
Analyzing DTLS security in Tamarin

- A master's thesis -

Project Report
cs-25-ds-10-10

Aalborg University
Computer Science



Computer Science
Aalborg University
<http://www.aau.dk>

AALBORG UNIVERSITY

STUDENT REPORT

Title:

Analyzing DTLS security in Tamarin

Theme:

Modeling security protocols

Project Period:

Spring Semester 2025

Project Group:

cs-25-ds-10-10

Participant(s):

Signe Kirstine Rusbjerg
Tobias Møller

Supervisor(s):

Tobias Worm Bøgedal
René Rydhof Hansen
Danny Bøgsted Poulsen

Page Numbers: 63

Date of Completion:

June 12, 2025

Abstract:

This report focuses on the 1.3 version of the DTLS protocol. Within the report, we model three parts of the DTLS protocol, to test the security of the protocol. These models are analyzed using the Tamarin Prover tool, where several security aspects are validated and verified. The report also proposes an idea to translate models created in Tamarin to the UPPAAL tool, in order to prove parts that might be hard in Tamarin.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Summary

In this Master's thesis, we verify the security of the DTLS 1.3 security protocol. The case was interesting, as DTLS 1.3 has been out for three years, yet almost no-one has publicly upgraded from the previous 1.2 version, even though DTLS 1.2 has proven security flaws. To prove the security of the protocol, we analyzed three models using the Tamarin Prover tool. The first model contains a full handshake, that establishes a connection between two first time peers, with the use of an additional cookie exchange. The second model is a handshake done by peers that have communicated earlier, and thus have a "Pre-Shared Key" that significantly shortens the handshake. The final model is of the record layer, which provides the encoding of a given message. All three models are based upon RFC 9147, which is the specification of DTLS 1.3, and RFC 8446, which is the specification of the TLS 1.3 protocol. The TLS specification is used, as the DTLS specification refers to it when there is no changes. The specification also contains some security requirements that must be upheld, and as such, these are the basis of the security analysis. However, we also evaluate DTLS against the CIAA tetrad. This tetrad focuses on Confidentiality, Integrity, Authenticity, and Availability. While all four are desired for a protocol like this, one must strike a balance between the four. In the report, the first three are analyzed using the Tamarin Prover tool, but the final property of availability, was not viable within the tool. This then led to a new idea to provide full insight into DTLS 1.3. In earlier work done by the group, the tool UPPAAL was used to model DTLS and analyze availability and power consumption. Thus, if we could develop a method to translate the Tamarin model into a UPPAAL model, we could check availability in a tool more suited for it. While a tool to do this automatically was not developed for this report, we propose a general method that should be able to work as a base for a translation between Tamarin and UPPAAL.

Contents

1	Introduction	1
1.1	Related Work	2
1.2	Paper structure	3
2	Secure communication	4
2.1	Security promises of Datagram Transport Layer Security	6
2.1.1	Authentication: Handshake security properties	6
2.1.2	Confidentiality and integrity: Record layer security properties	8
2.1.3	Availability: Denial-of-Service security properties	9
3	Datagram Transport Layer Security	11
3.1	Handshakes and key exchanges	12
3.1.1	Full handshake	12
3.1.2	Pre-shared keys	14
3.1.3	Key update	15
3.2	General message structure	15

3.2.1	DTLS handshake messages	17
3.2.2	DTLS handshake header	18
3.2.3	DTLS record headers	20
4	Modeling the protocol	22
4.1	Tamarin Prover	22
4.1.1	Threat Model	25
4.2	Datagram Transport Layer Security Model	25
4.2.1	Modeling: Handshake message	26
4.2.2	Modeling: Handshake header	28
4.2.3	Modeling: Record layer	29
4.2.4	Modeling: Additional behavior	31
4.2.5	Modeling: UDP	32
4.3	Model validation	32
4.4	Security properties	34
4.4.1	Handshake properties	34
4.4.2	Record properties	37
5	Analysis and results	40
5.1	Debug lemmas	41
5.2	Combating memory issues	41
5.3	Record layer model restrictions	43

5.4 Results	44
6 UPPAAL translation	46
6.1 Tamarin syntax and semantics	47
6.1.1 Tamarin variables translated to UPPAAL	47
6.1.2 Tamarin facts translated to UPPAAL	48
6.1.3 Pattern matching	49
6.2 Translation for a restricted Tamarin model	50
6.2.1 Tamarin symbolic variables into UPPAAL variables	50
6.2.2 Tamarin rules into UPPAAL processes	51
6.2.3 Tamarin facts in UPPAAL	52
6.2.4 Pattern matching for equality checks	53
6.2.5 Implementing cost	54
6.2.6 Adversary behavior in UPPAAL	54
6.3 Further development	54
7 Discussion	56
8 Conclusion	60
8.1 Future work	61
Bibliography	62
A Structs of DTLS	a

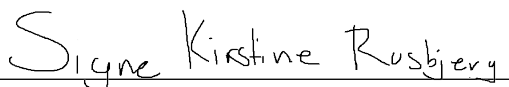
A.1 ClientHello	a
A.2 HelloRetryRequest	b
A.3 Second ClientHello	c
A.4 ServerHello	c
A.5 EncryptedExtensions	d
A.6 CertificateRequest	d
A.7 Certificate	e
A.8 CertificateVerify	f
A.9 Finished	g
A.10 Ack	g
B Validation Lemmas	i
B.1 Authentication	i
B.2 Cookie exchange	k
B.3 Finished Handshake	l
B.4 Order	m
C UPPAAL Translation ideas	o

Preface

Aalborg University June 12, 2025

This report was written by the computer science group "cs-25-ds-10-10" in the period from the 1st of February to the 13th of June 2025

The group would like to thank the supervisors René Rydhof Hansen, Tobias Worm Bøgedal and Danny Bøgsted Poulsen for their help and guidance through the project.



Signe Kirstine Rusbjerg
<srusb20@student.aau.dk>



Tobias Møller
<tmalle20@student.aau.dk>

Chapter 1

Introduction

The Datagram Transport Layer Security (DTLS) protocol has seen a lot of use in the world of Internet of Things (IoT). IoT devices have become commonplace in the modern world, spanning across many aspects of technology. Everything from private housing to large factories and power plants, utilize these small devices connected to the internet. Because of this abundance, it serves as a fine target for malicious actors. This is where many choose to utilize the DTLS protocol, as it claims to essentially provide the same amount of security as the widely used Transport Layer Security (TLS) protocol while being optimized to use considerably less power than TLS.

As of this paper, the newest version of DTLS is DTLS 1.3. Within this report, DTLS refers to DTLS 1.3 unless directly specified otherwise. The RFC [1] was published in April of 2022, approximately 4 years after its TLS counterpart TLS 1.3. In the three years since DTLS 1.3, almost no users have upgraded to this version [2]. This could be due to several things, like DTLS largely being used in hardware, making it more expensive to update. DTLS does however also see usage in VPN software, but even in this software based solution, only ExpressVPN has publicly adopted DTLS 1.3. In comparison, 47.3% of TLS users had updated to TLS 1.3 within a similar three year timespan according to Qualys SSL Labs [2]. One could wonder why TLS has significantly more users updating to the newest version.

This difference could stem from the earlier work done for each of the protocols. TLS 1.3 is supported by many libraries, and has had modeling work done on many aspects of the protocol, even ahead of the official release. With DTLS 1.3, the earlier work is limited to the library WolfSSL, and a paper focusing on availability and power consumption by modeling DTLS 1.3 in UPPAAL [3].

Within this report, we will model a segment of the DTLS protocol; a full handshake, a handshake using Pre-Shared Keys (PSK) and the record layer. We believe that this segment, presented in chapter 3, could produce an reasonable implementation suitable for an IoT device. We aim to verify all security claims of DTLS as per the specifications, but also discuss how these claims relate to the four CIAA properties;

- Confidentiality
- Integrity
- Authentication
- Availability

To create this model, the group has decided to use the Tamarin Prover security protocol verification tool, as it contains features that other competitors do not. Tamarin was chosen, as it allows for infinite instances of servers and clients. Similar work for TLS 1.3 also used Tamarin, and it has thus already proved it's potential as a tool for this task [4] [5].

The model is created using the specifications, similarly to the work done on TLS [4]. Modeling the specifications give us insight into the correctness of the template provided to protocol developers. This work will ensure that the documentation available is correct, which should positively effect future implementations.

1.1 Related Work

To the best of our knowledge, there is no existing work done for DTLS 1.3. However, both TLS 1.3 and DTLS 1.2 have papers that introduce models in Tamarin [4, 6].

As DTLS is built to achieve TLS security guarantees but over UDP, the security properties of DTLS 1.3 will mirror the security properties of TLS 1.3, although DTLS 1.3 has the additional property of protecting against further attacks such as DoS. There are however some major changes when it comes to modeling DTLS. The most obvious ones are the needs to handle unreliability and retransmission. These requirements result in some changes to every part of the communication, which means that an exploration of DTLS 1.3 is still prudent.

The paper on DTLS 1.2 ([6]) is now outdated, and there has been some major changes in the upgrade to DTLS 1.3. The biggest change between the versions, lies in the changes

within the handshake, which has been overhauled to shorten the exchange. The DTLS 1.2 paper does not exclusively focus on DTLS, and thus it has chosen to abstract away from some of the specifics. Within the paper, DTLS is only mentioned in the context of CoAP [6].

While both of these papers exist, we do not believe that it takes away from the importance and relevance of this paper.

1.2 Paper structure

DTLS promises a selection of security properties, which is presented and explained in chapter 2, along with the general security concerns of IoT devices. Chapter 3 then presents the selected segment of DTLS, with handshake examples and message structures.

Chapter 4 presents a Tamarin model featuring a segment of the DTLS 1.3 handshake, alongside session resumption using Previously Shared Keys(PSK) and the Record Layer. The model's security is then analyzed and verified using the Tamarin Prover's security protocol verification tool in chapter 5. Chapter 6 will provide an idea of how to transform the tamarin models to UPPAAL to prove the availability property. Finally, chapter 7 and 8 will reflect and conclude on the work done in the report.

Chapter 2

Secure communication

The need for strict security in IoT may in some cases be overlooked, since its importance is not immediately apparent [7]. Say a consumer owns some smart lights, which they can control from their phones. At first glance, whether or not a stranger knows that the light is on, seems insignificant. However, this information could be used by a thief to gauge whether or not the consumer is home, and if a break-in is possible. The question ensues; what security do we want from IoT devices?

For a running example, take the scenario in figure 2.1, with a consumer, Alice, communicating with an IoT device. To increase the severity of the situation, imagine that the IoT device is a home security camera, sending recordings from inside and outside the home. Alice can request the recordings at any point via the internet, which is controlled by a malicious user, Mallory.

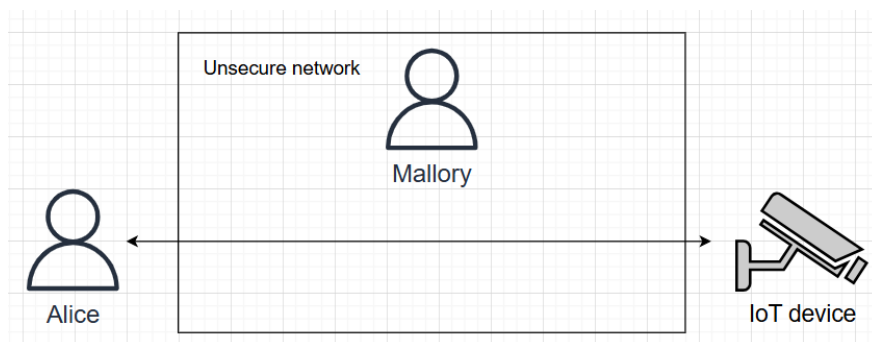


Figure 2.1: Communication with IoT on a unsafe network. Running example.

Confidentiality is desired since the recordings may contain sensitive information, that Alice wishes to keep a secret. Confidentiality can be achieved by correctly using cryptography to encrypt the data before it is sent, but confidentiality may also be broken with the use of side-channels. In the smart light example, confidentiality can be broken by analyzing the network traffic. In order to protect against this, the use of instant-message denial has been explored in other materials.

Authentication would also be valuable, since we only want the IoT device to send data to authenticated users. In the example, the IoT device should only send data to Alice, however if Mallory successfully impersonates Alice, the IoT device would send the potentially sensitive data to Mallory. This is not desirable, and such authentication is also needed.

Within authentication, each party has their own unique identity. This identity can be compared to real world examples like drivers license or passports, and is how a party can prove to be who they claim to be. A commonly used identity for secure communications is verified certificates issued by a trusted entity.

Integrity is especially desired in the case of security cameras, since we want to ensure that the recording has not been modified. If Mallory was able to change the content of the recording while it was being transferred from the IoT device to Alice, she would be able to delete evidence of a potential break in. Integrity gives Alice the trust that the recordings she sees are the actual recordings made by the security camera.

Availability is also a desired trait, since an unavailable device can be considered useless. A consumer should have access to the IoT device without unnecessary delay, and Mallory should not be able to excessively halt the communication. With the case of low-power devices, Mallory should also not be able to unnecessarily drain the battery of the device [3].

These four properties make the CIAA tetrad, which is commonly used to define desired security properties in communication over the internet. These properties however are not trivially upheld, and some mechanisms may uphold one property while negatively affecting another. Examples of conflicting properties is availability versus the rest. If the IoT device was not available, all other properties would be trivially upheld. More complex cases could be integrity versus confidentiality. Confidentiality negatively affects the integrity of the data, since it is only available to a select group of people, thereby making its integrity harder to verify.

DTLS is an example of a protocol, attempting to uphold all four properties to varying degrees. The following section 2.1 will examine all security promises made by DTLS and to what degree they uphold the CIAA tetrad.

2.1 Security promises of Datagram Transport Layer Security

DTLS divide its security promises into three aspects. The three aspects cannot strictly be divided into either of the four properties of CIAA, however they may focus on one aspect more than others.

- *Authentication* is provided from the DTLS handshake, which is an Authenticated Key Exchange (AKE) protocol. The AKE aims to exchange authenticated keys in a secure manner, thus providing authentication to the connection.
- *Confidentiality and integrity* is promised with the DTLS record layer protocol, which is defined as all communication after a successful handshake. Confidentiality is however only partly upheld, which is expanded upon in section 2.1.2. The record layer also enjoys authentication as a byproduct of the AKE.
- The entire DTLS protocol struggles with *availability*, suffering from the unreliability of UDP. There exists quite a few attacks against the DTLS protocol, which utilize the weaknesses of UDP. This is explained in further detail in section 2.1.3.

Passive and active attacks

As DTLS is used on the open internet, it is subjected to potential attacks, aiming to break some of the CIAA properties. DTLS generally aims to protect against an active attacker, with some exceptions, where protection is only guaranteed against passive attackers. An active attacker has complete control over the network, and may interfere with communication as they seem fit. Examples of interference could be something as simple as reading and/or modifying the packets, but they can also go a step further and completely block access for legitimate connections, using a Denial-of-Service attack. This attacker is quite strong in comparison to a passive attacker like an "eavesdropper". An eavesdropper attempts to steal the data as it is transmitted by "listening along" to a connection.

2.1.1 Authentication: Handshake security properties

The DTLS handshake provides one-party and mutual-party authentication. The specifications produce a list of eight security properties, that a successful handshake should provide the session:

- H1: Establishing the same session keys
- H2: Secrecy of the session keys
- H3: Peer authentication
- H4: Uniqueness of the session keys
- H5: Downgrade protection*
- H6: Forward secret with respect to long-term keys
- H7: Key Compromise Impersonation (KCI) resistance*
- H8: Protection of endpoint identities*

*Properties marked with a * are not directly modeled. Section 4.4 about the modeling of security properties, will argue why some properties have been omitted.*

These properties collectively define what successful handshake should provide, however a baseline for what a successful handshake entails is not specifically given. This paper regards the baseline for a successfully completed handshake as; a communication accepted by two uncorrupted parties, with matching communication history and session keys[8] [9].

Properties (H1) and (H2) follows from definition 1.1 and 1.2 in [8] respectively, and expand upon the definition of a successful handshake. The first property (H1) states that for a handshake to be successful, it must output the same session key. The second property (H2) states that the keys must be secret to any outsider, and there must be a negligible chance chance of guessing the key.

Property (H3) claims that when a client connects to a chosen server, then the identity of the peer should match the identity of the intended server. Additionally in the cases where the client is authenticated, the server should similarly be able to match the peers identity to the client.

The protocol is not restricted to just the handshake of a single run, and as such other properties are desired when expanding the view to multiple runs.

The fourth property (H4) develops on the relation between two distinct handshakes. It states that two distinct handshakes should produce distinct and unrelated session keys, and those individual session keys should also be distinct and independent.

The fifth property (H5) takes the versatility of servers providing multiple cryptographic functions, as well as earlier versions of DTLS into account. The property (H5) ensures the

quality of a connection by providing protection against downgrading. When a handshake is complete, both parties should have the same cryptographic parameters, and these parameters should be the same as if no attacker was present. This prevents an attacker from deliberately downgrading the connection to a less secure version.

The next three properties (H6-H8) all regard security properties after a breach. The sixth property (H6) considers Forward Secrecy (FS) with respect to long-term keys. FS is a common and desired property, however one should take care not to confuse it with its similarly named counterpart; Perfect Forward Secrecy (PFS). FS refers to the ability to keep a prior sessions confidential in the case of a breach of long-term identity keys [9]. PFS has the additional property that even if a session key is compromised, prior sessions will not be compromised.

The property (H6) is specifically with respect to the long-term signature keys and the long-term external/resumption pre-shared keys, where a new key was generated during the handshake (PSK with (EC)DHE). It does not cover the loss of pre-shared keys used without the DHE key-exchange.

The seventh property (H7) regards key compromise impersonation, and ensures that the compromise of long-term secrets should not allow an adversary to impersonate other actors that are communicating with the compromised actor.

The eighth and final property (H8) provides protection to the endpoints identities. It states that a server's identity should be protected against a passive attacker, where the client's identity should be protected against both a passive and an active attacker. The server will not be able to enjoy protection against an active attacker, since one side of a mutual authentication always has to go first, which for the case of DTLS is the server.

2.1.2 Confidentiality and integrity: Record layer security properties

DTLS's security properties differentiate from TLS when it comes to their record layers, since DTLS does not provide order protection/non-replayability.

The record layer is reliant on the handshake protocol producing strong traffic secrets. If this is the case and the keys are used no more than advised, the DTLS record layer will provide the following properties:

- R1: Confidentiality
- R2: Integrity

- R3: Length concealment
- R4: Forward secrecy after key change

The first property (R1) is limited to the concealment of plaintext. This showcases DTLS's limited sense of confidentiality, since as explained in the beginning of chapter 2, confidentiality entails much more.

The second property (R2) ensures that an adversary has not modified the contents of a message while it is in transit between the sender and receiver. This is generally enforced with the usage of a Message Authentication Code (MAC) or the expanded Hash-based Message Authentication Code (HMAC), as both of these provide integrity authentication.

The third property (R3) mitigates one potential side-channel by providing length concealment. Returning to the light switch example, the length of the package is quite prudent, as one might try to conceal the lack of traffic by sending smaller dummy messages. This could however lead to the adversary noticing the change in package size, and they could therefore still notice the anomaly.

The fourth property (R4) provides forward secrecy to the likes of what is explained in section 2.1.1. Using the earlier example of a security camera, a potential breach of a future connection does not allow the adversary to gain any meaningful information about an earlier session they might have eavesdropped upon. Forward secrecy will allow for these stipulations, given that the connection was secure when the old recordings were exchanged.

Forward secrecy only holds when old session keys are deleted, but when it comes to DTLS, there is an extra piece that needs to be accounted for. DTLS keeps a grace period where old session keys are still accepted even though they are marked to be deleted. This grace period is in place as DTLS does not promise in-order transmission, and some messages could still be in transit. The challenges of defining such a grace period can then be seen, since the longer the grace period, the less chance of delayed messages not being decryptable, and the higher the chance of the session being compromised.

2.1.3 Availability: Denial-of-Service security properties

When it concerns Denial-of-service (DoS) attacks on DTLS, there are two main concerns to be aware of:

- D1: Attackers can initiate a large amount of handshakes in order to drain the server's

resources. In the worst scenario, this attack will drain the power of a battery-driven server, or outright crash servers with a more consistent power supply.

- D2: Attackers can use the server as an amplifier, by sending a spoofed request from a compromised victim, which could lead the server to flood the victim with responses to the handshake. The worst case here is similar to the case in D1, but here the victim machine gets hit instead of the server.

Availability is not modeled in the Tamarin model, however the report will end with a proof of concept (chapter 6), of a method to translate Tamarin models into UPPAAL models. As seen in previous work [10], an UPPAAL model can be used to verify availability.

Both of these concerns are addressed by the stateless cookie exchange. This cookie exchange functions as a prelude to the proper handshake, by forcing both the client and the server to communicate using small packages (see section 3.1 for an example). Only after this exchange will the server see the client as a valid communication partner, and will thus begin using more resources to start the communication. This cookie exchange forces the attacker to be able to receive the cookie, and therefore, the attacker cannot use the spoofed method of requesting.

While the cookie exchange helps, it does not stop an attacker from gathering several cookies from different endpoints, which they can then use to flood the server. To counteract this, the secret value should be changed often, but a grace period of the older secret value should be considered due to the unreliability of UDP.

To combat D1, the server should as a general rule, limit their data sent to three times the amount of data received until the client is verified as valid. By enacting this rule, the server will never allocate an extreme amount of data, as the ClientHello message is quite small already.

The effectiveness of these DoS mitigation techniques were tested in an earlier paper using power consumption as a metric [3].

Chapter 3

Datagram Transport Layer Security

After looking at the security properties of DTLS, it is necessary to look at the structure of DTLS and how it works. DTLS is a security protocol that utilizes an Authenticated Key Exchange (AKE) protocol called the DTLS handshake, to exchange secure and authenticated keys between two endpoints. These keys are then used to perform safe and secure communication between the two endpoints for a period of time.

As mentioned in chapter 1, this report focuses on a segment of the DTLS protocol. This includes two different handshake options, a full DTLS handshake, as well as a handshake using Previously Shared Keys (PSK). Additionally, it includes the record layer, which consists of application data exchanges, as well as a key update that can happen after a valid handshake. Keys become less secure the more times they are used, and if the same key is used over a long period of time, then the odds of it being leaked increase. Modeling a key update and its effect on the security would be necessary in a realistic implementation.

This chapter will first explain the two handshake options and the key update, before looking at the structure of the handshake messages, and their headers. The handshakes and key update are explained using flight diagrams, and the message and header structure are presented as pseudo-code similar to c-structs. The structs will later be used in section 4.2, to explain how the model was created to mimic the structs from the specifications closely.

3.1 Handshakes and key exchanges

Before going in depth with the proper handshake, we would like to present a major issue when using Datagram transport protocols, and how DTLS solves said issue:

Issue 1; Datagram transport protocols are susceptible to abusive behavior effecting DoS attacks against nonparticipants [1].

Solution 1; DTLS adds a return-routability check, with the strongly advised inclusion of a cookie exchange. The cookie exchange forces a client to show commitment to the connection before the server invests a large amount of space and computation power into said connection. The cookie exchange also prevents the server from becoming an amplifier in itself, since UDP forgery is very easy, and a server may unknowingly flood a nonparticipant with large amount of data [1].

3.1.1 Full handshake

Figure 3.1 shows how a DTLS handshake can look, between two first time peers. The handshake consists of three round trips, with a round trip being one sent message and one response.

[1] The client initiates the handshake with a ClientHello message. The message contains some metadata and a Diffie Hellman keyshare.

[2] The server will respond with a HelloRetryRequest. This is the start of the previously mentioned cookie exchange, as the message contains a cookie.

[3] The client will resend the ClientHello, now with the received cookie attached, finishing the cookie exchange.

[4] Since the client now has shown a commitment to the handshake, the server will continue the handshake. The server sends a server hello, which will also contain a Diffie Hellman keyshare.

With its own keyshare and the one received from the client, the server can generate the traffic key. This is used to encrypt the next few messages.

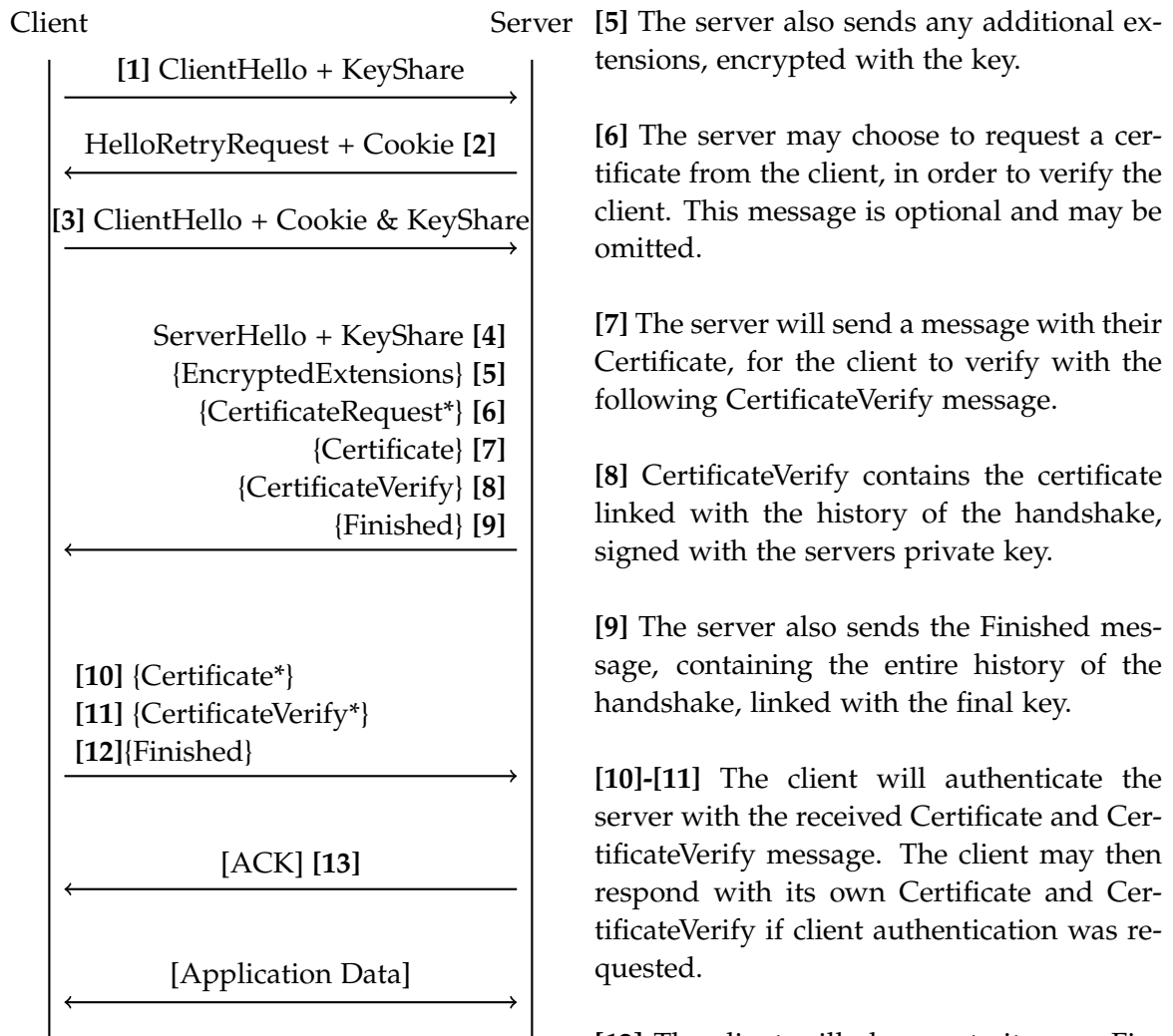


Figure 3.1: Full DTLS 1.3 handshake [1].

{ } denotes encryption made with the traffic key and [] denotes encryption made with session key.

* denotes optional or situational messages, and + denotes extensions.

The server may receive the Certificate and CertificateVerify messages, and would use them to authenticate the client.

[13] The server also receives the Finished message from the client. The server will ensure that the received Finished message is identical to its own Finished message, and will conclude the handshake with an ACK message.

The ACK message contains the record number of all records sent in the handshake. This is the final message that concludes the handshake.

3.1.2 Pre-shared keys

DTLS may also utilize a PSK, to simplify consecutive handshakes. Figure 3.2 shows how a handshake could look, when the two endpoints have previously completed a handshake and shared keys. Both the cookie exchange and the authentication step is skipped. Since the client has previously shown commitment, the cookie exchange is not necessary. The server is also indirectly authenticated by the PSK, so there is not a need to exchange certificates.

By utilizing PSK a lot of power and traffic is saved. This is the case since an entire round trip is saved, and the need for authentication is indirectly handled by the PSK.

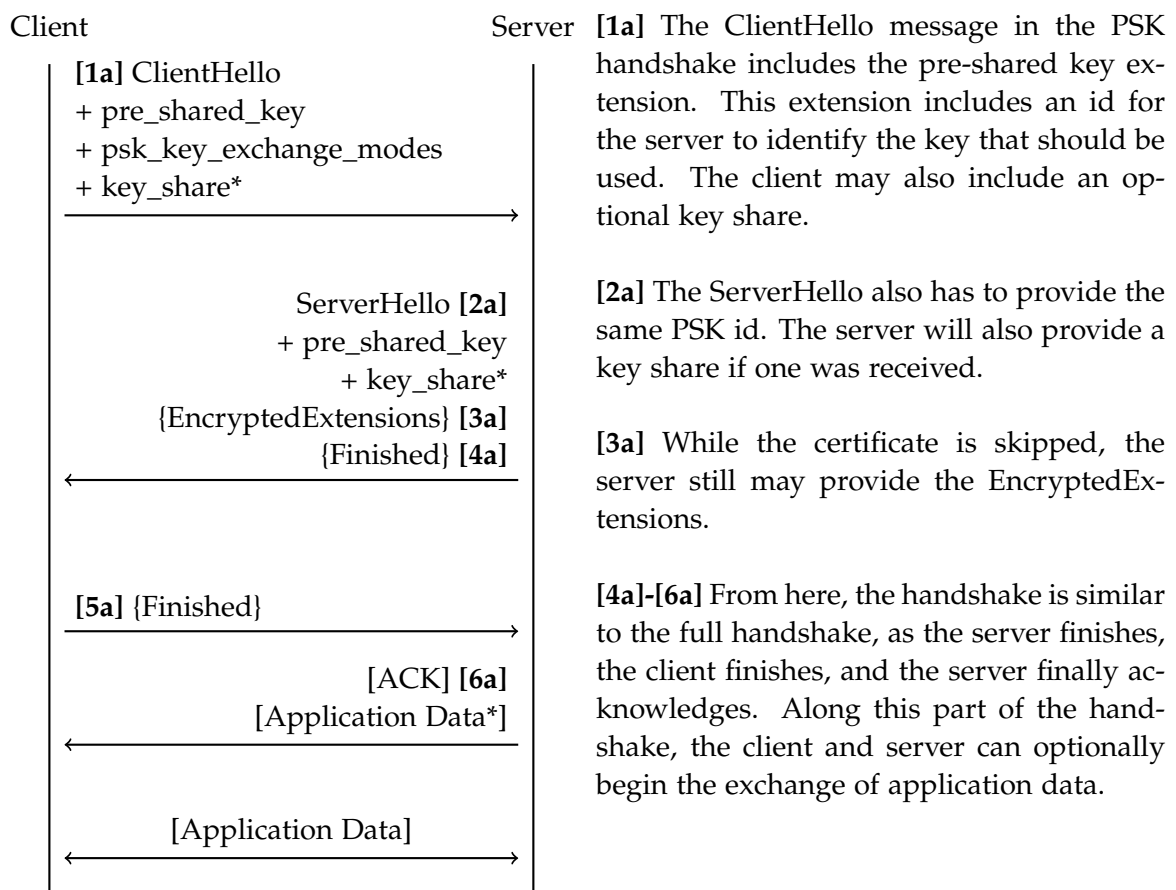


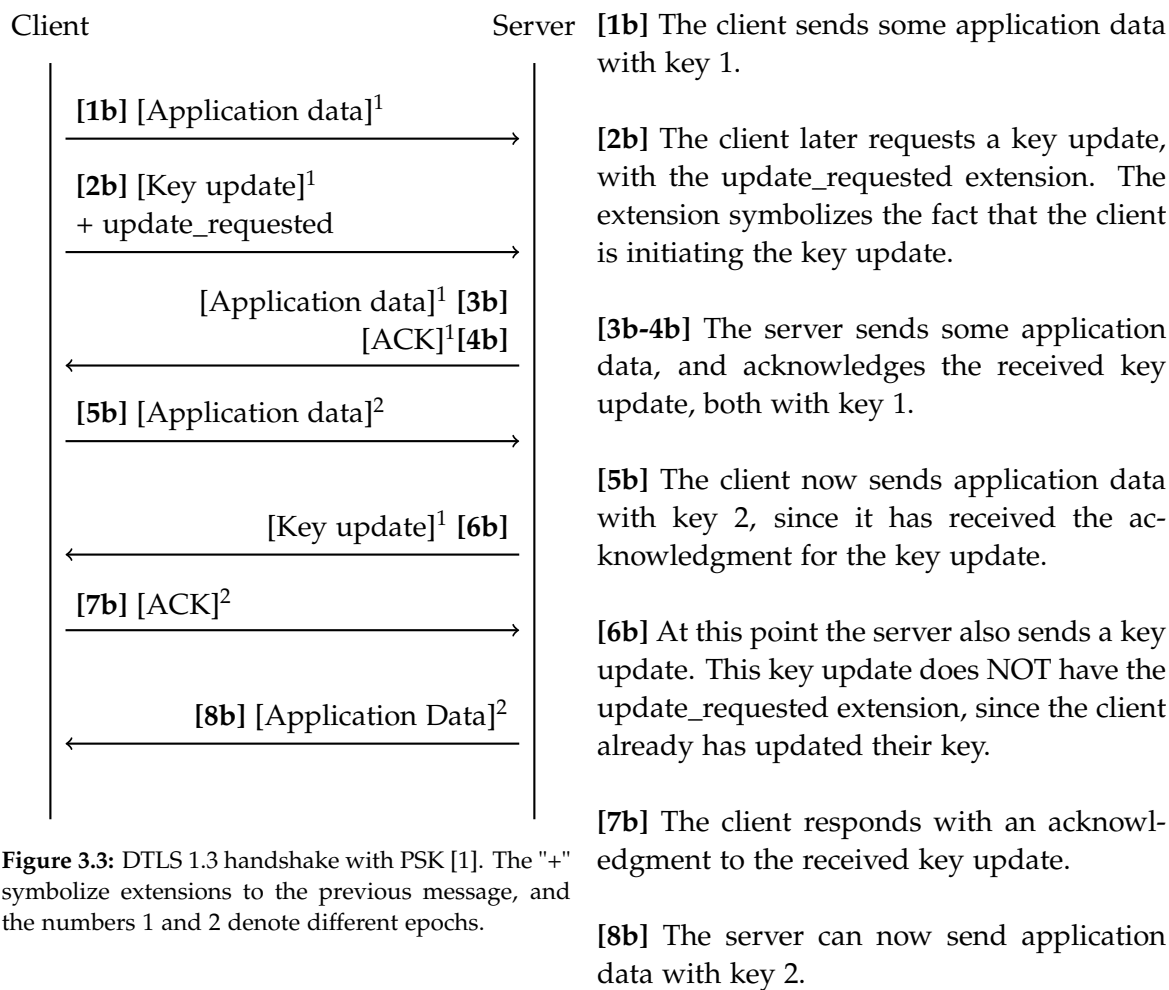
Figure 3.2: DTLS 1.3 handshake with PSK [1].

{ } denotes encryption made with the traffic key and
[] denotes encryption made with session key.

* denotes optional or situational messages, and + denotes extensions.

3.1.3 Key update

A session key will be less and less secure the longer and more it is used. The key must be updated during the transmission of data, and DTLS provides a method to do exactly this. Figure 3.3 shows how such a series of flights may look. The square brackets are marked with a number 1 or 2. This symbolizes different epochs, so when the number changes from 1 to 2, it means the key has been updated.



EncryptedExtensions message would look with their headers. The dashed lines symbolize un-encrypted information and the solid lines symbolize encrypted information.

ClientHello is sent in plaintext and is therefore un-encrypted, however the EncryptedExtensions message must be encrypted, which means that only the record header is plaintext with the rest being encrypted. This is also reflected in the information in each header.

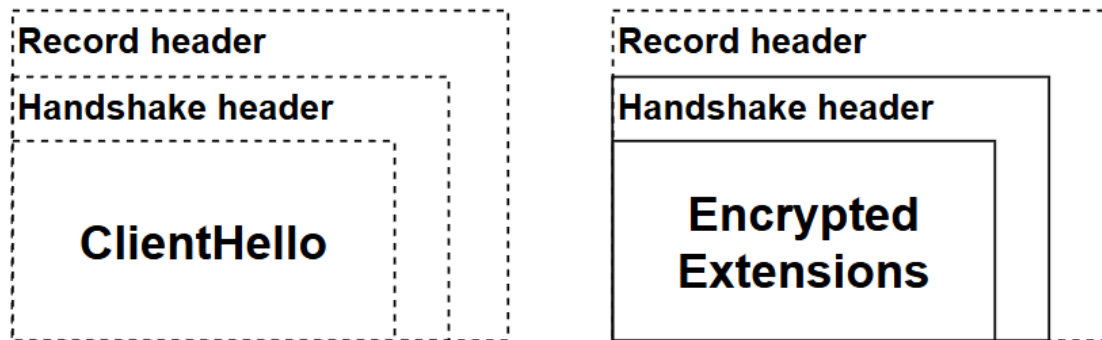


Figure 3.4: ClientHello and EncryptedExtensions messages with headers. Solid lines symbolize encrypted content, dotted lines symbolize un-encrypted content.

The record header contains the information needed for de-protection. The need for this information to be in plain text is highlighted by the two issues with attached solutions below. Since UDP is unreliable and unordered, DTLS cannot utilize the same mechanisms that TLS does.

Issue 2; TLS uses an implicit sequence number which is kept and maintained locally and independently on each side of the connection. The sequence number is used together with the keys to remove the record protection and such a wrong sequence number will result in a "bad_record_mac". Unlike TCP, UDP does not promise in-order or reliable transmission, and each endpoint needs to be able to deduce the sequence number of each individual received record [1].

Solution 2; As the solution to this issue, DTLS adds an explicit sequence number in the header of the record, which can then be used to de-protect the record. This however can lead to some initial concerns, since sequence numbers in TLS enjoy a greater protection, by never being sent over the exposed internet. For DTLS however, sequence numbers cannot enjoy this same protection, and has to be sent in plaintext in some capacity. It is interesting to see how this change effect the security of the overall protocol.

Issue 3; TLS also uses a lock-step cryptographic protocol, which requires packages to arrive

in-order. UDP does not promise in-order transmission and such the endpoint needs a way to figure out what parameters to use of decryption [1].

Solution 3; Again the sequence number along with an epoch is used to order the out of order messages. However since these values are needed for decryption, both the sequence number and epoch must be sent in plaintext for the endpoint to analyze.

The handshake header on the other hand, can enjoy protection and contain information used for reassembly after fragmentation. Fragmentation is how DTLS solves the final major issue.

Issue 4; Handshake messages are potentially too large to fit within a single datagram, which is an issue, since the handshake message must be correctly reconstructed before it can be processed [1].

Solution 4; DTLS solves this by adding a field to the handshake header in order to handle fragmentation and reassembling. These extra fields can be used to reassemble all parts of a record, and does not have to be sent in plaintext, and can such enjoy encryption.

The rest of the section will go into detail with the structure of the three message parts.

3.2.1 DTLS handshake messages

The structure of all DTLS handshake messages, with a more detailed explanation can be found in Appendix A, but for this section we will take a look at ClientHello.

All handshake messages follow the same structure and contain a number of fields. The ClientHello message contains 7 fields with varying complexity. The extension field will generally be one of the most complex, as it can contain any number of extensions within.

The client may append a number of extensions for the server to acknowledge. One extension usually sent here is the *Keyshare* extension, which contains a list of Diffie-Hellman groups and keys. This field however, is used for many different extensions, including the cookie extensions. The report will highlight important extensions when they become relevant and we refer to the RFC [1] for a full list.

```
1 struct {
2     ProtocolVersion legacy_version = { 254,253 };
3     Random random;
4     opaque legacy_session_id<0..32> = [];
```

```
5     opaque legacy_cookie<0..28-1> = [];  
6     CipherSuite cipher_suites<2..216-2>;  
7     opaque legacy_compression_methods<1..28-1> = 0;  
8     Extension extensions<8..216-1>;  
9 } ClientHello;
```

3.2.2 DTLS handshake header

The listing below shows the DTLS handshake header, with three new fields added in order to accommodate for solution 4 (see section 3.2).

```
1 struct {  
2     HandshakeType msg_type;      /* handshake type */  
3     uint24 length;              /* bytes in message */  
4     uint16 message_seq;         /* DTLS-required field */  
5     uint24 fragment_offset;     /* DTLS-required field */  
6     uint24 fragment_length;     /* DTLS-required field */  
7     opaque body;  
8 } DTLSHandshake;
```

Fragmentation

Fragmentation of handshake messages is handled by DTLS in order to avoid fragmentation on the underlying layer. See figure 3.5, for an example of fragmented packages.

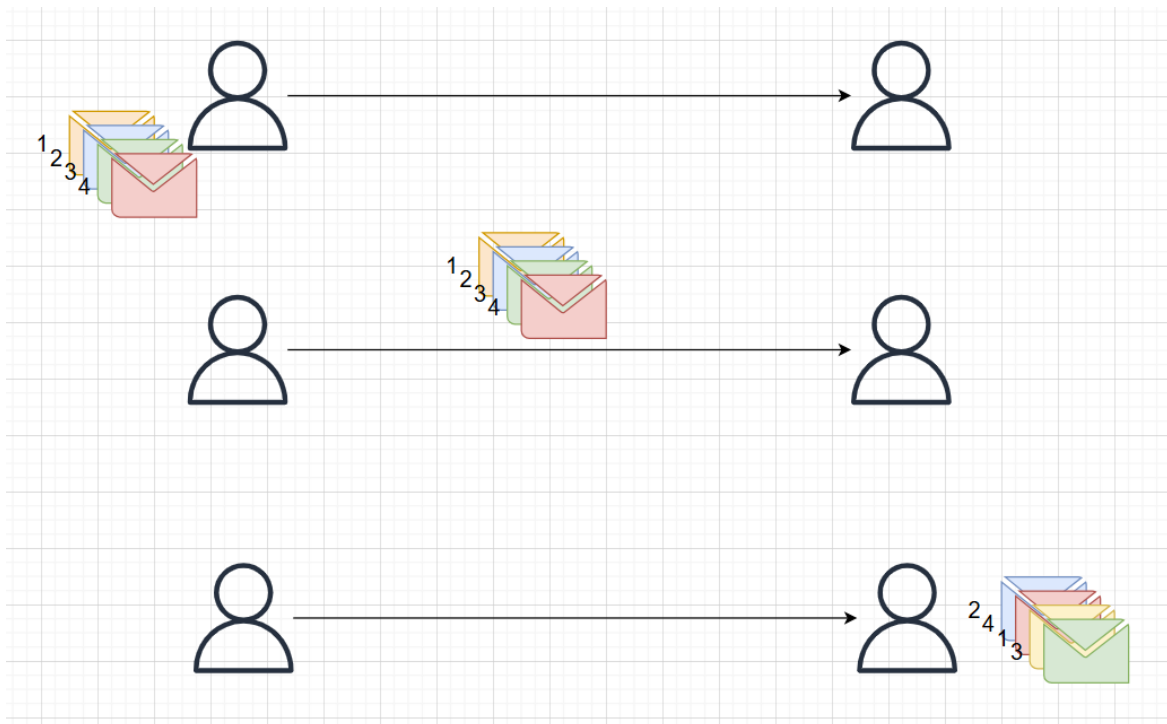


Figure 3.5: Un-ordered fragments example

Alice initially sends four fragments in the correct order, however under transmission the packages get reordered, and Bob receives the fragments out of order. To solve this issue, each package has a sequence number, a fragment offset, a fragment length and a message length in their header. For the example, take the values below:

- Fragment: {sequence number: 1, fragment offset: 200, fragment length 300, length: 700}
- Fragment: {sequence number: 2, fragment offset: 0, fragment length 300, length: 300}
- Fragment: {sequence number: 1, fragment offset: 0, fragment length 200, length: 700}
- Fragment: {sequence number: 1, fragment offset: 500, fragment length 200, length: 700}

The fragments above are ordered as Bob receives them, and the reassembly can be seen in figure 3.6. Bob sees that the first fragment is part of a message with sequence 1, however since the fragment offset is 200, it is not the first fragment of said message. Bob stores the fragment for now and reads the next one.

The next fragment is of sequence 2. The fragment offset is 0, so it must be the first fragment of the message, however since the fragment's length and the length of the message are both 300, this message has not been fragmented, and thus the full message is already constructed. The message however is a sequence number ahead, and must be stored for now.

The third fragment is of sequence number 1, and with a fragment offset of 0. This symbolizes that it is the first fragment in the message. Bob already has a fragment with a sequence number of 1, and compares the two fragments. The fragment length of 200 in the third fragment matches the fragment offset of the stored fragment, and the two fragments should appear one after the other. However, since the combined length of the fragments does not equal the final length of the message, Bob knows that more fragments are still in flight.

The fourth fragment is again from sequence number 1, and belongs with the other two fragments. The fragment offset is equal to the total length of the two stored fragments, and thus this fragment should come after them. The three fragments' combined length equals the length of the entire message, and thus the message has been reconstructed, and can be processed.

Finally the stored fragment with sequence number 2 can also be processed.

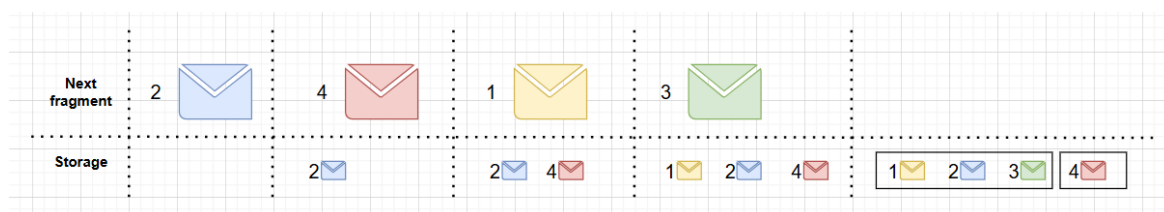


Figure 3.6: Fragment reassemble example

3.2.3 DTLS record headers

There exists two types of DTLS record headers, one for encrypted records and one for unencrypted records. Within the specification, these are known as DTLSPlaintext and DTLSCiphertext. We see both sequence number and epoch appearing in clear text in both of these headers in order to solve issue 2 and 3 (see section 3.2).

Plaintext

The structure of the plaintext record header can be seen in the listing below. The header contains six fields, with some being there for legacy reasons. Plaintext records have a fixed epoch of zero, since this symbolizes an un-encrypted record.

```
1 struct {
2     ContentType type;
3     ProtocolVersion legacy_record_version;
4     uint16 epoch = 0
5     uint48 sequence_number;
6     uint16 length;
7     opaque fragment[DTLSPlaintext.length];
8 } DTLSPlaintext;
```

Ciphertext

The header for ciphertext is divided into two sub-headers; the ciphertext header and the DTLS inner plaintext header. Both the ciphertext header and the inner plaintext header can be seen in the listing below.

The ciphertext header appears in plaintext, whereas the inner plaintext header is encrypted. As such the ciphertext header contains information for deprotection. The inner plaintext header then has the content type and the actual content of the message along with some padding.

```
1 struct {
2     opaque content[DTLSPlaintext.length];
3     ContentType type;
4     uint8 zeros[length_of_padding];
5 } DTLSInnerPlaintext;
6
7 struct {
8     opaque unified_hdr[variable];
9     opaque encrypted_record[length];
10 } DTLSCiphertext;
```

Chapter 4

Modeling the protocol

The DTLS protocol was modeled and analyzed using Tamarin Prover. The Tamarin Prover is explained in detail in section 4.1, and utilizes the threat model explained in section 4.1.1.

A total of three models were created; one modeling the full DTLS handshake explained in section 3.1.1, one modeling a handshake using PSK explained in section 3.1.2 and finally one modeling the record layer following a successful handshake explained in section 3.1.3.

While modeling, a modular approach was taken. All handshake message was modeled in a similar fashion, closely resembling the structures explained in section 3.2 and the specifications [1]. Each header also follows the same process, and both the handshake header and record header was modeled according to the structure explained in section 3.2.2 and 3.2.3. Elements of the models that were changed in comparison to the specifications are highlighted throughout the subsections in section 4.2.

After presenting these building blocks in section 4.2, the models are validated in section 4.3. The security properties mentioned in section 2.1 were translated into Tamarin lemmas and validated in section 4.4.

4.1 Tamarin Prover

Tamarin Prover is a security protocol verification tool that allows users to prove and disprove symbolic models. The formalism used in Tamarin is based on multiset rewriting and allows for creation of detailed models and attackers in the style of Dolev-Yao [11].

Tamarin also offers symbolic Diffie-Hellman support, the ability to have an unbounded amount of instances running simultaneously, and the addition of a graphical interface to visualize proofs. These features make Tamarin a fitting choice as a tool for modeling DTLS and other complex protocols.

As mentioned previously, Tamarin uses multiset rewriting, which is used for the “rules” within the language. An example of a rule inspired by the Tamarin documentation can be seen below:

```

1 rule example:
2   [ In(X),
3     Fr(~msg),
4     !Ltk($A, key) ]
5   --[ Sessionkey($A, key) ]->
6   [ Out(senc(~msg, key)) ]

```

Tamarin rules consist of three separate parts: The first part in line 2-4 (premises), The middle part in line 5 (actions) and the final part in line 6 (conclusions). Both the premises and the conclusions contain what is known as “facts”. Tamarin functions by having a global state which is defined by a multiset (a collection) of facts. The multiset is modified each time a rule is executed.

Tamarin makes use of different types of variables:

- \sim denotes fresh variables. These variables are unknown to the adversary on creation, and are unguessable.
- $\$$ denotes public variables.
- $\%$ denotes a numerical variable. Numerical variables can be compared to each other in terms of value. The only natural number that exists in Tamarin is $1:\text{nat}$. It is possible to create all positive numbers by adding one to it repeatedly.
- $\#$ denotes a temporal variable. This is the type of all timestamps.
- $'c'$ are string constants, which are also public.

Tamarin has multiple types of reserved facts.

- The "In" and "Out" facts are reserved for communication over an unsafe channel. All "Out" facts are processed by the adversary, allowing them to get knowledge from the input, before creating a matching "In" fact.

- The "Fr" fact is a built-in fact that denotes a freshly generated name. Such \sim variables should always on creation appear in the "Fr" facts in the premise of the rule.
- Facts prefixed with an ! are persistent. Rules will consume facts, however persistent facts can be consumed multiple times.

Returning to the example, the rule can only be executed if all facts in the premise are available for consumption. In order for "In(X)" to be in the global state, there must have been an "Out(X)" in a previous rule conclusion. The freshly generated variable $\sim\text{msg}$ is always available for consumption. Finally the "!Ltk(\$A, key)" requires a previous conclusion containing the fact, however since the fact is persistent, it may have been consumed before.

The conclusion sends out a message with the "Out" fact. The message is symmetrically encrypted using built-in cryptographic functionality with the "senc" function. The message $\sim\text{msg}$ is encrypted with the key and sent out.

Actions specify observable events, and is used to express certain security properties. Security properties are encoded as lemmas, an example of which can be seen below:

```
1 lemma example_property:
2   "All Actor key #i. Sessionkey(Actor, key) @ i
3   ==> not (Ex #j. K(key) @ j)"
```

The lemma uses one action *SessionKey*, and ensures that any case where this action is executed, does not result in the adversary knowing the key. These actions are executed at certain time slots (#i in the case of *SessionKey*). These time slots can be assigned to be before or after any other given time slots, but in the case of this lemma, the time slots #i and #j are independent of each other. For a more detailed look at Tamarin, we refer to the Tamarin manual [12].

Tamarin makes great use of pattern matching, and uses Tamarin tuples for this. The syntax for the tuples is the < and >. The example below shows tuples in Tamarin. The first rule "Tuple1" sends out the variables x, y and z. These variables are then received in the *In* fact in rule "Tuple2", however the tuple with the variables y and z are just received as a new variable rest, which is equal to "<y, z>".

```
1 rule Tuple1:
2   [ ... ]
3   -->
4   [ Out(<x, <y, z>>)]
5
```



```

6 rule Tuple2:
7     [ In(<x, rest>) ]
8     -->
9     [ State(z, rest) ]

```

4.1.1 Threat Model

Tamarin, and by proxy, the model makes use of a Dolev-Yao attacker. The attacker has complete control over the network and can not just eavesdrop, but also intercept, replay, delete and send synthesized messages on the network. The attacker can gain information about all sent messages on the network, and can extract information from protected messages given they have knowledge about the necessary keys. This information can be used to create new messages, which can be sent out in the network.

The threat model assumes perfect cryptographics, which limits the attacker to only breaking cryptographic properties if they have knowledge about the necessary keys. The adversary can also make use of all functions, but cannot break hash functions. The attacker is provided with further power to compromise long-term keys used for signing and the ability to compromise DH values. These extra abilities are added to the adversary to allow us to check for forward secrecy which revolves around an adversary that has gained compromised long-term keying material. The rules are shown in the listing below.

```

1 // Adversary power
2 rule Reveal_DHExp:
3     [ DHExp(~x, ~tid, $A) ]
4     --[ RevDHExp(~x, ~tid, $A), Corrupt($A) ]->
5     [ S_Out(~x) ]
6
7 rule Reveal_LTK:
8     [ !Ltk($A, ~ltk) ] --[ Corrupt($A) ]-> [ S_Out(~ltk) ]

```

4.2 Datagram Transport Layer Security Model

The model was created using a modular approach, modeling each part of a message individually according to the specifications [1]. Each message is divided into three parts, as explained in section 3.2. The modeling process takes a similar approach, and models the handshake messages, the handshake header and the record header separately.

This section will not explain all rules in detail, but rather the overall idea for the model. All models can be found in the attached material or on GitHub [13].

Key Servers

In a real life scenario, the public keys of peers are accessible using key servers, but within the context of this paper, the idea of key servers are abstracted away from. As such, each peer simply has knowledge of every public key in the system.

4.2.1 Modeling: Handshake message

We return to the example of ClientHello in section 3.2. The messages modeled in Tamarin can be seen in the listing below. The resemblance between the struct in section 3.2 and the Tamarin tuple is obvious. This is done in order to mimic the specifications as closely as possible, to limit the possibility of introducing errors into the model.

Some fields have been omitted since they only exist for legacy reasons and have a constant value. In the case of the ClientHello, these fields are the *legacy session id*, the *legacy cookie* and the *legacy compression method*. We also see the client keyshare extension, which was briefly discussed earlier. All extensions and other message structs can be found in the specifications [1].

```

1 ClientHello =
2   <
3     /*ProtocolVersion*/ protocolVersion(),
4     /*Random32*/ crandom,
5     /*CipherSuite*/ 'CipherSuits',
6     /*Extension*/ <
7       /*ExtensionType*/ '51',
8       /*KeyshareClientHello*/ <
9         /*KeyshareEntry*/ <'g', ga>
10      >
11    >
12  >

```

Aside from the removed fields, other modifications are also made to the specifications in order to properly model the protocol. One of the larger ones are the certificate and certificate verify messages.

Certificate and Certificate Verify

The server certificate is just the constant 'serverCert', which is sent in the certificate message. As for the certificate verify message, removing the headers leaves us with the message shown in the listing below. The verify message contains the entire history of the handshake and the certificate. This is all signed with the servers private key.

The client will use the certificate message and the certificate verify, in order to authenticate the server. Only if the history and the certificate matches, and the server's private key was used to sign the message, will the client accept. In the model we have full trust in the key-server, however as explained in section 4.1.1, the long-term server private key may be compromised.

```

1 CertificateVerify =
2   <
3     /*algorithm*/ $alg,
4     /*signature*/
5       sign(hmac(<h1(
6         <ClientHello,
7         HelloRetryRequest,
8         ClientHelloCookie,
9         ServerHello,
10        EncryptedExtensions,
11        ServerCertificate>),
12        'serverCert'>), serverPrivateKey)
13   >

```

Other changes include;

- Both the *HelloRetryRequest* and *ServerHello* have some legacy fields removed; *legacy_session_id_echo* and *legacy_compression_methods*.
- The protocol version is omitted since it is a constant value of {254,253}.
- *EncryptedExtensions* just contains a string, since we do not actually exchange any extensions.
- The *Certificate* message just includes a constant with either 'clientCert' or 'serverCert'. We do not model the direct way that a certificate is made or the methods used. This is enough since Tamarin promises perfect cryptographic.
- For the *CertificateVerify* the algorithm has been replaced with the public value *\$alg*. The rest follows the specifications closely.

4.2.2 Modeling: Handshake header

With the DTLS handshake header, we must properly handle the message sequence number and the fragmentation. The handshake header for ClientHello can be seen in the listing below.

The handshake type for ClientHello is set to a fixed '1'. The `msg_len` is the length of the message, which is a freshly generated value. The length is set to a fresh value. This is to make the length unknown to the adversary on creation. This allows queries to check whether the adversary can deduce the length solely from the content of the message. This will not ensure that the adversary does not know the length of the message, which is discussed further in section 4.4.

The first `1:nat` represents the sequence number for the message. Since the ClientHello is the first message in a connection, it has a sequence number of 1. Both epoch and sequence numbers normally start at 0, but since Tamarin natural numbers start from 1, the model sequence number and epoch likewise start from 1. This makes no difference to the output of the model.

```

1 HandshakeHeader =
2   <
3     /*HandshakeType*/ '1', <
4     /*length*/ ~msg_len,
5     /*sequence number*/ 1:nat,
6     /*fragment offset*/ 1:nat,
7     /*fragment length*/ 1:nat>,
8   ClientHello
9   >

```

The fragment offset and fragment length, has been modeled in a different way to its specifications explained in section 3.2.2. In the model, the fragment offset represents the fragment's order in the entire record and the fragment length represents the total amount of fragments.

In the ClientHello handshake header, the fragment offset and the fragment length are both set to 1, which means the fragment is this records first fragment out of 1. This also means that the record is not fragmented.

To showcase a fragmented example, see the listing below, which show the first and last fragment of a fragmented certificate message. We see that fragment offset is set to 1 in the first message, and to the max (5 in this case) in the last message. The fragment length

is also max, and this information alone is enough to inform us that the first message is the first fragment and the last message is the last fragment. Section 3.2.2, explains fragmentation in detail.

```

1  DTLSCipherTextSC_1 =
2    <
3      /*HandshakeType*/ '11', <
4        /*length*/ msg_len_1,
5        /*sequence number*/ %next_seq,
6        /*fragment offset*/ 1:nat,
7        /*fragment length*/ %max>,
8      /*certificate_entry*/ 'ser'
9    >
10
11 DTLSCipherTextSC_5 =
12   <
13     /*HandshakeType*/ '11', <
14       /*length*/ msg_len_5,
15       /*sequence number*/ %next_seq,
16       /*fragment offset*/ %max,
17       /*fragment length*/ %max>,
18     /*certificate_entry*/ 't'
19   >

```

4.2.3 Modeling: Record layer

The record layer structure was explained in section 3.2.3, and the model follows this specification closely. The record header structure does not change much between different flights, however there are two distinct cases; plaintext header and ciphertext header.

Plaintext record header

For the plaintext record header we use the example of ClientHello. The Tamarin implementation can be seen in the listing below, with DTLSHandshakeMsg, being the DTLS handshake message with the handshake header.

```

1  DTLSPlaintext =
2    <
3      /*ContentType*/ '22',

```

```

4      /*epoch*/ 1:nat,
5      /*sequence_number*/ 1:nat,
6      /*length*/ ~msg_len,
7      DTLSHandshakeMsg
8  >

```

The record header consists of a ContentType, Epoch, SequenceNumber, the length of the fragment, and the fragment itself. The ProtocolVersion seen in the struct for the plaintext record layer has been omitted, as it is legacy and will always be the same. The ContentType is '22' for all handshake messages.

Ciphertext record header

For the Ciphertext record header we use EncryptedExtensions as an example. The Ciphertext can be seen in the listing below, and we see that the unified_hdr has the bit configuration 001-0-1-1-10. This bit configuration specifies that it is an encrypted message, without connection id, with a 16 bit sequence number, a length present, and the epoch 2. The sequence number is also included in the un-encrypted part of the message in order to aid de-protection.

Ciphertext also contains the DTLSInnerPlaintext that encrypts the content of the message, a padding of zeros (in the case of our model, it is a singular zero) and the content type using the traffic key. The next sequence number is carried between flights, and both the server and the client keeps track on what sequence number is next.

```

1  DTLSInnerPlaintext =
2      senc{<
3          /*content*/ <
4          /*HandshakeType*/ '8',
5          <
6          /*length*/ ~msg_len,
7          /*sequence_number*/ %next_seq,
8          /*fragment_offset*/ 1:nat,
9          /*fragment_length*/ 1:nat
10         >,
11         EncryptedExtensions
12     >,
13     /*zeros*/ <'0'>,
14     /*ContentType*/ '22'
15 >}traffic_key

```

```

16
17
18 DTLSCipherText =
19   <
20     /*unified_hdr*/ <
21       /*first three bits*/ <'0','0','1'>,
22       /*Cid flag*/ '0',
23       /*Sequence number length flag*/ '1',
24       /*Length flag*/ '1',
25       /*First two bits of epoch*/ <'1','0'>
26     >,
27     /*sequence number*/ %next_seq,
28     /*encrypted_record*/ DTLSInnerPlaintextEE
29   >

```

4.2.4 Modeling: Additional behavior

Some of the model's behavior is modeled using restrictions. These restrictions allow us to limit the model in a way, such that unintended traces are not generated. An example of such a restriction is the *Dont_essablsh_session_with_self* restriction seen in the listing below.

The restriction states that an actor may not establish a session with itself. This is unintended behavior from an implementation point of view, and we therefore want to limit the model's behavior as well.

```

1 restriction R2_Dont_essablsh_session_with_self:
2 "All actor1 actor2 role session_id #i.
3   Start(actor1, actor2, role, session_id) @ #i
4   & actor1 = actor2 ==> F"

```

Another example is ordered fragments, which were explained in 3.2.2. The fragments should be ordered according to their value, which is enforced with the lemma in the listing below.

```

1 restriction R4_Fragments_in_order:
2 "All %frag_offset_1 %frag_offset_2 %frag_offset_3 %frag_offset_4
3   %frag_offset_5 #i.
4   FragmentsInOrder(%frag_offset_1, %frag_offset_2,
5                     %frag_offset_3, %frag_offset_4,
6                     %frag_offset_5) @ i

```

```

7      ==> %frag_offset_1 < %frag_offset_2
8          & %frag_offset_2 < %frag_offset_3
9          & %frag_offset_3 < %frag_offset_4
10         & %frag_offset_4 < %frag_offset_5 "

```

4.2.5 Modeling: UDP

In previous work [10] the unreliability of UDP was explicitly modeled. However, explicit modeling of UDP may be simplified, since the adversary provides implicit resending of unreceived messages.

First, it is vital to distinguish between the adversary "resending" a dropped message, and "replaying" a message. Replaying of messages is unwanted behavior, since it involves the message being processed more than once. Sequence and epoch are among other things included to avoid this very issue. Resending of dropped messages are on the other hand wanted behavior, since the message will only be processed once.

Unreliability can easily be modeled with the single rule in the listing below. This rule will silently consume *In* facts. The *re-send* properties are provided solely by the adversary. In tamarin, the adversary gains knowledge about the message when the *Out* is consumed, and the *In* is created. The adversary may then resend the messages, since they have knowledge about them.

Since the message that is resent, is exactly the same as the initial message, this approach does not limit the model.

```

1 rule Drop :
2   [ In(x) ] --> [ ]

```

4.3 Model validation

Model validation is a crucial step of any model creation, as this is the part where the model is evaluated against the specifications. This helps ensure that the model behaves as the specifications, or as a potential implementation. The model is solely validated against the specifications, as the only library for implementation, WolfSSL did not work for us.

Documentation versus Implementation

When one traditionally validates a model, the model will be directly compared to a trace of a real implementation. However, in the case of DTLS 1.3, there are limited implementation resources at the time of writing this paper. From the research done, the two main libraries that support DTLS is OpenSSL and WolfSSL. While OpenSSL supports DTLS, it currently does not support DTLS 1.3 and is capped at DTLS 1.2. A DTLS 1.3 update for OpenSSL is in the works, but it is currently still in the early stages, and cannot be used for validation [14]. WolfSSL has supported DTLS 1.3 in some capacity since June 2022 [15], but the compilation process has been troublesome even though the group has previous experience with WolfSSL. Since neither of the main implementations of DTLS are viable for the project, the only remaining option would be to implement a simple setup ourselves, but this is outside the scope of the project.

Thus, the validation of the model is based upon the documentation seen in RFC 9147.

Removing the adversary

To validate correctly, the adversary has to be removed, as validation focuses on the pure model. The adversary can mistakenly solve issues with the model, by altering the order or content of messages. This can consequently lead to the model seemingly functioning correctly, even with errors present.

Tamarin, unlike other similar tools, has a built-in adversary. This then proves to be a challenge, as Tamarin seemingly has no built-in method to disable the adversary. As a result, "Secure channels" were developed as a means to remove the adversary's influence, since an adversary can neither learn nor modify messages sent in these channels. The code for the secure channels can be seen in the listing below.

```
1  /* Channel rules */
2  rule Secure_Chan :
3      [ S_Out(m) ] --> [ S_In(m) ]
```

Validation

In order to validate the models, several lemmas were made within four major categories relating to parts of the DTLS handshake.

- Authentication
- Cookie exchange
- Finished handshake
- Order

These four are focused upon, as we believe them to be the key points that could result in failure if not thoroughly validated. The validation process highlighted some mistakes with the order in which messages were sent. Client authentication also was not possible initially, which was only found out later, since it is an optional branch.

Furthermore the handshake security properties in section 2.1.1 and record layer security properties in section 2.1.2 were all tested on the restricted model. These properties are important to test before the adversary is included, as they should be upheld in a legitimate DTLS connection. The full list of lemmas for the four main categories including a overview table, can be seen in Appendix B, where a short explanation of the lemmas also exists.

4.4 Security properties

With the DTLS models of the two handshakes and the record layer, the focus can now shift towards proving the security properties stated in section 2.1. The section narrowed the security promises of DTLS down to eight handshake properties (H1-H8), four record layer properties (R1-R4), and two general concerns about abusive behavior enabling potential DoS attacks (D1 & D2). Only the handshake and record layer properties were modeled, and all lemmas were tested against a mutated model, with intentional issues, in order to verify that the lemmas are correct and behave as expected. This section will present how these properties were modeled in Tamarin.

4.4.1 Handshake properties

Five of the eight handshake properties have been modeled in Tamarin.

Establishing the same session keys (H1) is per [8] limited to completed matching sessions between two uncorrupted parties. The only sessions that are completed with a *HandshakeComplete* action from both the client and the server, are those with a matching finished message,

that contains the entire session history. The lemma which can be seen in the listing below, claims that if the handshake was completed by both the client and the server, with matching nonce, then the session keys should be the same.

The lemmas was tested against a model where the server would commit another key than the one agreed upon. This would make the committed keys not the same, and the lemma would fail as expected.

```

1 lemma H1_Establishing_the_same_session_keys:
2 "All C S client_sid server_sid keyC keyS nonce #i #j.
3   HandshakeComplete(C, S, 'client', client_sid, keyC, nonce) @ #i
4   & HandshakeComplete(S, C, 'server', server_sid, keyS, nonce) @ #j
5   & not(Ex #p. Corrupt(C) @ p)
6   & not(Ex #q. Corrupt(S) @ q)
7   ==> keyC = keyS"

```

Secrecy of the session keys (H2) however asks the question of whether or not the session key is secure, or if the adversary has knowledge about the key. The lemma can be seen in the listing below, and regards the secrecy of a authenticated session key. The lemma states that if the actor has accepted a session key from an authenticated peer, while neither they themselves or their peer has been corrupted, then the adversary will not have knowledge about the key.

The lemma only regards authenticated session keys. This is done since an adversary could start a session and exchange unauthenticated keys with the server, and trivially have knowledge about the session key. The only time that the server can enjoy session key secrecy, is when they authenticate the client as well. However, since the completed sessions as per property (H1) ensure that the client and server has the same session key, since the client enjoys secrecy of session keys, the server will also enjoy secrecy in those cases.

In attempts to test the validity of the session key secrecy lemma (H2), issues with computation arose. For a more thorough explanation of computation issues we refer to chapter 5, but the issues resulted in a model where the traffic key was leaked too early, which resulted in the lemma not being computable. The lemma was also changed with these tests since an earlier version would incorrectly be marked as true, when the model leaked the key on purpose. We theorize that the unexpected pass is due to some timing in Tamarin, however the change fixed the issue.

```

1 lemma H2_Session_key_secrecy:
2 "All actor peer role session_id key #i.
3   SessionKey(actor, peer, role, session_id, key, 'auth') @ i

```

```

4      & not(Ex #p. Corrupt(actor) @ p)
5      & not(Ex #q. Corrupt(peer) @ q)
6      ==> not(Ex #j. K(key) @ j)"

```

Peer authentication (H3) ensures that when an actor believes they are talking to a peer, then the identity of the peer should match the actors idea of the peer. An example is when the client believes they are talking to the server, then it should be the case. This is proved by the lemma in the listing below, by having an actor commit to a peer's identity, when they accept it. When the actor has committed to the identity, then there should exist a peer that are currently running a matching session, which is the *HandshakeHistory*.

The lemma was tested on a mutated model, where the client would not ensure that the certificate had matching history. The lemma was expectedly false, in the mutated model.

```

1 lemma H3_Authentication:
2 "All actor1 actor2 role1 role2 HandshakeHistory #i.
3   CommitAuth(actor1, actor2, role1, HandshakeHistory) @ i
4   & not(Ex #p. Corrupt(actor1) @ p)
5   & not(Ex #q. Corrupt(actor2) @ q)
6   ==> (Ex #r. Running(actor2, role2, HandshakeHistory) @ r)"

```

Uniqueness of the session keys (H4) can be seen in the listing below and checks for machining session keys across multiple sessions. The lemma is restricted to check for matches within roles, so two different clients may not have the same session key, and two different servers may not have the same session key. This is done since a client and a server who established a session will share a session key, making the lemma fail unintentionally.

The mutated model for H4 has simply changed the traffic key for each peer. By doing this, the mutated model encountered some computation issues. Computation issues are discussed further in chapter 5.

```

1 lemma H4_Unique_session_keys:
2 "All actor1 actor2 peer1 peer2 role key session_id1 session_id2
3   auth1 auth2 #i #j.
4   SessionKey(actor1, peer1, role, session_id1, key, auth1) @ i
5   & SessionKey(actor2, peer2, role, session_id2, key, auth2) @ j
6   ==>
7     #i = #j
8     | (Ex #p. Corrupt(actor1) @ p)
9     | (Ex #q. Corrupt(actor2) @ q)
10    | (Ex #p. Corrupt(peer1) @ p)
11    | (Ex #q. Corrupt(peer2) @ q)"

```

Downgrade protection (H5) was not modeled within either of the three models. We do not regard lower versions of DTLS, and do not negotiation cryptographic configurations.

Forward secrecy with respect to long-term keys (H6) can be seen in the listing below and regards secrecy similar to (H2), but in scenarios where one of the actors has been corrupted. The lemma similarly to (H2) has a completed handshake between two peers, where none of them are corrupted before the handshake is completed. The server is the final part to complete the handshake, and the corruption must therefore happen after the timestamp $\#j$. The lemma then states that if the handshake is complete and one of the endpoints are corrupted after completion, then the adversary will not gain knowledge about the session key, if they did not have knowledge of it prior.

The mutated model for H6 should in theory be the same as H2, but by testing H6 on that model, it continued infinitely. This could be because the traffic key is leaked just as the handshake ends, and the lemma will therefore disregard it as a part of the handshake. By moving the "Out" one step back to the ClientFinished, the lemma results in a fail as expected.

```

1 lemma H6_Forward_secrecy:
2   "All C S session_id key nonce #i #j.
3     HandshakeComplete(C, S, 'client', session_id, key, nonce) @ i
4     & HandshakeComplete(S, C, 'server', session_id, key, nonce) @ j
5     & ((Ex #p. Corrupt(C) @ p & #j < #p)
6       | (Ex #q. Corrupt(S) @ q & #j < #q))
7     & not((Ex #k. K(key) @ k & #k < #j))
8     ==> not((Ex #k. K(key) @ k))"
```

Key Compromise Impersonation (KCI) resistance (H7) was not modeled, due to the way certificates are handled in the model.

Protection of endpoint identities (H8) was not modeled since the abstractions made for certificates, did not allow the certificates to be unknown to the adversary, since they must be directly known to the server and client. In section 4.2.1, we mention how certificates are modeled as fixed strings. These are known to the adversary per default, and such a lemma could not be written for the current models that could verify H8. This is a limitation of the models and not DTLS.

4.4.2 Record properties

All four record properties, explained in section 2.1.2, were modeled.

As discussed *confidentiality* (R1) can mean a lot of things, however DTLS restricts confidentiality to the concealment of plaintext and length concealment. Concealment of plaintext can be seen in the listing below. It states that for all messages sent, if the adversary knows the message, it must be because they know the session key as well.

```

1 lemma R1_Confidentiality:
2 "All actor1 actor2 msg session_key len #i #j.
3   SendMsg(actor1, actor2, msg, session_key, len) @ i
4   & K(msg) @ j
5   ==> Ex #l. K(session_key) @ l"

```

The *integrity* (R2) lemma can be seen in the listing below, and ensures that all messages received, are the actual messages sent. The lemma is limited to traces where the adversary does not know the session key.

```

1 lemma R2_Integrity:
2 "All actor1 actor2 msg session_key len #i.
3   ReceivedMsg(actor1, actor2, msg, session_key, len) @ i
4   & not(Ex #l. K(session_key) @ l & #l < #i)
5   ==>
6     Ex #j. SendMsg(actor2, actor1, msg, session_key, len) @ j
7     & #j < #i"

```

Ensuring *length concealment* (R3), was a lot harder. The length of the message may be deduced from other parameters than the written length in the header, and this model does not promise complete length concealment, however it does ensure that the adversary will not be able to deduce the length of the message from the content of the message. The lemma in question can be seen in the listing below.

```

1 lemma R3_Length_concealment:
2 "All actor1 actor2 msg session_key len #i.
3   SendMsg(actor1, actor2, msg, session_key, len) @ i
4   ==> not(Ex #j. K(len) @ j)"

```

The final record layer property *forward secrecy* (R4) regards key updates. The lemma can be seen in the listing below, and states that if a message is sent out on an old key, and the adversary does not know that message, then the introduction of a new session key, which the adversary knows, will not reveal the old message.

```

1 lemma R4_Forward_Secrecy:
2 "All actor1 actor2 role msg_old session_key_old session_key_new
3 len #i #j #m.
4   SendMsg(actor1, actor2, msg_old, session_key_old, len) @ i

```

```
5      & not(Ex #k. K(msg_old) @ k & #k < #j)
6      & RegisterSessionkey(role, 'send', session_key_new) @ j
7      & K(session_key_new) @ m
8      & #j < #m
9      & #i < #j
10     ==> not(Ex #l. K(msg_old) @ l)"
```

Chapter 5

Analysis and results

In order to verify the security properties, the adversary had to be introduced back into the model. The verification was done on a group member's home computer, using Windows Subsystem for Linux with 32 GB allocated ram. This introduced some computation issues, due to the complexity of the model, with both computation time and memory being an issue.

Computation time alone does not necessarily propose an issue when it comes to verification, since the verification only has to take place once. Whether it takes a week or a day to verify a model, doesn't really matter in the long run, since it does not have to be run repeatably.

The issues with computation time presented themselves during the development of the model and verifications queries. The model took ~ 20 minutes to load and another ~ 20 minutes to open. This leaves a ~ 40 minute overhead on changes done to the model during development. This also does not take into account the time it takes to run the queries. Measures were implemented to limit the resources used during development, which is explained in section 5.1.

While issues with computation time was not a complete barrier, the same can not be said about memory. If the hardware runs out of memory before the verification is complete, either the model and verification query must be changed, or better hardware must be used. Measures were taken in order to tackle the growing state-space and ease verification, and is explained in section 5.2.

5.1 Debug lemmas

In order to ease the development of the lemmas, two limiting debug lemmas were introduced. These lemmas would limit the amount of clients and servers present in the model.

One of the strong advantages of tamarin, is its ability to host an unlimited amount of clients and servers. This is a large benefit compared to model checkers like UPPAAL and ProVerif, whose potential were explored in an earlier report [16]. However this also comes with a greater cost, since an unoptimized model may have major computation issues. As such, the debug lemmas seen in the listing below were introduced to aid the development process of the security lemmas.

```

1  restriction DEBUG_Only_one_session_client:
2  "All a1 a2 a3 a4 sid1 sid2 #i #j.
3      Start(a1, a2, 'client', sid1) @ i
4      & Start(a3, a4, 'client', sid2) @ j
5      ==> #i = #j"
6
7  restriction DEBUG_Only_one_session_server:
8  "All a1 a2 a3 a4 sid1 sid2 #i #j.
9      Start(a1, a2, 'server', sid1) @ i
10     & Start(a3, a4, 'server', sid2) @ j
11     ==> #i = #j"

```

5.2 Combating memory issues

In order to limit the issues with memory use, the amount of accessible states must be reduced. The state-space can be reduced by introducing some additional information into the model.

Limiting the amount of rules that can provide a fact

One way additional information was introduced directly into the model, was by limiting the amount of rules specific facts could be gathered from. We will explain this with an example from the model.

An earlier iteration of the model made use of two facts to store both the current sending

and receiving sequence number and epoch. The two facts would contain the next sequence number and epoch to use, and the next sequence number and epoch to receive.

These two facts were named *ServerNextSeqAndEpoch* and *ServerRecSeqAndEpoch*, and would be present in almost all rules premise and conclusion. This was an issue, since during verification, if a rule had *ServerNextSeqAndEpoch* in its premise (which they almost always had), then it could be received from almost all other rules (since almost all rules had the facts in their conclusion as well). Due to the protocol like structure of the model, at any time, only one or two rules are relevant, and all other rules with *ServerNextSeqAndEpoch* would not themselves be executable at that time. However, the prover would not know this, and would check them anyways.

This lead to a change to the model. Instead of having two facts *ServerNextSeqAndEpoch* and *ServerRecSeqAndEpoch*, they were combined into one fact *ServerSeqAndEpochACK*. The fact was also made rule specific, which did not compromise the model. An implementation would also know what message to expect next, and would hold the sequence number and epoch up against that information as well. This greatly improved computation, since the Tamarin Prover now did not have to check all the rules repeatably.

This was later expanded upon further, and the final model only has one fact for the state of each rule (ex *St_ClientHello*). This way the Tamarin Prover would not attempt to gather the current state from multiple different instances, just to later realize that this was not possible.

Dividing the lemmas into smaller parts

Another way to reduce the state-space was to divide the verification into smaller parts. Tamarin allows for lemmas with the "reuse" property, which we will call "reuse lemmas". This allows the lemmas to work as a restriction for other lemmas.

Reuse lemmas and restriction differentiate in a very interesting way. Restrictions cannot be verified, and its use should be kept to the modeling aspect, by using it to model intended behavior as seen in section 4.2.4. Reuse lemmas however can be verified, and can thus be used to ease the verification of larger lemmas, by restricting already verified paths.

An example of this is the Diffie-Hellman challenge. The Diffie-Hellman challenge is a reuse lemma that was introduced to help the adversary know when it is possible to deduce the traffic key, created using the Diffie-Hellman method.

During verification, a lot of traces check for the adversaries knowledge of the private parts

of the Diffie-Hellman key (a and b). This took up a lot of time and memory, and therefore the reuse lemma was made in order to settle whether or not the adversary is able to know the key under perfect conditions. By verifying this reuse lemma, it can be used as a restriction to the verify the lemmas in section 4.4.1.

The lemma can be seen below and is heavily inspired by the work made by [4].

```

1 lemma dh_challenge[reuse]:
2 "All actor1 actor2 random1 random2 g a b ga gb gab #i #j #r.
3   DHChal(actor1, random1, g, a, ga, gb, gab) @ i
4   & DHChal(actor2, random2, g, b, ga, gb, gab) @ j
5   & K(gab) @ r
6   ==>
7     (Ex #p. (RevDHExp(a, random1, actor1) @ p & #p < #r)) |
8     (Ex #q. (RevDHExp(b, random2, actor2) @ q & #q < #r))"
```

However, even lemmas such as the Diffie-Hellman challenge, may be too large to verify, so other lemmas like the "session_id_invariant" below was created.

This reuse lemma is quite a bit simpler, stating that rules that take place in the middle of a handshake may only occur if the handshake has been initiated. Tamarin, while being a good tool for protocols, still relies on multiset-rewriting, and all rules are checked to see whether they aid the current goal or not. This reuse lemma stops Tamarin from considering rules that appear in the middle of a handshake that has not yet been started.

```

1 lemma Session_id_invariant[reuse]:
2 "All session_id actor role #i.
3   Instance(session_id, actor, role) @ i
4   ==> (Ex peer #j.
5     Start(actor, peer, role, session_id) @ j & (#j < #i))"
```

In order to run all the lemmas created in section 4.4 a handful of these reuse lemmas had to be created. For a complete list of all the reuse lemmas we refer to the attached models, or the model on GitHub [13].

5.3 Record layer model restrictions

The record layer model required additional restrictions. The model has been restricted to only allows for one key update, and for application data to be sent twice. These restrictions

were set in place, since in contrary to a handshake, the record layer does not have a theoretical end to it.

The model also had to be further restricted, by only having one client/server group. The handshakes are relatively linear, with only one potential branching (optional client authentication). The record layer can constantly choose between the options of updating the server key, updating the client key or sending application data. This results in a plethora of branches, with what is to be expected similar results. Some of the branches can be seen in table 5.1.

This branching resulted in additional time and memory issues, compared to the handshake. The record layer is less complicated, but ultimately the record layer had to be restricted in order to verify the security properties R1-R4.

	Branch 1	Branch 2	Branch 3	Branch 4
Step 1	Client app data	Client app data	Client app data	Client app data
Step 2	Client app data	Client app data	Client key update	Server key update
Step 3	Client key update	Server key update	Server key update	Client key update
Step 4	Server key update	Client key update	Server app data	Server app data
Step 5	Server app data	Server app data	Client app data	Client app data
Step 6	Server app data	Server app data	Server app data	Server app data

Table 5.1: Four of the branches that the record layer creates

5.4 Results

The lemmas mentioned in section 4.4 were checked against the model using Tamarins interactive prover. All modeled security properties were successfully verified in all three models, with various restrictions. The results and restrictions can be seen in table 5.2.

This shows us that the additional functionality and changes made to DTLS in comparison to TLS does not introduce any new security breaches, within the scope of our models. The addition of sequence numbers and epochs in clear text and the additions of a cookie exchange does not affect confidentiality, integrity nor authentication.

Property	Lemma name	Passed	Restrictions
H1	H1_Establishing_the_same_session_keys	✓	Unrestricted
H2	H2_Session_key_secrecy	✓	Unrestricted
H3	H3_Authentication	✓	Unrestricted
H4	H4_Unique_session_keys	✓	Unrestricted
H5	Not tested	X	-
H6	H6_Forward_secrecy	✓	Unrestricted
H7	Not tested	X	-
H8	Not tested	X	-
R1	R1_Confidentiality	✓	Only one server/client
R2	R2_Integrity	✓	Only one server/client
R3	R3_Length_concealment	✓	Only one server/client
R4	R4_Forward_Secrecy	✓	Only one server/client

Table 5.2: The results of the lemmas tested.

Chapter 6

UPPAAL translation

While Tamarin has proven to be a valuable tool when it comes to analyzing authentication, confidentiality, and integrity mentioned in section 2, analyzing availability does not come intuitively with a Tamarin model. Tamarin makes use of natural numbers to a limited extent, with its recommended use limited to simplistic counters.

Analyzing availability with a Tamarin model, may not be impossible, however being able to translate a Tamarin model directly into a UPPAAL model, would allow access to UPPAAL's more powerful notion of numbers and time. Analysis on DTLS power consumption has been modeled previously in UPPAAL [3], showing that these precise clocks and large numbers make analysis of availability possible.

UPPAAL is a tool that models in a network of timed automata, that are extended to include traditional data types. We refer to the UPPAAL documentation for more precise descriptions [17, 18].

This chapter will in section 6.1 present the thoughts and considerations put into a possible translation of Tamarin models into UPPAAL models. Section 6.2, then presents a translation method that works for simple and limited Tamarin models. Finally, in section 6.3 this is followed up by thoughts and considerations on what changes are needed in order to include more of Tamarins features.

It should be noted that the result of this chapter is proof of concept, and not the complete work. The translation presented, does not utilize all Tamarin features, and the model itself is also still an over-approximation. We do however believe that this draft has potential, and with further work this idea could become a realistic translation method.

6.1 Tamarin syntax and semantics

The first step, was to look at the syntax and semantics of Tamarin, in order to effectively map it to UPPAAL's syntax and semantics. Both the syntax and the semantics of Tamarin differ a lot from UPPAAL's, and previous work done by the group [16] highlights the formal differences between the two, detailing the large difference in formalisms.

In Tamarin, four major aspects were noted as crucial in the translation process.

- Variables
- Facts
- Pattern matching
- Actions when they are involved in restrictions

6.1.1 Tamarin variables translated to UPPAAL

Tamarin variables, with the exception of natural numbers, are just a symbolic representation of a name. This differs a lot from UPPAAL, where most variables contain actual values.

The rule in the listing below creates a fresh variable $\sim var$. When using the Tamarin Prover, the first time the rule is executed, a variable $\sim var$ is created. If the rule was executed again, another variable of the same name is created. Internally these two variables are distinct, and will in the interactive prover most likely be represented as var and $var.1$.

```
1 rule Variable_example:
2   [Fr( $\sim var$ )] --[ ]-> []
```

We can see this as well if we notice the updated rule in the listing below. We have added an action *NewVar*, which takes the fresh variable as an input. We also have a lemma, that asks whether two variables not created at the same time ($not(\#i = \#j)$) can ever be the same. The lemma will confirm our intuition, that they cannot, thereby indeed being distinct.

```
1 rule Variable_example:
2   [Fr( $\sim var$ )] --[ NewVar( $\sim var$ ) ]-> []
3
```

```

4 lemma Are_variables_distinct:
5 " All var1 var2 #i #j.
6   NewVar(var1) @ i & NewVar(var2) @ j & not(#i = #j)
7   ==> not(var1 = var2)"

```

Regarding the other variable types shown in section 4.1, public variables work in a similar way. See the example rule "Public_variable_example" in the listing below.

```

1 rule Public_variable_example:
2   [] --[] -> [State($var)]

```

If this rule is run twice, we have two cases. We may have an instance of the fact *State* with a public variable *\$var* and another with *\$var.1* or we have two instances of the fact state with *\$var*.

The case of strings is different however. They are constant, and if we have a rule similar to "Variable_example" but with strings, the lemma would not hold, and the opposite would be true. In all cases the variables would be the same. The example can be seen in the listing below.

```

1 rule String_variable_example:
2   [] --[] -> [State('var')]

```

Finally numerical variables are the only variable within Tamarin, that has an actual "value". They work as expected, and the number 1 is equal to another number 1, and 2 is larger than 1.

With this understanding of variables in Tamarin, the variables were modeled in UPPAAL in section 6.2.1.

6.1.2 Tamarin facts translated to UPPAAL

Translating the facts was a bit more convoluted than translating the variables. Facts contain two parts, the facts themselves, and the variables they contain. Looking at only the facts themselves, facts in tamarin are "created" and "consumed" as mentioned in section 4.1. When a fact is created, it is entered into the global multiset, where it is kept until possibly being consumed by a later fact.

Two facts are differentiated by both its name and amount of inputs. An example of this is the fact $Fact(x, y)$ and $Fact(x, y, z)$. If $Fact(x, y)$ is in the current multiset, it cannot be

consumed as $Fact(x, y, z)$. Two facts with different names $Fact1(a, b)$ and $Fact2(a, b)$ also are not related. This gives us a lot of leeway when translating facts. This can be seen with the two ideas in appendix C and the final idea in section 6.2.3

Special facts

There exists some special facts, two of which being the *In* and *Out* facts. These facts concern the adversary, and the content of an *Out* fact is parsed by the adversary before becoming an *In* fact. All *In* and *Out* facts also only take one variable.

6.1.3 Pattern matching

Pattern matching plays a large part in Tamarin, and can roughly be divided into two parts. Firstly, pattern matching can be used to ensure variables have the same value. An example of this can be seen in the rule "Pattern_matching1" in the listing below. The two facts $Fact_A1$ and $Fact_A2$ shared the same variable id . This means that id must be the same symbolic value, in both the facts $Fact_A1$ and $Fact_A2$.

```
1 rule Pattern_matching1:
2   [ Fact_A1(id, a),
3     Fact_A2(id, b) ]
4   -->
5   [ ... ]
```

The second part of pattern matching, is the tuples in Tamarin. The two rules in the listing below, showcase this. Both these facts have two variables, and "Pattern_matching2b" may consume the fact in "Pattern_matching2a".

```
1 rule Pattern_matching2a:
2   [ ... ]
3   -->
4   [ Fact(id, <a, b>) ]
5
6 rule Pattern_matching2b:
7   [ Fact(id, c)]
8   -->
9   [ ... ]
```

6.2 Translation for a restricted Tamarin model

This section will present a translation from Tamarin to UPPAAL that works on restricted Tamarin models. The models may not use actions outside of verification purposes (no restrictions or reuse lemmas), and may not use the Tamarin tuples. The work presented here functions as proof of concept, and its use has only been theorized and not tested. The translation itself has been manually tested on a very small example Tamarin model. We provide this work as inspiration for further development, since its potential is apparent.

This draft utilizes aspects from all previous ideas, which can be found in appendix C. The main focus was to utilize more major aspects of UPPAAL. The section will first present how the symbolic variables from Tamarin is modeled in UPPAAL. Later, how rules are represented, and how both variables and facts act to create the global state is presented. Finally, the way in which individual facts were modeled, will be presented with examples.

6.2.1 Tamarin symbolic variables into UPPAAL variables

Section 6.1.1 highlighted how variables behave in Tamarin, with each variable being recognized by its name, and an index. As such, the two symbolic variables *A.1* and *A.2* are not the same. The struct in the listing below was created in order to contain these variables in UPPAAL. All symbolic variables are recognized by their name and their index, and two variables are only equal if both name and index are the same.

```
1 typedef struct {
2     int name;
3     int index;
4 } variable;
```

Since UPPAAL's use of strings are very limited, each variable were assigned an id representing its name. Since each individual name gets a unique number, this works identically to using strings. An example can be seen in the listing below. The index is modeled as an integer that is incremented, whenever a new instance of its corresponding variables is created.

```
1 const int id_identifier = 1;
2 const int a_identifier = 2;
3 const int b_identifier = 3;
4 const int x_identifier = 4;
5 const int y_identifier = 5;
```

```

6  const int A_identifier = 6;
7  const int B_identifier = 7;

```

With this variable struct, it is possible to create symbolic Tamarin variables in UPPAAL. We return to the example in section 6.1.1 (re-pasted below, with a different variable name), where the rule generates the fresh variable *id*. When this rule is executed twice, we will have two distinct variables *id* and *id.1*. In UPPAAL with this method, this rule would generate the two variables, *var1* and *var2*, in the UPPAAL code in the listing below.

```

1  /* TAMARIN CODE */
2  rule Variable_example:
3    [Fr(~id)] --[ ]-> []
4
5  /* UPPAAL CODE */
6  variable var1;
7  var1.name = id_identifier;
8  var1.index = id_index++;
9
10 variable var2;
11 var2.name = id_identifier;
12 var2.index = id_index++;

```

6.2.2 Tamarin rules into UPPAAL processes

Each Tamarin rule is translated into a single process in UPPAAL. This process mirrors the Tamarin rule, with its premise and conclusion. The process' purpose is to update the state with its conclusion, when its premise is true. Figure 6.1 shows an example of such a process.

The process has an initial location. This initial location can only be exited, if the equivalent Tamarin premise is true. This is highlighted with the guard *premise_check* and the variable *premise*. Each edge going from the initial location back into the initial location represents a single fact in the premise. If the fact is true in the global state, it can be consumed, and satisfy that part of the premise. When all parts of the premise is true, the process may leave the initial location.

The process can also at any point release the consumed facts back into the global state. This is done, since Tamarin will only consume facts when all facts in the premise is true and the rule is executed. In UPPAAL we consume the facts one at a time. The next section, explains how facts are modeled in UPPAAL, but ultimately facts must be immediately

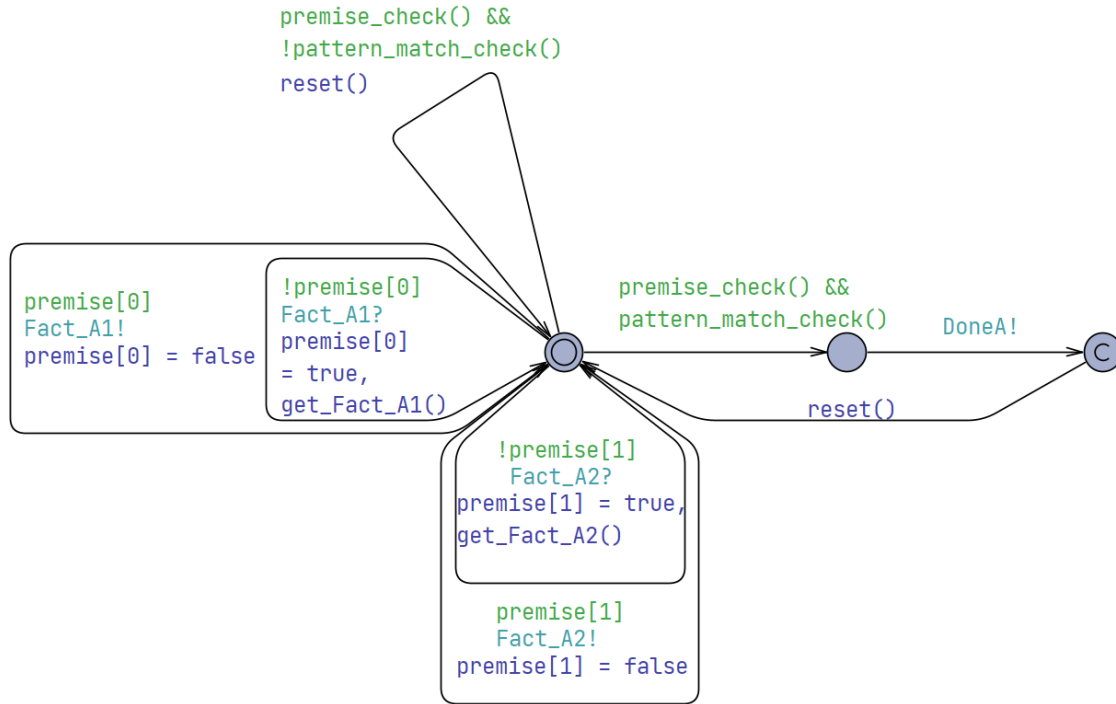


Figure 6.1: A basic example of the proposed model. In this case there is two facts.

consumed upon creation in this translation. We therefore have processes "unconsume" facts again, to mimic the facts being free in the global state. Processes will in this case hold onto facts until they themselves or another process need them.

All following locations and edges represents the conclusion. Here the global state is updated with new facts.

6.2.3 Tamarin facts in UPPAAL

Facts were ultimately modeled as broadcast channels. This was decided in order to allow processes to take a transition, even if there were no responders. This represents a fact that will not be consumed. This has introduced some errors into the translation, since with the use of broadcast channels, more than one other process can synchronize on the channel. This is unwanted behavior for non persistent facts, and a channel that synchronizes with either one or zero other processes should be used instead.

Facts also contain variables, which must be transmitted from one rule to another when

the fact is consumed. This is done with the use of UPPAAL's global variables, where the rule that creates the fact, will save its variables in the global variables, for the rule who consumes the fact to fetch. These variables are then stored in the process' local variables.

Each fact has a fixed amount of variables, and can therefore be hardcoded as seen in the listing below. One could suspect issues could arise, where two sets of variables from the fact may need to be stored at the same time. This is not the case however, since the variables are only needed while the rule is actively being executed. This is also the case in Tamarin, where only one rule is executed at a time as well.

```
1 variable Fact_A1_var1;
2 variable Fact_A1_var2;
3
4 variable Fact_A2_var1;
5 variable Fact_A2_var2;
```

6.2.4 Pattern matching for equality checks

Earlier in section 6.1.3, the two types of pattern matching were explained. This draft only handles the first kind of pattern matching, where variables must have the same symbolic value. We reuse the example from said section, together with how this is modeled in UPPAAL. Both the Tamarin code and the UPPAAL code can be seen in the listing below.

In Tamarin we have the two facts *Fact_A1* and *Fact_A2* sharing the same value for *id*. In UPPAAL the check would look as the function *pattern_match_check()*, which compares the name of *Fact_A1*'s first variable with the name of *Fact_A2*'s first variable, as well as their index. Only if both match, do the two facts share the same value for their first variable. The previous figure 6.1 also shows how this pattern match check must be true before the rule is allowed to proceed, even if all facts of the premise have been consumed.

```
1 /* TAMARIN CODE */
2 rule Pattern_matching1:
3   [ Fact_A1(id, a),
4     Fact_A2(id, b) ]
5   -->
6   [ ... ]
7
8 /* UPPAAL CODE */
9 bool pattern_match_check() {
10   return Fact_A1_var1.name == Fact_A2_var1.name
11     && Fact_A1_var1.number == Fact_A2_var1.number;
```

12 }

6.2.5 Implementing cost

Finally, in order to have the translation work as a medium for availability analysis, some sort of cost should be implemented. The cost can be attached to the *In* and *Out* facts, in order to model the cost of receiving and sending messages. One may also associate some cost with other facts, in order to model the cost of memory. In this work, a template for cost was thought out and mildly experimented with, however this is detached from the Tamarin model itself, since it does not consider cost.

The representation of cost is ultimately dependent on the subsequent queries created and used for verification. This factor may vary from purpose to purpose, however we believe that cost associated with *In* and *Out* facts, can be used to compare the load of receiving and sending messages between honest users and the adversary. If the adversary is able to send messages at a cost that is much lower than the cost of receiving and answering to the messages, then this could be an potential attack on availability.

6.2.6 Adversary behavior in UPPAAL

The special facts *In* and *Out* is modeled very similar to all other facts, with the exception that another process "Adversary" will receive all *Out*'s and create all *In*'s. This is also similar to the adversary implementation in Tamarin. The adversary may create additional *In* facts, using previously learned values. The adversary is in a very early state of development in this proof of concept, and does not encompass all aspects of the adversary in Tamarin. Ways to store known values and how the adversary can generate new values must be thought up.

6.3 Further development

This section work as inspiration for future readers, attempting to expand on this translation idea. The section contain reflections from the group, on areas of Tamarin not yet covered. These ideas have not been tested, however we believe in their potential.

Pattern matching

The pattern matching issue can in theory be solved by implementing pattern matching in UPPAAL. Pattern matching is also a "nice to have" in Tamarin, and models can be made without using it, however this may not be a realistic requirement for developers of large and complex models.

Cryptography

One of the large upsides of Tamarin, is its built-in support for cryptography. We see the majority of this being modeled in UPPAAL without major issue. Only entities that have the key should be able to read the content of the message.

Broadcast channels

The choice of channel in UPPAAL was discovered flawed late in the project. We however, do believe that this could be easily fixed by limiting the broadcast channels, to one zero or one synchronizations.

Restrictions

In this work restrictions were not explored much, however we believe these could be implemented in UPPAAL. Restrictions are first order logic on actions, and all Tamarin actions could therefore be modeled in UPPAAL by adding the actions to the end of a rule execution in UPPAAL. Referring back to the example in figure 6.1 in section 6.2.2, the actions would happen on the final edge back into the initial location.

Actions would be stored as a timestamp, in which *action1* would be a clock, and have the value of the time the rule was executed. The restriction could thus be implemented as a guard with the first order logic, to ensure the model never reached a state where the restriction was true.

Chapter 7

Discussion

This chapter will reflect on several aspects of the report. This is done to highlight certain points of interest.

Modeling a specification

The work done within the report, and the resulting models created are solely based on the official specifications for DTLS 1.3 - RFC 9147 [1]. The models were also solely validated against the wording in the specifications.

The specification functions as the truth of how one should implement (or in this case, model) the DTLS protocol. Future implementations may introduce errors that the model does not currently capture, and may not be subject to the same security properties. It is important to acknowledge that there exists a gap between specifications and implementations, but proving the correctness of the specifications give a great baseline for future implementations.

Specifications has been modeled before, providing tangible results that aided in the final stages of creation. During the development of TLS 1.3, models of the TLS 1.3 drafts were made in order to analyze and locate security flaws in the drafts. Issues with the drafts were found, and subsequently fixed.

Closely following the specifications

While modeling, the specifications were followed quite closely, in order to ensure a high accuracy, and avoid any potential errors that could be introduced elsewhere. Tamarin also works great for this particular purpose, since its syntax follows the flow of a security protocol very closely.

We find this aspect of modeling security protocols in Tamarin very useful. Earlier work done by the group [3] [16], highlighted how modeling communications in a protocol into an arbitrary modeling language requires some language specific translations. In [3] the act of sending a ClientHello message, was modeled as a synchronization on a channel called ClientHello. This can still correctly model DTLS, however the option in Tamarin, to send out a message that structurally looks similar to a ClientHello in the specifications, provides further reassurance that the specifications are being correctly modeled.

Expanding the model further

Due to the scope of the project, the model is based on a segment of the DTLS protocol. We believe that an implementation of this segment would be able to at least support realistic IoT communication, however it does have potential for expansion. The modular method of modeling the protocol presented in section 4.2 allow for systematic expansion, with the major limiting factor being the final size of the model.

The start of chapter 4 presents the challenges of expanding the models further, with ever increasing saturation and computation times. This is also the reason that the three models are disjoint, since loading and analyzing a full model became infeasible. The steep rise in complexity was introduced with the addition of natural numbers in the tamarin model.

An older model without natural numbers would finish fully loading in ~ 2 -3 minutes, while the final model loads in ~ 40 -50 minutes. The Tamarin manual [12] and the examples available on the Tamarin Github [5], only includes limited mentions and examples of models using natural numbers. It is hard to say whether the natural numbers are the direct reason for the long saturation times, however it does seem that way. The numbers are also currently strictly needed to properly model fragmentation.

Analyzing availability with UPPAAL

The extent of analysis targeting availability has been limited within the Tamarin models, however knowledge was gained from the analysis process. The analysis highlighted repeated attempts by the adversary to use the weak addresses provided by UDP to breach the connection. We find that this further highlights the importance for an analysis on this aspect of the protocol.

Ways to analyze availability within Tamarin was theorized, however it was ultimately decided against in favor of a possibly more lucrative approach. Previous work attempting to analyze availability with the use of UPPAAL [10], showcased UPPAAL's potential in this area. The group looked into ways to translate the Tamarin model into a potential UPPAAL model, for availability purposes. This resulted in chapter 6.

The UPPAAL translation is in a very early state due to time constraints, it does however show the potential of translating a model made in Tamarin, prime for analysis on confidentiality, integrity and authentication, into an UPPAAL model for analysis on availability. If this translation is sound, this would allow for models created in Tamarin to verify both confidentiality, integrity and authentication properties as well as availability properties.

Tamarin Prover

The Tamarin Prover was created with the purpose of analyzing security protocols [19], and is specialized in this exact task. Furthermore, Tamarin has previously shown itself as a very valuable tool for analyzing protocols, with numerous models being created for a plethora of different protocols [4] [5]. It also proved to be the most beneficial when three tools were compared in the report preceding this one [16].

Tamarin draws its major strengths from its formalism allowing infinite instances of servers and client, something not seen in other tools using different formalisms. It also models cryptographic very well, and has great support for cryptographic features.

One thing to be aware of when choosing Tamarin, is the lack of resources. For the most part, the documentation contains everything, and the documentation is seemingly up to date. It contains a few very simple examples, and is good for an easy introduction. There is a lack of more medium complex examples, with examples either being very simple or very complex. The Tamarin GitHub [5] contains a large collection of examples, however these are not regularly updated and are not explained in all cases. This leaves Tamarin with a steep learning curve, with limited resources when encountering errors outside the

documentation.

DTLS is a large and complex protocol, and the model quickly rose to a size where additional information outside the documentation was needed. Accessing this information, was a tedious process which slowed down development for a period of time. This resulted in a lot of trial and error in order to get new technologies to work within the model. A large example of this was the addition of natural numbers. This along with the long load times also affected the speed of development.

To use the Tamarin Prover effectively to verify lemmas, one needs a rather deep knowledge of multi-set rewriting. This is to be expected, but it is worth noting that users must gather this information elsewhere, since it is not part of the documentation.

Results

The results are very promising, with the two handshakes passing all modeled properties. The record layer was subject to some restrictions, by only having one server/client group, however we do not believe this limitation to have major consequences for the results. The adversary still has all of its power with this restriction, and the record layer only takes place after a successful handshake, which we have proven to be safe and have certain properties. The result of the handshake is a set of secret and authenticated keys, and we do not believe that the adversary would gain any meaningful information from another client/server group. The adversary themselves should at this point not be able to impersonate either client nor server, and the only information the adversary should be able to get from the connection is in the case where old keys are leaked. This however has not been proven, and is only our opinion after working extensively with the protocol, and also from working with the unrestricted record layer model.

The results show the theoretical correctness of a segment of the DTLS protocol, which we believe could be used in a realistic IoT implementation. We believe that the lack of people updating to DTLS 1.3 must be reasoned in something not related to the protocol itself, with the leading theories being the lack of libraries supporting DTLS 1.3, and the high cost of updating hardware.

Chapter 8

Conclusion

In conclusion, this work has proved that a segment of the DTLS 1.3 protocol, is secure in regards to the 9 tested security properties promised by the specifications. We believe that the chosen segment of the protocol is detailed enough, that it would be able to work as a realistic protocol for an IoT device.

The model closely resembles the specifications, and we believe fully in its correctness to the extent that it is a model of the protocol. While making a model, some abstractions will always be present. In this work, the major abstractions lie with the cryptographic aspects of the protocol. We assume perfect cryptographic, and do not model certificates to their full extent. The lemmas created in order to verify the security properties were mutation validated to ensure that they function as expected, and we believe in their correctness as well.

We conclude that the DTLS protocol is a good choice of a protocol for a lot of IoT devices. A small analysis of the security desires of IoT devices and the security properties of DTLS was made during this report. It was shown that all the properties in the CIAA tetrad are desired traits for various IoT devices. It was concluded that no protocol could provide all four traits to their fullest, and with the two examples; a security camera and smart lights, DTLS would effectively only defend against one of the two theorized attacks.

8.1 Future work

If one were to expand the work made in this project, a few options arise. First of all, the three models should be combined into one model in order to capture potential synergies. Other additional models could also be made to test other aspects of DTLS that was not covered by the three original models within the report. If the models were to be optimized to a point, where additional functionality could be added without state-space explosion issues, then the models are prime for expansion with the modular approach taken.

A translation of UPPAAL could be another focus point. This could, with ample time, be developed into a fully fledged script that could automate the proposed idea. With a complete UPPAAL translation, it would be possible to use Tamarin for the aspects where it excels; Confidentiality, Integrity and Authentication, and use UPPAAL for Availability.

Bibliography

- [1] Eric Rescorla, Hannes Tschofenig, and Nagendra Modadugu. *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*. RFC 9147. Apr. 2022. DOI: 10.17487/RFC9147. URL: <https://www.rfc-editor.org/info/rfc9147>.
- [2] *Qualys SSL Labs - SSL Pulse*. [Online; accessed 2025-05-28]. URL: <https://www.ssllabs.com/ssl-pulse/>.
- [3] Lise Bech Gehlert et al. “Modelling and Analysis of DTLS: Power Consumption and Attacks”. In: *International Conference on Formal Methods for Industrial Critical Systems*. Springer. 2024, pp. 136–151.
- [4] Cas Cremers et al. “A Comprehensive Symbolic Analysis of TLS 1.3”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1773–1788. ISBN: 9781450349468. DOI: 10.1145/3133956.3134063. URL: <https://doi.org/10.1145/3133956.3134063>.
- [5] tamarin-prover. *GitHub - tamarin-prover/teaching: Teaching materials related to the Tamarin Prover*. [Online; accessed 2025-06-03]. URL: <https://github.com/tamarin-prover/teaching>.
- [6] Jun Young Kim et al. “Automated Analysis of Secure Internet of Things Protocols”. In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACSAC ’17. Orlando, FL, USA: Association for Computing Machinery, 2017, pp. 238–249. ISBN: 9781450353458. DOI: 10.1145/3134600.3134624. URL: <https://doi.org/10.1145/3134600.3134624>.
- [7] Alex Hern. “Hacking risk leads to recall of 500,000 pacemakers due to patient death fears”. In: (2017). URL: <https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update>.
- [8] Ran Canetti and Hugo Krawczyk. “Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels”. In: *Advances in Cryptology — EUROCRYPT 2001*. Ed. by Birgit Pfitzmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 453–474. ISBN: 978-3-540-44987-4.

- [9] Whitfield Diffie, Paul C. Van Oorschot, and Michael J. Wiener. “Authentication and authenticated key exchanges”. In: *Des. Codes Cryptography* 2.2 (June 1992), pp. 107–125. ISSN: 0925-1022. DOI: 10.1007/BF00124891. URL: <https://doi.org/10.1007/BF00124891>.
- [10] Christoffer Brejnholm Koch et al. *IoT Power Consumption & DTLS Modelling*. Student report, Dept. Computer Science, Aalborg University. Available on GitHub: https://github.com/Goggon/DTLS_Paper_Models. Jan. 2024.
- [11] D. Dolev and A. Yao. “On the security of public key protocols”. In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650.
- [12] David Basin et al. *Tamarin Prover Manual*. <https://tamarin-prover.com/>. (Accessed on 11/11/2024).
- [13] *Repository for the project*. URL: <https://github.com/Goggon/P10>.
- [14] *GitHub - openssl/openssl at feature/dtls-1.3*. (Accessed on 04/23/2025). URL: <https://github.com/openssl/openssl/tree/feature/dtls-1.3>.
- [15] wolfSSL. *DTLSv1.3 support by rizlik · Pull Request #4907 · wolfSSL/wolfssl · GitHub*. [Online; accessed 2025-04-23]. URL: <https://github.com/wolfSSL/wolfssl/pull/4907>.
- [16] Signe Kirstine Rusbjerg and Tobias Møller. *DTLS 1.3 and modeling security in Tamarin*. Student report, Dept. Computer Science, Aalborg University. Jan. 2025.
- [17] *UPPAAL Documentation*. [Online; accessed 2025-05-12]. URL: <https://docs.uppaal.org/>.
- [18] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*. <https://homes.cs.aau.dk/~adavid/RTSS05/UPPAAL-tutorial.pdf>. (Accessed on 12/04/2023). 2005.
- [19] *Tamarin Prover*. URL: <https://tamarin-prover.com/>.
- [20] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Aug. 2018. DOI: 10.17487/RFC8446. URL: <https://www.rfc-editor.org/info/rfc8446>.

Appendix A

Structs of DTLS

These sections provide explanations of the structure for the different messages in DTLS.

A.1 ClientHello

The initial part of a handshake will always be a *ClientHello*, as the server will not actively initiate a connection. *ClientHello* is quite a small message, as it only includes the available cipher suits. As this message initiates the connection, it cannot be encrypted, and thus it is sent in plain text.

Technical details

A handshake is initiated by the client, with an initial *ClientHello* message, that follows the structure in the listing below. Lots of parameters exists for compatibility between earlier versions, so we focus on the *CipherSuit* and *Extension*. The client will send a list of valid cipher suits, that it is willing to use. The client can also append a number of extensions that the server might acknowledge. One extension usually sent here is the *Keyshare* extension, which is specific for *ClientHello* is the *KeyShareClientHello*, and contains a list of Diffie-Hellman groups and keys. This concludes the *ClientHello*, and the message is sent to the server.

```
1 struct {  
2     ProtocolVersion legacy_version = { 254,253 };
```



```

3     Random random;
4     opaque legacy_session_id<0..32> = [];
5     opaque legacy_cookie<0..28-1> = [];
6     CipherSuite cipher_suites<2..216-2>;
7     opaque legacy_compression_methods<1..28-1> = 0;
8     Extension extensions<8..216-1>;
9 } ClientHello;

```

A.2 HelloRetryRequest

Upon receiving the *ClientHello*, the server will respond with a *HelloRetryRequest*, with a cookie extension. This is done to mitigate DoS attacks. The *HelloRetryRequest* message type does not actually exist, and is just a *ServerHello* message with the Cookie extensions included. Both the official documentation and this paper will refer to this message as *HelloRetryRequest* for simplicity and understanding.

Technical Details

The *HelloRetryRequest* similarly to the *ClientHello* contains a lot of legacy parameters. The *Random* value is a fixed value for all *HelloRetryRequests* and is used to determine a *HelloRetryRequest* from a *ServerHello*. The *CipherSuite* is a single cipher suit chosen from the list of available cipher suit from the initial *ClientHello*. The *Extensions* contain a keyshare, the cookie extension and a supported versions extension.

The keyshare extension of *HelloRetryRequest* is called *KeyShareHelloRetryRequest*, and only contains the *NamedGroup*. The supported version extension will contain the version that the server will use. For the case of DTLS 1.3 this will always be the value *0xfefc*.

The cookie extension itself is rather simple, since it will just contain the cookie. However, the cookie has some important restrictions and requirements it has to adhere to.

```

1 struct {
2     ProtocolVersion legacy_version = { 254, 253 };
3     Random random = CF 21 AD 74 E5 9A 61 11 BE 1D 8C 02 1E 65 B8 91
4                     C2 A2 11 16 7A BB 8C 5E 07 9E 09 E2 C8 A8 33 9C;
5     opaque legacy_session_id_echo<0..32> = [];
6     CipherSuite cipher_suite;

```

```
7      uint8 legacy_compression_methods<1..28-1> = 0;  
8      Extension extensions<8..216-1>;  
9 } ServerHello (HelloRetryRequest);
```

A.3 Second ClientHello

When receiving a *HelloRetryRequest* from the server, the client must respond with a *ClientHello* that is identical to the one that triggered the *HelloRetryRequest*, with a few modifications. Two important ones are listed here but we refer to the RFC [20] for the full list. If the *HelloRetryRequest* contains a keyshare, the client should limit their cipher suites and keyshare to a list of a single keyshare from the same group. The client should also copy the content of the *HelloRetryRequest*'s cookie extension into its own cookie extension.

Technical Details

This *ClientHello* utilizes the same structure as the initial *ClientHello*, with the addition of the provided cookie from the *HelloRetryRequest*. It is paramount that the contents of the struct is the exact same as the original *ClientHello*, as the connection will be dropped otherwise.

A.4 ServerHello

When the server receives a *ClientHello* with a valid cookie extension, it may continue the handshake. The goal of this message is to ensure that the client accepted and provided the cipher suit that was sent in the *HelloRetryRequest*. This is also the final message sent in plaintext, as the connection is now ready to begin initial encryption.

Technical Details

A *ServerHello* is sent to the client containing a random value and the keyshare from the group agreed on between the *HelloRetryRequest* and the second *ClientHello*. The keyshare for the *ServerHello* is named *KeyShareServerHello*, and contains a single key and its named group. This message also concludes the series of messages sent in plain text.

The server can at this point calculate the *server_handshake_traffic_secret*, since it has chosen a keyshare group for the connection, and has both the clients and its own part of the Diffie-Hellman key.

```

1 struct {
2     ProtocolVersion legacy_version = { 254,253 };
3     Random random;
4     opaque legacy_session_id_echo<0..32> = [];
5     CipherSuite cipher_suite;
6     uint8 legacy_compression_methods<1..2^8-1> = 0;
7     Extension extensions<8..2^16-1>;
8 } ServerHello;
```

A.5 EncryptedExtensions

The server must send the *EncryptedExtensions* message immediately after the *ServerHello*. The message contains protected extensions that are not needed for the cryptographic context of the handshake, nor the individual certificates. The extensions used in this message can be seen in the RFC [20].

Technical Details

This is the first message to be sent with the encryption derived from keys. The structure of *EncryptedExtensions* is quite simple, and only contains a list of extensions.

```

1 struct {
2     Extension extensions<0..2^16-1>;
3 } EncryptedExtensions;
```

A.6 CertificateRequest

The *CertificateRequest* message is used by the server to request a certificate from the client. In a DTLS handshake the server must always be authenticated, whereas authentication is optional for the client. This means that for handshakes that use certificates as a way of authenticating, the server must deliberately request a certificate from the client. This is done

with the *CertificateRequest* message, which contains the *signature_algorithms* extension, that specifies the certificate algorithms the server is willing to use.

Technical Details

The *CertificateRequest* message includes two parts, a list of extensions, and the *certificate_request_context* which must be included in the *certificate* sent by the client in response.

```

1 struct {
2     opaque certificate_request_context <0..28-1>;
3     Extension extensions <2..216-1>;
4 } CertificateRequest;
```

A.7 Certificate

The certificate is used to prove an agent's identity. The *Certificate* message can be sent by both the server and the client, but it has a few differences depending on the sender. While the server **MUST** send their certificate, it is optional for the client, who only has to supply a certificate if they receive the *CertificateRequest* message.

Technical Details

The *Certificate* message contains a list of *certificate_request_contexts* and a list of *CertificateEntries*. The *certificate_request_context*'s value depends on whether the certificate was requested or not, and such differs between the client and the server. The *CertificateEntry* contains the needed information in order to process the *CertificateVerify* message that immediately follows the *Certificate* message. The information depends on the chosen *certificate_type*.

```

1 enum {
2     X509(0),
3     RawPublicKey(2),
4     (255)
5 } CertificateType;
```

```

7 struct {
8     select (certificate_type) {
9         case RawPublicKey:
10             /* From RFC 7250 ASN.1_subjectPublicKeyInfo */
11             opaque ASN1_subjectPublicKeyInfo<1..224-1>;
12
13         case X509:
14             opaque cert_data<1..224-1>;
15     };
16     Extension extensions<0..216-1>;
17 } CertificateEntry;
18
19 struct {
20     opaque certificate_request_context<0..28-1>;
21     CertificateEntry certificate_list<0..224-1>;
22 } Certificate;

```

A.8 CertificateVerify

This message is used to provide explicit proof that the endpoint has access to the long-term signing key (the private key). The message also provides integrity for the handshake up to this point. The *CertificateVerify* message contains the algorithm used for signing, and a hash of the entire handshake up to this point concatenated with the certificate sent in the *Certificate* message.

The receiver can then verify that the unsigned certificate is the same as the one that was sent in the *Certificate* message, and that the handshake context is the same.

Technical Details

The structure of the *CertificateVerify* message contains two parts, a *SignatureScheme* which specifies the algorithm used, and the *signature* which is used to prove ownership of the certificate signature.

```

1 struct {
2     SignatureScheme algorithm;
3     opaque signature<0..216-1>;
4 } CertificateVerify;

```

A.9 Finished

Both endpoints must send, receive and verify a *Finished* message. The message is essential to prove authentication of the handshake and the computed keys. This step is also where the key for transmitting application data is generated, and therefore this message is the final message to use the initial encryption for each peer.

Technical Details

The structure of the finished message can be seen in the listing below.

At this point the traffic key (*finished_key*) is computed, and used along with the hash of all previous handshake messages to create a final hash that the peer can verify. This message connects the final key with the entire handshake context, and the peer can verify the hash, by constructing their own and compare.

```

1 finished_key =
2     HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)
3
4 struct {
5     opaque verify_data[Hash.length];
6 } Finished;
7
8 verify_data =
9     HMAC(finished_key,
10         Transcript-Hash(Handshake Context,
11                         Certificate*, CertificateVerify*))
12
13 * Only included if present.
```

A.10 Ack

Since DTLS may be run on an unreliable and unordered transport layer, the only way both parties can be sure that the handshake is successful, is if the retrieval of the final message of the handshake is acknowledged. The *Ack* message contains a list of the record numbers that has been used in the handshake, either processed or buffered. It should be noted

that the Ack is technically not a part of the handshake, but generally it is seen grouped together with the handshake, as it is directly tied to it.

Technical Details

The structure of the *Ack* message only contains one part, that being the *RecordNumber* which is a list of the records containing handshake messages in the current flight which the endpoint has received and either processed or buffered, in numerically increasing order (sequence and epoch numbers).

```
1 struct {  
2     RecordNumber record_numbers<0..216-1>;  
3 } ACK;
```

Appendix B

Validation Lemmas

The lemmas are split into the same categories as 4.3.

B.1 Authentication

We have three lemmas that concern authentication validation. The first lemma disallows authentication after the handshake is finished, whereas the two remaining lemmas concern client authentication, and whether or not this should be provided.

The first lemma "prove_server_auth" seen in lines 1-12, ensures that that in cases where the handshake is at it's end, the server has been authenticated. This authentication uses the handshake history seen in lines 5 and 11.

The two remaining lemmas "client_cert_when_req" seen in lines 14-21 and "no_client_cert_req" seen in lines 23-31, focus on whether or not the client is requested to authenticate or not, and whether or not the client provides authentication in these cases. Since these lemmas passed, it is clear that the client will only send its certificate in a case where it is requested, as expected.

```
1 lemma prove_server_auth:
2   "All C S history session_idC #i #j.
3     Send(C, S, 'CF', session_idC) @ i
4     & CommitAuth(C, S, 'client', history) @ j
5     & not(Ex #p. Corrupt(C) @ p)
6     & not(Ex #q. Corrupt(S) @ q)
```


Lemma Description	Category	Result
When both sides are finished, their handshake history should be the same.	Authentication	Pass
Client Certificate will only be supplied to the server if requested	Authentication	Pass
Client Certificate will not be sent if it is not requested	Authentication	Pass
ServerHello must come after the full cookie exchange	Cookie Exchange	Pass
HelloRetryRequest must come before ClientHello+Cookie, and the cookie must be identical between the two	Cookie Exchange	Pass
ClientHello must come before ClientHello+Cookie, and the contents of each ClientHello must be identical	Cookie Exchange	Pass
Both sides must be finished with the handshake, before the server sends the final acknowledgment	Finished Handshake	Pass
The server must verify itself by sending their certificate, and their certificate verification before the server is finished	Finished Handshake	Pass
All client messages in a given handshake contains the same session_id	Finished Handshake	Pass
All server messages in a given handshake contains the same session_id	Finished Handshake	Pass
The sequence numbers for each epoch increments correctly for the client	Order	Pass
The sequence numbers for each epoch increments correctly for the server	Order	Pass

Table B.1: Table showcasing all validation lemmas.

```

7      ==>
8      (Ex session_idS #k. Send(S, C, 'SF', session_idS) @ k
9      & #k < #i)
10     & (Ex #l. Running(S, C, 'server', history) @ l)"
11
12 lemma client_cert_when_req:
13 "All C S session_idC #i.
14   Send(C, S, 'CC', session_idC) @ i
15   ==>
16   (Ex session_idS #j. CertificateRequested(S, C, 'True',
17     session_idS) @ j & #j < #i)"
18
19 lemma no_client_cert_req:
20 "All C S session_idC #i.
21   Send(C, S, 'CF', session_idC) @ i

```

```

22    & (Ex session_idS #j. CertificateRequested(S, C, 'False',
23    session_idS) @ j & #j < #i)
24    ==>
25    not(Ex #k. Send(C, S, 'CC', session_idC) @ k)"

```

B.2 Cookie exchange

The cookie exchange has three validation lemmas in the model.

The first lemma "SH_after_cookie" in lines 1-8, ensures that the handshake cannot reach the ServerHello message without a cookie exchange. The cookie exchange being complete is marked by the previous existence of a HelloRetryRequest and a ClientHello+Cookie message.

The second lemma "HRR_before_CHC" in lines 10-19, makes sure that a HelloRetryRequest Precedes the ClientHello+Cookie message. Additionally, it checks that the cookie is the same between the two, as the RFC specifies that the server must reject in cases where they differ.

Finally, the lemma "CH_before_CHC" in lines 21-29, ensures that the ClientHello is sent before an eventual ClientHello+Cookie, at which point the content of the messages should be equivalent. This equivalence is again due to the RFC, wherein a server must reject a client if they do not the send exact same message alongside the cookie.

These three lemmas allows the cookie exchange to function as expected when comparing it to the documentation.

```

1  lemma SH_after_cookie:
2  "All C S session_idS #i.
3    Send(S, C, 'SH', session_idS) @ #i
4    ==>
5    (Ex session_idC #j. Send(C, S, 'CHC', session_idC) @ j
6    & #j < #i)
7    & (Ex #k. Send(S, C, 'HRR', session_idS) @ k & #k < #i)"
8
9  lemma HRR_before_CHC:
10 "All C S session_idC #i.
11   Send(C, S, 'CHC', session_idC) @ #i
12   ==>

```

```

13      (Ex session_idS #j. Send(S, C, 'HRR', session_idS) @ j
14      & #j < #i)
15      & (Ex cookie #k #l. HRR_cookie(S, C, cookie) @ k
16      & CHC_cookie(C, S, cookie) @ l)"
17
18 lemma CH_before_CHC:
19 "All C S session_id #i.
20   Send(C, S, 'CHC', session_id) @ #i
21   ==>
22   (Ex #j. Send(C, S, 'CH', session_id) @ j & #j < #i)
23   & (Ex content #k #l. CH_content(content) @ k
24   & CHC_content(content) @ l)"

```

B.3 Finished Handshake

The end of the handshake has four validation lemmas concerning it. The first lemma ensures that both sides has sent their "finished" message before the Server concludes the handshake with the Ack message.

The second lemma claims that if the handshake is finished, the server must be authenticated, as server authentication is required in DTLS. Finally, we have two lemmas that compare the session_id of the finished messages of the client and server respectively, to their earlier messages. This ensures that no steps have been skipped in the handshake.

```

1 lemma CF_and_SF_before_Ack:
2 "All C S session_idS #k .
3   Send(S, C, 'ACK', session_idS) @ k
4   ==>
5   (Ex #i. Send(S, C, 'SF', session_idS) @ i)
6   & (Ex session_idC #j. Send(C, S, 'CF', session_idC) @ j)"
7
8 lemma SC_before_SF_and_after_SH:
9 "All C S session_id #i.
10   Send(S, C, 'SF', session_id) @ i
11   ==>
12   (Ex #j #k. Send(S, C, 'SH', session_id) @ j & #j < #i
13   & Send(S, C, 'C', session_id) @ k & #k < #i & #j < #k)"
14
15 lemma all_client_messages_share_same_session_id:

```

```

16 "All C S session_id #i.
17   Send(C, S, 'CF', session_id) @ #i
18   ==>
19     (Ex #j. Send(C, S, 'CH', session_id) @ j & #j < #i)
20     & (Ex #k. Send(C, S, 'CHC', session_id) @ #k & #k < #i)"
21
22
23 lemma all_server_messages_share_same_session_id:
24 "All C S session_id #i.
25   Send(S, C, 'SF', session_id) @ #i
26   ==>
27     (Ex #n. Send(S, C, 'HRR', session_id) @ #n & #n < #i)
28     & (Ex #j. Send(S, C, 'SH', session_id) @ j & #j < #i)
29     & (Ex #k. Send(S, C, 'EE', session_id) @ #k & #k < #i)
30     & (Ex #l. Send(S, C, 'C', session_id) @ #l & #l < #i)
31     & (Ex #m. Send(S, C, 'CV', session_id) @ #m & #m < #i)"

```

B.4 Order

As mentioned in the specification, DTLS has to be able to reorder messages such that they are received in the intended order. DTLS uses the sequence numbers and epochs to achieve this knowledge of the intended order. To ensure these are utilized correctly, we have created two lemmas, one for the client, and one for the server. These lemmas compare the message sequence numbers, by checking whether or not the previous number in the sequence has been received already.

In the case of a message with the sequence number 2 and epoch 1, the lemma checks for the message with sequence number 1 and epoch 1, and if this message does not exist, then the lemma fails.

```

1 lemma correct_seq_per_epoch_client:
2 "All C %X Epoch session_id #i.
3   C_SeqEpoch(C, session_id, %X %+ 1:nat, Epoch) @ i
4   ==>
5     (Ex #j. C_SeqEpoch(C, session_id, %X, Epoch) @ j
6     & #j < #i) | %X %+ 1:nat = 1:nat"
7
8 lemma correct_seq_per_epoch_server:
9 "All S %X Epoch session_id #i.

```

```
10   S_SeqEpoch(S, session_id, %X %+ 1:nat, Epoch) @ i
11   ==>
12   (Ex #j. S_SeqEpoch(S, session_id, %X, Epoch) @ j
13   & #j < #i) | %X %+ 1:nat = 1:nat"
```

Appendix C

UPPAAL Translation ideas

This section will present the thought process throughout the time taken to translate Tamarin to UPPAAL. To see the final idea see section 6.2.

The flower

From here, we started considering different ways of presenting the model in UPPAAL. The earliest idea was a singular location with a large amount of edges that functioned as the rules in Tamarin. Each edge would act as a rule, and the single location would act as the multiset. As the state changed, so would the multiset. See figure C.1 for a visual representation.

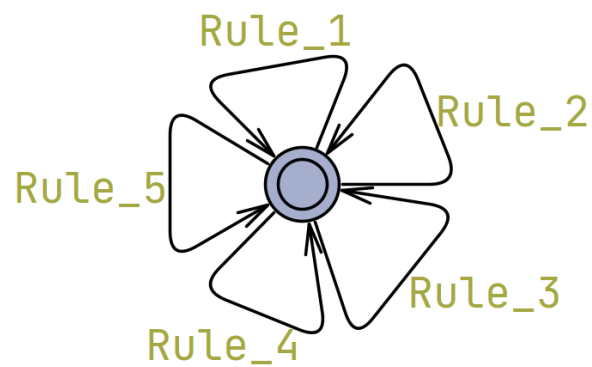


Figure C.1: Example of how a "flower" model would look.

Major concerns with this approach includes storage of this ever growing multiset. Storing the multiset would be limited to the capabilities of UPPAAL variables. Ignoring the potential of multiple locations, edges and processes seemed unwise, and such the idea was expanded.

Client-Server spawning

The next idea relied on the notion that the translation process would only work for Tamarin models that follow a "client-server protocol structure". The idea was to use the "spawning" feature in UPPAAL SMC to create any given amount of client and server processes from a main process.

The name of each rule in Tamarin would be assigned a prefix; *client_* or *server_*. This way rules would be modeled in either the spawned client or server process. Other than this, the approach is similar to the first approach. See figure C.2 for a visual representation.

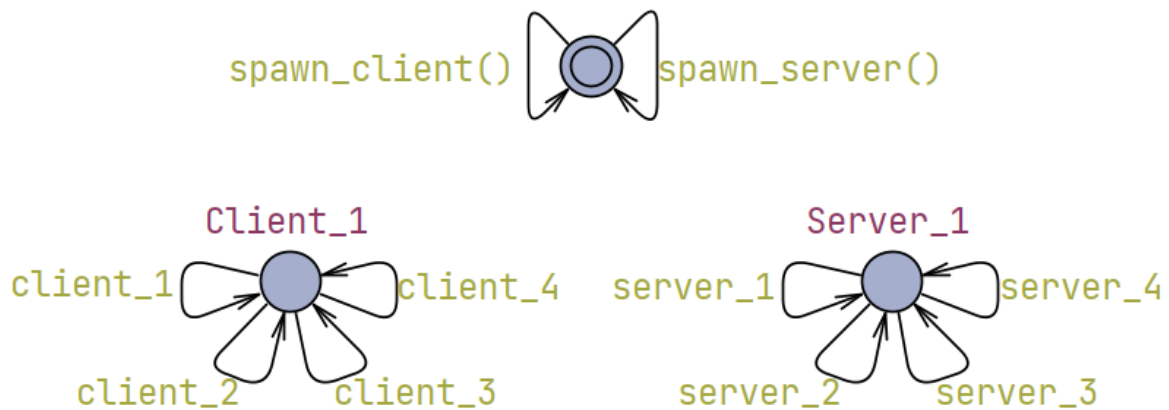


Figure C.2: Example of client-server spawning

This idea still suffers from the downsides of the initial "flower" idea. The information located in a single process would be less bloated, since it would be limited to an individual instance of a client or a server. This idea also sparked the method used in the final draft for sharing variables and facts between processes.

This approach also sparked interest in spawning individual handshakes, however the translation procedure became more and more specialized to work with just a few select protocols types, which was not desired.