2.

5.3 Modeling the IKEv2 protocol

In order to formally verify that IKEv2 protocol is not vulnerable to the key renegotiation attack, we need to construct a Scyther model of the IKEv2 protocol, similar to the TLS protocol model we created in 4.2. We use models developed in [5]. In particular, we combine different existing models:

- The models for *IKE_SA_INIT* and *IKE_AUTH* describing the exchange and authentication conducted by the IKEv2 protocol to set up the initial security association *SA*.
- The model for creating the *CHILD_SA* and renegotiation.

By combining these models, we receive the complete IKEv2 model that can be used to verify the re-keying in IKEv2. We conducted the automatic verification of the complete IKEv2 protocol model using Scyther and detected no key renegotiation problem.

6 Analyzing the SSH2 protocol

6.1 Overview of the SSH2 protocol

Secure Shell, abbreviated as SSH is a rather complex network protocol. It is a replacement for the Telnet protocol and currently, the functionalities such as logging into and executing commands on a remote computer typically use SSH. We will focus on SSH2, the more recent version of the Secure Shell, formally described in a span of at least 5 Request For Comments of the Internet Engineering Task Force.

The SSH2 protocol has a layered structure, that is, it consists of layers of protocols operating on top of each other. The lowest level sub-protocol of the SSH2 is the SSH Transport Layer Protocol, responsible for establishing a Secure Shell connection above the Transport Layer. The SSH Transport Layer Protocol ensures the authentication of the server and the confidentiality and integrity of the communication [14] [7].

Other sub-protocols of the SSH2 protocol, such as the SSH Authentication protocol, work on top of the communication channel established by the SSH Transport Layer Protocol. We will focus on modeling and analyzing the SSH Transport Layer Protocol, because it is the sub-protocol of the SSH2 where the initial key negotiation and subsequent key-renegotiations happen. Other sub-protocols of the SSH2 protocol are the users of the functionality provided by the SSH Transport Layer Protocol.

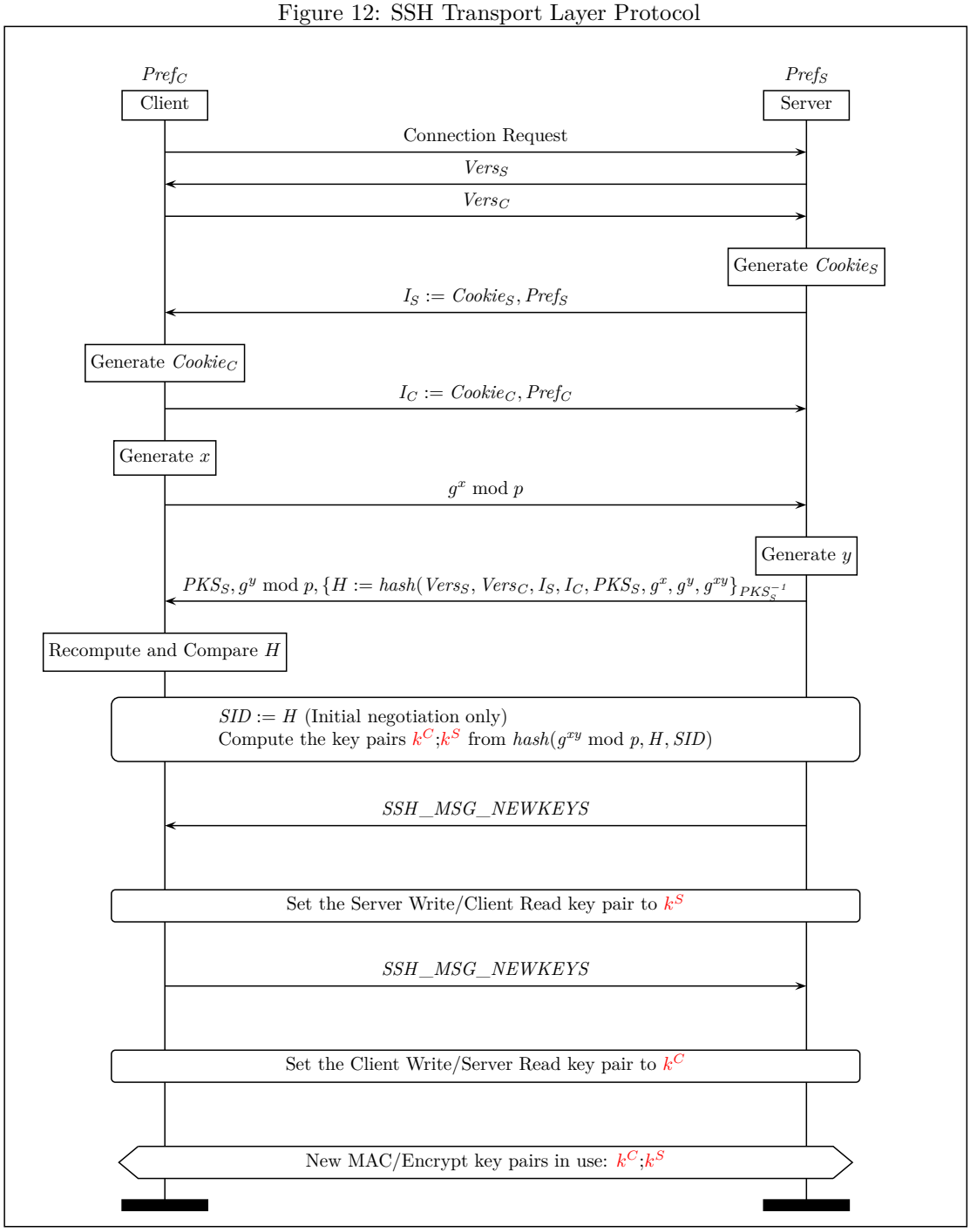
6.2 SSH Transport Layer Protocol

The message flow chart in Figure 12 describes the workflow of the SSH Transport Layer Protocol to establish the initial communication.

The conventions used in the message flow chart in Figure 12 closely match the conventions used before for modeling the TLS protocol. More precisely:

- $Pref_S$ and $Pref_C$ still contain the preferences of the parties.
- $Vers_S$ and $Vers_C$ denote the versions of SSH for the Server and the Client, respectively.
- PKS_S is the public signing key of the Server. This variant of the protocol assumes that the Client is able to verify the public key of the Server.

Figure 12: SSH Transport Layer Protocol



- SID is the Session Identifier of the communication. The session identifier is computed during the initial negotiation and remains unchanged during the key-renegotiations.
- *SSH_MSG_NEWKEYS* for the SSH Transport Layer Protocol is exactly the same as the *change_cipher_spec* for the TLS protocol.
- $x, y, Cookie_C$ and $Cookie_S$ are the nonces, that is, random values generated by the client or the server.
- p is a large safe prime. g is a generator for a subgroup of $GF(p)$.

The key renegotiation happens like in the TLS protocol: the parties just renegotiate the keys, but the communication is conducted through the existing SSH channel. The only difference is that the session identifier stays invariant and is never recomputed.

The key pairs are computed using very similar methods as in the TLS protocol. We should just notice that the negotiated keys depend on the shared secret g^{xy}, the value H, and the unique session identifier SID.

We can now proceed to build a formal model of SSH2 to automatically verify the key renegotiation.

6.3 Modeling the SSH Transport Layer Protocol in Scyther

We developed a Scyther model of the SSH Transport Layer Protocol according to the message flow chart in Figure 12. As for the TLS protocol models, we omitted the initial connection request and version exchange messages. We also had to use the same techniques as in Section 4.3 to model the Diffie-Helman key exchange scheme and the resulting model is very similar in structure to the TLS models with the Diffie-Helman exponentiation. As the Server is authenticated and the Client is assumed to have the capability to verify the Server's public key, we have to place the Commit, representing the agreement on the data claim in the Client role in the SSH protocol model.

Scyther detects the attack on the Agreement on Data property. This is somewhat surprising, because the unique session identifier affects the keys of every session and provides the cryptographic binding among all the SSH communication sessions between the Client and the Server. All of the sessions between the Client and the Server are bound to the initial session, when the SID is computed.

To understand why the attack works, we need to remember that it is not sufficient to only have the cryptographic binding. As outlined previously in Section 5.2, the values that mismatch due to the mismatching cryptographic bindings need to be verified by the protocol. The new keys mismatch in the SSH Transport Layer Protocol but they are never verified. Were the new keys verified, the attack would be denied.

The only value verified in SSH Transport Layer Protocol is the value H, but the only cryptographic binding among all the SSH communication sessions is the session identifier SID. So, if H would depend on value of SID, the attack would be denied. Unfortunately, we can not just include SID as the part of the computation of H, because SID is itself defined as the value of H for the initial negotiation and we would receive an infinite recursive definition.

We could think of three ways of solving the problem and denying the attack:

- Let the Server send an additional verification message to the client, that is secured by the new keys. The attack is be denied, because the new keys depend on the value of SID. Whether or not we include SID in the verification message does not have a crucial importance. This is similar to what happens at the end of the TLS protocol.

- Instead of securing the verification message by new keys, we could also make the server sign it, because for SSH we have the signing keys of the Server which the Client can verify. It is important that the verification message contains SID , but it is also very important that the verification message is different for each session, because otherwise the attacker could just reuse it. A possible choice for the additional verification message is $hash(H, SID)$ and it can be sent along with the message containing H.
- Like in the TLS protocol suggested fix, we can define an extension and let it have some fixed value for the initial negotiation. The extension should be included in the computation of H. For every renegotiation let the extension be SID.

We modeled all of these possible fixes in Scyther and all of them deny the attack.

6.4 The importance of the attack

Even though, just like for the TLS protocol, the agreement on the data property is violated for the SSH2 protocol, there is a substantial difference when it comes to the importance of the attack: In the TLS protocol, the Client and the Server have the same keys and continue the communication, whilst in the SSH2 protocol, the keys are different. The keys being different means that the communication channel can not be used. The very first time when a party would try to use it, the other party would abort. In particular, the other party would abort on the MAC checking phase, because the MACs would be different.

Based on this exact argument, Damien Miller in [13] claims that the key renegotiation vulnerability does not affect SSH. Such an assumption that everything is fine is a dangerous assumption that was also made about the TLS protocol for a long time, before the HTTPS exploit was discovered. Just as the TLS Transport Layer Protocol did not fulfil the properties that it intuitively should, SSH2 Transport Layer Protocol also does not - in the attack scenario it is completed successfully, but in reality the parties end up with different keys, and providing the shared keys is the very task of the SSH2 Transport Layer Protocol, as specified in the official descriptions[7]. In such sense, SSH2 Transport Layer Protocol has even a bigger problem than the TLS counterpart. Whereas, because of the key mismatch, the probability of ever exploiting the broken property of agreement on data seems to be very low. One possibility would be that the attacker makes the Server abort the connection attempts from the same Client with the MAC error over and over again. If some implementation of the Server would then identify the Client as suspicious and for instance black-list it, we would be facing an exploit.

7 Conclusions

In this thesis, we have analyzed attacks on key renegotiation in key exchange protocols. We started by the TLS protocol and then considered re-keying in two other key exchange protocols, IKEv2 and SSH2. We have systematically explored the key renegotiation related aspects of these protocols and developed corresponding formal models in the Scyther framework. Automatic verification of our TLS models detected the infamous TLS key renegotiation attack [15], that was not previously detected by formal verification attempts of TLS [6]. The same approach can be taken and the same methodology can be used to analyze, model and verify key renegotiation of many other key exchange protocols.

Additionally, we discovered a dual attack of the TLS key renegotiation attack. Using the advanced functionalities of Scyther, we analyzed the multi-protocol interaction [2] of the different variants of the TLS protocol as well as their security against Compromising Adversaries [1]. We

also independently re-discovered a key renegotiation weakness [13] in SSH2 when the protocol completes successfully without creating a secure encrypted channel.

The results varied for the three key exchange protocols for which we explored the re-keying aspects, as shown in Figure 13.

Figure 13: Comparison of findings for key exchange protocols

	TLS	IKEv2	SSH2
Cryptographic Binding	No	Yes	Yes
Verification of Binding	N/A	Yes	No
Key renegotiation attack	Yes	No	Yes
Exploitable	Yes	N/A	No

Our work brings the key renegotiation aspects from different key exchange protocols into the same context. The need to consider securing the key renegotiation in the key exchange protocols into one context is illustrated in Figure 13. The IKEv2 protocol is not vulnerable to the key renegotiation attack, so if the practices and measures used to design IKEv2 would be deployed in designing the TLS protocol, TLS would also be secure.

We have also gained insight in the necessary measures to fix the key renegotiation weakness in key exchange protocols by discussing and comparing several potential solutions, including the official suggested fix for TLS [10]. For all the Scyther models that suffered from a key renegotiation attack we implemented possible fixes and verified that the fixed models were attack-free.

As a side benefit, we contributed to the further development of the Scyther framework by reporting unexpected or undesired behavior experienced during the protocol verification process.

In the next sections, we sum up the important lessons learnt from our work and mention potential future work directions.

7.1 Lessons learnt

Based on the work conducted in the thesis, we can now summerize what it takes to design a protocol with a non-vulnerable key renegotiation functionality between the parties participating in the protocol.

- We have to make sure that there is a cryptographic binding between all the subsequent sessions between the parties. This can be accomplished by either cryptographically binding every session to the initial session, or binding every session to the immediately preceding session. If we need some unique identifier for all the sessions in the communication between the parties, then we should bind the sessions to the initial session and use the binding as an identifier, like in SSH2.
- We have to make sure that the parameters that mismatch due to the mismatching cryptographic binding are actually verified. It is enough to verify the parameters only in the initial negotiation.
- We have to consider and verify all possible identity switches in the protocol. For example, in TLS it is essential to also consider the possibility of the server identity switch, leading to a discovery of the dual key renegotiation attack.

7.2 Future Work

There are several interesting possibilities for the future work on the subject:

- While verifying the protocol models using the Scyther framework, we faced very fast growth of the verification state space. Therefore, due to the time constraints, most of the verification had to be conducted with the maximum number of protocol runs set to two. It is desired to consider more runs to account for more complex possible interactions, especially for the models involving the Diffie-Hellman exponentiation using the helper protocols. Because the verification is conducted on the limits of what Scyther can currently do, it would be interesting to explore the ways of improving Scyther and scaling up the verification space.
- Use another modeling framework, such as [16], to precisely model the Diffie-Hellman exponentiation instead of using the Scyther approximation. This resulting models would allow to better reason about the variants of the protocols examined in the thesis, such as the TLS and SSH2 protocols involving the Diffie-Hellman exponents.
- Explore the possibilities to construct an exploit, based on the SSH2 Transport Layer Protocol key renegotiation vulnerability.
- Analyze other protocols and sub-protocols that involve key renegotiation in a similar fashion to detect potential key renegotiation problems. Use the insights and techniques gathered throughout existing models and fixes to propose solutions to any detected vulnerabilities.
- Analyze different variants of the protocols and sub-protocols, like we have done for the TLS protocol. Such analysis is important, because for instance the dual key renegotiation attack in the TLS protocol works in only one variant - *DH_ANON*. One interesting task would be to analyze the SSH2 variant that uses RSA cryptography instead of the Diffie-Hellman key exchange. Diffie-Hellman key exchange was the only option when the SSH2 protocol was introduced, but later the RSA mode was added.

References

[1] D. Basin and C. Cremers. Modeling and analyzing security in the presence of compromising adversaries. In *Computer Security - ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 340–356. Springer, 2010.

[2] C. Cremers. Feasibility of multi-protocol attacks. In *ARES 2006*, 2006.

[3] C. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Proc. CAV*, volume 5123, pages 414–418, 2008. Available for download at `http://people.inf.ethz.ch/cremersc/scyther/`.

[4] C. Cremers. Session-state reveal is stronger than eCK's ephemeral key reveal: Using automatic analysis to attack the naxos protocol. *International Journal of Applied Cryptography (IJACT)*, 2010.

[5] C. Cremers. Key Exchange in IPsec revisited: Formal Analysis of IKEv1 and IKEv2. In *ESORICS 2011*, 2011.

[6] S. Farrell. Why didn't we spot that? *IEEE Internet Computing*, 14:84–87, 2010. `http://doi.ieeecomputersociety.org/10.1109/MIC.2010.21`.

[7] I. E. T. Force. The secure shell (SSH) transport layer protocol. 2006. http://tools.ietf.org/html/rfc4253/.

[8] I. E. T. Force. The transport layer security (TLS) protocol. 2008. http://tools.ietf.org/html/rfc5246/.

[9] I. E. T. Force. Internet key exchange protocol version 2 (IKEv2). 2010. http://tools.ietf.org/html/rfc5996/.

[10] I. E. T. Force. Transport layer security (TLS) renegotiation indication extension. 2010. http://tools.ietf.org/html/rfc5746/.

[11] C. He, M. Sundararajan, A. Datta, A. Derek, and J. C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *ACM Conference on Computer and Communications Security*, pages 2–15, 2005.

[12] G. Lowe. A hierarchy of authentication specifications. pages 31–44. IEEE, 1997.

[13] D. Miller. SSL vulnerability and SSH. 2009. http://lists.mindrot.org/pipermail/openssh-unix-dev/2009-November/028003.html.

[14] E. Poll and A. Schubert. Rigorous specifications of the SSH transport layer. Technical Report ICIS-R11004, Radboud University Nijmegen, 2011.

[15] M. Ray and S. Dispensa. Renegotiating TLS. 2009.

[16] B. Schmidt, S. Meier, C. Cremers, and D. Basin. Automated Analysis of Diffie-Hellman Protocols and Advanced Security Properties. In *CSF 2012*, 2012.

[17] G.-S. Thiery Zoller. TLS/SSLv3 renegotiation vulnerability explained. 2011. http://www.g-sec.lu/practicaltls.pdf.

[18] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *USENIX 1999*, 1999. Revised version, http://www.schneier.com/paper-ssl.html.

A Appendix: Scyther models

A.1 TLS with RSA model

Here is the code of the TLS-RSA-attack.cpp, that upon being preprocessed by the C preprocessor (cpp) produces the basic scyther model of the TLS with the RSA encryption key.

```
#define MSG1 client,nc1,ns1,sid1,pc1,ps1,CERT(server),{pms1}pk(server)
#define M1 hash(pms1,nc1,ns1)
#define F1 hash(M1,MSG1)
#define CLIENTK1 wrap(hash(M1,false))
#define SERVERK1 wrap(hash(M1,true))

#define MSG2 client,nc2,ns2,sid2,pc2,ps2,CERT(server),{pms2}pk(server)
#define M2 hash(pms2,nc2,ns2)
#define F2 hash(M2,MSG2)
#define CLIENTK2 wrap(hash(M2,false))
#define SERVERK2 wrap(hash(M2,true))

#define MSG3 c,nc3,ns3,sid3,pc3,ps3,CERT(server),{pms3}pk(server)
#define M3 hash(pms3,nc3,ns3)
#define F3 hash(M3,MSG3)
#define CLIENTK3 wrap(hash(M3,false))
#define SERVERK3 wrap(hash(M3,true))

#define MSG4 client,nc4,ns4,sid4,pc4,ps4,CERT(server),{pms4}pk(server)
#define M4 hash(pms4,nc4,ns4)
#define F4 hash(M4,MSG4)
#define CLIENTK4 wrap(hash(M4,false))
#define SERVERK4 wrap(hash(M4,true))

usertype Params, Bool, SessionID;

const wrap: Function;
const CERT: Function;
hashfunction hash;
const false,true: Bool;

protocol tlsrsaencrypt(client,server)
{
        role client
        {
                fresh pc1: Params;
                fresh nc1: Nonce;
                fresh sid1: SessionID;
                fresh pms1: Nonce;
                var ns1: Nonce;
                var ps1: Params;

                fresh pc2: Params;
```

```
        fresh nc2: Nonce;
        fresh sid2: SessionID;
        fresh pms2: Nonce;
        var ns2: Nonce;
        var ps2: Params;

        send_!1( client,server, client,nc1,sid1,pc1 );
        recv_!2( server,client, ns1,sid1,ps1 );
        recv_!3( server,client, CERT(server) );
        send_!5( client,server, { pms1 }pk(server) );
        send_!7( client,server, { F1 }CLIENTK1 );
        recv_!8( server,client, { F1 }SERVERK1 );
        claim(client, Commit, server, MSG1);

        send_!9( client,server, {client,nc2,sid2,pc2}CLIENTK1 );
        recv_!10( server,client, {ns2,sid2,ps2}SERVERK1 );
        recv_!11( server,client, {CERT(server)}SERVERK1 );
        send_!13( client,server, {{ pms2 }pk(server)}CLIENTK1 );
        send_!15( client,server, { F2 }CLIENTK2 );
        recv_!16( server,client, { F2 }SERVERK2 );
}

role server
{
        var pc3: Params;
        var nc3: Nonce;
        var sid3: SessionID;
        var pms3: Nonce;
        fresh ns3: Nonce;
        fresh ps3: Params;

        var pc4: Params;
        var nc4: Nonce;
        var sid4: SessionID;
        var pms4: Nonce;
        fresh ns4: Nonce;
        fresh ps4: Params;

        var c: Agent;
        recv_!1( c,server, c,nc3,sid3,pc3 );
        send_!2( server,c, ns3,sid3,ps3 );
        send_!3( server,c, CERT(server) );
        recv_!5( c,server, { pms3 }pk(server) );
        recv_!7( c,server, { F3 }CLIENTK3 );
        claim(server, Running, c, MSG3);
        send_!8( server,c, { F3 }SERVERK3 );

        recv_!9( c,server, {client,nc4,sid4,pc4}CLIENTK3 );
        send_!10( server,c, {ns4,sid4,ps4}SERVERK3 );
```

```
                send_!11( server,c, {CERT(server)}SERVERK3 );
                recv_!13( c,server, {{ pms4 }pk(server)}CLIENTK3 );
                recv_!15( c,server, { F4 }CLIENTK4 );
                send_!16( server,c, { F4 }SERVERK4 );
        }
}
```

A.2 TLS with Diffie-Hellman exponentiation

There are several important differences from the TLS model with the RSA encryption:

- The hashfunction p and *const* g.
- The certificate is defined as

  ```
  #define CERT(x) {p(g,sk(x))}sk(x)
  ```

- The client sends $p(g, dh)$ for some nonce dh instead of the encrypted pre-master-secret.
- Correspondingly, the messages are updated and the shared secret is defined as $p(p(g, sk(server)), dh)$:

  ```
  #define MSG1 client,nc1,sid1,pc1,ps1,ns1,sid1,ps1,CERT(server),p(g,dh1)
  #define M1 hash(p(p(g,sk(server)),dh1),nc1,ns1)
  ```

- The helper protocol for public key publicity is implemented the following way

  ```
  protocol @publickeys(PK) {
          role PK
          {
                  send_!1(PK,PK, p(g,sk(PK)));
          }
  }
  ```

- The helper protocol for replacing expressions has the following structure (it is not complete here, although it can be argued that these roles might actually be sufficient for this model)

  ```
  protocol @DHapproximation(RA, RB) {
          role RA
          {
                  var dh: Nonce;
                  recv_!ra1(RA,RA,p(p(g,dh),sk(RB)));
                  send_!ra2(RA,RA,p(p(g,sk(RB)),dh));
          }
          role RB
          {
                  var dh: Nonce;
                  recv_!rb1(RB,RB,p(p(g,dh),sk(RB)));
                  send_!rb2(RB,RB,p(p(g,sk(RB)),dh));
          }
  }
  ```

A.3 Incorporating the TLS suggested fix into the models

In order to incorporate the TLS suggested fix into the models, we need to define an usertype Extension. Then for the negotiation we write

```
fresh ext1: Extension;
send_!1( client,server, client,nc1,sid1,pc1,ext1 );
recv_!2( server,client, ns1,sid1,ps1,ext1 );

var ext3: Extension;
recv_!1( c,server, c,nc3,sid3,pc3,ext3 );
send_!2( server,c, ns3,sid3,ps3,ext3 );
```

and for the renegotiation

```
send_!9( client,server, {client,nc2,sid2,pc2,F1}CLIENTK1 );
recv_!10( server,client, {ns2,sid2,ps2,F1}SERVERK1 );

recv_!9( c,server, {client,nc4,sid4,pc4,F3}CLIENTK3 );
send_!10( server,c, {ns4,sid4,ps4,F3}SERVERK3 );
```

On the client and server sides, respectively.

Verification messages also include *ext1*, *F1*, *ext3* and *F3*. Plus there is another Commit after the client renegotiation and Running after the server renegotiation to make sure there is no attack in the renegotiation phase too.

B The attack detected by Scyther

Figure 14 shows the attack as detected by Scyther.

C Appendix: Lookup Table

MAC Message Authentication Code.

TLS The Transport Layer Security protocol. See Subsection 2.1.

HTTPS Hyper Text Transfer Protocol Secure, a protocol using TLS.

SMTPS Simple Mail Transfer Protocol Secure, a protocol using TLS.

FTPS File Transfer Protocol Secure, a protocol using TLS.

RSA, *RSA_PSK* Key Exchange Algorithms in TLS using the Server's RSA encryption public key. See Subsubsection 2.5.2.

PKE_S *RSA* public key of the server that can be used for encryption.

DH_ANON A Key Exchange Algorithm in TLS using Diffie-Helman without Server Authentication. See Subsubsection 2.5.5.

DHE_DSS, *DHE_RSA* Key Exchange Algorithms in TLS using the Server's signing public key. See Subsubsection 2.5.4.

DH_DSS, *DH_RSA* Key Exchange Algorithms in TLS using the Server's Diffie-Hellman public key. See Subsubsection 2.5.3.

PMS Pre-master-secret value in the TLS. See Subsubsection 2.4.2.

MS Master secret value in the TLS. See Subsubsection 2.4.2.

EXT_C^{FIX} TLS renegotiation indication extension.

PRF_n(**secret, label, seed**) n bytes produced by the pseudo-random function with 3 arguments.

$Vers_C$, $Vers_S$ Protocol version for the Client and the Server.

$R_C, R_S, x, y, Cookie_C, Cookie_S$ Random values generated by the Client or the Server.

SID Session Identifier.

IKE The Internet Key Exchange protocol. See Subsection 5.2.

IPsec Internet Protocol Security Suite.

SA Security Association. A shared security state between the parties in IKE.

SSH2 The Secure Shell protocol. See Subsection 6.1.

spdl Security Protocol Description Language, the language used to models security protocols in Scyther.

RNR An advanced capability of an adversary in Scyther to learn random values used during the protocol execution.

LKRothers A standard capability of an adversary in Scyther to compromise long-term keys of a third party.

Figure 14: Attack on TLS as detected by Scyther