SCTP

Strategies to Secure End-To-End Communication

D I S S E R T A T I O N

to obtain the academic grade
doctor rerum naturalium
(Dr. rer. nat.)
in Computer Science

Submitted to the
Faculty of Economics and Business Administration
Institute for Computer Science and Business Information Systems
University of Duisburg-Essen

by
Robin Seggelmann, M.Sc.
born on July 29th, 1982, in Oelde, Germany

Reviewers:

1. Prof. Dr.-Ing. Erwin P. Rathgeb
2. Prof. Dr. Bruno Müller-Clostermann

Date of Disputation: October 22nd, 2012

Abstract

The Stream Control Transmission Protocol (SCTP) is a fairly recent generic transport protocol with novel features, like multi-streaming, multi-homing, and an extendable architecture. This, however, prevents existing approaches to secure end-to-end connections from being used without limiting the supported SCTP features. New solutions also exist, but require extensive modifications that are difficult to realize and deploy. Hence, there is no widely deployed solution to secure SCTP-based connections.

In this thesis, possible strategies to secure end-to-end SCTP connections are analyzed. For each strategy, a viable solution that does not limit the features of SCTP is presented, with a focus on deployability in terms of standardization as well as implementation. Implementations based on common open source tools are developed and used to conduct functionality and performance measurements, with simulated and real systems, to prove the usefulness of the suggested approaches.

Keywords: SCTP, Security, DTLS, SSH, Tunneling

Acknowledgements

This thesis is the result of my work as a research associate in the Network Programming Lab of the Department of Electrical Engineering and Computer Science at the Münster University of Applied Sciences.

First and foremost, I would like to thank Prof. Dr. Erwin P. Rathgeb for his support and supervision of this thesis and his invaluable comments on my work. I also thank Prof. Dr. Bruno Müller-Clostermann for the review of this thesis.

I would like to gratefully acknowledge the enthusiastic support and inspiration of Prof. Dr. Michael Tüxen at the Münster University of Applied Sciences. Furthermore, I am grateful to my colleague Dr. Irene Rüngeler, as well as Dr. Thomas Dreibholz, Martin Becke and Thomas Beermann for their assistance, the technical discussions and motivation.

In addition, I would also like to thank the Deutsche Forschungsgemeinschaft (DFG) for supporting this project.

Contents

1 Introduction **1**
- 1.1 Motivation 2
- 1.2 Goals 3
- 1.3 Organization of this Work 3

2 Network and Transport Protocols **5**
- 2.1 Internet Protocol 5
- 2.2 Transmission Control Protocol 6
- 2.3 User Datagram Protocol 6
- 2.4 Datagram Congestion Control Protocol 6
- 2.5 Stream Control Transmission Protocol 7
  - 2.5.1 Protocol Design 7
  - 2.5.2 Association Setup and Teardown 8
  - 2.5.3 Data Transfer 9
  - 2.5.4 Streams 10
    - 2.5.4.1 Sender Scheduling 10
    - 2.5.4.2 Receiver Scheduling 11
  - 2.5.5 Multi-homing 12
  - 2.5.6 Notifications 12
  - 2.5.7 Extensions 13
- 2.6 Mobility Solutions 14

3 Security Protocols **17**
- 3.1 Internet Protocol Security 17
- 3.2 Transport Layer Security 18
  - 3.2.1 Record Layer 19
  - 3.2.2 Handshake and ChangeCipherSpec Protocol 19
  - 3.2.3 Session Resumption 20
  - 3.2.4 Alert Protocol 21
- 3.3 Datagram Transport Layer Security 22
  - 3.3.1 Record Layer Modifications 22
  - 3.3.2 Handshake Message Modifications 23
  - 3.3.3 Client Verification with Cookies 24

3.3.4 Handshake Reliability 24
3.3.5 Alert Protocol 26
3.4 Secure Shell 26
3.4.1 Transport Layer 26
3.4.2 Authentication Protocol 27
3.4.3 Connection Protocol 28

4 Tools and Software **31**
4.1 Simulation 31
4.1.1 OMNeT++ 32
4.1.2 INET Framework 32
4.2 OpenSSL 33
4.2.1 Architecture 33
4.2.2 Context Objects 34
4.2.3 Session Objects 34
4.2.4 Basic Input/Output Objects 34
4.2.5 DTLS Implementation 35
4.3 OpenSSH 35

5 Securing SCTP-based Applications **37**
5.1 Security-aware Applications 38
5.1.1 Integrated Mechanisms 38
5.1.2 External Mechanisms 38
5.2 Security-agnostic Applications 39
5.2.1 Network Protocol Tunneling 39
5.2.2 Transport Protocol Tunneling 40
5.2.3 Application Protocol Tunneling 41
5.3 Comparison 42
5.4 Existing Security-aware Solutions for SCTP 43
5.4.1 TLS over SCTP 43
5.4.2 DTLS over SCTP 44
5.4.3 Secure SCTP 44
5.4.4 Secure Socket SCTP 44
5.5 Existing Security-agnostic Solutions for SCTP 45
5.5.1 Internet Protocol Security 45
5.5.2 TLS/DTLS-based Tunneling 45
5.6 Conclusion 45

6 SCTP-aware DTLS **47**
6.1 General Considerations 47
6.2 Open Issues and Proposed Solutions 47
6.2.1 Encryption and Authentication 48
6.2.2 Ensuring Order and Reliability 48
6.2.3 Message Loss Prevention 49

6.2.4 Renegotiations 49
6.2.5 Shutdown 50
6.2.6 Session Resumption 51
6.2.7 Generic Adaptations 51
6.2.8 SCTP Extensions 52
6.3 Implementing SCTP-aware DTLS 53
6.3.1 Existing Implementation 53
6.3.2 BIO Object 53
6.3.3 SSL Object 54
6.4 Performance Evaluation 55
6.4.1 Single Core Throughput Measurements 55
6.4.2 Dual Core Throughput Measurements 57
6.4.3 Renegotiations 58
6.5 Optimizations 60
6.5.1 Handshake Message Synchronization 61
6.5.2 Avoiding Duplicate HMACs 62
6.5.3 Allowing Multiple Epochs 62
6.6 Conclusion 64

7 DTLS-based Tunneling 65
7.1 General Considerations 65
7.2 Heartbeat Extension 66
7.3 Path MTU Discovery 67
7.4 Mobility Extension 69
7.4.1 General Considerations 69
7.4.2 Concept 71
7.4.3 Security Considerations 73
7.4.4 ICMP Considerations 74
7.5 Implementing Heartbeats and DTLS Mobility 75
7.6 Evaluation and Measurements 77
7.6.1 Direct Use 77
7.6.2 Tunneling 80
7.7 Conclusion 83

8 SSH Tunneling 85
8.1 SSH Forwarding of SCTP 85
8.1.1 SSH Modifications 85
8.1.2 Limitations 87
8.2 Forwarding over SCTP 87
8.2.1 Usage of Multi-Homing 88
8.2.2 Usage of Multi-Streaming 88
8.2.2.1 Security Considerations 91
8.2.2.2 SCTP Data Chunk Header Encryption 91
8.2.3 Stream Scheduling Considerations 92

8.2.3.1 First-Come, First-Served Scheduling 92
8.2.3.2 Round-Robin Scheduling 92
8.2.3.3 Fair Bandwidth Scheduling 93
8.2.3.4 Priority Scheduling 94
8.2.3.5 Per Packet Scheduling 95
8.2.3.6 Preemptive Scheduling 95
8.2.4 Congestion Control Considerations 95
8.2.5 Flow Control Considerations 96
8.3 Forwarding other Transport Protocols 96
8.3.1 SCTP Forwarding Characteristics 97
8.3.2 UDP Forwarding Characteristics 98
8.3.3 DCCP Forwarding Characteristics 98
8.4 Implementing Forwarding in OMNeT++/INET 98
8.4.1 Stream Scheduling 99
8.4.2 Forwarding Application 99
8.5 Implementing SSH over SCTP 99
8.5.1 Adding Support for SCTP 99
8.5.2 Adding Support for Multi-Streaming 100
8.5.3 Forwarding Other Transport Protocols 100
8.5.4 Pluggable Schedulers in the Kernel 100
8.6 Performance Evaluation 101
8.6.1 Simulation and Real Setup 101
8.6.2 Reliability with Multi-Homing 102
8.6.3 Throughput with Multi-Homing and CMT 104
8.6.4 Delay with Mapping Channels onto Streams 106
8.6.5 Delay with Different Stream Schedulers 107
8.7 Conclusion 109

9 Conclusion **111**
9.1 Achieved Results 111
9.2 Future Work 113

List of Figures **115**

List of Tables **117**

List of Abbreviations **119**

Bibliography **123**

Chapter 1

Introduction

The Internet is ubiquitous nowadays, and security has become a major concern. This is not anymore an issue concerning only applications for which security requirements are obvious, like online banking or confidential business communication. There is a highly dynamic evolution of applications providing a wide variety of services, which attract enormous numbers of users in a very short time, as exemplified by Skype, YouTube, Facebook or Twitter. All of these distributed applications involve massive communication and create many new possibilities for fraud, identity theft and data misuse. Besides securing the applications as such against attacks and securing the data storage, the exchange of potentially critical data also has to be secured. Depending on the application scenario, mechanisms like encryption to prevent eavesdropping, integrity checks to prevent tampering with the exchanged data and authentication to confirm the identity of each involved party have to be provided. One obvious way to provide the appropriate level of security is to integrate the required security functionality directly into the application design. However, in a situation where everyone can create and easily deploy applications on a global scale, it cannot be assumed that all application programmers have the highly specific knowledge to ensure secure operation and communication of their applications – apart from the effort required on an application per application basis. Therefore, mechanisms have to be provided that offload the task of securing end-to-end communication from the application developers. Considering the layered protocol architectures used today, a variety of strategies to do this exist and are actually deployed in today's networks.

The Stream Control Transmission Protocol (SCTP) is a recent transport protocol standardized by the Internet Engineering Task Force (IETF) in 2000 and is currently described in RFC 4960 [96]. Initially, it has been developed to migrate telephone signaling to Internet Protocol (IP) [69] based networks, but evolved into a general-purpose transport protocol during its development process. As an alternative to the established transport protocols, Transmission Control Protocol (TCP) [70] and User Datagram Protocol (UDP) [67], SCTP offers more features than both of these combined, since its design allows it to be extended easily, without modifying the base specification. Implementations of SCTP are available for all major operating systems by now.

Although security and privacy being such an important issue, there is no actually deployed solution for SCTP yet.

1.1 Motivation

The need for security solutions for SCTP is urgent, because insecure data transmission, over the Internet in particular, is not acceptable anymore. The main issue, however, is that there are many different scenarios in which SCTP-based applications can be used, which cannot be covered all by a single strategy.

The first scenario in which SCTP was used, and for which it has been developed initially, is the telephone signaling network with the Signaling System No. 7 (SS7) [20] protocols. These protocols have been developed for proprietary networks, but have been ported to SCTP for the use in IP-based networks. Since their original area of deployment were closed networks with proprietary hardware, no security features have been considered.

Reliable Server Pooling (RSerPool) [17] is an architecture to manage a pool of servers, to increase the reliability or available performance with distributed computing. The protocols to manage the available servers and to assign tasks to them are based on SCTP. Security has been an issue during the design, so appropriate features can be used with the protocols directly, when available. The same is true for IP Flow Information Export (IPFIX) [103], which is used to export flow information from routers and similar devices, which is used for billing for instance.

Another scenario is to provide encryption and authentication to single connections, for example to grant access to an internal network from a remote location securely, without the need to expose the services to the Internet. This can be realized for TCP with Secure Shell (SSH), but no solution is available for SCTP.

The available security solutions for TCP, for example, have been developed before SCTP had been introduced and thus do not take its specific characteristics into account. Other approaches proposed until today also have drawbacks, which limit the features of SCTP that can be used, or are difficult to realize and deploy. Additionally, the existing solutions are only suitable for applications, for which security has been considered in their design, and thus are able to use these features. Such applications are called security-aware in the following, while applications for which security was not considered at all, are called security-agnostic. To prevent any attacker from eavesdropping or tampering with the data of security-agnostic applications, some kind of tunneling is required. Tunneling means to encapsulate the application data, alone or with underlying protocols in a security protocol. The encapsulated data is then transmitted and unpacked at its destination, instead of being sent directly. Unfortunately, existing tunneling solutions have not been developed with SCTP in mind either, and thus are not supporting it or only with limitations.

1.2 Goals

The existing suggestions to secure SCTP all have serious drawbacks, which prevented them from being widely deployed so far. They either limit the number of features of SCTP that can be used, for instance to only those also available with TCP, or are unlikely to be realized and deployed, because they require extensive modifications, for instance of the SCTP implementation in the operating system kernel. Additionally, all existing proposals assume that the applications are security-aware, that is implement a mechanism to protect their connections. All other applications, for which security has not been considered in their design, are not regarded and thus there is no approach how to tunnel single connections. The only available option to provide security for those applications is to set up a generic tunnel on or below the IP layer, which is called a Virtual Private Network (VPN).

Therefore, the main goal of this work is to suggest new solutions for securing both kinds of applications using SCTP, by collecting generic requirements and examining the drawbacks of already available solutions. In the process, the focus is on easy deployability, while still supporting all features of SCTP. A further goal is to optimize the suggested solutions to achieve an equal or even better performance than the commonly used options for securing TCP. The highest potential has the tunneling for generic applications, since there are multiple approaches possible, one even being to use SCTP itself to realize a tunnel and thus making use of its extended features.

To verify feasibility and functionality and to measure the performance, implementations are necessary. These can then be tested either in a simulation environment or with a real setup, depending on what is to be measured. In this way it is ensured that the proposed solutions can actually be used, and also have an adequate performance compared to TCP. As a result, applications using SCTP can be secured without limitations or performance issues.

1.3 Organization of this Work

The next Chapters 2 and 3 provide an introduction to SCTP and other common transport and security protocols, respectively. The real and simulated implementations were realized with open source tools, described in Chapter 4. Chapter 5 discusses the general considerations on securing SCTP-based applications, as well as the motivation for the solutions proposed subsequently. Security-aware applications are addressed and a new approach is suggested in Chapter 6. Subsequently, Chapters 7 and 8 are concerned with security-agnostic applications and propose two different tunneling solutions. The conclusion and an outlook on future work are given in Chapter 9.

Chapter 2

Network and Transport Protocols

The Internet model [35] provides an abstraction of the communication system and is structured into logical layers as shown in Figure 2.1. Each layer provides a certain service to the layer above, which is provided by using the capabilities of a specific protocol. Common protocols for the network and transport layer will be described in the following sections.

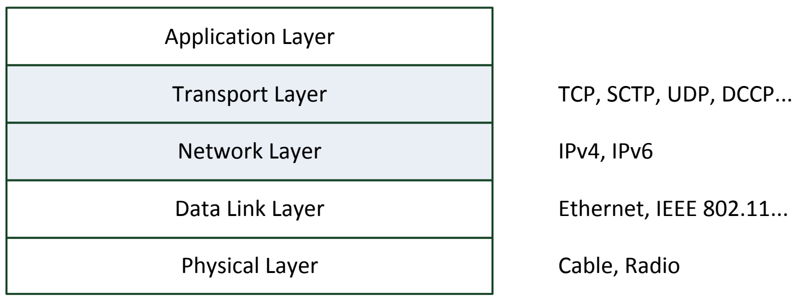


Figure 2.1: The Internet Model

2.1 Internet Protocol

The most widely deployed network protocol is the Internet Protocol Version 4 (IP, IPv4) [69], which is gradually superseded by Version 6 (IPv6) [12]. The purpose of the protocol is the routing of data packets between two hosts, which are identified by unique source and destination addresses. Version 4 of the protocol supports 32 bit addresses, which equals about 4.3 billion possible values. All addresses have already been assigned until January 2011, so the successor IPv6 features 128 bit addresses, with about $3.4 * 10^{38}$ possible values to solve this issue. Packets have an identifier displaying their type of data. The service provided by IP is a simple, unreliable message delivery. Fragmentation is possible for large messages and the boundaries are preserved. IP provides no reliability mechanisms, so messages may get lost, duplicated, reordered, or even corrupted.

2.2 Transmission Control Protocol

A reliable connection can be provided by the transport protocol Transmission Control Protocol (TCP) [70]. A three-way handshake is used to establish a connection between two peers. This is realized with flags, a packet with the SYN flag set initiates a new connection, which is responded with a packet having the SYN flag set, and also the ACK flag to acknowledge the initial packet. This is again acknowledged and the connection is established. Port numbers are used to differentiate multiple connections between the same hosts and a checksum is used to detect corrupt packets. The data transfer is bytestream-oriented, so there are no message boundaries, and received data is acknowledged. Lost packets are repeated until acknowledged, or the sender has given up after too many unsuccessful attempts and terminated the connection. TCP also restores the order in which the data has been sent, in case it was reordered. Furthermore, a Flow Control [95] prevents overloading the receiver by limiting the amount of data that is allowed to be sent. Network overloading is also prevented with a Congestion Control [108] and Explicit Congestion Notification (ECN) [72], which reduce the throughput when packets are lost. A detailed introduction to the protocol can be found in [27].

2.3 User Datagram Protocol

The User Datagram Protocol (UDP) [67] is a connectionless and unreliable transport protocol. Hence, data can be sent to arbitrary destinations at any time, even to all hosts on the same network segment at once, which is called broadcasting. Like TCP, port numbers are also used to allow multiple data transfers between the same hosts, and a checksum detects transmission errors. It is message-based, so message boundaries are preserved during transfer, but fragmentation is not available. There is no protection against overloading either the network or the receiver. Lost messages are not repeated and the order is not retained.

2.4 Datagram Congestion Control Protocol

Also an unreliable and message-oriented transport protocol is the Datagram Congestion Control Protocol (DCCP) [45]. Contrary to UDP, a reliable handshake is performed to establish a connection between two hosts. DCCP also supports multiple Congestion Control algorithms and ECN to prevent network overloading. This requires the acknowledgement of received messages, but there is no retransmission for lost ones.

2.5 Stream Control Transmission Protocol

Although initially developed for telephone signaling, the Stream Control Transmission Protocol (SCTP) had evolved to a generic purpose transport protocol when its first specification [100] has been released in 2000 by the Internet Engineering Task Force (IETF). Seven years later, the current revised specification, RFC 4960 [96], has been published. SCTP is reliable and connection-oriented like TCP, but message-oriented and features multi-streaming as well as multi-homing. With multi-streaming, multiple unidirectional channels can be used within a connection. With multi-homing, multiple addresses can be used for a single connection to allow a failover for increased reliability. Furthermore, the protocol design is extendable to allow adding new features without changing the base specification. A detailed introduction to the protocol has been given in [102] and more recently also in [19].

2.5.1 Protocol Design

Every SCTP packet starts with a Common Header, as shown in Figure 2.2. This header consists of the source and destination port numbers, a 32 bit Verification Tag, and a Checksum to detect corrupted packets. The Verification Tag is a random value, which is unique per direction and exchanged during the connection establishment. The tag chosen for each direction has to be used for all packets sent during the connection lifetime. This mitigates the risk of blind attacks, where an attacker guesses the port numbers used for a connection and tries to insert packets. With the Verification Tag, this has to be guessed as well.

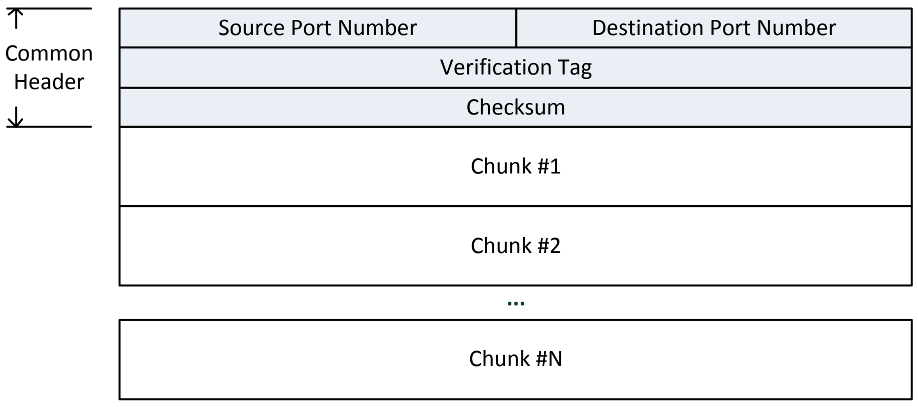


Figure 2.2: SCTP Packet Format with Common Header and Chunks

The Common Header is followed by an arbitrary number of chunks, as illustrated in Figure 2.2. A chunk can either be a control chunk or contain user data. Control chunks are used to manage the connection, called “association” in SCTP. Hence, there are different chunk types for setup and teardown of the association, keep-alive, and error handling. The information within chunks is usually stored in Type-Length-Value (TLV) parameters. This design allows extending the protocol easily by defining new chunks and parameters.

2.5.2 Association Setup and Teardown

SCTP uses a four-way handshake to establish a new association, which is shown in Figure 2.3. Either endpoint can initiate the handshake with an INIT chunk, which is answered with an INIT-ACK chunk. These chunks exchange the Verification Tags, additional IP addresses or hostnames for multi-homing, the number of streams that can be used, and supported extensions. The INIT-ACK chunk also contains a cookie, which consists of all the information necessary for this association. This cookie has to be returned unmodified in a COOKIE-ECHO chunk. By using this additional exchange during the handshake, the receiver of an INIT chunk does not need to allocate any resources for the new association immediately, it just stores all necessary information in the cookie. This is contrary to TCP, which has to allocate resources for each SYN packet immediately. A malicious user can therefore send many faked SYN packets with different source addresses to a host until its resources are exhausted, thus performing a Denial-of-Service (DOS) attack [64]. The attacker

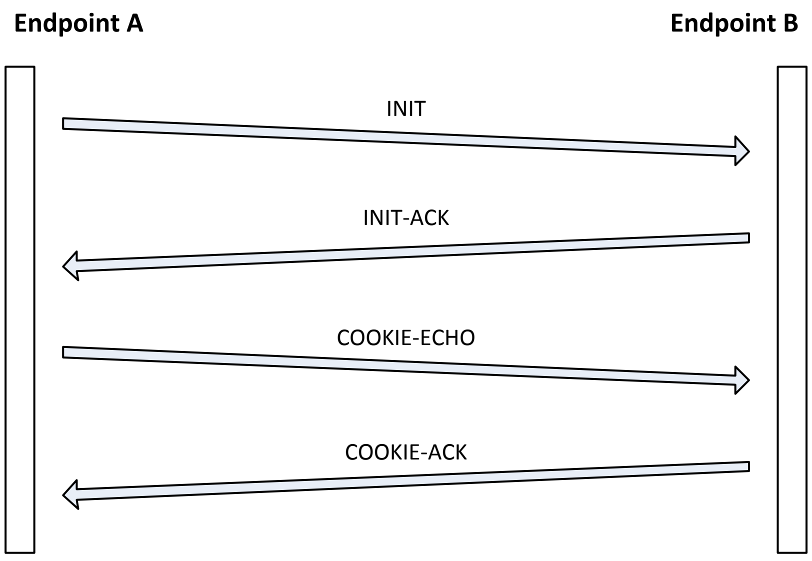


Figure 2.3: SCTP Association Establishment

also has to store resources for the association to be able to return the cookie and cannot use faked addresses, so this attack is not possible anymore. To avoid attacks based on cookie manipulations, the cookie should also contain a signature calculated over the association data and a secret key. Before processing a received cookie, the signature can be verified to ensure the integrity. If the verification has been successful, the handshake is concluded with a COOKIE-ACK chunk.

Since SCTP is connection-oriented and reliable, a method to gracefully teardown an association is necessary, which also has to ensure that all outstanding data has been received. The message flow is depicted in Figure 2.4. When one of the endpoints wants to end the association, no more data is accepted from the application, and after all still outstanding data has been acknowledged, a SHUTDOWN chunk is sent. After receiving this chunk, the other endpoint also stops accepting any data from the application and awaits the acknowledgement of all still outstanding data. Then, a SHUTDOWN-ACK is sent, which is acknowledged with a SHUTDOWN-COMPLETE and the association is closed.

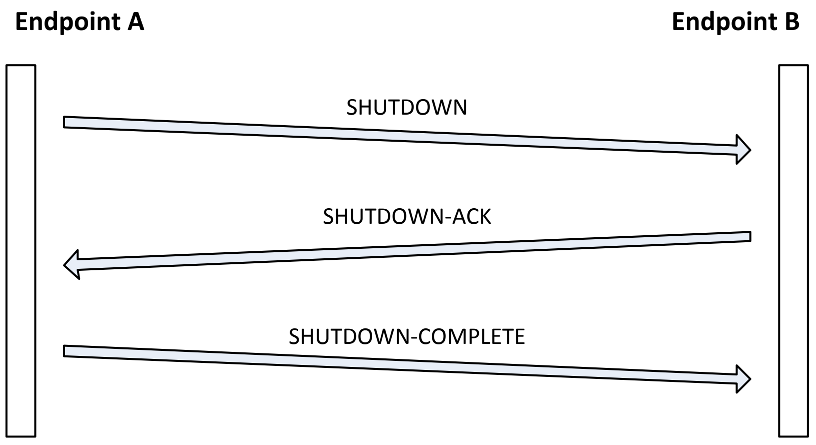


Figure 2.4: SCTP Association Teardown

2.5.3 Data Transfer

The data transfer of SCTP is reliable, therefore the successful reception is acknowledged with selective acknowledgment chunks (SACK). Every DATA chunk containing user data, illustrated in Figure 2.5, has a Transmission Sequence Number (TSN) assigned, which is used for the acknowledgement. The SACK chunks not only report the highest TSN of continuous data, but also the TSNs of out-of-order data chunks that have already been received in so-called gap reports. Selective acknowledgments have been introduced as an option for TCP in [49] to reduce the number of retransmissions. The space available for TCP options is limited, so only

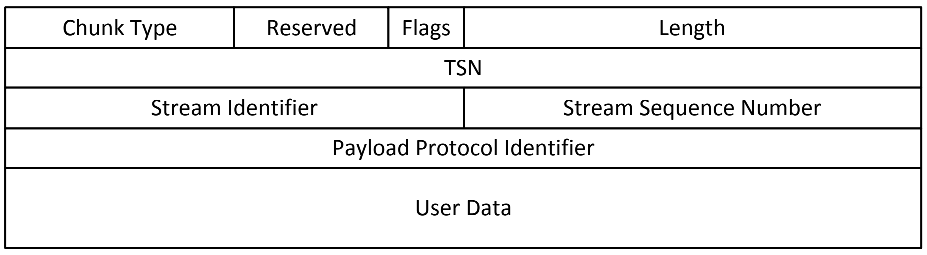


Figure 2.5: SCTP Data Chunk Format

very few out-of-order packets can be acknowledged. SCTP, on the other hand, supports an arbitrary number of gap reports until an entire packet is filled, because this feature has been integrated from the very beginning. Due to the message-orientation of SCTP, every user message is sent in its own DATA chunk, if possible. Large messages that exceed the maximum packet size will be fragmented and sent with multiple DATA chunks, each in its own packet. The receiver reassembles the message using the TSNs to restore the order. Multiple smaller messages can be sent in a single packet, which is called bundling, to reduce the otherwise necessary overhead.

2.5.4 Streams

A distinctive feature of SCTP is multi-streaming, hence its name. Streams are unidirectional logical channels within the data transfer. The user can assign a message to a stream by setting the Stream Identifier (SID). The SID is 16 bit, so the possible number of streams is 2^{16} or 65,536. The order of the messages is only retained within a single stream, so when a message is lost, only the following messages of the same stream have to be delayed until a retransmission has been received. Otherwise, the original order of the messages could not be restored. Without streams, all messages have to be delayed, which is a common issue with TCP and called head-of-line blocking. A Stream Sequence Number (SSN), which is maintained per stream and increased for every message, is used to restore the order of the messages, if necessary. The user can, however, choose to leave messages unordered even within a stream, in which case the SSN is ignored and usually set to 0.

2.5.4.1 Sender Scheduling

After messages have been assigned to different streams, all of them still have to be transmitted over a single association. This requires a scheduler, which decides on the order in which messages of different stream queues are sent, as depicted in Figure 2.6. The specification of SCTP does not state any requirement for such

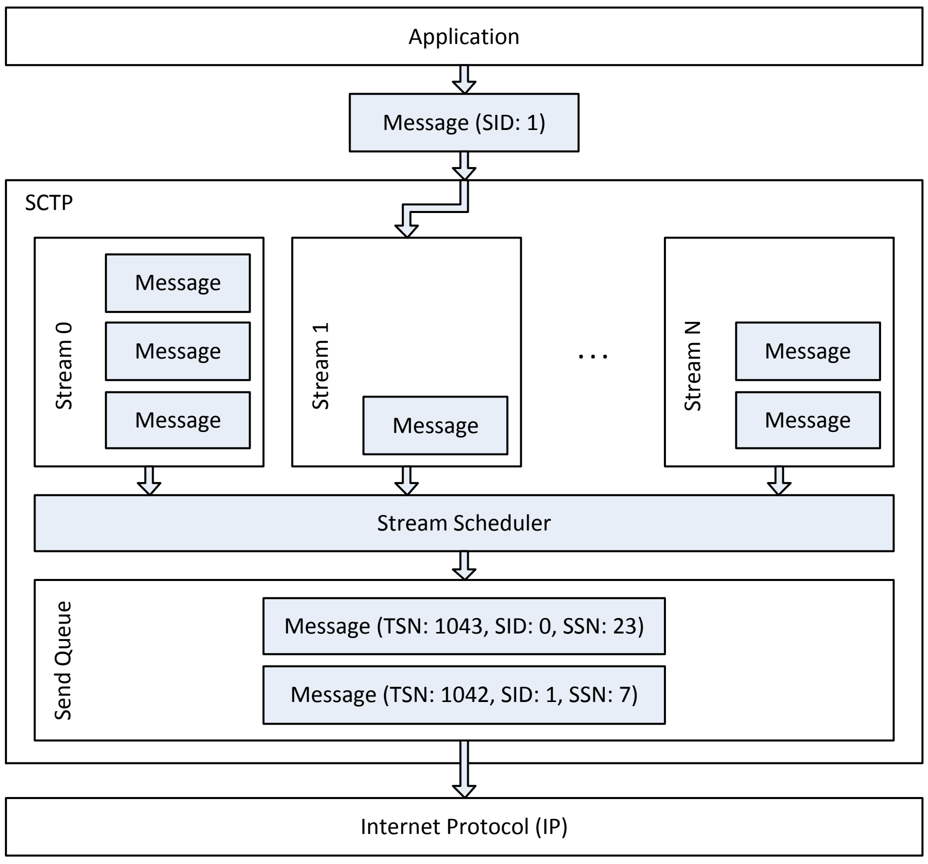


Figure 2.6: SCTP Sender Stream Scheduling

a scheduler; therefore it is up to the implementation to choose an appropriate approach. Commonly used algorithms are first come, first-served (Linux, Solaris) or round-robin (FreeBSD, Mac OS X). However, since the order in which the messages are sent can affect the behavior on the wire, specialized scheduling may be a benefit in certain scenarios. This will be discussed in further detail in Section 8.2.3.

2.5.4.2 Receiver Scheduling

After messages have been received and added to the appropriate stream queue, a scheduler is again necessary to decide on the order in which the messages are passed to the application. This is shown in Figure 2.7. Contrary to the scheduling before sending, which influences the message order on the network, the received messages are passed to the application immediately. Therefore, different scheduling algorithms have little to no effect, so all implementations just use first-come, first-served here.

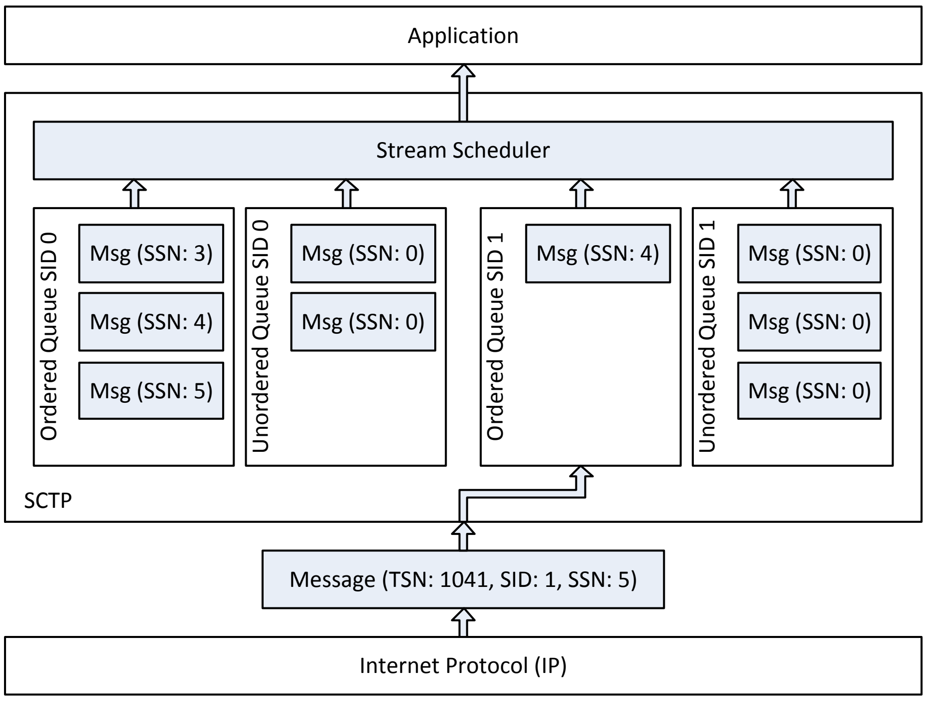


Figure 2.7: SCTP Receiver Stream Scheduling

2.5.5 Multi-homing

SCTP supports multi-homing, that is multiple addresses can be used for a single association. Each pair of addresses of both peers represents a path usable for the association, as shown in Figure 2.8. Only one of the paths is actually used for data transfer, and is called primary path, while the others remain idle. Every destination address is frequently tested for its availability with HEARTBEAT chunks. In case the primary path is not available anymore, one of the idle paths will become the new primary path as a failover mechanism, and the association remains alive.

2.5.6 Notifications

The Application Programming Interface (API) of SCTP supports so-called notifications to communicate information to the user about events or errors that occurred. The SCTP sockets API specification [99] lists various events that the user can subscribe to receive the corresponding notifications. This can be for example SCTP_ASSOC_CHANGE, which invokes a notification every time an association has been set up or closed, or SCTP_REMOTE_ERROR, which gives more detailed information in case an error occurs. The notifications are passed to the applica-

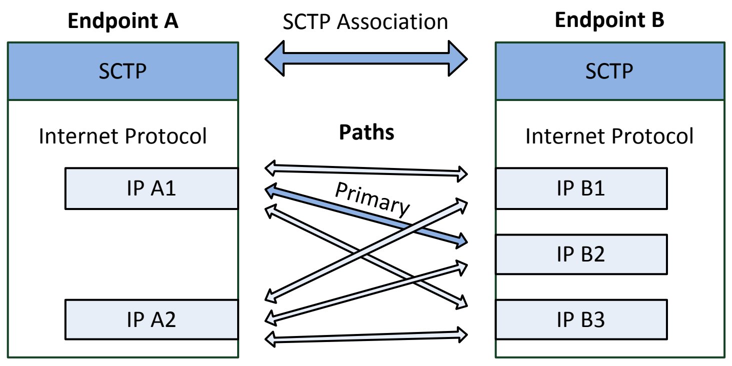


Figure 2.8: SCTP Multi-homing

tion through the `recvmsg()` call, but with the MSG_NOTIFICATION flag set, to distinguish them from regular messages.

2.5.7 Extensions

The extensible protocol design allows adding new features to SCTP easily. Various extensions have been standardized and implemented already. The most notable are PR-SCTP, SCTP-AUTH, ADD-IP and CMT.

SCTP provides reliable data transfer, but with the Partial Reliability (PR-SCTP) extension [98], this can be limited. The extension allows defining policies for when messages may be discarded, such as a maximum number of retransmissions or a maximum lifetime. The new chunk FORWARD-TSN is used to notify the receiver to ignore the absence of discarded messages.

The Authenticated Chunks for SCTP (SCTP-AUTH) extension [106] adds a new parameter to the INIT and INIT-ACK, with which each peer can announce that it will only accept certain types of chunks with protected integrity. This is realized by adding a new chunk carrying a Hash-based Message Authentication Code (HMAC) [3] that has been calculated over all the chunks following it in the same packet. The shared key necessary for the HMAC is either calculated with random numbers also added to the INIT and INIT-ACK, or can be set by the user.

Additional addresses can be added or removed dynamically after the association has already been established with the Dynamic Address Reconfiguration (ADD-IP) extension [101]. Additionally, the user can change the primary path, which is usually chosen by SCTP automatically. To prevent attacks that for example remove all valid addresses or the primary path is changed to hijack the connection, the ASCONF and ASCONF-ACK chunks introduced with this extension have to be protected with SCTP-AUTH.

Although SCTP supports multiple paths per association, only one is used for data transfer. The Concurrent Multipath Transfer (CMT) extension [36] allows using all existing paths simultaneously to maximize the possible throughput. The extension has not yet been standardized since research is still in progress, but the benefit of using multiple paths simultaneously has already been proven [18].

2.6 Mobility Solutions

With the increasing popularity of mobile Internet access, the necessity for mobile connections becomes more and more urgent. A smartphone user, for instance, frequently has access to multiple different networks, since he can use local Wi-Fi networks and the cell phone network in between. Every time he connects to another network, the IP address of his device changes and until today this causes the loss of all established data connections until a reconnect is done with the new address. Several approaches to provide mobility have already been proposed and some even standardized. The most important ones will be introduced in this section.

Virtually all data services offered to end-users are based on the Internet Protocol, so making it mobile seems natural. Mobile IP [65] as well as Mobile IPv6 [38] have already been defined for this purpose. It is assumed that a host, called Mobile Node, has a permanent address in its home network, and different ones in foreign networks. Therefore, a Home Agent in the home network keeps track of the current foreign address of the host that is provided by a Foreign Agent. The Mobile Node announces every new address, called Care-of Address, to its Home Agent, which will provide a tunnel to it. So, whenever the Mobile Node is not in its home network, the Home Agent will forward any data sent to the home address. With IPv4 the host will tunnel its response back through the home agent, while IPv6 allows continuing a direct communication with the sender. Similar to Mobile IP is the MSOCKS [47] approach using transport layer mobility. It uses a TCP proxy in the home network to relay the data to mobile hosts. The proxy accepts incoming connections, connects to the mobile host and forwards the data transparently. This is done by modifying the TCP packets. The forwarding connection is combined with the original one, so they appear as a single ongoing connection. Unfortunately, the deployment of these solutions needs not only support on the mobile device but also infrastructure support. A Home Agent or proxy in the home network has to be set up and has to be available at any time. Common middleboxes, such as firewalls or routers with NAT, have to allow the mobile host's connection attempt to the home agent for address announcement and have to pass through connections forwarded by the home agent. Hence, because of these issues neither solution has been widely deployed.

TCP Redirection (TCP-R) [28] realizes mobility by using new TCP options to notify and authenticate an address change. When the mobile host detects an address change, it notifies the peer, which requests an authentication. A drawback of this approach is that the mobile devices need an operating system with the modified

TCP implementation. NAT is also an issue because the mobile device may not notice that its public address changed. The address change is done without a SYN flag set, which is necessary to enable NAT for TCP and to open a firewall port for the connection. Also based on the transport layer is SCTP Mobility [80], using SCTP's multi-homing features with the ADD-IP extension. This allows reconfiguring the multiple paths for an established association, if necessary. When the host gets a new address after joining a new network, it simply adds the new address as a new possible path for its connections, while addresses that are not used anymore will be removed. Unfortunately, there is no SCTP support on common mobile devices yet. Furthermore, middleboxes performing NAT still cause issues with SCTP's multi-homing, because they can modify the addresses in the IP header, but not those in the INIT, INIT-ACK, and ASCONF (ADD-IP) chunks. This would require a modified NAT implementation that is able to modify these chunks, and also calculate a new checksum after doing so. Multipath-TCP (MP-TCP) [25] adds multi-homing features to TCP. To handle NATs, the connection is set up over a single path first and then additional paths are added. This is similar to SCTP's ADD-IP extension and therefore can be used for mobility in the same way. This concept is, however, still experimental and not fully specified yet.

Chapter 3

Security Protocols

In this chapter, several common security solutions are introduced. All of them are standardized, but use different approaches to secure end-to-end connections. The support of security features can either be included in an application, or added externally by using secure tunneling for the application protocol. Most security solutions do not rely on specific algorithms for key exchange, encryption and hash computation, but rather support arbitrary algorithms. This allows to add more secure algorithms and abandon weak ones without changing the protocol. A combination of key exchange, encryption and hash algorithms is called a cipher suite. A common cipher suite, for instance, is an RSA key exchange [81], encryption with AES-256 [24], and SHA-1 [23] for hashes. Hence, the protocols can be discussed independently of the algorithms.

3.1 Internet Protocol Security

Internet Protocol Security (IPsec) [43] is a security suite for the Internet Protocol (IP) providing authentication and encryption features. It consists of two protocols, the Authentication Header (AH) [41] and Encapsulating Security Payload (ESP) [42], which extend IP packets. The key used for the security features can for example be pre-shared or negotiated with the Internet Key Exchange (IKE) protocol [40]. The Authentication Header ensures integrity and allows authentication for the payload and header of IP packets. A sliding window for sequence numbers is also maintained, so old or duplicated packets can be discarded to prevent an attacker from eavesdropping and resending packets, called replay attacks. Confidentiality with encryption, and optionally also authentication and integrity for the payload can be achieved with Encapsulating Security Payload. Yet the ESP does not secure the IP header.

The protocol architecture does not explicitly specify the algorithms that have to be used for key exchange, encryption and hash calculation, but provides a framework for arbitrary cipher suites. When establishing a connection, both peers exchange their supported algorithms to negotiate a mutually supported one. IPsec

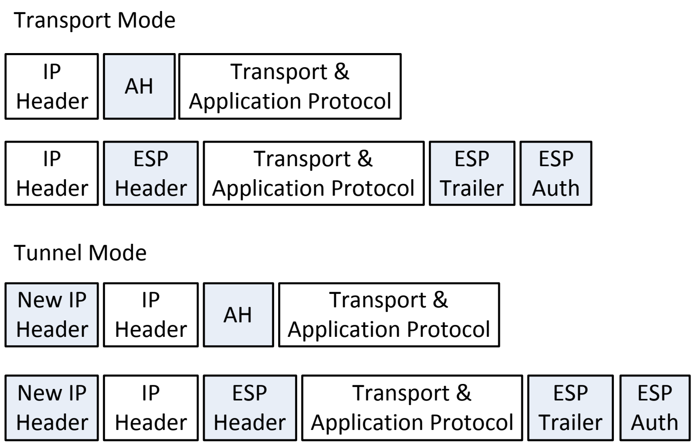


Figure 3.1: IPsec Transport and Tunnel Modes

maintains a Security Policy Database (SPD) to handle established connections, also called Security Associations (SA).

Two different modes are available for each association, transport and tunnel mode, which are illustrated in Figure 3.1. In transport mode, the original IP packet is extended with AH and ESP, but encryption is limited to the payload. The tunnel mode can be used to encrypt the IP header as well. In this mode the entire original IP packet is sent as the payload of a new IP packet with IPsec features. This mode also allows realizing a VPN, because the source and destination addresses in the original and the new IP header do not have to be identical. Hence, a SA in tunnel mode can be used to transport IP packets between two different network segments, for example over the Internet. A comprehensive introduction to IPsec is given in [16].

3.2 Transport Layer Security

Transport Layer Security (TLS) [14] is a security suite that adds an additional layer between the transport and application protocol to provide authentication and encryption for the application protocol. It has been designed for reliable transport protocols, so it requires the transport layer to retain the message order and retransmit lost messages. These requirements are met by TCP, and by limiting its usable features also by SCTP. A TLS connection between two hosts is called session. A session can remain alive over multiple actual connections, because it can be resumed to avoid a new full handshake.

3.2.1 Record Layer

The Record Layer is the basis for application data exchange and also for three sub-protocols, the Handshake, ChangeCipherSpec and Alert protocol, illustrated in Figure 3.2.

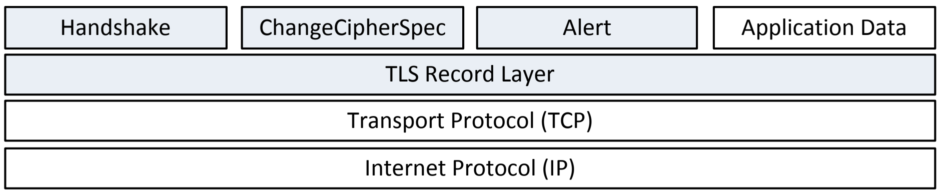


Figure 3.2: TLS Protocol Structure

The header of the Record Layer consists of the Content Type that indicates which sub-protocol it is carrying, the Protocol Version and its Length, depicted in Figure 3.3. It retains message boundaries, in case the transport layer does not. Each Record message has a unique sequence number that is increased with every message sent. This number is not transmitted; each peer maintains it internally instead. Nonetheless, it is used for the calculation of the HMAC, which is used to ensure the integrity of the message. If a received message does not have the expected sequence number, the hash cannot be verified and the connection is dropped. This is done because the transport protocol is expected to be reliable, so the only case where messages may be reordered is after manipulation by an attacker.

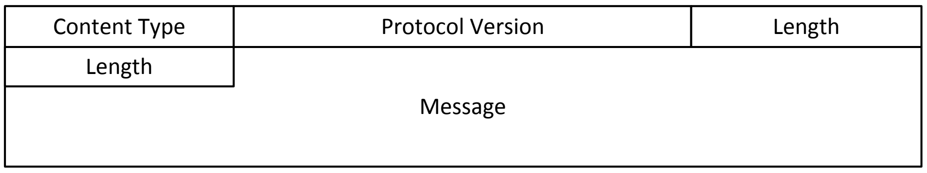


Figure 3.3: TLS Record Header

3.2.2 Handshake and ChangeCipherSpec Protocol

To set up a new connection and negotiate the security parameters, like cipher suite or compression algorithm, the Handshake protocol is used. The message format is depicted in Figure 3.4, and Figure 3.5 shows the message sequence. The client initiates the handshake by sending a ClientHello message to the server. This message contains the supported cipher suites, compression algorithms and a random number. The server is supposed to respond with a ServerHello, which contains the

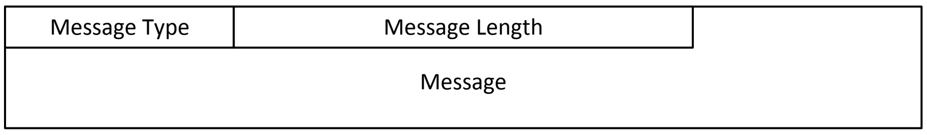


Figure 3.4: TLS Handshake Message Header

cipher suite and algorithms the server has chosen from the ones the client offered and also a random number. Both random numbers will be used, among other data, to calculate the master secret. The server may continue with a ServerCertificate with its certificates to authenticate itself, if necessary. In that case it can also send a CertificateRequest, to provoke the client to authenticate, too. For some cipher suites, additional data is necessary for the calculation of the secret, which can be sent with a ServerKeyExchange. Since the last three messages mentioned are optional, a ServerHelloDone indicates when no more messages follow from the server.

After the ServerHelloDone, the client has to send its certificates with a ClientCertificate if the server requested authentication. This is followed by a ClientKeyExchange that contains its public key or other cryptographic data, depending on the cipher suite used. Also depending on the cipher suite is whether a CertificateVerify to verify a signed certificate has to be sent. At this point both peers have enough information to calculate the master secret. Thus, the client sends the ChangeCipherSpec, to announce that the negotiated parameters and the secret will be used from now on. Its last message is the Finished that contains a hash calculated over the entire handshake and is encrypted already. If the Finished can be decrypted, the negotiation of the cipher suite and shared key was successful. The included hash value is used to verify the integrity of the handshake, so potential manipulations of the messages by an attacker can be detected. If the decryption and verification fails, the connection must be terminated. The server concludes the handshake by also sending the ChangeCipherSpec and the Finished.

3.2.3 Session Resumption

TLS supports session resumption, that is using the already negotiated parameters from an earlier connection to avoid a costly full negotiation. The ClientHello of every new session contains the session identifier 0, which causes the server to assign a new one and communicate it with the ServerHello. If the client still has the information of the previous session, it may reuse the identifier in its ClientHello. If the server still recognizes the client's session identifier, an abbreviated handshake, as shown in Figure 3.6, can be performed and the cipher suite parameters of the former connection are used further on without full negotiation, but with the new random numbers of the ClientHello and ServerHello, respectively. So no more messages are

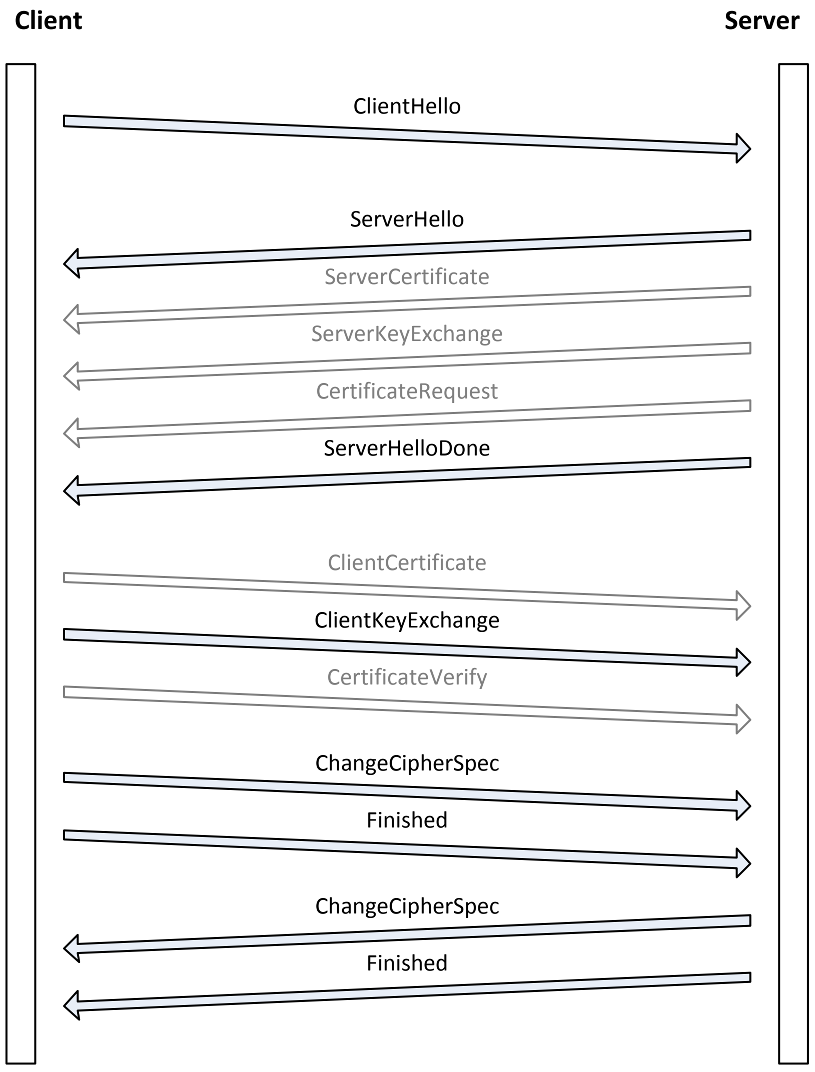


Figure 3.5: TLS Handshake

necessary, and the server sends the ChangeCipherSpec and a Finished immediately after the ServerHello. The client also sends its ChangeCipherSpec and Finished to conclude the handshake.

3.2.4 Alert Protocol

The Alert protocol is used to exchange notifications on warnings or errors that might have occurred, for example when a certificate could not be verified. While errors are always fatal and lead to the immediate shutdown of the connection, warnings are informational and the connection can remain established. Additionally, alert messages are also used to gracefully shut down the connection. When a peer has nothing to send anymore, it should send a CloseNotify alert. The connection is closed after both peers have sent it.

3.3 Datagram Transport Layer Security

Since TLS is limited to reliable protocols, Datagram Transport Layer Security (DTLS) [78] has been developed to secure unreliable protocols, such as UDP or DCCP. It is a modification of TLS developed with the intention to make as few changes as possible. The main issues with TLS over an unreliable transport protocol are reordering and loss of messages, because it relies on messages arriving reliable and in sequence and otherwise assumes an error or attack and simply drops the connection.

3.3.1 Record Layer Modifications

The DTLS Record Layer is basically the same as the one of TLS, but had to be extended to tolerate message loss and reordering. Figure 3.7 illustrates the extended header. The sequence number used for the HMAC calculation is added to the record header to allow the verification of the integrity independent of the order

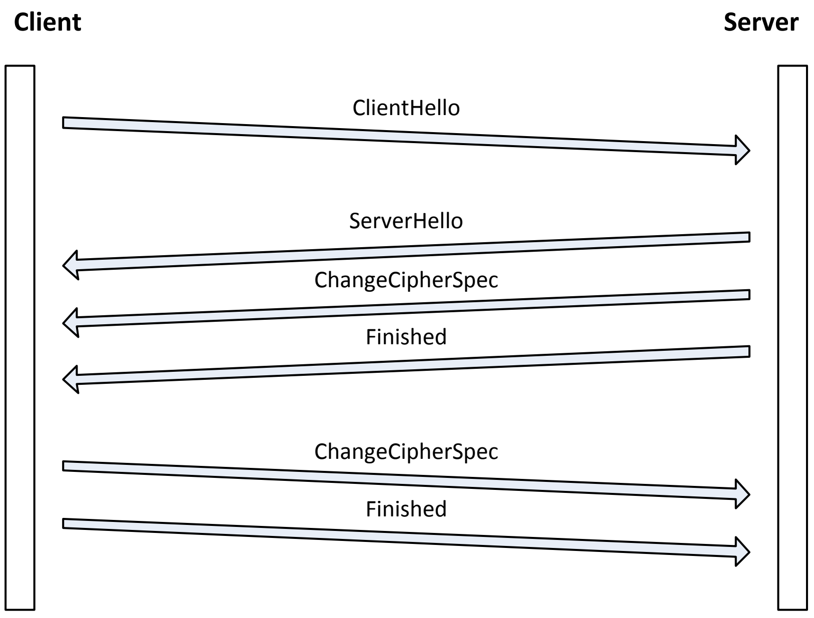


Figure 3.6: TLS Handshake for Session Resumption

in which the message has been received. The sequence numbers are reset after every successful handshake, so different records with the same sequence number may arrive. To distinguish identical numbers, an Epoch is also added, which is increased with every successful handshake and also used for the calculation of the HMAC.

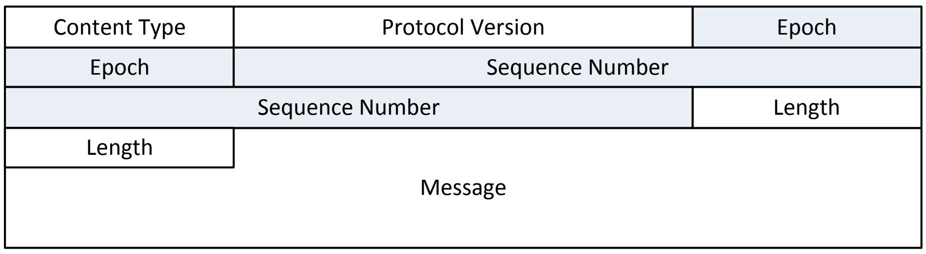


Figure 3.7: DTLS Record Header

To prevent a replay attack, where an attacker intercepts and resends valid messages, for example in attempt to reissue a command, a replay check is also done. A window of acceptable sequence numbers is maintained and every sequence number that is outside the window or has been marked as received within the window will be discarded.

3.3.2 Handshake Message Modifications

Another issue is that most protocols other than TCP are message-oriented, while TCP is bytestream-oriented. TCP does not care how large a TLS Record is; it will just split it in as many parts as necessary to send it. Message-oriented protocols, on the other hand, conserve the message boundaries, but may not have a mechanism to fragment and reassemble messages, which is the case with UDP, for instance. The consequence is that only messages smaller than the current Path-MTU can be sent. The Path Maximum Transmission Unit is the maximum message size every router on the path between the peers can handle. Especially the messages containing certificates may be larger than the current Path-MTU. To still be able to transfer these messages, DTLS has to provide its own fragmentation mechanism. This is achieved by extending the Handshake Message Header, as shown in Figure 3.8. With TLS every handshake message starts with its Message Type and its Length. For DTLS a Fragment Offset and Fragment Length entry is added.

DTLS also has to deal with reordered messages, which can likely occur with unreliable transport. To handle handshake messages arriving in the wrong order, the Handshake Message Header is further extended and a Message Sequence Number is added. This allows restoring the correct sequence of the handshake.

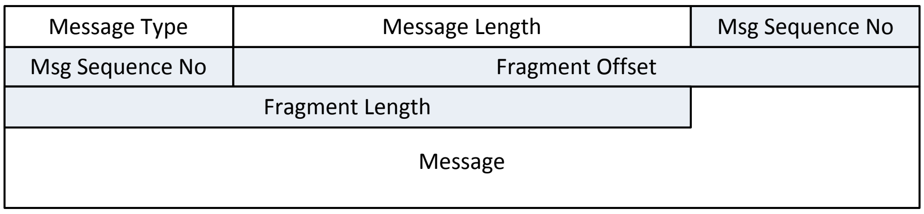


Figure 3.8: DTLS Handshake Message Header

3.3.3 Client Verification with Cookies

A new connection is always initiated by the client sending its ClientHello. The server responds with its ServerHello, Certificate, maybe the optional ServerKeyExchange and CertificateRequest, followed by the ServerHelloDone. This part of the handshake is a problem with connectionless transport protocols, because there is no transport connection setup necessary, and an attacker could just send an arbitrary number of ClientHellos to a server. This could be used for a Denial-of-Service attack against the server, which will start a new session, thus allocating resources, for every ClientHello. It could also be used against another victim by redirecting the much larger response of the server to it, thus multiplying the attacker's bandwidth. To prevent this issue, DTLS uses an additional handshake message, called HelloVerifyRequest. The extended handshake is depicted in Figure 3.9. It can be sent optionally in response to the ClientHello and contains a so-called cookie of arbitrary data. If this message is used, the server will not allocate any resources yet. It requires the client to repeat its ClientHello with the unmodified cookie attached. This allows the server to verify that the client uses a valid address and responds. To prevent manipulations or reusing an old cookie, the cookie should contain information about the client, like its address, a timestamp and a signature to ensure the integrity of the data. If the verification was successful, the server can process the ClientHello, allocate the necessary data and continue with the regular handshake. With this mechanism the server's resources cannot be exhausted with many faked ClientHellos, because they won't be repeated with a valid cookie. A bandwidth multiplication is also prevented, because the HelloVerifyRequest is smaller than a ClientHello.

3.3.4 Handshake Reliability

The handshake cannot be completed if one or more messages are missing, so it has to be performed reliable. With an unreliable transport, DTLS has to ensure the reliability of handshake messages itself. Therefore, it needs a timer to retransmit lost messages. For increased efficiency, DTLS does not use a timer for every message, but for bundles of messages, called flights. A flight contains all messages

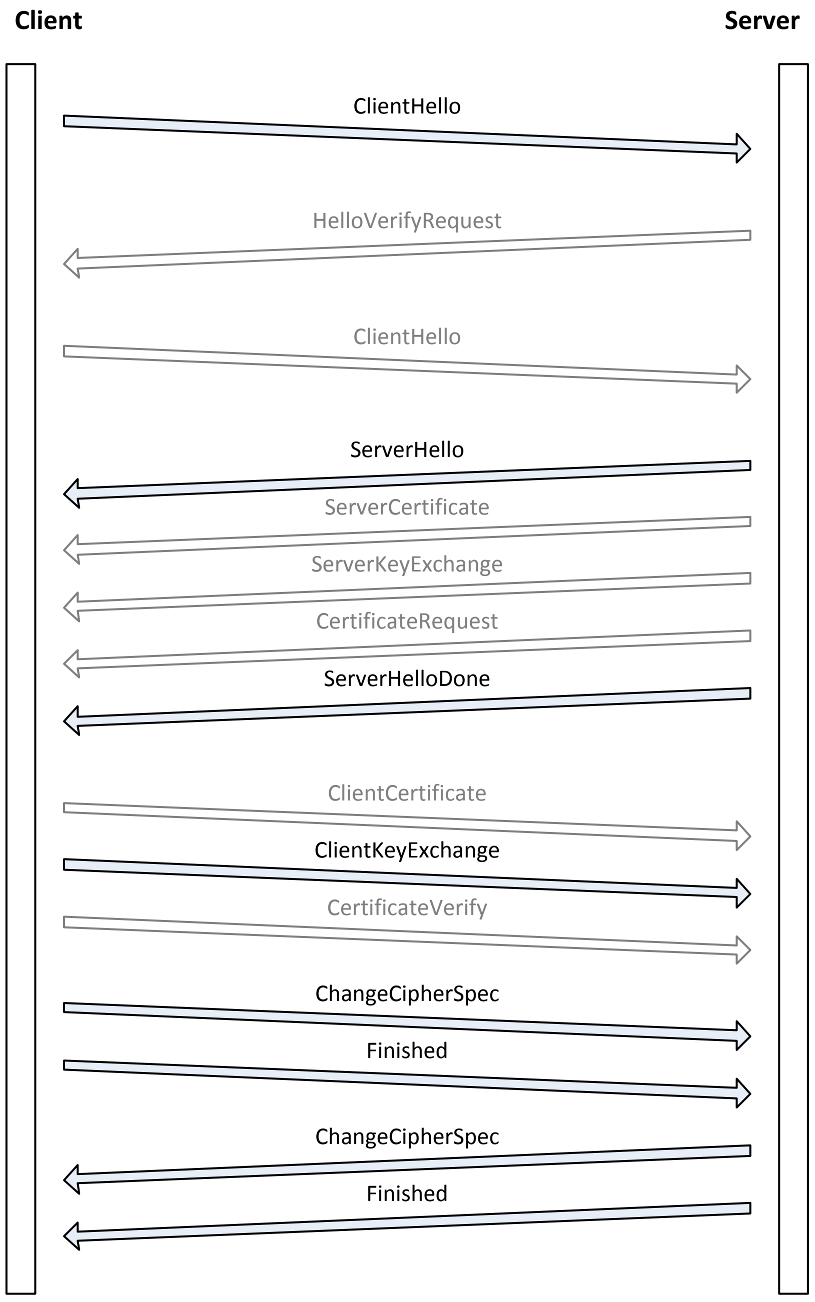


Figure 3.9: DTLS Handshake

before the sending side changes, so for example all messages from ServerHello to ServerHelloDone form a flight (compare Figure 3.9). For every flight sent, a timer is started, and if there is no response until the timer expires, the entire flight will be retransmitted. The retransmission is limited to handshake messages and cannot be used for application data.

3.3.5 Alert Protocol

Because of the connectionless transport protocol, DTLS sends some errors just as warnings. This is done when the HMAC of a message could not be verified (BadRecordMAC), the Record length is too large (RecordOverflow), or the decryption failed (DecryptionFailed). Otherwise an attacker could just send random data to one of the peers to cause an error that immediately terminates the connection.

3.4 Secure Shell

Secure Shell (SSH)[113] is a security protocol providing bidirectional channels for various services over a single secure connection. Possible services are data transfer, interactive remote shells, X11 forwarding and connection forwarding.

3.4.1 Transport Layer

The SSH Transport Layer [114] runs on top of a reliable transport protocol, typically TCP, and provides an encrypted and integrity ensured transport for the two other SSH sub-protocols, the Authentication and the Connection protocol. The structure of SSH and its sub-protocols is illustrated in Figure 3.10.

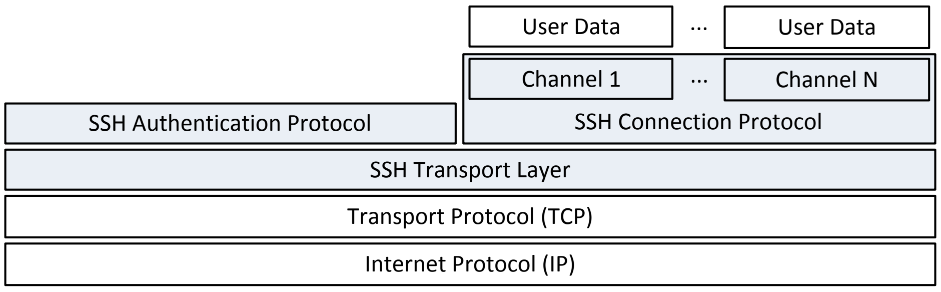


Figure 3.10: SSH Protocol Structure

The SSH Transport Layer performs host-based authentication, and security parameters as well as the shared key are negotiated as shown in Figure 3.11. The connection establishment is always initiated by the client sending a Protocol Version message. This is also done by the server and after that Key Exchange Init messages are exchanged, which list the supported cypher suites and key exchange algorithms.

Depending on the mutually supported algorithms, a shared key is negotiated with messages corresponding to the chosen algorithm, which is usually the Diffie-Hellman method [15], since it requires no certificates or other data that has to be provided in advance. The New Keys messages announce the use of the previously negotiated parameters and key, and conclude the connection setup.

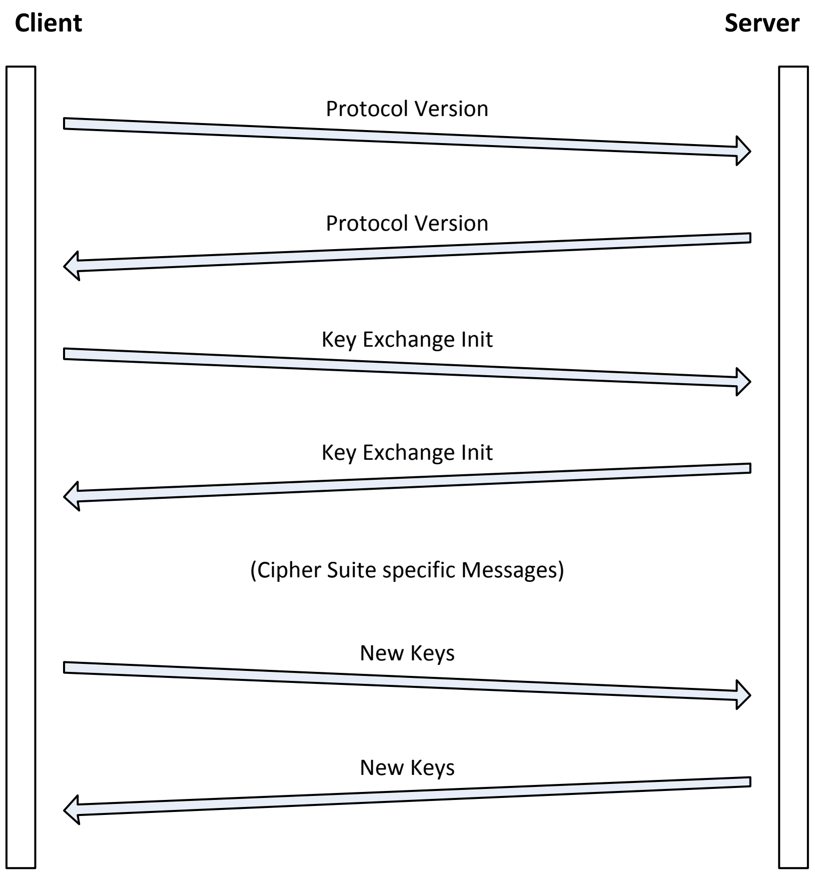


Figure 3.11: SSH Key Exchange Message Flow

3.4.2 Authentication Protocol

The SSH Authentication [111] protocol is used by the client to request the permission to use a specific service. It will provide a username, and the server offers acceptable authentication methods, by default host-based, password, or public-key. The client can use any offered authentication method in arbitrary order until the authentication is successful, or the server has disconnected because of too many failed attempts, or a timeout occurred.

3.4.3 Connection Protocol

Upon successful authentication, the client is allowed to actually request a service, such as data transfer, interactive remote shells, connection forwarding, or the forwarding of the X Window System (X11) [9], the GUI of many UNIX- and Linux-based systems. The services are realized with the SSH Connection [112] protocol. This protocol provides channels for each offered service, which can be opened and closed arbitrarily within the secure connection.

The connection forwarding supports TCP connections only, and is illustrated in Figure 3.12. An SSH connection is set up between two hosts, and one of them is configured to accept TCP connections on a specific local port. The destination to where all data should be forwarded has to be configured as well. The data received from connections accepted on the local port will be forwarded over the SSH connection to the other SSH endpoint, which will establish a new TCP connection to deliver the forwarded data to the previously configured destination host. Every accepted incoming TCP connection triggers the opening of a forwarding channel within the SSH connection, and the establishment of a new TCP connection to the configured destination by the other SSH endpoint. The data transmission is now separated into three segments. The payload of the initial TCP connection is received by the first SSH endpoint, sent over the forwarding channel to the other SSH endpoint, which sends it over the new TCP connection to the destination and vice versa. This service can be used to secure an otherwise unsecured connection, since the SSH connection is encrypted, or to tunnel a connection to a protected host, since SSH requires authentication.

SSH maintains a Flow Control for every data transmission channel, realized with a window indicating the amount of data that is allowed to be sent on this channel. The size of the window is initially announced during the channel opening and then continuously adjusted with the SSH_MSG_CHANNEL_WINDOW_ADJUST message. This allows slowing down the transfer rate of a forwarded sender, in case the receiver cannot deliver the incoming data to the destination quickly enough or not at all.

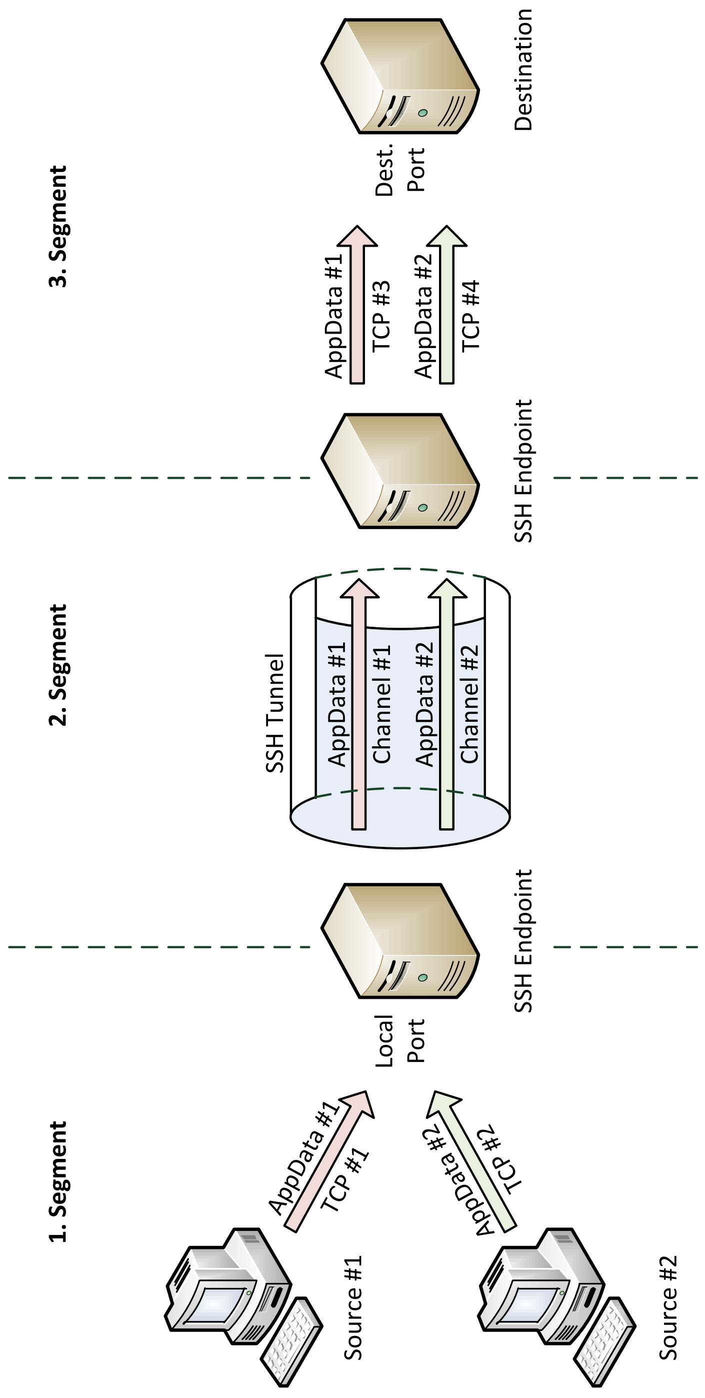


Figure 3.12: SSH Forwarding

Chapter 4

Tools and Software

To evaluate the functionality and possible benefits of the solutions proposed in this thesis, appropriate tools and software are necessary. Measurements can either be done by simulation to analyze the behavior of network connections in a controlled environment, or on real systems, to investigate the impact of external influences, such as hardware limitations, for instance.

4.1 Simulation

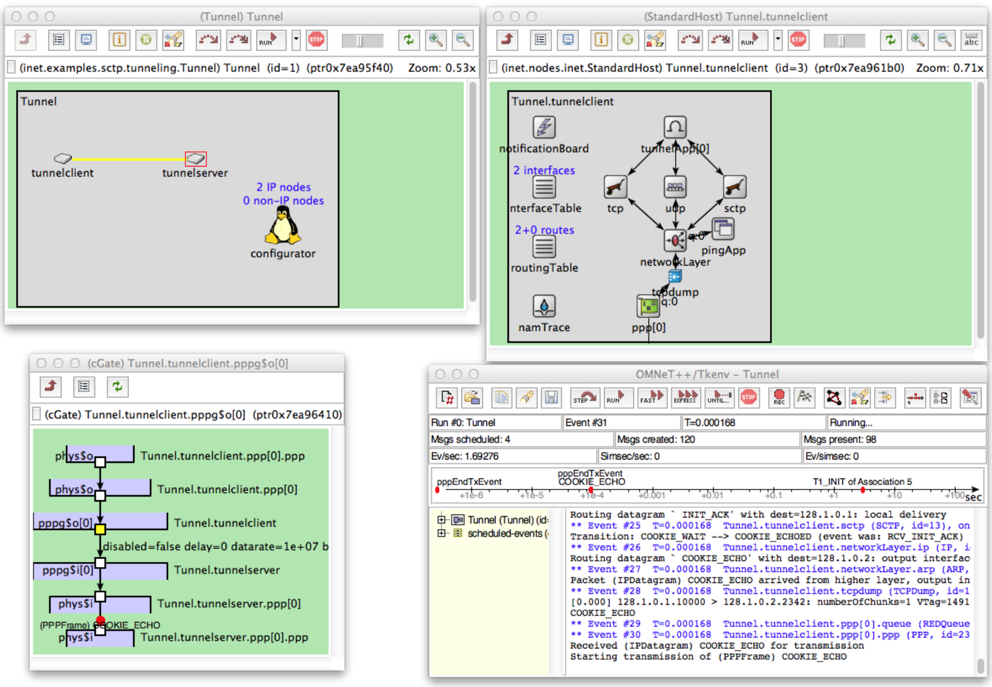
A widely used method to examine the behavior of a proposed solution to a problem is a simulation [115]. For networks, a discrete event simulation is usually used, which simulates a sequence of events occurring at a certain time. Each event alters the state of the system and may trigger other events [83]. The simulation allows performing repeatable and deterministic measurements in a controlled environment, in which selected parameters can be varied without external influences, like hardware limitations. Hence, the impact of the proposed changes can be measured without possible side effects to evaluate potential benefits under optimal conditions.

There are only few network simulation tools providing SCTP implementations. The most notable are the commercial OPNET Modeler [63] and QualNet [86], as well as the open source tools Network Simulator 2 (NS-2) [56] and OMNeT++ [107] with its INET framework [34]. To avoid expensive licensing and to allow easier bug tracking, if necessary, an open source solution is preferred. While NS-2 contains the base specification of SCTP provided by the University of Delaware, the implementation in OMNeT++ with INET has been completed with all available SCTP features and extensions by our research collaboration of Münster University of Applied Sciences and the University of Duisburg-Essen. The source code has been provided to the OMNeT++ developers and is included in the official releases. Therefore, OMNeT++ is the tool of choice for simulations and will be introduced in the following section.

4.1.1 OMNeT++

OMNeT++ is a discrete event simulation environment. The program provides the infrastructure to develop functional modules interacting with each other, which are written in C++ with the interfaces provided by OMNeT++. Parameter configuration and analysis methods support the creation of parameter studies and the evaluation of the results.

The simulations can either be run from the command line, or with a Graphical User Interface (GUI), as depicted in Figure 4.1. This allows efficient use of resources, but also the possibility to monitor single events and the state of the system at any given time.



Run #0: Tunnel | Event #31 | T=0.000168 | Running...
Msgs scheduled: 4 | Msgs created: 120 | Msgs present: 98
Ev/sec: 1.69276 | Simsec/sec: 0 | Ev/simsec: 0

```
Routing datagram ` INIT_ACK' with dest=128.1.0.1: local delivery
** Event #25  T=0.000168  Tunnel.tunnelclient.sctp (SCTP, id=13), on
Transition: COOKIE_WAIT --> COOKIE_ECHOED (event was: RCV_INIT_ACK)
** Event #26  T=0.000168  Tunnel.tunnelclient.networkLayer.ip (IP, i
Routing datagram ` COOKIE_ECHO' with dest=128.1.0.2: output interfac
** Event #27  T=0.000168  Tunnel.tunnelclient.networkLayer.arp (ARP,
Packet (IPDatagram) COOKIE_ECHO arrived from higher layer, output in
** Event #28  T=0.000168  Tunnel.tunnelclient.tcpdump (TCPDump, id=1
[0.000] 128.1.0.1.10000 > 128.1.0.2.2342: numberOfChunks=1 VTag=1491
COOKIE_ECHO
** Event #29  T=0.000168  Tunnel.tunnelclient.ppp[0].queue (REDQueue
** Event #30  T=0.000168  Tunnel.tunnelclient.ppp[0].ppp (PPP, id=23
Received (IPDatagram) COOKIE_ECHO for transmission
Starting transmission of (PPPFrame) COOKIE_ECHO
```

Figure 4.1: OMNeT++ Graphical User Interface

Furthermore, we have developed an extension [87] that manages the parallel computation of simulation runs with a multi-core CPU or multiple computers in a network with Xgrid [1] based on Mac OS X Server.

4.1.2 INET Framework

The INET framework provides models and implementations of networking protocols for OMNeT++. The support includes link layer protocols, such as Ethernet, Point-to-Point (PPP), and IEEE 802.11 (Wi-Fi), network layer protocols, such as IPv4 and IPv6, various routing protocols, and the transport protocols TCP, UDP, and SCTP.

Models of basic devices typically found in an IP-based network are also available. This includes hosts and routers, which consist of a full link layer, IPv4/IPv6, a transport protocol stack and routing capabilities, respectively. The network configuration, IP addresses and routing information for example, can either be provided manually or configured automatically. There are some default applications a host can run, for instance simple traffic generators, but more advanced applications can be added.

4.2 OpenSSL

The OpenSSL toolkit [60] is the most widely used library for TLS-based security and is available for Windows, most UNIX- and Linux-based, and other operating systems [37]. Furthermore, it also contains the most advanced open source DTLS implementation. It was first added by the developer of DTLS, Nagendra Modadugu [51], for release 0.9.8 [61] and has been maintained and extended by us since 2008 [53]. Hence, prototype implementations for TLS- or DTLS-based security will be based on OpenSSL.

4.2.1 Architecture

The architecture of OpenSSL is separated into multiple objects interacting via abstracted interfaces. Figure 4.2 illustrates the architecture and relations between the different objects of OpenSSL. Generic parameters, such as the used protocol, are stored in Context objects (CTX). For every new connection, a new Session object (SSL) is derived from the context. Each session uses Basic Input/Output objects (BIO) to communicate via networking sockets or read and write files. The relations are one-to-one, so every session object is derived from a single context and has BIO objects for both reading and writing, although this can also be handled by a

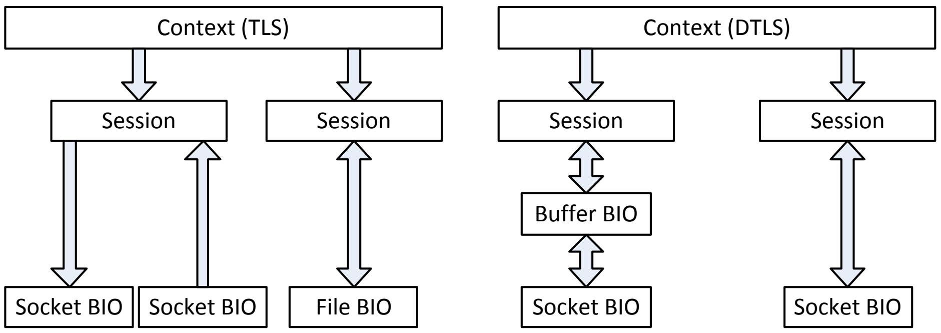


Figure 4.2: OpenSSL Architecture

single BIO object. There are different kinds of BIO objects, they can either access a networking socket or file, or have a special intermediate functionality, like buffering, in which case they will use another BIO object for the actual I/O operation.

4.2.2 Context Objects

Before any connection can be established, a context has to be created to store the necessary parameters. The most obvious is which protocol should be used. OpenSSL currently supports TLS 1.2 (since release 1.0.1), TLS 1.0, Secure Sockets Layer (SSL) 2.0 and 3.0, as well as DTLS 1.0. Upon creation of a context object with a protocol, an SSL_METHOD object will be assigned. Every protocol has its own object, it contains protocol-specific functions for sending, receiving, handshaking, and so forth. The certificates and private keys for RSA [81] or parameters for Diffie-Hellman key exchanges that should be used for all sessions derived from this context, can then also be specified.

If session resumption should be allowed, this can be activated for the context, which also caches the session information of previous connections, so they can be reused with a new session object. Furthermore, generic parameters can also be set, such as timeouts or the allowed cipher algorithms.

4.2.3 Session Objects

For every new connection, a session object is derived from an appropriate context. The session object holds all parameters and information necessary for the connection. It also provides a generic and protocol-independent API to perform tasks like connection establishment, or sending and receiving data with the protocol-specific functions inherited from the context. Furthermore, options can be changed from the inherited default values for the connection handled by it.

The session object holds two BIO objects, which have to be assigned before usage. One is used for receiving data and the other for sending data. It does not matter to the session object whether the BIO objects use networking sockets or files, and whether two different or a single object is used for both sending and receiving. However, it is assumed that everything read and written from and to these BIO objects belongs to the connection the session object is managing.

4.2.4 Basic Input/Output Objects

All input and output operations are handled with BIO objects. This ensures that the upper layers do not need any knowledge about different transport protocols or file formats. BIO objects can also be chained, so a buffering or filter BIO can be used before the actual I/O operation is performed with a file BIO, for instance.

Every kind of BIO object provides options to retrieve and set parameters for its provided functionality. In case of a transport protocol like TCP, this may be socket errors or even an interface for accepting new connections.

4.2.5 DTLS Implementation

OpenSSL has originally been developed for SSL and later TLS, so the addition of DTLS required more extensive modifications than just adding the protocol specifications. Adding a new SSL_METHOD object with DTLS-specific functions does the latter. That is already enough to use the DTLS protocol, although only over TCP, because only BIO objects for handling TCP connections were available at this point. Hence, for an additional transport protocol like UDP, a new kind of BIO object is necessary to handle its characteristics.

While UDP supports one-to-many style sockets, that is a single socket can be used to communicate with multiple peers, the opposite is true for TCP where multiple connections are handled with a separate socket for each. SSL and TLS are based on TCP, so the one-to-one approach was adopted to the OpenSSL architecture (compare Figure 4.2). Therefore, one-to-many style sockets cannot be realized without modifying considerable parts of the architecture and API, to allow multiple session objects share a single BIO object handling all their UDP communication. The use of connected UDP sockets is necessary to simulate a one-to-one behavior, which can be realized just as TCP. Connecting a UDP socket means binding it to an address and port, thus limiting the communication to this destination.

Additionally, as part of my work the API has been extended to support the new cookie mechanism of DTLS. Creating a new session object and thus allocating resources for every incoming ClientHello can be used for DOS attacks (compare Section 2.5.2). Therefore, a new listening API was necessary that responds to ClientHellos with HelloVerifyRequest messages only, until they return a valid cookie. The cookie creation and verification mechanism also required extending the API, so I added appropriate callback methods. The user has to implement the functions and register them for callback, so the amount of security measures used can be adapted to the application's needs.

4.3 OpenSSH

The Secure Shell protocol [113] has been developed by Tatu Ylönen as a more secure replacement for remote login protocols, such as Telnet [71], and was first released as freeware, but still proprietary software. Based on version 1.2.12, the last one released as freeware, a fork was developed with many bug fixes by Björn Grönvall, called OSSH and was also released as freeware. This again was forked by the OpenBSD project and integrated into their system as OpenSSH [59]. Since then, it has been developed as an open source project under the BSD license and became part of many UNIX- and Linux-based systems, so it is now the most widely deployed SSH implementation.

The implementation of OpenSSH is separated into client, server and common parts. The latter include the basic handling of encryption and decryption of SSH packets, as well as the management and services realized with channels. The client-

and server-specific parts handle initiating and accepting new connections, respectively, and also the corresponding handshaking behavior for the connection establishment. Both allow configuration via command line parameters or config files. The server uses separate sockets for every address it is supposed to listen on, and forks itself for every incoming connection. This ensures that each connection is handled by its own process, which allows to restart the SSH server while staying logged in remotely, for example.

Chapter 5

Securing SCTP-based Applications

Existing security solutions, such as TLS, have been developed based on the assumption that TCP is used as transport protocol. While SCTP is rather similar to TCP in some aspects, it offers distinct features that limit the applicability of these existing solutions severely. Therefore, adaptations, extensions and even new approaches are necessary to define optimized security solutions for SCTP-based communication.

To identify applicable strategies for securing application protocols, the applications can be categorized into two distinct classes with respect to their ability to contribute to securing their end-to-end communication. The first class contains applications that provide functionality, or at least support, for securing their communication and will be called "security-aware". For these applications, security was already considered to some extent during their design and development. This can range from integrating security features directly into the application and its communication protocol to simply defining, or adapting and extending, the lower layer interface to enable the cooperation with external security mechanisms. Examples for such applications are web browsers and email clients, which can secure their protocols Hypertext Transfer Protocol (HTTP) [76] and Internet Message Access Protocol (IMAP) [10], respectively. Applications belonging to the other class are called "security-agnostic", meaning that they are totally unaware of the communication security issue, like Telnet, or intentionally rely on other mechanisms, like Virtual Network Computing (VNC) [79]. This is not typical anymore for the majority of newly designed applications. However, most of the widely used legacy applications and protocols fall into this category. As adding security support may require significant redesign, standardization activities and could compromise compatibility, this is not always the preferred option. The same holds for cases where the source code is not open in the first place or the user has specific security requirements and is providing own security policies. Therefore, flexible and efficient strategies to secure the communication of security-agnostic applications are definitely still needed. My detailed analysis of possible strategies to secure end-to-end connections in this chapter has also been published in [92].

5.1 Security-aware Applications

Security-aware applications provide or at least support security mechanisms for their data exchange. To do this, the required functions and mechanisms can either be integrated directly into the application, or make use of an external library or the operating system to provide such features. In either case, this does not require any modifications of the transport protocol and the lower layers, since the application data is already secured before handing it down to the transport layer. However, this does not secure the transport protocol itself. All information contained in the headers of the transport protocol and the lower layers remains exposed to attacks.

5.1.1 Integrated Mechanisms

One strategy to secure the communication protocol used by an application is to integrate security features right into it. This has the advantage, that the security solution is custom-made to perfectly fit the application's needs. Furthermore, there is no dependency on external libraries or any other kind of support, because the application comes with everything necessary already. This strategy has been used for Skype [50], for instance, whose proprietary protocol is secured with RC4 and AES encryption [5]. The disadvantage of integrating such a custom-made solution is that the design and implementation of the protocol and its security features have to be repeated for every single application. This requires not only an extensive amount of work for the design and implementation, but also for testing, since this strategy is more prone to flaws in concept and realization, which may lead to severe security issues.

5.1.2 External Mechanisms

Rather than designing and implementing security features for every specific application, generic mechanisms provided by an external library or the operating system can be used. This reduces the necessary effort of development and testing, because such solutions are usually standardized or at least already widely deployed. Since generic solutions have to be independent of the application protocol that is to be secured, the realization can either be done by adding an additional layer between the application and transport protocol or by using features of a transport protocol providing security features. Figure 5.1 illustrates these two generic methods. As a consequence, the transport protocol chosen by the application limits the possible solutions. In both cases, the application has to be aware of the external security mechanisms and has to access them via appropriate interfaces.

The approach to insert a security layer is used by TLS for TCP and DTLS for UDP, respectively. Common protocols, which make use of TLS, are HTTP [44] or

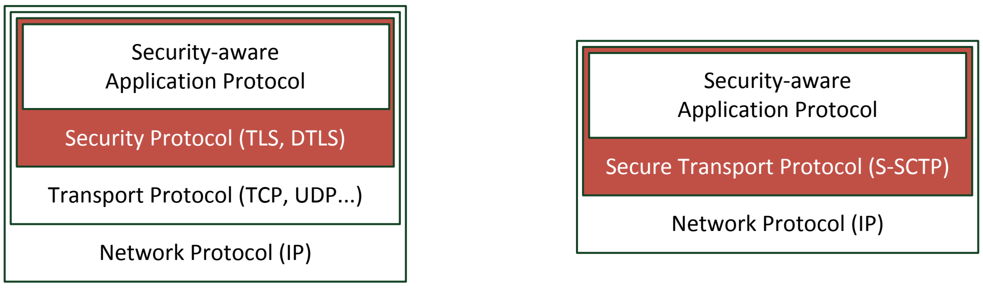


Figure 5.1: External Security

IMAP [57]. Transport protocols with inherent security functionality are not readily available yet. However, the Secure SCTP [22] concept described in Section 5.4.3 is an example for such a protocol, which demonstrates the potential advantages.

5.2 Security-agnostic Applications

Many widely used legacy applications were designed without considering security features, because at the time of their development or in their designated field of use security was not an issue. To secure their protocols with the methods described in the previous sections, these applications have to be modified and extended, which has already been done for most popular applications like Simple Mail Transfer Protocol (SMTP) [30], File Transfer Protocol (FTP) [26], and many others. However, in some cases this may not be possible or would create compatibility problems. Hence, another common strategy is to leave the application and its protocol untouched, and to use a tunnel instead that provides the necessary security features. Such a tunnel can be realized in several ways, depending on the network layer on which it is located.

5.2.1 Network Protocol Tunneling

The most generic approach is to tunnel the network protocol, since de facto there is only the Internet Protocol that has to be considered. The IP packet will be encapsulated in a security layer instead of being sent directly. This can be either an IP packet extended with security features or an upper layer protocol, as shown in Figure 5.2. The former is typically realized with the tunnel mode of IPsec, which encapsulates the original IP packet in another IP packet. The latter can be done with several protocols, for example TLS, which is partly used by OpenVPN [62], or DTLS, which is used by the Cisco VPN software [6]. The TLS- and DTLS-based solutions create more overhead than IPsec, but can be realized in an application without support of the operating system that is necessary for IPsec.

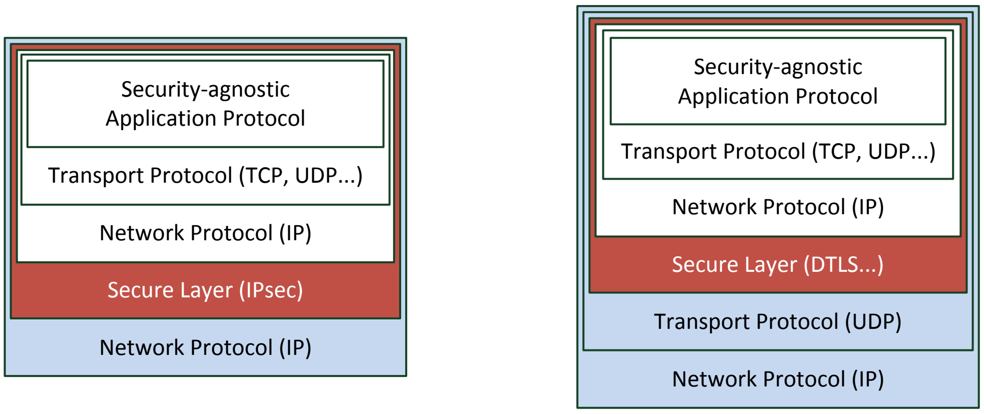


Figure 5.2: Network Protocol Tunneling

A tunnel realized by these methods is often referred to as a Virtual Private Network (VPN). This setup has the advantage that it does not matter which protocols are used above IP, so it basically works with every application protocol on an arbitrary transport protocol. This solution, however, requires a manual configuration and setup of the tunnel in advance, which includes the distribution of a shared secret and, in case of IPsec, an appropriate configuration for firewalls and routers with Network Address Translation (NAT). This limits the possible application scenarios significantly, so this approach is usually used to link two networks and tunnel an arbitrary number of connections between them, rather than tunneling a single connection between two hosts.

IPsec also supports a transport mode, which adds an authentication and encryption layer to protect the payload of an IP packet, that is the upper layers, without the encapsulation of the tunnel mode. Hence, this method can only be used for direct connections between two hosts, but still has the disadvantages of the manual setup and issues with routers and firewalls, and will therefore not be discussed in more detail.

5.2.2 Transport Protocol Tunneling

A tunnel can also be created for the transport protocol that the application chose to send its messages on. This is realized by adding a secure layer, usually with its own transport protocol, between the network and transport layers. For the secure layer, the common protocols TLS and DTLS can be used. A benefit of this method is that the transport protocol and all its information, which may be helpful for an attacker, are secured as well. Furthermore, the once established secured layer could be used for different transport protocols without additional renegotiation. This is especially helpful when the Interactive Connectivity Establishment (ICE) [84] protocol had to be used to connect through firewalls and NAT routers, and thus every reconnect is costly. This approach to security is illustrated in Figure 5.3.

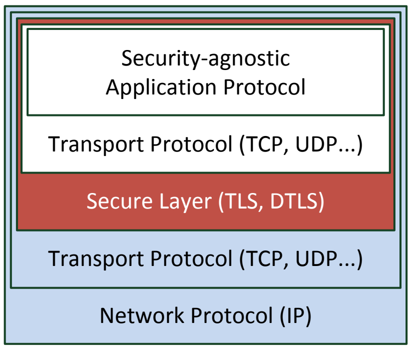


Figure 5.3: Transport Protocol Tunneling

In [74] this concept is used to improve the performance of The Onion Routing (TOR), a network that makes TCP connections anonymous. Additionally, the “Real-Time Communication in WEB browsers” (RTCWeb) working group at the IETF is currently evaluating how direct and secure communications between web browsers can be realized [75]. Since the focus is on Voice over IP (VoIP), but generic data transfer should be possible as well, this security technique was chosen to support multiple transport protocols. After a DTLS connection is established with ICE, it can either be used with SCTP to transfer generic data or only its key material for the Secure Real-time Transport Protocol (SRTP) [2] to transfer media streams.

The realization, however, requires support of the operating system for the necessary encapsulation of the transport protocol into a secure layer, which is currently not available. Otherwise, the application has to be modified to use a user-space implementation of the tunneled transport protocol. This strategy for security is only beneficial in very specific scenarios, because it requires more effort than simply using TLS or DTLS directly.

5.2.3 Application Protocol Tunneling

The application protocol itself can be tunneled directly as well, which is commonly called forwarding. The end-to-end connection is segmented in this case, because the application’s transport protocol connection is terminated at one of the tunnel endpoints, which forwards only the payload (the application data) to the other endpoint of the tunnel, from where a new transport protocol connection is established to the actual destination. This setup is illustrated in Figure 5.4.

The tunnel does not necessarily have to be realized with the same transport protocol that has been used by the application. However, explicit support for the

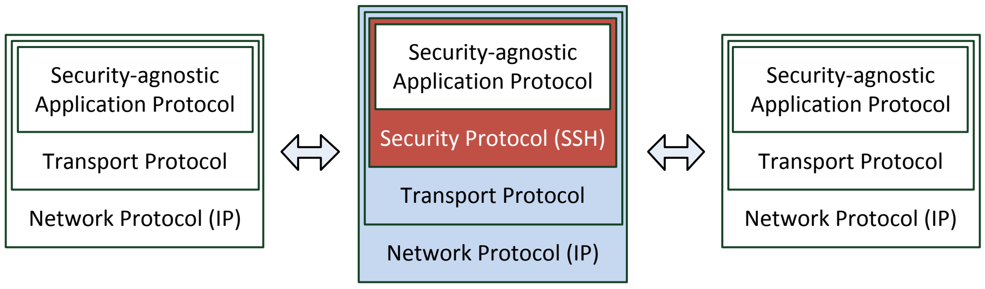


Figure 5.4: Application Protocol Tunneling

transport protocol originally used by the application is necessary, otherwise some transport protocol features may be unavailable. For reliable protocols in general a Flow Control is necessary, because if one host stops receiving data from the tunnel and the sender does not reduce its transmission rate, the buffers within the tunnel will run full and data may be lost. If the transport protocol has specific features, like the streams of SCTP, these have to be explicitly supported, since the information has to be preserved within the tunnel and used again for the connection at the far end. The setup is static, so the tunnel can only be established for a connection between two hosts, and has to be done in advance. This kind of tunneling is currently done with Secure Shell (SSH) [113], which supports forwarding for TCP connections.

5.3 Comparison

Each of the previously introduced strategies to secure the communication of an application has its pros and cons. Depending on the applications, security requirements and usage scenario, different approaches will be preferred. If the application in question can be modified and adapted, then basically all solutions can be considered. However, extending the application's protocol with security features is usually not desirable, because of the significant effort for every single application. Tunneling, on the other hand, often requires specific support by the operating system or manual configuration in advance. Therefore, the use of external mechanisms to add an additional security layer, such as TLS, is usually the method of choice. If the application is security-agnostic and cannot be modified, then tunneling is the preferred strategy. Which kind of tunneling is optimal for a scenario depends on the given circumstances, for example, which features are supported by the operating system. Table 5.1 provides an overview of the pros and cons of each solution and lists some of the major aspects to be considered when choosing the most appropriate strategy for a specific scenario.

	Security-aware Applications		Security-agnostic Applications (Tunneling)		
	Internal	External	Network	Transport	Application
Application protocol remains untouched	-	x	x	x	x
Application can remain untouched	-	-	x	x[1]	x
Reusable for different applications	-	x	x	x	x
No operating system support required	x	x	-	x[1]	x
Transport protocol features are secured	-	-	x	x	x
Transport protocol independent	x	x	x	x	-
Connections can be set up on demand	x	x	-	x	-
No overhead for multiple connections	x	-	x	x	x
Examples	Skype	HTTP, IMAP, XMPP	IPsec, Cisco VPN, OpenVPN	RTCWEB	SSH

[1] either operating system support or modifications of the application required

Table 5.1: Comparison of Security Solutions

5.4 Existing Security-aware Solutions for SCTP

The common security-aware solutions TLS and DTLS, developed for TCP and UDP, respectively, can also be used with SCTP. Yet the features of SCTP have to be limited to the ones of the originally supported protocols. Additionally, two approaches to extend SCTP with security features have been proposed.

5.4.1 TLS over SCTP

The most obvious solution to secure SCTP-based communication is using TLS, which has been standardized in 2002 [39]. However, this solution is not widely deployed because of some severe limitations. The main issue is that TLS has been defined assuming reliable and in order transport as offered by TCP. As a consequence, successive TLS data units depend on each other and it is not possible any more to process them at the receiver if loss or reordering occurs. SCTP only retains the order of messages within a stream, not across multiple streams in order to minimize the head of-line blocking effect. This leads to systematically occurring reordering at the receiver if multiple streams are used. To avoid this situation, the multi-streaming feature of SCTP cannot be used in combination with TLS – unless a separate TLS connection is managed for every active stream. Unreliability with PR-SCTP cannot be used at all. Furthermore, TLS encrypts each application message separately while SCTP can bundle several small application messages into one SCTP packet to increase transport efficiency. Therefore, TLS encryption leads to unnecessary overhead for small messages. Since cryptographic security with TLS is provided above SCTP, the transport protocol itself and its control messages remain unprotected, and thus a target for eavesdropping and tampering.

5.4.2 DTLS over SCTP

Using DTLS to secure SCTP associations has been discussed in [31]. DTLS has been developed to tolerate reordering and packet losses that occur with unreliable transport protocols. So contrary to TLS, the use of multiple streams and partial reliability is possible. However, since with DTLS it is assumed that packet loss is acceptable, it may discard messages under certain circumstances, for example because they cannot be processed immediately while a handshake is in progress. This counteracts the reliability of SCTP, because the messages are discarded after being successfully delivered by SCTP. The same limitations as with TLS regarding the overhead for small messages and an unprotected transport layer still apply.

5.4.3 Secure SCTP

Secure SCTP [22] integrates encryption and authentication features into the SCTP protocol. Different levels of security are supported, ranging from no security (level 0) to authentication, encryption for all chunks, and integrity checks for all packets (level 3). The integration of security features has the advantage that not only application data, but transport protocol information as well is secured. By allowing to encrypt chunks selectively with a flag (level 2) or all at once (level 3), the necessary computational effort is minimized. The main issue with this solution, however, is that it would require major extensions to the SCTP standards and to their implementations in the various operating system kernels. Additionally, the key and certificate management is vulnerable to Denial-of-Services attacks as it may be used to block the kernel and so stop the entire system, in case only one CPU core is available.

5.4.4 Secure Socket SCTP

With combining different approaches, Secure Socket SCTP [46] tries to avoid the drawbacks of the other solutions. Integrity is provided with the SCTP-AUTH extension for all chunks, while the negotiation of parameters and the encryption is done with TLS functions. However, an issue of this approach is the use of the Payload Protocol Identifier (PPI) in the DATA chunk header to determine whether this chunk is encrypted and which key and algorithm have been used. The PPI actually indicates which upper layer protocol is used. This prevents the use of several protocols which require a specific PPI to be set. Another major issue is an elaborate implementation and standardization of the entirely new protocol.

5.5 Existing Security-agnostic Solutions for SCTP

A common solution to provide security for security-agnostic applications is to tunnel their connections. This is usually done at the network or application layer. While network tunneling basically supports all transport protocols that run on top of the network protocol, that is SCTP as well in case of IP, application layer tunneling requires explicit support for each protocol.

5.5.1 Internet Protocol Security

SCTP over IPSec [4] has also been standardized shortly after SCTP in 2002. The security protocol suite IPsec is an extension for IP to secure IP and the layers above without specific requirements concerning the transport protocol. Therefore, securing SCTP with IPSec is possible and there is no hard limitation of SCTP's features. However, the multi-homing support of SCTP is still an issue, because multiple IP addresses are used. To enable multi-homing, an IPsec connection has to be established for every possible pair of addresses, which results in an excessive overhead in the SPD. As a solution, the SCTP over IPsec standard suggests lists of multiple addresses for an IPsec connection, but there is no implementation available yet. SCTP supports fragmentation to split up large messages for sending them in multiple IP packets. IPsec encrypts each IP packet individually, so sending large messages creates additional overhead.

5.5.2 TLS/DTLS-based Tunneling

As an alternative to IPsec, network tunneling can also be realized over TLS or DTLS. OpenVPN and the Cisco VPN software use this technique for instance. The advantage is that no support of the operating system is required, besides virtual interfaces to route the network traffic. Since the network layer or even the link layer (optional for OpenVPN) is tunneled, SCTP is also supported. The same limitations as with IPsec, however, still apply. Multi-homing is not supported without the setup of multiple tunnels, although it may be realized based on DTLS with its connectionless transport protocol UDP.

5.6 Conclusion

The importance of security is well known nowadays, so it will be considered in the design of most newly developed applications. Therefore, a security-aware solution to secure such applications using SCTP is inevitable. The existing ones, however, have severe limitations regarding usable SCTP features and deployability so a new approach is still necessary. Tunneling SCTP-based traffic of security-agnostic applications is basically possible by using a VPN, and only multi-homing remains an issue. Application protocol tunneling, on the other hand, which is commonly realized with SSH, is still not available.

Chapter 6

SCTP-aware DTLS

To overcome the limitations of the existing security solutions for security-aware applications, a new approach is still necessary. The focus has to be on deployability, and feature limitations have to be avoided. This can be achieved by analyzing the relevant characteristics of SCTP and adjusting the new concept to them. In this chapter, a new solution for security-aware applications using SCTP is introduced, which has been developed as part of this project and also been published in [88].

6.1 General Considerations

The need for encryption, authentication and integrity for the payload is obvious, but SCTP is also a reliable transport protocol, so the reliability should be ensured as well. Contrary to TCP, SCTP does not necessarily retain the order of all messages. They can be reordered across streams or can even be sent unordered explicitly. This means the solution has to ensure the order of messages sent in order, but also be able to handle unordered messages. Like DCCP, SCTP uses a Congestion Control to protect the network, so the possibility of unnecessary retransmissions overloading the network has to be considered as well. The solution also has to allow the usage of any other SCTP feature, like multi-homing, streams and the various extensions.

Additionally, it has to be avoided to modify the SCTP protocol itself, since this would require an elaborate standardization process and the corresponding adaption of existing implementations. This would make a wide deployment much more difficult. The same applies to the encryption layer that preferably relies on existing work to also prevent an elaborate standardization and a new and therefore error-prone implementation.

6.2 Open Issues and Proposed Solutions

With the requirements identified above, it is now possible to design a solution that meets all of them. The most promising candidate to start with is DTLS as proposed in [31], since it is already standardized and does not prevent any SCTP features from being used. However, it has some issues with reliability and providing the required security features, like protecting the transport protocol.

6.2.1 Encryption and Authentication

DTLS is an adaptation of TLS, so it provides the same encryption, authentication, and compression features. An HMAC is calculated of the compressed application data and both are encrypted and sent as the payload of the transport protocol, which itself is unprotected. The DATA chunks of SCTP contain more information than a TCP packet, that is the stream information and the Payload Protocol Identifier (PPI). This data can remain unencrypted, because an attacker cannot use this information to presume the application data, and otherwise the encryption would have to be realized as part of SCTP in the kernel. However, in [31] it is suggested to use SCTP-AUTH to ensure the integrity of control chunks and DATA chunk headers. This is discussed in the following section.

6.2.2 Ensuring Order and Reliability

TLS ensures the order and reliability of TCP by maintaining internal sequence numbers for every record and using them for hash calculations. This approach cannot be used with SCTP because of streams and the possibility to send messages unordered. However, with DTLS these sequence numbers are part of the record header to allow hash calculation independently of the order of the messages, so no ensuring is done at all anyway. An attacker could tamper with SCTP's sequence numbers to manipulate the order of messages as shown in Figure 6.1.

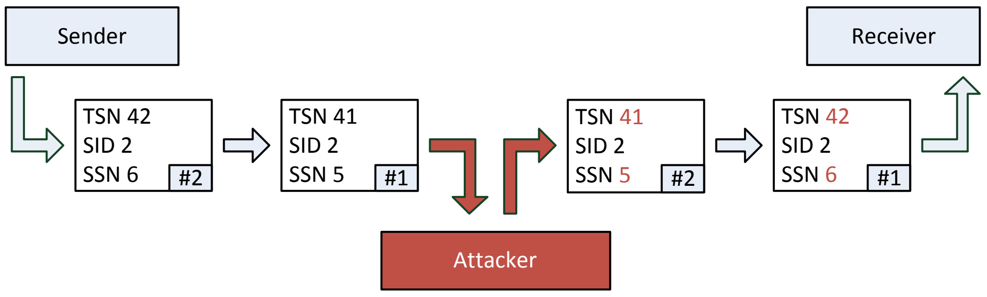


Figure 6.1: Switching TSNs for Reordering Attack

In [31] it is suggested to use SCTP-AUTH for DATA, SACK and FORWARD-TSN chunks to ensure their integrity, since DTLS cannot protect any parts of the transport protocol. Unfortunately, there is no reasoning why these chunks have to be protected. The DATA chunk is obvious, because an attacker must not be able to modify any sequence numbers or stream information to cause reordering or change the assigned stream of a message. Regarding the base specification, it is proposed to not protect other chunks, because their modification would not have an impact on the content to be transmitted. Requirements for SCTP extensions are discussed in Section 6.2.8. The key required for SCTP-AUTH can be derived from the negotiated master secret as described in [77].

6.2.3 Message Loss Prevention

DTLS does not rely on reliable transfer and can discard out of order messages. This is no problem with the unreliable UDP, but with SCTP a reliable service is expected. To avoid message loss, certain situations require in order delivery, for instance when application data might arrive after the epoch already changed and therefore the message became invalid.

Although SCTP offers in sequence transfer, this only applies to messages within a stream. Messages of different streams may be reordered and can still arrive late. To allow the use of multiple streams anyway, in [31] it is suggested to enforce the message sequence across streams in such situations. That is basically ensuring that all previous messages have been received, and continue on a single stream as long as necessary until application data can be transferred normally again.

It is proposed to maintain the message sequence across all streams by using the SENDER_DRY event notification we added to SCTP. This notification occurs as soon as every sent message has been acknowledged and there are no further messages pending or still in flight. Every time before protection against message loss is required, the SENDER_DRY event can be used to wait until there is no data left on any stream. Then sensitive messages can be sent on one stream only and the order is retained by SCTP. This ensures in sequence transfer, even if multiple streams are used.

Unfortunately, this does not cover all possibilities of reordering, because it does not ensure the in sequence reception of the messages for the application. There can still be messages of other streams in the receive buffer, which have not been read by the application when the first message after the SENDER_DRY event arrives. It may occur that the SCTP stack passes the latest message arrived to the application first, for example because it cycles through streams during delivery, so the messages are reordered again. Therefore, whenever a SENDER_DRY event is awaited on the sender side, it is proposed that the receiver has to read everything from the socket to empty the receive buffer first.

6.2.4 Renegotiations

A handshake can also be performed for an already established connection to renegotiate key material and cipher suite. This also changes the key for SCTP-AUTH, which is derived from every new key, except for the initial handshake, which is performed unprotected. Contrary to the initial handshake, application data has already been sent when renegotiating. This is critical when multiple streams are used, since data across different streams is likely to be unordered. If application data arrives after a ChangeCipherSpec, the key for SCTP-AUTH already changed and the packet gets dropped because its HMAC cannot be verified anymore.

To avoid such a message loss, it has to be ensured that there is no application data in flight when changing keys. This has been described in [31], although without specifying when the message loss prevention is necessary during a handshake. The new key material is used after the ChangeCipherSpec, so it is proposed that before sending it, a SENDER_DRY event should be awaited. For the client this is after sending the CertificateVerify and for the server before sending the ServerHelloDone, because after that the client continues and the server's next message is the ChangeCipherSpec. However, it is recommended that the client still sends its Finished with the old SCTP-AUTH key. The Finished is part of the same flight as the ClientKeyExchange, which is required for the key computation. Since messages of the same flight are sent virtually at the same time, the server may not be able to compute the new key before the Finished arrives. In this case, it will be discarded by SCTP-AUTH because the new key that has been used for it is still unknown, and a retransmission becomes necessary. Furthermore, to prevent application data from being processed after the ChangeCipherSpec in the socket, the server has to read all pending messages from its buffer after the CertificateVerify, and the client after the ServerHelloDone before continuing.

The server concludes the handshake with the Finished and resumes sending application data. Since the client has already computed the new key, it can be used with SCTP-AUTH for the Finished already. Unfortunately it cannot start sending application data immediately, because different streams can cause it to pass the Finished, which is a protocol violation. The client may then discard the application data or drop the entire connection, which has to be avoided.

It is advised that the server also waits for a SENDER_DRY event after sending the Finished, to ensure the client has received it before continuing with application data. Since there is also still a small chance that application data passes the Finished in the receive buffer of the client, it is also advised that the client buffers all application data arriving between ChangeCipherSpec and Finished, and processes it after the Finished has been read and thus the handshake is completed. The client cannot just read all data after the ChangeCipherSpec, because this would be a security risk. If the client would pass all messages to the application without having seen the Finished, an attacker could intercept the server's Finished to prevent the verification of the handshake with its hash value. Sending application data after manipulating the handshake may then be possible.

6.2.5 Shutdown

To gracefully shut down a DTLS connection, CloseNotify alerts are used. A peer announces that it finished sending data with the CloseNotify alert, but continues reading. The connection is shut down and the sockets can be closed once the other peer also confirmed to have finished sending. After receiving a CloseNotify, a peer will discard every following message arriving. This can lead to data loss when shutting down a connection, because application data arriving after the alert message will be dropped. This has to be avoided, so it is suggested to await a SENDER_DRY event before sending a CloseNotify alert. Additionally, all pending messages in the read buffer have to be processed at the receiver side, before changing the state to 'CloseNotify received'.

6.2.6 Session Resumption

DTLS supports session resumption like TLS, that is using the already negotiated parameters from an earlier connection. If the server still recognizes the client's session identifier, an abbreviated handshake can be performed and the cipher suite parameters of the former connection are used further on without negotiation.

This kind of handshake is not only shorter than the normal full handshake, the sequence of messages also differs in such a way that the client sends the last Finished. Hence, the precautions against losing data have to be changed, what has not been considered before. It is proposed that the client awaits a SENDER_DRY event notification before sending application data after the Finished in this case, while the server has to buffer messages arriving between ChangeCipherSpec and Finished. The server sends a single flight including the ChangeCipherSpec and Finished, so it is recommended to use the old key for SCTP-AUTH also for the Finished. Otherwise the client may not have the appropriate key computed in time.

This is sufficient if the abbreviated handshake is done for session resumption. If it is done just to refresh the key material while the connection remains established, more measures have to be taken. Application data has likely been sent before, which has to be kept from getting discarded because of arriving after the Change-CipherSpec message. Before the client sends the ClientHello message, it is advised to await a SENDER_DRY event to ensure no application data is in flight anymore. The server has to read all pending messages from the receive buffer and has to await a SENDER_DRY event before answering with the ServerHello message. Because the ServerHello message is followed by the ChangeCipherSpec and Finished, the client has to read all pending application data before both handshake messages can be processed.

6.2.7 Generic Adaptations

SCTP already performs retransmission and replay checks, so these have to be deactivated in DTLS, as suggested in [31]. Otherwise a lost message may be retransmitted twice, which causes unnecessary load on the network. This also solves the problem that DTLS may trigger a retransmission despite the message is only delayed because of a network congestion. Fragmentation is also provided by SCTP, so DTLS can make use of it and does not need to discover the Path MTU. It is proposed to achieve this by increasing the maximum message size for DTLS to 2^{14}

bytes, which is equal to the limitation of the record length caused by the encryption algorithms of TLS. Handshake messages are mostly smaller, so the fragmentation of DTLS will rarely be used.

6.2.8 SCTP Extensions

SCTP extensions also have to be considered for a solution with DTLS, in case they can pose a security risk. According to [31], the DATA, SACK, and FORWARD-TSN chunks have to be protected with SCTP-AUTH. The PR-SCTP extension, which limits the reliability, can be a possible target for attacks. An attacker could drop a message by intercepting it and sending a fake FORWARD-TSN chunk to the receiver, so it ignores the loss, as shown in Figure 6.2

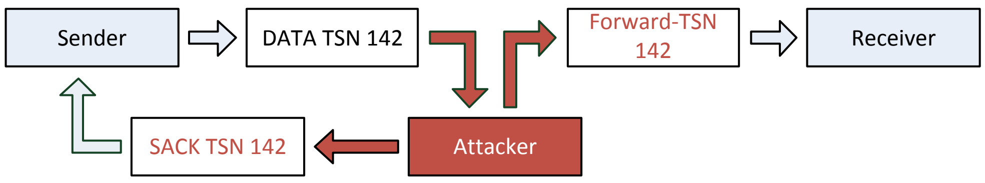


Figure 6.2: Message Drop Attack with PR-SCTP

To successfully drop a packet, an attacker would have to modify a FORWARD-TSN and a SACK chunk. The FORWARD-TSN chunk has to convince the receiver that it is ok that the message is missing and the modified SACK chunk has to contain the acknowledgement of reception for the sender, so it does not detect the loss and retransmits the message. With the protection of the FORWARD-TSN chunk this attack is already impossible, because although SACK chunks could still be modified, the receiver would still await the missing data and not continue before receiving it. Having SCTP-AUTH calculate an HMAC for every SACK chunk is therefore no security benefit and only costs performance. Therefore, it is proposed to only secure the FORWARD-TSN and DATA chunks with SCTP-AUTH.

The ADD-IP extension [101] adds or removes addresses of an established association. Multiple addresses are no issue with DTLS, so this extension can be used. Its ASCONF chunk, which is used to add or remove addresses for an established association, is protected by SCTP-AUTH by default, so DTLS does not need to take any measures. It also cannot be used for content modification or message dropping. The same applies to the Stream Reset extension [97] that just resets the SSN.

6.3 Implementing SCTP-aware DTLS

To evaluate the performance of SCTP-aware DTLS, an implementation is necessary. Since the OpenSSL toolkit contains the only open-source implementation of DTLS available at this time, this will be the basis for my further developments. In the following, the necessary adaptations to make the UDP-based implementation aware of SCTP are described.

6.3.1 Existing Implementation

The implementation of DTLS in OpenSSL has been introduced with release 0.9.8 [61]. While some improvements have been made in the security updates until 0.9.8i, the state of the implementation was still rather experimental than productive. Therefore, the development of an SCTP-aware prototype required at first fixing numerous issues with the base implementation, before the new features could actually be added. My fixes have been included in the recent stable release 1.0.1, as well as in 0.9.8 and 1.0.0 with the latest security updates [53]. The development for an SCTP-aware implementation was done for 1.0.1, which was the next stable release. Modifications were necessary for BIO and SSL objects, as illustrated in Figure 6.3.

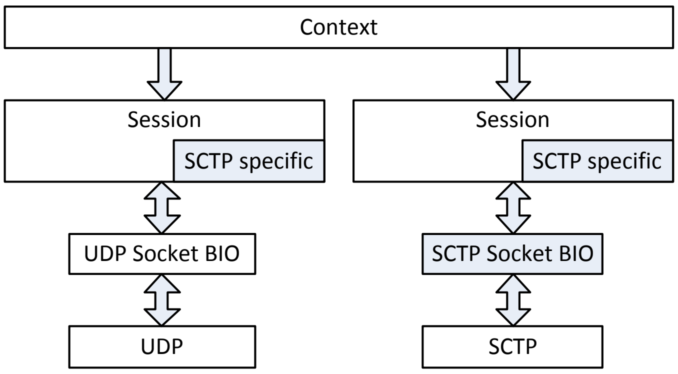


Figure 6.3: Modifications to OpenSSL to support DTLS-aware SCTP

6.3.2 BIO Object

The BIO objects are used to provide an abstraction layer of the underlying socket to an SSL object, which implements the security protocol layer of OpenSSL. For DTLS, a new BIO object type to support UDP-specific features has been introduced. Accordingly, as part of my work, an additional object type for SCTP to handle SCTP-specific features has been created. This object has to activate SCTP-AUTH for DATA and FORWARD-TSN chunks. It also has to provide methods for the SSL object to add keys derived from the master secret and their activation and deactivation, respectively. Also necessary are methods to await a SENDER_DRY event, to read everything pending in the receive buffer, and to ignore all stream information set by the application to enforce the in order transfer on a single stream during handshakes. Since the application uses the API calls provided by OpenSSL to send and receive messages instead of system calls, an interface to set and read stream information and the PPI have to be provided. SCTP supports notifications that are passed to the application with read calls and the MSG_NOTIFICATION flag set. The BIO object has to recognize notifications and avoid passing them to the SSL object, as shown in Figure 6.4, because they would be handled as a corrupt record message and therefore discarded. The BIO object should rather allow the application to access the notifications by adding callback methods the application can register and which are called whenever a notification occurs.

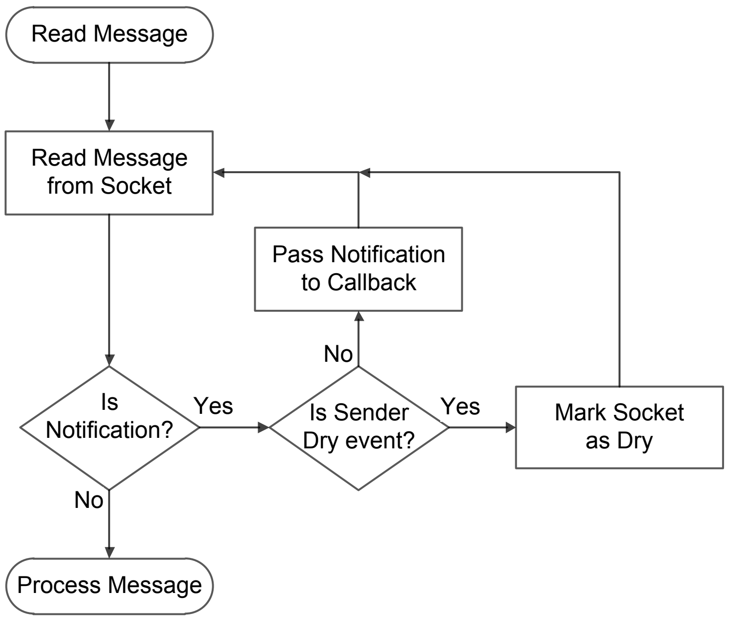


Figure 6.4: Read SCTP Message Flow Chart

6.3.3 SSL Object

The protocol implementation of DTLS is realized with protocol-specific functions used by the SSL objects. In the course of my work they have been extended to check the type of the underlying BIO object and in case of SCTP behave accordingly. Whenever a new master secret has been negotiated, a key for SCTP-AUTH has to be derived and passed to the BIO object. After sending the ClientVerify or before sending the ServerHelloDone, the SENDER_DRY event has to be awaited and everything still in the receive buffer has to be processed. The beginning and completion of a handshake have to be notified to the BIO object, so that the use of a single stream can be enforced. The replay check, handshake message fragmentation as well as retransmission timers have to be deactivated.

6.4 Performance Evaluation

To measure the performance of SCTP-aware DTLS, multiple series of measurements have been taken. The setup consisted of two identical hosts connected by 1000 MBit/s links with an MTU of 9000 bytes over a switch, illustrated in Figure 6.5. Each host was equipped with a Core 2 Duo 3.3 GHz CPU and running FreeBSD 10.0 (CURRENT). To avoid unpredictable scheduling effects with two CPU cores, one of them was disabled at first, but the measurement was repeated with both CPU cores enabled for comparison. In each measurement, one host sent as many messages as possible, while the other received them and recorded the throughput. The duration of the measurements was always 60 seconds and the length of the messages was constant for a single measurement, but increasing in a series. To be able to calculate 95% confidence intervals, each measurement has been repeated 25 times.

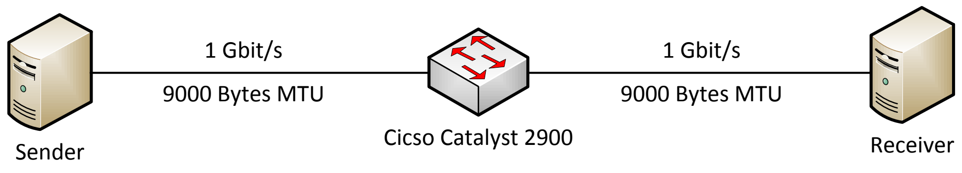


Figure 6.5: Measurement Setup

These measurements have been chosen to determine the impact of DTLS on the possible throughput the hosts can handle and how the additional computing power necessary for the security features slows the connection down. Different network conditions or certain features of SCTP like PR-SCTP, multi-homing or multiple streams were ignored. While just transferring data, DTLS does not care about reordering or message loss, so it has no influence and only the behavior of SCTP would be relevant in these scenarios.

6.4.1 Single Core Throughput Measurements

The first measurements were done with standard (unsecured) SCTP and TCP connections as a reference, illustrated in the both topmost graphs in Figure 6.6. Since modern CPUs were used, enough computing power was available to fully load the link, which limited the throughput. The throughput was measured for the application layer, and can be calculated for SCTP according to [33] by considering the overhead of the IP and SCTP headers and of SCTP's message bundling feature. With the given link, this resulted in a maximum throughput of about 995 MBit/s or 121,527 KBytes/s. For very small messages, however, SCTP was less efficient than TCP, because of its message-orientation, so every single message requires its own DATA chunk with an eight bytes header, resulting in more overhead per packet.

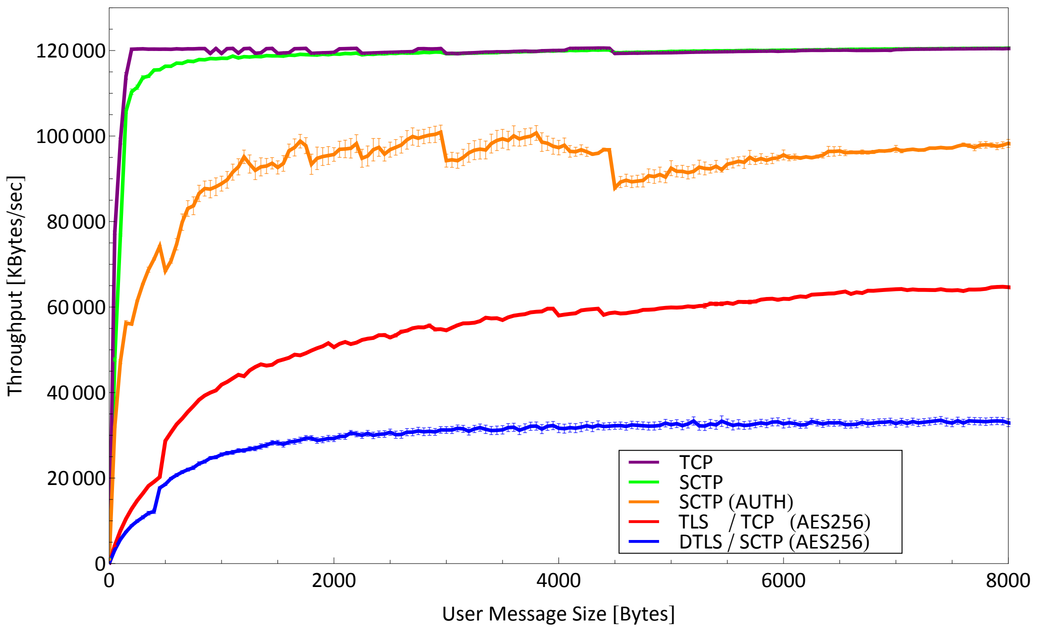


Figure 6.6: SCTP-aware DTLS and TLS / TCP (Single Core)

The next step was to activate SCTP-AUTH for DATA chunks to determine the impact on the performance of this solution to ensure the integrity. The slightly lower graph in Figure 6.6 shows that the use of SCTP-AUTH already reduces the throughput by about 20%. This is mainly because the data was not just passed to the network interface but also processed in advance, in this case to calculate an HMAC and add the AUTH chunk. The varying throughput up to 4500 bytes message size was due to the message-orientation of SCTP in combination with its bundling feature [33]. As many small messages are bundled into one SCTP packet as possible, and the drops indicate the message lengths where one message less can be bundled – which decreased the overall throughput by resulting in smaller packets and more overhead. This also constantly changed the amount of data the HMAC had to be calculated over, and resulted in a varying CPU load to process a packet. At 4500 bytes message size only a single one could fit into a packet, using only about 50% of the possible packet size. Therefore, the throughput dropped and continued to increase steadily with the increasing message size, because the number of messages in a packet could not be increased anymore. A fully secured DTLS connection, represented by the lowest graph, reduced the throughput again to about 25% of an unsecured SCTP connection. The steep increase at 512 bytes message size is because OpenSSL uses a different algorithm for larger messages as an optimization.

Finally, TLS over TCP was added for comparison, which was significantly faster than DTLS over SCTP. This was because TCP requires less computing power and there was no calculation of AUTH chunks necessary.

6.4.2 Dual Core Throughput Measurements

Modern CPUs usually have at least two cores, so measurements with both CPU cores activated were interesting, too. Using multiple cores had no noticeable effect when the application did not consume any significant computing power and only passed data to the SCTP socket. This also applied to SCTP-AUTH, which is processed in the kernel. When DTLS was enabled, the application needed much more computing power to perform the encryption that can be done on the other core. This resulted in a slightly higher performance with dual cores as seen in Figure 6.7. So when more than one CPU core was available, SCTP-aware DTLS came closer to TLS over TCP, because of the additional computing power.

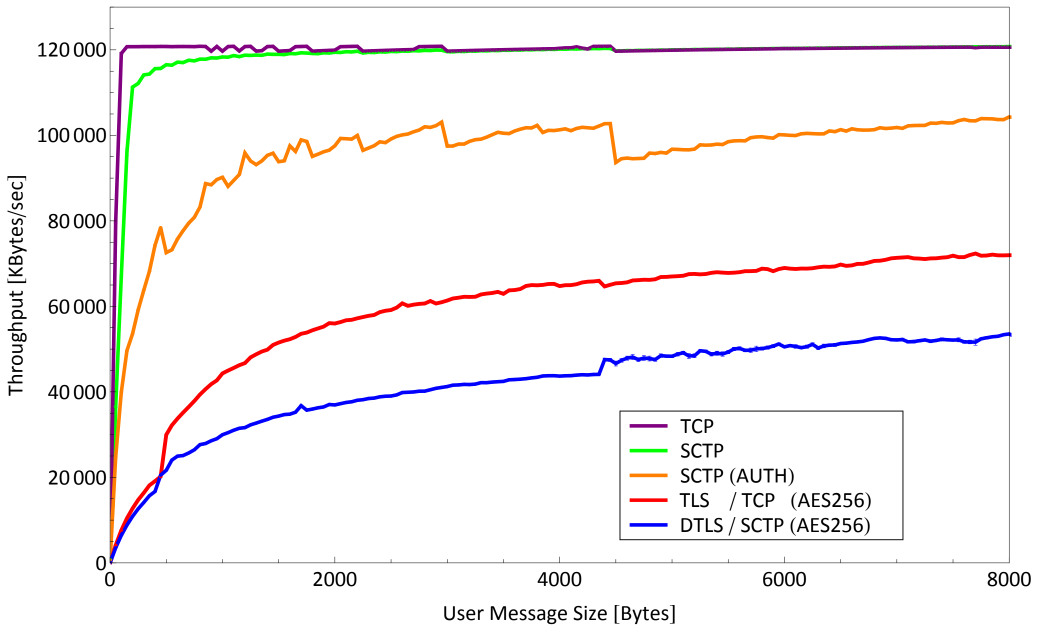


Figure 6.7: SCTP-aware DTLS and TLS / TCP (Dual Core)

6.4.3 Renegotiations

Because of the inevitable synchronization of handshake messages, renegotiations may cause a slowdown of the data transfer. If there is a high amount of data to be sent on multiple streams, all of them have to be drained before the handshake can be performed successfully.

To determine the impact of renegotiations, at first the duration of a full handshake had to be identified. The timing of the messages is listed in Table 6.1. Since the actual timing depends on the latency of the network, the timings are given in multiples of the Round-Trip Time (RTT). An RTT is the time necessary to send a message to the other endpoint and receive its response, therefore it is equal to twice the delay of the network. According to Table 6.1, it takes four times the RTT until application data is sent again.

Time [RTT]	**Msssages**
0	ClientHello
0.5	ServerHello, Certificate
1.0	SACK (Sender Dry)
1.5	ServerHelloDone
2.0	ClientKeyExchange
2.5	SACK (Sender Dry)
3.0	ChangeCipherSpec, Finished
3.5	ChangeCipherSpec, Finished
4.0	Application Data

Table 6.1: DTLS Handshake Timing

The maximum throughput that is possible depends on multiple factors. At first, and obvious, it is limited by the link speed, which was 1000 MBit/s, resulting in about 120,000 KBytes/s for the application (compare Section 6.4.1) in the scenario used to measure DTLS. The next limiting factor for the throughput is the CPU. According to Section 6.4.2, the maximum throughput of an SCTP association secured with DTLS was about 53,000 KBytes/s on the hardware used. The last limiting factor is the link delay in combination with the Flow Control. The overall delay of the link is considered, so the specific reason for it, for example a long distance link or queueing effects, can be neglected. The Flow Control prevents the overloading of the receiver by maintaining a window size that equals the amount of data the sender is allowed to send. With an increasing link delay, a larger window is necessary, because it takes longer to acknowledge the received data and increase the window again. This dependency can be described with $T = \frac{Window}{RTT}$, with T being the possible throughput. Since the maximum window size is usually fixed, a large delay can therefore also be a limiting factor. With the CPU limit and the default window size of 1820 KBytes, the maximum delay that does not affect the bandwidth can be calculated as in Equation 6.1, according to Little's Law.

$$T = \frac{Window}{RTT} \quad \Rightarrow \quad RTT = \frac{Window}{T} = \frac{1820}{53,000} = 0.034\ s = 34\ ms \tag{6.1}$$

As a consequence, for delays smaller than 34 ms the throughput was limited by the CPU, while otherwise the delay became the limiting factor. This allows to calculate the expected throughput with a given delay and a given number of renegotiations performed per minute. The duration of a handshake is four times the RTT, so with n renegotiations performed per minute the time spent renegotiating is $\frac{n}{60} * 4 * RTT$ per second. During this time, no data transfer is allowed. This allows to calculate the throughput T_n for n renegotiations per minute, depending on the RTT, and with the given CPU limit and window size as in Equation 6.2.

$$T_n = T * (1 - (\frac{n}{60} * 4 * RTT)) \quad \text{with} \quad T = \begin{cases} 53,000 & \text{for } RTT \leq 34ms \\ \frac{1820}{RTT} & \text{for } RTT > 34ms \end{cases} \tag{6.2}$$

Figure 6.8 shows graphs for the delays 0 ms, 5 ms, 25 ms, and 50 ms. Without any delay, the renegotiation takes no time to complete, and therefore does not affect the achieved throughput. With 5 ms delay the CPU is still the limit, but an increasing number of renegotiations has a noticeable effect already. The delays 25 ms and 50 ms result in an RTT larger than 34 ms, so the maximum throughput is reduced even without any renegotiation. Due to the longer time it takes to complete

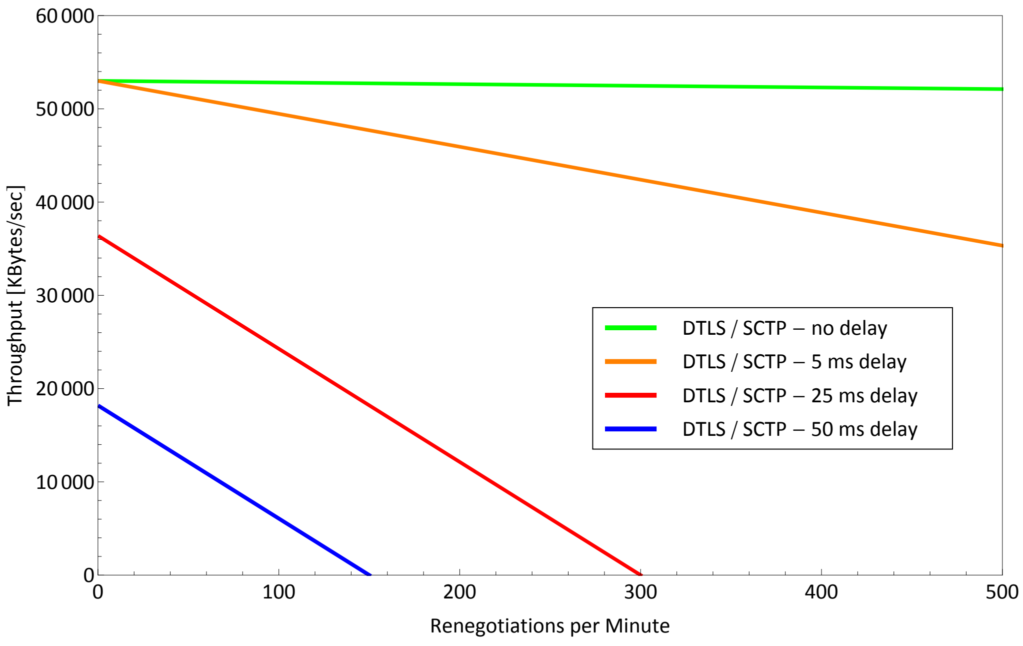


Figure 6.8: Expected Throughput with Renegotiations

a handshake with a larger delay, the impact of renegotiations is quite significant. With 150 renegotiations per minute there is no throughput possible anymore with a delay of 50 ms. With 25 ms delay at least 300 renegotiations per minute are possible until no time is left for data transfer.

The maximum number of renegotiations per minute can also easily be calculated as in Equation 6.3. Hence, with 5 ms delay no throughput would be possible anymore with 1500 renegotiations per minute.

$$n_{max} = \frac{60}{4 * RTT} \quad \text{with} \quad RTT > 0 \tag{6.3}$$

To confirm the validity of the calculations above, the throughput depending on the number of renegotiations per minute with the same delays of 0 ms, 5 ms, 25 ms, and 50 ms has also been measured. The results can be seen in Figure 6.9, and are exactly as predicted with Little's Law. As a consequence, many renegotiations can reduce the throughput significantly, in particular if the link has a large delay, as characteristic for cell phone networks or intercontinental connections for instance.

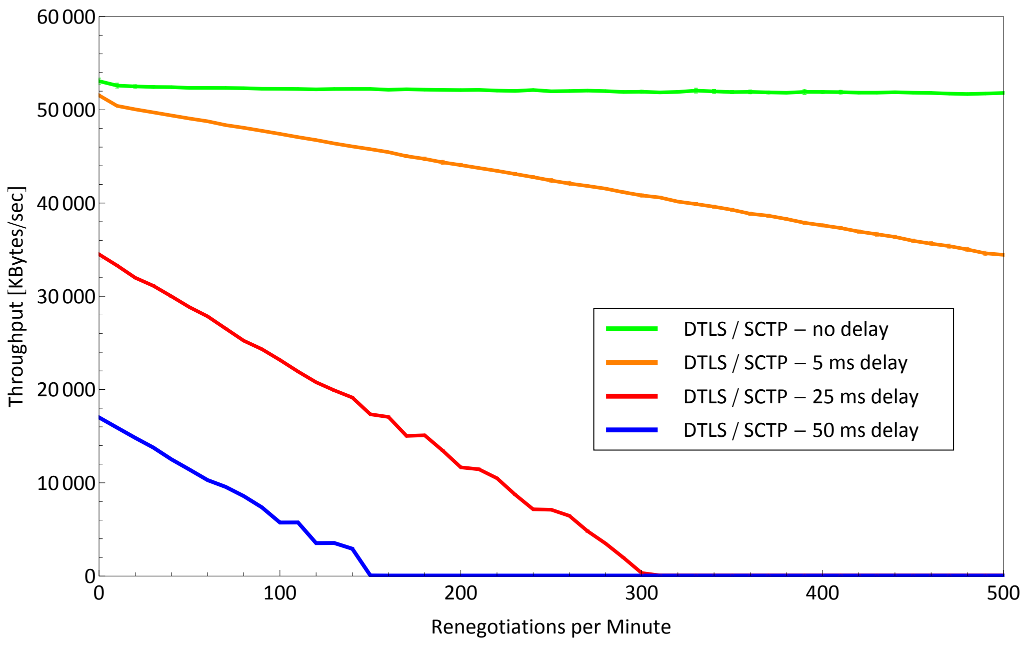


Figure 6.9: Measured Throughput with Renegotiations

6.5 Optimizations

With the modifications of DTLS identified in the previous sections, it can be used for SCTP-based applications without limiting the possible features. However, enforcing the order during handshakes can reduce the performance significantly, as shown in Section 6.4.3. Additionally, having an HMAC calculation done twice, once by DTLS and again by SCTP-AUTH is also detrimental to the performance. Therefore, although SCTP-aware DTLS is a viable concept, there is still room for optimizations.

6.5.1 Handshake Message Synchronization

A problem besides the draining of the data transfer on all streams for a handshake is that the receiver, according to the SCTP standard, waits 200 ms for another message to arrive before sending an acknowledgment. This prevents sending selective acknowledgments (SACKs) for every single message. In the case of DTLS, the next message while handshaking may not follow until the previous one has been acknowledged because of the SENDER_DRY event. If that occurs, the receiver always waits the full 200 ms period before sending its SACK to acknowledge the previous message, and the handshake can continue.

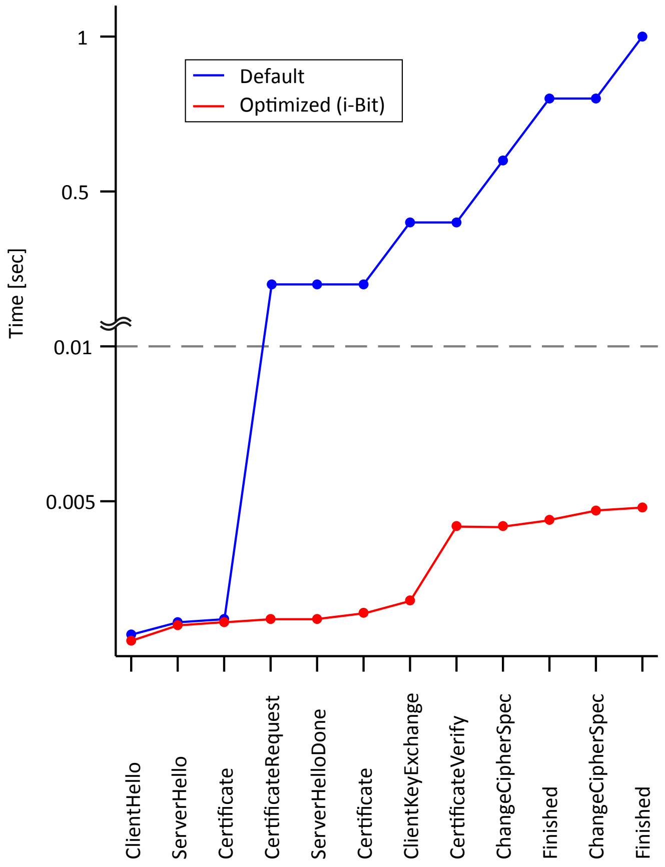


Figure 6.10: DTLS/SCTP Handshake with SACK Immediately Optimization

This problem can be mitigated with the SACK Immediately extension [104]. It allows the sender to set a so-called I-bit to request the receiver to acknowledge the respective data immediately, regardless of any timer or other condition. It is, therefore, proposed to set this I-bit for every handshake message, if available in the current implementation. This is done by the OpenSSL implementation, so the impact can be measured by tracking the time elapsed since the handshake has been initiated for each handshake message. Figure 6.10 shows the resulting graphs for a default handshake and again with the SACK Immediately extension. The 200 ms delays because of the delayed acknowledgements are clearly visible for the default handshake and resulted in a total time of about 1 second until the handshake has been completed. The same handshake, optimized by setting the I-bit for every handshake message, only took about 5 ms to finish, because no delays had to be awaited and the messages followed each other almost instantaneous.

6.5.2 Avoiding Duplicate HMACs

A further optimization is to define cipher suites with NULL_HMAC algorithms, that is using no separate HMAC for DTLS. DTLS does not need to calculate its own HMAC to ensure integrity, because this is done by SCTP-AUTH for the entire message anyway. Although the HMAC is now calculated after encryption and is not encrypted itself anymore, this does not affect security because an attacker still cannot alter any data without recalculating the HMAC, which is impossible without knowing the secret key. This is expected to reduce the necessary computing power for every packet.

The result with one CPU core can be seen in Figure 6.11, SCTP-aware DTLS became much faster and had almost a comparable performance to TLS over TCP. The difference narrowed even more when two CPU cores were used, as shown in Figure 6.12. This proved that this optimization is a significant improvement for the performance and makes SCTP secured with DTLS a viable alternative to TLS with TCP. The DTLS measurements showed a gap at 4500 bytes message size, which was again caused by SCTP's message-orientation. Only a single one fit into a packet when the message size was 4500 bytes or larger. The OpenSSL algorithms used for encryption and hash computation were more efficient with fewer but larger message sizes, hence the increase for the unoptimized measurement. The HMAC algorithm of SCTP-AUTH, however, is less optimized and performs better with multiple smaller messages. Therefore was the throughput reduced with SCTP-AUTH only, as well as for optimized DTLS without the second HMAC calculation.

6.5.3 Allowing Multiple Epochs

When renegotiations are frequently performed, an epoch usually does not last very long, so as another optimization, messages of the previous epoch can be accepted as well. The DTLS specification allows the last epoch still to be used for up to 120 seconds. Unfortunately, this does not guarantee that there will not be any messages

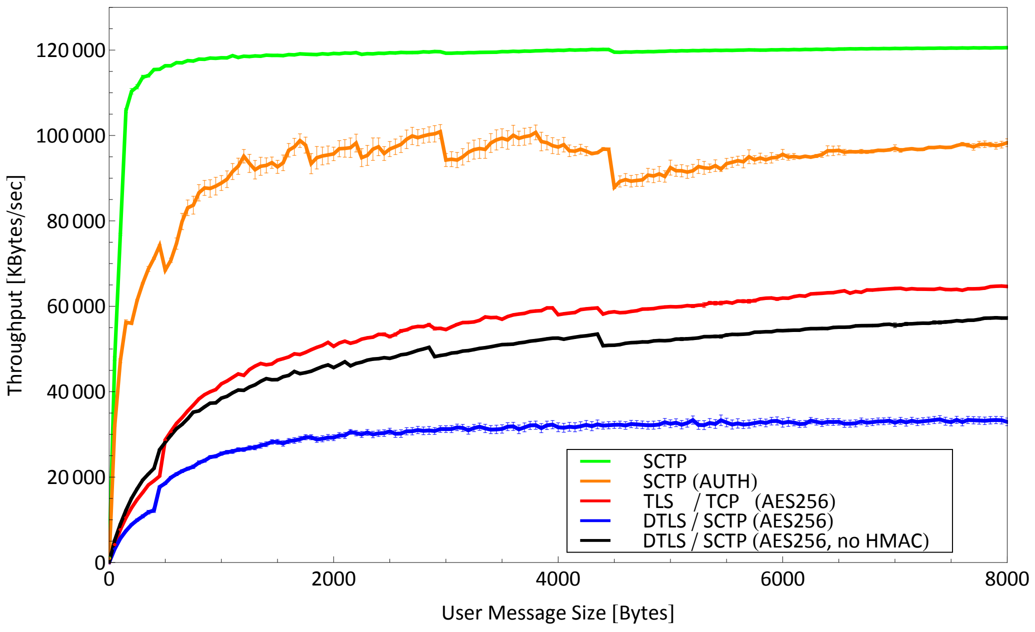


Figure 6.11: SCTP-aware DTLS with HMAC Optimization (Single Core)

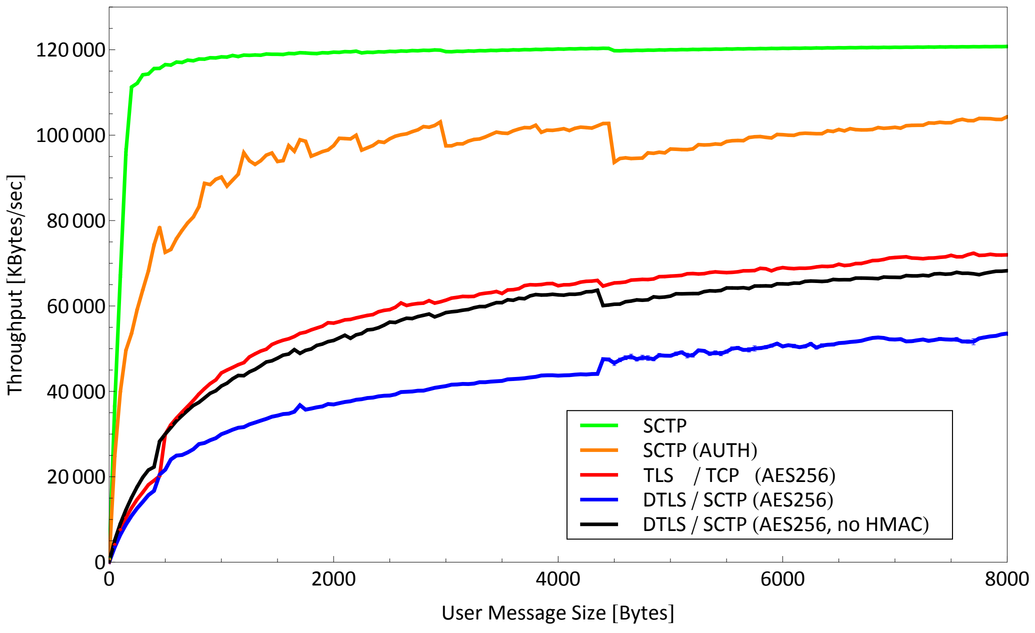


Figure 6.12: SCTP-aware DTLS with HMAC Optimization (Dual Core)

arriving even later and will be discarded as discussed in Section 6.2.3. Since message loss subverts the reliable transfer of SCTP, without any other measures, this is an unacceptable approach.

To avoid message loss, the previous one or more epochs can be always acceptable. Before handshaking, it has to be ensured that all messages of the allowed previous epochs have been received. The more epochs are still accepted, the higher is the probability that all past messages have been received and no delay is necessary. However, the more epochs are accepted, the higher is the risk of a successful attack. Therefore, as few epochs as possible should be permitted, and this optimization is not recommended for connections with rare renegotiation. Especially the first epoch for connection establishment is critical since it is completely unsecured. Hence, application data sent with epoch 0 parameters must not be accepted at any time.

6.6 Conclusion

In this chapter, SCTP-aware DTLS has been proposed as a new security solution for security-aware applications. The basic idea in [31] has been extended with respect to some crucial aspects, for example regarding message loss prevention. The change of key material and association shutdowns are critical with respect to data loss, because SCTP messages on different streams do not have to be kept in sequence. The suggested solution uses SENDER_DRY event notifications to drain the data transfer.

Performance measurements proved that an optimized SCTP-aware DTLS can be almost competitive to TLS over TCP. The presented solution based on DTLS allows to use encryption and authentication with SCTP features fully supported without security issues but with a reasonable performance. It can be realized in a user-space library, such as OpenSSL. This has been done and the maintainers of OpenSSL have accepted the implementation described in Section 6.3 and integrated it into stable release 1.0.1 [61]. The adaptations have also been introduced to the IETF for standardization, which has accepted and published them in RFC 6083 [105].

Chapter 7

DTLS-based Tunneling

One possibility to secure security-agnostic SCTP applications is to tunnel the network protocol (IP) underneath the SCTP associations over DTLS. This is done with Cisco's VPN software and discussed in Section 5.2.1. The unreliability of DTLS provides an efficient transport for the likewise unreliable network protocol, without adding unnecessary retransmissions. DTLS is a protocol that has almost no requirements to its lower and upper layers that have to be considered, so it can rather easily be extended with new features to increase the efficiency of the transfer even further. The extensions proposed in this chapter have been published in [90].

7.1 General Considerations

DTLS has been specified for the use with unreliable and connectionless transport protocols, but is session-oriented itself. That causes the issue that despite the transport protocol being connectionless, a handshake is necessary for the establishment. It has to be determined if the peer is still available or became unreachable, so the session must be closed. Additionally, if there has been no data transfer for a certain time, intermediate firewalls or NAT routers may discard the corresponding state that opened the necessary ports. Therefore, a mechanism used periodically to check the peer's availability and to keep the state of possible middleboxes is required. The only mechanism DTLS offers is to perform a handshake for renegotiation, which is a costly operation. A more efficient keep-alive mechanism would have to be implemented for the application protocol.

Furthermore, UDP is a message-oriented protocol, so determining the maximum possible message size is crucial. If the Maximum Transmission Unit (MTU) is assumed too high, then intermediate routers may discard messages, or IP fragmentation will be used. If it is assumed too low, the transmission is less efficient than actually possible. To address this issue, the DTLS specification requires performing a Path MTU Discovery [52] to determine the maximum message size each router on the path between the peers can handle. Unfortunately, unlike the also message-oriented SCTP, UDP has no mechanism to perform such a discovery.

Another generic problem with tunneling is the lack of mobility. With modern mobile devices having multiple network interfaces, such as Wi-Fi and Universal Mobile Telecommunications System (UMTS, 3G) [85], the points of attachment frequently change, which at the current state always terminates every connection and all have to be reestablished. Although there are several approaches to this issue, as discussed in Section 2.6, because of various drawbacks none has been widely deployed so far.

7.2 Heartbeat Extension

Due to the connectionlessness and lack of a keep-alive mechanism of UDP, the availability of the peer can only be determined by performing a handshake or implementing a keep-alive mechanism for every application protocol. To provide an application protocol independent mechanism, an extension for DTLS that provides the required functionality is suggested here. A request and response protocol is most suitable to realize a keep-alive method, because each peer can send requests periodically and await the response. Therefore such messages have to be added with a new extension and since keep-alive is also known as heartbeating, the extension is named the Heartbeat extension. It consists of the two mentioned messages, which are called HeartbeatRequest and HeartbeatResponse accordingly. Upon reception of a HeartbeatRequest, a HeartbeatResponse must immediately be sent as the answer. A timer similar to the handshake timers is started for every request, and if no response has been received until the timer expires, the HeartbeatRequest is retransmitted. This is done until the maximum number of retransmissions is reached and the connection is closed. Requests can be sent at any time during the connection lifetime, except when a handshake is in progress, because the handshake and its timers will perform the same task.

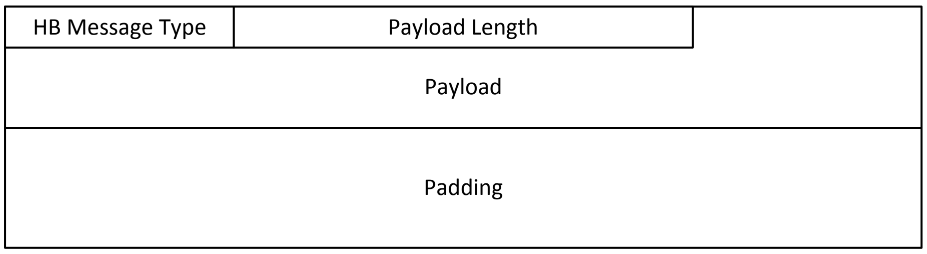


Figure 7.1: Heartbeat Message

The format of the new messages can basically be arbitrary, because for keep-alive no further information than the message type is necessary. However, to make the extension as versatile as possible, an arbitrary payload and a random padding is preferred, illustrated in Figure 7.1. The payload of the HeartbeatRequest can be chosen by the implementation, for example simple sequence numbers or something more elaborate. The HeartbeatResponse must contain the same payload as the request it answers, which allows the requesting peer to verify it. This is necessary to distinguish expected responses from delayed ones of previous requests, which can occur because of the unreliable transport. The padding, on the contrary, always has to be of random data and is not sent back. The length of the padding is arbitrary, but should always be 16 bytes or more for security reasons, to prevent statistical attacks in case the payload is predictable. This can be the case if an implementation only uses sequence numbers for the payload of its requests. Without additional random bytes this payload can be guessed easily, thus making it prone to a Known-Plaintext Attack (KPA) [94].

The support of the extension is negotiated with Hello-Extensions, an additional parameter for the ClientHello and the ServerHello, depicted in Figure 7.2. This is done for backward compatibility in case one of the peers does not yet support this protocol extension. The Hello-Extensions also contain a Mode field to either allow or forbid the peer to use this protocol extension. This can be used to avoid continuous requests by the server that could drain the battery of mobile devices, for instance.

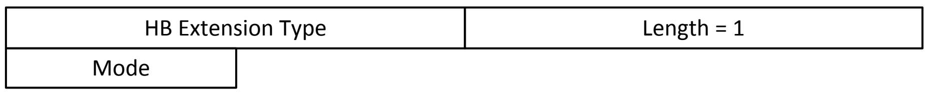


Figure 7.2: Heartbeat Hello-Extension

The Heartbeat Protocol can be used to test if the peer is still alive, by sending a HeartbeatRequest periodically. If no HeartbeatResponse has been received after a certain time, the peer can be considered unreachable and the DTLS session can be closed. The protocol can also be used to perform a Path MTU Discovery, as described in the next section.

This extension is also crucial for DTLS Mobility, introduced in Section 7.4.

7.3 Path MTU Discovery

A generic mechanism to perform the necessary Path MTU Discovery for DTLS is preferred to having every application implement their own. The proposed Heartbeat extension can be used to realize a Path MTU Discovery for packet-based protocols, as described in [48]. Only the payload of a HeartbeatRequest is returned in the response, so varying the padding of HeartbeatRequests, while keeping the padding of HeartbeatResponses at minimum, can be used to determine the maximum message size of one direction of the current path.

The Path MTU is the minimum MTU of all involved devices on a path, therefore it can never be larger than the MTU of the local network interface. Hence, trying to send a HeartbeatRequest of the size of the local MTU is a good start. The message has to be sent with the *Don't Fragment* bit to disable automatic IP fragmentation. This is not necessary with IPv6, which does not allow intermediate routers to perform IP fragmentation anyway. If the HeartbeatResponse is returned, the path can handle messages of this size and no further actions are necessary. If any of the devices on the path signals a *Fragmentation needed* error via the Internet Control Message Protocol (ICMP) [68], the Path MTU is smaller than the local one. The ICMP message can contain the MTU of the relevant device, which then should be tested next. However, to read ICMP messages administrator privileges are usually required. Most applications are run with lesser privileges, to limit the possible damage in case they become compromised, so this information may not be available. Security issues with ICMP messages in general are discussed in Section 7.4.4. If an ICMP message contains no MTU value or cannot be read, there is only a limited number of possible MTU values, so it is more efficient to keep a list of these values and continue with the next smaller one instead of decreasing the request message size byte by byte. Table 7.1 shows a list of some common values. If ICMP is disabled or the message was lost, a too large HeartbeatRequest will simply be dropped without any notification. In this case a timer has to be started every time a request is sent. If it expires because of no response, the discovery can be continued with a decreased request size. This process is illustrated in Figure 7.3.

MTU	**Protocol**
65535	Maximum specified by RFC 791
65535	Hyperchannel
17914	IBM Token Ring
9000	Ethernet Jumbo Frames
4352	FDDI
2048	Wideband Network
1500	Ethernet, PPP
1492	IEEE 802.3, PPPoE
1460	L2TP
1372	PPTP
576	x.25, SLIP
68	Minimum specified by RFC 791

Table 7.1: Some Possible MTU Values

As soon as a HeartbeatResponse has been received, the Path MTU has been found and can be used. Since it can be different for each direction of the path, both peers have to determine this value independently. The Path MTU Discovery should be performed periodically, in case the routing and therefore the path between the hosts changes.

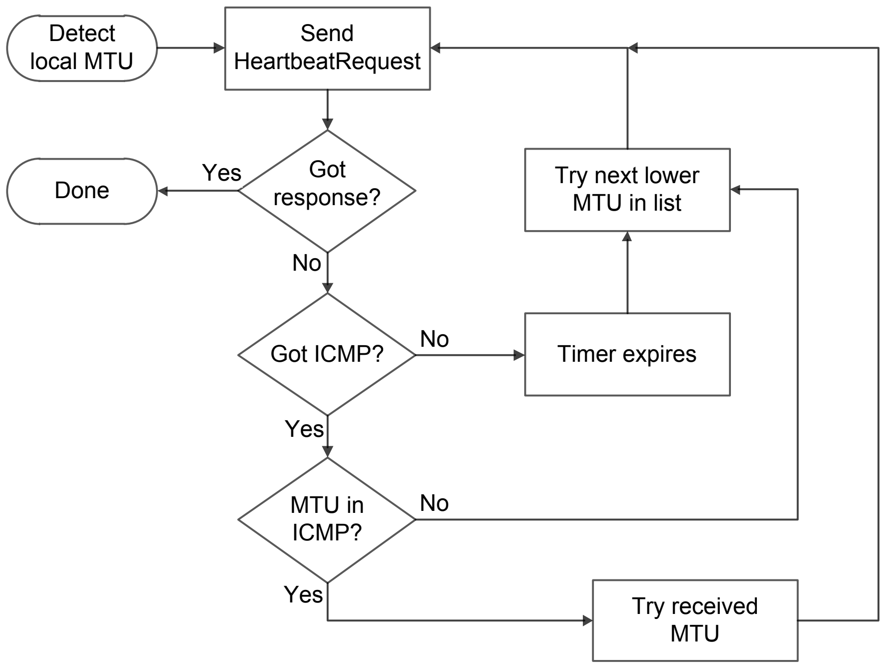


Figure 7.3: Path MTU Discovery Flow Chart

7.4 Mobility Extension

Several solutions to provide mobility for end-to-end connections have been introduced in Section 2.6. Because of the drawbacks of the present approaches, the requirements for a more easily deployable solution will be analyzed in the following and an approach based on DTLS will be introduced. A rough draft has been described in [109], yet lacking any security mechanisms and other details necessary for an actual realization. DTLS Mobility offers a secure connection based on UDP and can be used directly for data transmission, or as a tunnel for the actual application protocol. It can be implemented in a library and deployed together with the application that wants to make use of it. No infrastructure support is required, and middleboxes performing NAT are supported.

7.4.1 General Considerations

Whenever a host changes its point of attachment, it should be able to keep its connections alive and continue the communication without reconnecting. This can be achieved either by an infrastructure that knows the location of a device and routes the data to its address accordingly, or by the hosts themselves, by being able to handle changing addresses. The problem with mobility-aware infrastructure, like cell phone networks, is that the connection is still lost as soon as another network with a different infrastructure, like a Wi-Fi network, is joined. Hence, only possible solutions that allow mobility across different kinds of networks will be considered.

An essential requirement for handling changing addresses is a unique identifier, because the address obviously cannot be used for the distinction of multiple connections anymore. This identifier only has to be unique per direction, so both peers can either use the same or different identifiers for the same connection.

The notification of an address change can be either explicit or implicit. An explicit notification uses a designated message carrying the new address, while with an implicit notification the receiver determines the address change by a different source address of a specific or an arbitrary message. The advantage of explicit notifications is that within a secured connection the notification messages will be secured as well, while with implicit notifications additional security measures have to be taken. They are necessary because the source address is usually unprotected and can easily be modified by an attacker. However, the explicit notification requires knowledge of the current address, that is it needs layer 3 and 4 information. This is not only a violation of the layer model, it also does not work with NAT, because a middlebox performing NAT will overwrite the source address of the endpoint, but not the addresses that may be carried in the payload of a message. Even a customized NAT implementation would not be able to modify the payload because of the encryption and integrity checks that are used. Finally, the address change has to be known before it occurs in order to send a notification, which is impossible to realize in most circumstances.

Therefore, it is suggested here to use implicit notifications, either with a designated message or an arbitrary one. Using a specific message has similar issues as the explicit notification, since the host must be aware of the address change in order to send the notification. This is not the case with NAT middleboxes, which are quite common in today's networks. So the only reasonable solution is to use an implicit notification with all messages, although this requires an additional mechanism to secure the address change. The unprotected source address can easily be modified by an attacker, which can be used to drop the connection with an invalid source address or to perform a Denial-of-Service attack. The attacker establishes a regular connection, requests a large amount of data and changes the source address to the victim's one, so all the data will be sent to the victim.

An address change likely results in a different path to the peer, so a Path MTU Discovery has to be performed to determine if the MTU has changed. This prevents packet losses and less efficient transfer, respectively.

A new approach introduced in the next section is to use DTLS for mobility as discussed above. The benefit of integrating mobility into a security solution is that applications already having secured connections can be mobile without additional measures. This is particularly important for mobile devices where resources are limited. Since DTLS is designed for unreliable and connectionless protocols, it can use any given address for a connection. Therefore, the only issue is to verify an address change before the transmission can continue with the new destination. DTLS is implemented in a library, like OpenSSL, so if an application wants to use mobility, it just needs to include the library. These advantages, especially the easy integration, make DTLS Mobility a promising solution to be actually deployed. An overview of the features of this suggestion compared to existing solutions is given in Table 7.2.

	Mobile IP	MSOCKS	TCP-R	SCTP	MP-TCP	DTLS Mobility
No specific infrastructure required	-	-	+	+	+	+
Operating system independent	-	+	-	-	-	+
Deployable as part of application	-	+	-	-	-	+
NAT support	- [1]	- [1]	-	- [2]	+	+
Reliable transfer	-	+	+	+	+	-
Unreliable transfer	+	-	-	+ [3]	-	+
Multi-homing	-	+	-	+	+	-

[1] explicit configuration of NAT required - [2] UDP encapsulation and ADD-IP extension required - [3] PR-SCTP extension

Table 7.2: Comparison of Different Approaches to Mobility

7.4.2 Concept

To extend DTLS with mobility, an identifier is required to map an incoming message to a specific connection even if the source address changes. This can be achieved by adding a new Connection Identifier (CID) to the Record header, as illustrated in Figure 7.4. A length of 32 bit is considered appropriate here, but it could be extended if necessary. The CID is placed before the length field to allow network packet analyzers to distinguish between regular and extended headers more easily. In case the analyzer has only seen an excerpt of a connection or only single messages, it may not know if the extended header is used and thus a heuristic is necessary. This can be done by verifying the Length field of the regular Record header. If the value of the field does not match the length of the message, then the extended header is probably used, with the actual length field in a different position due to the CID. We implemented this strategy for the network packet analyzer Wireshark [110] and tests proved it to be very reliable.

For backward compatibility, the modified header can only be used after both peers have announced their mobility support. This can be done by adding a Hello-Extension to the ClientHello and the ServerHello, respectively. If both support the extension, the modified header can be used as soon as the sending side changes,

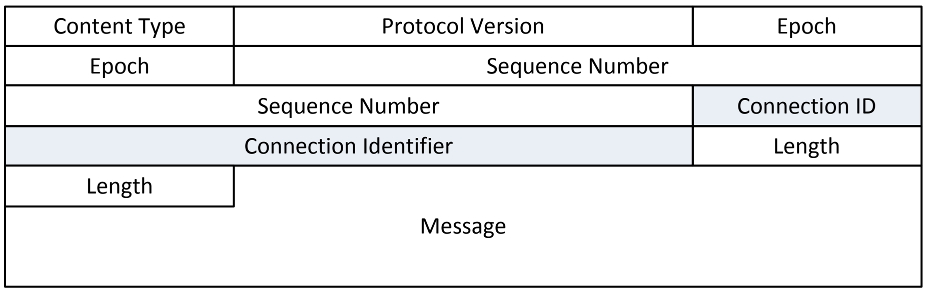


Figure 7.4: DTLS Record Header with Added Connection Identifier

that is after the ClientHello for the client and after the ServerHelloDone for the server (compare Figure 3.9).

The CID of the modified header has to be unique for each connection on both sides. Since the peers cannot know which values are already in use on the opposite side, the Hello-Extension can be used to provide an available identifier, as illustrated in Figure 7.5, which the receiver uses subsequently for its messages.

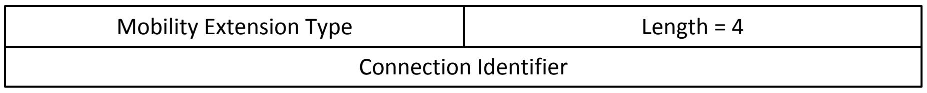


Figure 7.5: Mobility Hello-Extension

An address change will be notified implicitly, which is prone to attacks since the source address is not secured by DTLS. However, it can be secured by using the Heartbeat extension to verify the new address. Whenever a different source address is detected, the endpoint sends a HeartbeatRequest to the new address, but continues sending everything else to the previous one. After a valid HeartbeatResponse has been received, the endpoint will start using the new address.

If the mobile host is not sending any data that would allow the peer to notice the address change, periodic Heartbeats can be sent. Observing the local interfaces and sending a HeartbeatRequest immediately after the address has changed can also optimize this. However, periodic requests are still necessary to handle address changes of intermediate routers performing NAT.

An address change with Heartbeat verification is illustrated in Figure 7.6. The address of one host is overwritten by a NAT router, which then changes its address, for example because it had to reconnect to the Internet and got a different dynamic address assigned. In this case the first data message after the router changed its address will be lost, because the new address has not been verified yet, and therefore

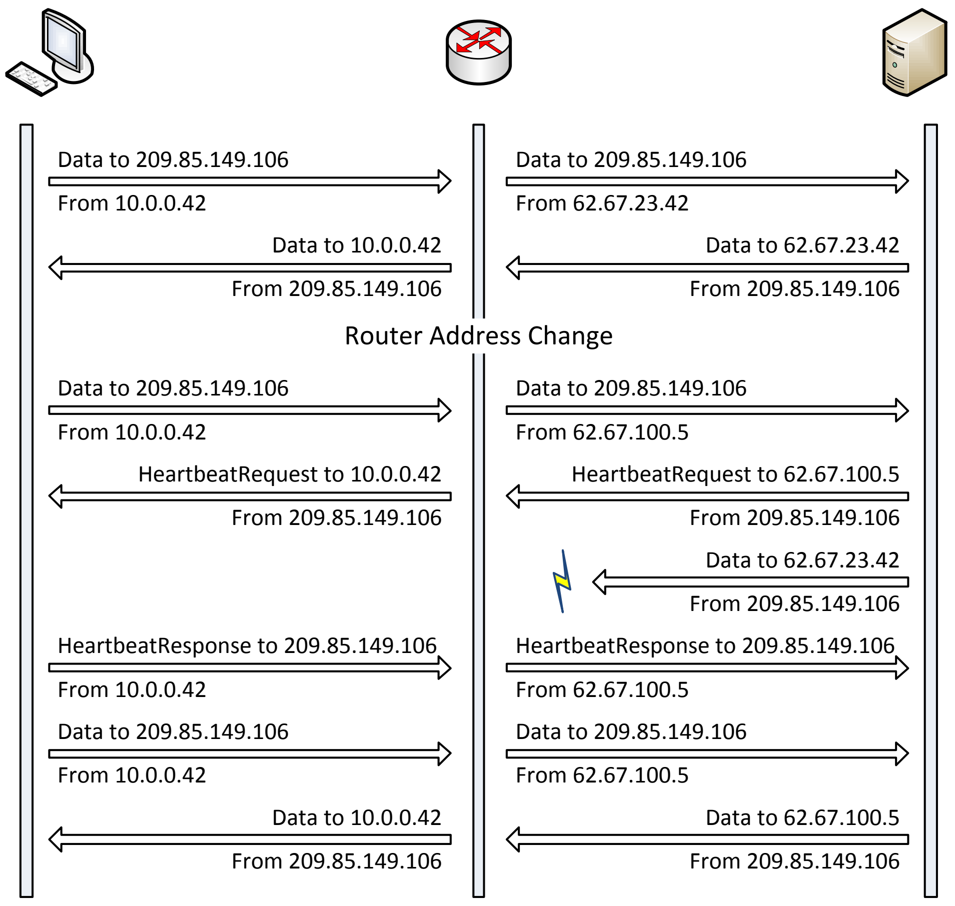


Figure 7.6: Implicit Address Change with Verification

the old one has to be used. In case another host started using the previous address, it may send an ICMP error message, which has to be ignored as discussed in the next section.

7.4.3 Security Considerations

The 32 bit Connection Identifier is used to map incoming messages with varying source addresses to a known connection. For this task, the identifier could simply be set to 1 for the first connection and increased for every other one. The knowledge of the CID does not help an attacker in terms of injecting messages or manipulating the address changes because of DTLS' security features. An attacker can, however, send random messages with valid CIDs but faked source addresses to a server to waste computing power. While every message with an unknown CID will be discarded immediately, those with valid CIDs will be processed until the decryption and MAC validation failed. So with many concurrent connections and therefore the possibility to send many manipulated messages to all of them, a considerable amount of CPU load can be generated. To mitigate this risk, the CIDs should rather be random than sequential, thereby impeding the guessing of valid ones.

Securing the address change with Heartbeats is sufficient, since they are fully secured by DTLS. The endpoint must have knowledge of the previously negotiated master secret to be able to respond correctly, which is impossible without the private keys. Also, a previously captured valid HeartbeatResponse cannot be resent by an attacker because of the replay check done by DTLS. A Denial-of-Service attack, where a malicious user requests a large amount of data before changing the address to a victim's one, is also prevented, because the victim would never send a valid response. A man-in-the-middle can perform a Denial-of-Service attack by changing the address to an invalid one, thus basically shutting down the connection. This is possible by modifying the source addresses of regular messages and the Heartbeat exchange. However, with the privileges necessary for this, the attacker could rather drop all messages more easily for the same effect.

According to the specification of the Heartbeats, only a single one can be in flight at a time. A timer has to be used to retransmit a lost one until a response arrives or the connection is dropped because of too many failed attempts. When securing an address change, retransmissions must obviously not be done, because a single message with a random source address, from where no valid response can be expected, will lead to the termination of the connection. Hence, only a single HeartbeatRequest should be sent for each verification attempt. The limitation that only a single request can be in flight can then not be enforced, because if there is no response, a timer has to expire before a new address change verification can be done with a new HeartbeatRequest. This could also be used to perform a Denial-of-Service attack against the address change. When an attacker sends so many faked messages, that a request is always in flight, or more specifically a timeout is always awaited, a real address change cannot be validated. Sending a single request for every single message with a different source address can prevent this. The payload can be used to carry all the necessary information, so no resources have to be allocated. Once a valid HeartbeatResponse has been received, the new address can be accepted. This also serves to increase the reliability, because otherwise there are no retransmissions for lost verification requests. Although this increases the load on the network, it cannot be used for attacks, because only valid DTLS messages will trigger a request.

7.4.4 ICMP Considerations

The Internet Control Message Protocol (ICMP) and its version for IPv6, ICMPv6 [8], are used for signaling at the IP layer, for example that a message is undeliverable for some reason. Although these messages are usually handled by the operating system, the *Destination Unreachable* messages result in failing send/receive calls and are thus notified to the application layer, if connected sockets are used. The application has to behave according to the reason why this error occurred.

If *Fragmentation needed* or *Packet Too Big* is received, a Path MTU Discovery should be performed, because the routing and thus the Path MTU probably changed. In the other cases the peer is not available anymore. Without mobility, these errors should rarely occur as soon as the connection has been established, since either the network has to go down or the peer has to decline messages. An attacker, however, could easily fake such ICMP messages in an attempt to drop the connection. Hence, they should be ignored and the reachability of the peer tested otherwise, for example by sending a HeartbeatRequest or with the application protocol, if possible. With mobility these errors are more likely to occur, for example if the mobile client joins a network with limited connectivity. Hence, with DTLS Mobility an ICMP error indicating unreachability of the peer must trigger a HeartbeatRequest to check the connectivity.

7.5 Implementing Heartbeats and DTLS Mobility

As part of this work, the Heartbeat extension has been implemented for OpenSSL, which was rather straightforward as illustrated in Figure 7.7. The new message types had to be added, as well as the Hello-Extensions. The handling of incoming messages had to be extended to answer every reception of a HeartbeatRequest with a HeartbeatResponse immediately. An API had to be provided to send a HeartbeatRequest. The retransmission timer for DTLS handshake messages was also used for Heartbeat messages. The connection is closed if there are too many unsuccessful retransmissions, so by using the same timer this also applied to Heartbeats and no further action was required.

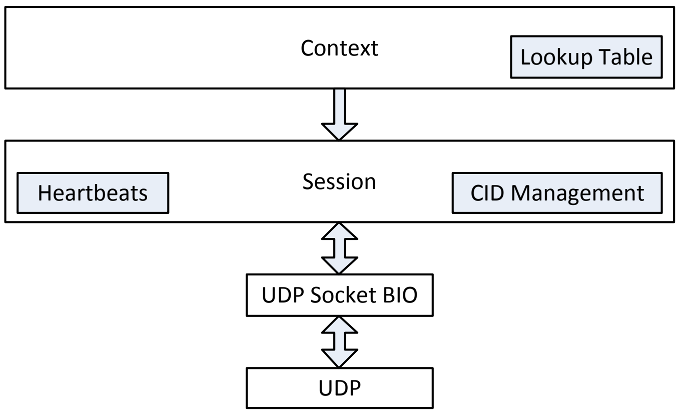


Figure 7.7: Modifications to OpenSSL to support Heartbeats and DTLS Mobility

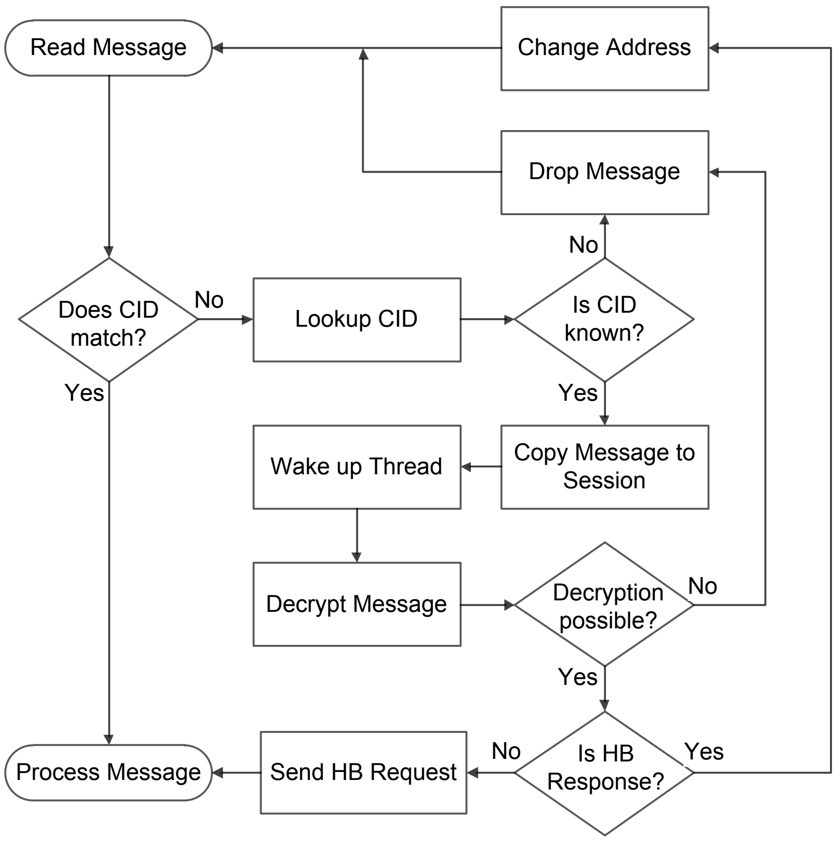


Figure 7.8: DTLS Mobility Flow Chart

The modifications for the DTLS Mobility prototype required the session objects to handle both the default and the extended Record Header. Additionally, a lookup table was necessary, which was added to the context objects similar to the session caching functionality for session resumption.

Figure 7.8 shows how mobility messages are processed within the implementation. Every time a new connection has been successfully set up, the Connection Identifier, the corresponding session object and the thread used to handle the connection have to be added to the lookup table. Usually, each established connection using connected sockets, that is they are bound to a fixed destination, is handled by its own thread. The kernel automatically passes incoming data to the proper socket, and so also to the proper session. If the source address changes, the kernel cannot match the socket anymore and the message ends up at the listening socket for new connections, which receives everything else. The lookup table is used to find the appropriate session by comparing the CID. If the lookup is successful, the message can be copied into an additional buffer that had to be added to the session object, which is checked and processed every time before a new message is read from the socket. In the case the socket is in a blocking read call waiting for incoming messages, a signal is sent to the thread, so the read call is interrupted, and the new message can be processed. If the message is valid and has been successfully decrypted, a HeartbeatRequest is sent to the new address for verification. The response will again end up at the listening socket and has to be passed to the appropriate session object, but now that the new address has been verified, the connected socket of the corresponding session can be changed to the new address.

These changes to realize DTLS Mobility allow an easy deployment, since the resulting library is all that is necessary. The DTLS library can be provided by the system and linked dynamically by applications. If not yet available it can also be linked statically, that is bundled with an application.

7.6 Evaluation and Measurements

DTLS with the mobility extension can either be used directly for an application protocol or as a tunnel. By tunneling the IP layer, this can be used to provide security to generic connections (compare Section 5.1.2), or to connect two separated networks in a secure way, that is a VPN (compare Section 5.2.1). With the mobility extension, such a tunnel can even be mobile and thus add mobility to generic connections transparently. Compare the protocol stacks in Figure 5.1 and 5.2 for an application protocol using DTLS directly and being tunneled, respectively.

To verify the functionality, direct connections have been tested in common scenarios with address changes. Once it was ensured that everything worked as expected, the tunneling could be examined as well by measuring a connection over a mobile tunnel.

7.6.1 Direct Use

To validate the concept and verify the functionality of DTLS Mobility and the Heartbeat extension, three common scenarios with network address changes have been set up. The time necessary until the DTLS connection was fully operational again has also been measured. The prototype implementation for OpenSSL was used with an echo server and client, the latter sending messages continuously, which were returned by the server, a Mac Mini running Mac OS X 10.6.4. For every address change, the server measured the time until the transmission continued, that is until the physical link was reestablished, and DTLS had verified the new address. The measurements were repeated 10 times to calculate an average value. The Wi-Fi networks used were connected via a high speed connection to the server over the internet, which added almost no latency to the connection.

Common scenarios for mobility are the address change of a middlebox, usually unnoticed by the client, the change from one Wi-Fi network to another, either because of a location change or a failure, and the change between Wi-Fi networks using the cell phone network in between. The first scenario was measured with a laptop running Ubuntu Linux 10.04 LTS, connected to a Wi-Fi network. The router of the network was an Apple Airport Extreme that performed NAT and was connected to the internet via Digital Subscriber Line (DSL) with a dynamic IP address that was likely to change with every reconnect. The client did not notice this, because it only knew its address in the local network. The setup is illustrated in Figure 7.9.

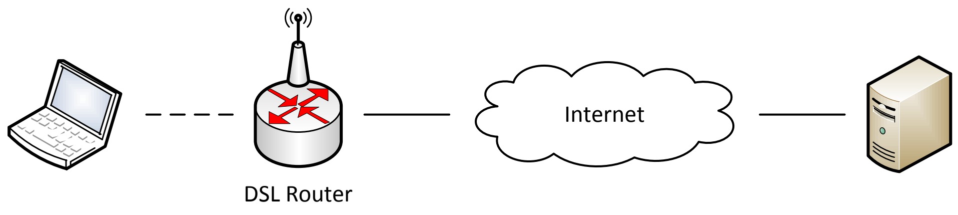


Figure 7.9: Laptop Connected over a NAT Middlebox

In the second scenario, shown in Figure 7.10, there were two Wi-Fi networks, and the client changed from one to the other, because the first either went down or moved out of range. For the measurement, one router was turned off manually for the loss of connectivity during the measurement. The same laptop was used and for comparison an iPod touch (iOS 4.2.1) and an Android 2.1 smartphone, both with Wi-Fi networking enabled. The client application had to be ported to these two devices. Both are able to run applications written in C, so this was done by implementing a User Interface using their native development environments (Objective-C and Java) for the original program, and linking the OpenSSL library statically.

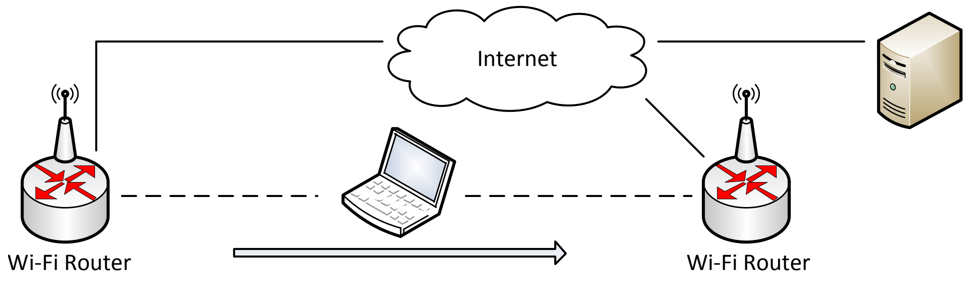


Figure 7.10: Laptop Moving to Another Wi-Fi Network

The third scenario was basically the same as the second one, but this time with an iPhone with 3G capabilities, to make use of the cell phone network when no Wi-Fi connection was available, as depicted in Figure 7.11. As before, one Wi-Fi router was turned off manually.

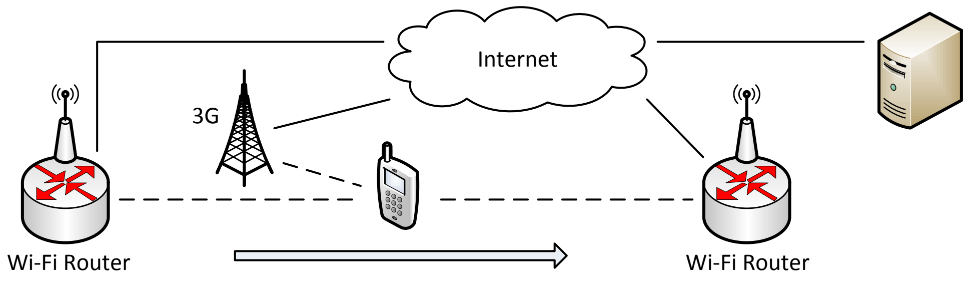


Figure 7.11: Cell Phone Moving to Another Wi-Fi Network with 3G

All scenarios worked as expected, and the connection was continued despite the address changes. The necessary time to reestablish the physical link, as well as performing the DTLS address change verification is listed for each measurement in Table 7.3. While the time for verifying the new address roughly equaled a Round-Trip Time (RTT) in the given network, the time to reestablish the physical link was depending on the device and network used. This is particularly true for the time necessary to detect the loss of a Wi-Fi connection, which differed significantly between the laptop and the handheld devices. This was most likely because the handheld devices are optimized to use as little power as possible for increased battery lifetime, and therefore check the availability of the Wi-Fi network less frequently. The use of 3G as an always-available fallback shortened the downtime to the detection of a connection loss and was used until a new Wi-Fi connection had been established. So while DTLS Mobility is basically suitable even for time-critical applications, such as audio and video, the time necessary for the hardware and the

Device		**Link**	**DTLS**
Laptop	DSL Router Reconnect	6.6 s	42.0 ms
	Wi-Fi → Wi-Fi	5.5 s	3.2 ms
iPod touch	Wi-Fi → Wi-Fi	17.8 s	16.5 ms
HTC (Android)	Wi-Fi → Wi-Fi	14.8 s	7.1 ms
iPhone	Wi-Fi → 3G	13.3 s	273.5 ms
	3G → Wi-Fi	0.1 s	15.0 ms

Table 7.3: Mobility Delays

operating system to reestablish the physical connectivity has yet to be optimized to avoid outages of several seconds. This, however, is a generic problem for any mobility solution.

7.6.2 Tunneling

After the tests with a connection making direct use of DTLS Mobility have proven that everything works as expected, and the different kinds of address changes are handled easily, DTLS can be used as a tunnel for other connections.

To examine the behavior of a tunneled connection, a tunnel between two hosts has been set up, which was used by two additional hosts communicating over it, as shown in Figure 7.12. The tunnel server was a Mac Pro connected to the internet directly via a high speed link and the tunnel client a MacBook Pro (MBP) connected to the Internet via a 100 MBit/s cable, a Wi-Fi connection, and also via the cell phone network with 3G. The tunnel was realized with divert sockets, which redirected the IP packets to be tunneled to the tunnel applications. After passing the tunnel, the IP packets were sent with a raw socket to their actual destination.

A second Mac Pro was connected to the tunnel server and hosted a server accepting TCP connections and logging their throughput each second. A Mac Mini connected to the MacBook Pro runs the TCP client that connected to the server over the tunnel and sent as much data as possible. All hosts were running Mac OS X 10.7.3 and all measurements were repeated 10 times to calculate average values and 95% confidence intervals.

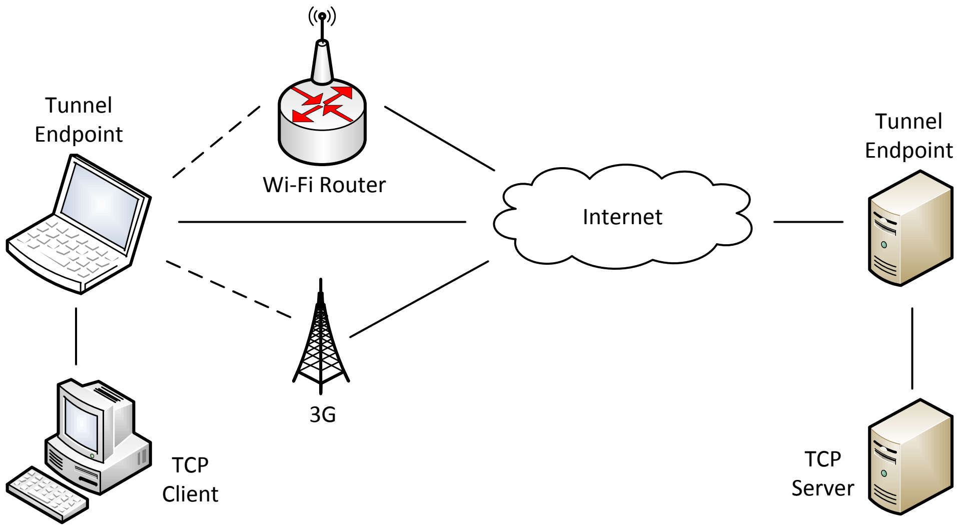


Figure 7.12: TCP Connection over a Mobile DTLS Tunnel

At first, the time necessary to change from the cable to the Wi-Fi connection and again to the 3G link in this setup was measured, in the same way as in the previous measurements. The results listed in Table 7.4 are necessary to interpret the behavior of the tunneled connection.

Device		**Link**	**DTLS**
MacBook Pro	Cable → WLAN	2.0 s	20.3 ms
	WLAN → UMTS	13.3 s	121.4 ms

Table 7.4: Mobility Delays with a DTLS Tunnel

The MBP had three links to the Internet and was set up to always prefer the fastest possible method. Hence, at first the cable link was used when the TCP connection was established over the tunnel. After 15 seconds the cable link was lost by pulling the cable, and the DTLS connection changed to the Wi-Fi link and performed an address change verification. According to the results in Table 7.4 this took about 2 seconds. The Wi-Fi network was lost 60 seconds later by turning off the Wi-Fi router, so only the 3G link was available anymore. It took about 13 seconds for the MBP to notice the loss of the connectivity before it started to use the cell phone network.

The graph in Figure 7.13 shows that while the 100 MBit/s cable link was available, the throughput was accordingly at about 11 MByte/s, which is slightly lower than the theoretical maximum because of the additional overhead due to the tunneling. When the cable link was lost, the Wi-Fi link was used as expected about 2 seconds later. The achieved throughput varied between 1 and 2 MByte/s and increased only slowly, because the signal strength and link quality varied, which also resulted in large confidence intervals. When this link was lost as well, as shown in Figure 7.14, it took about 15 seconds until there was noticeable throughput over the 3G link. This is slightly longer than the 13 seconds of the address change itself, and was caused by the retransmission timers of TCP that had to expire before the sending of data resumed. The transfer continued on the 3G link with about 0.2 MByes/s.

This measurement shows not only that DTLS is a viable option to realize a VPN by tunneling IP packets, but also that mobile tunnels are possible without any limitations, even for reliable transport protocols. The tunneled connection resumes immediately after the first retransmission was successfully sent over the new link. This is significantly faster than awaiting a timeout to realize the connectivity changed and reconnect with a new address. The limitation is again the time necessary for the hardware and the operating system to detect the loss of connectivity, and to restore it with a different network interface.

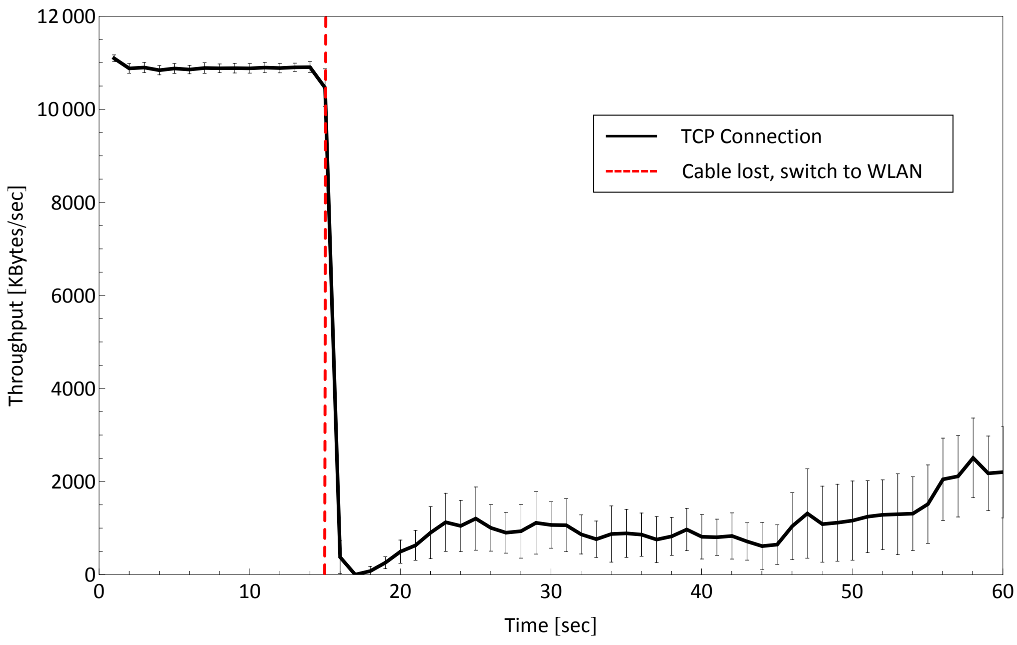


Figure 7.13: Throughput of the TCP Connection, Cable to WLAN Change

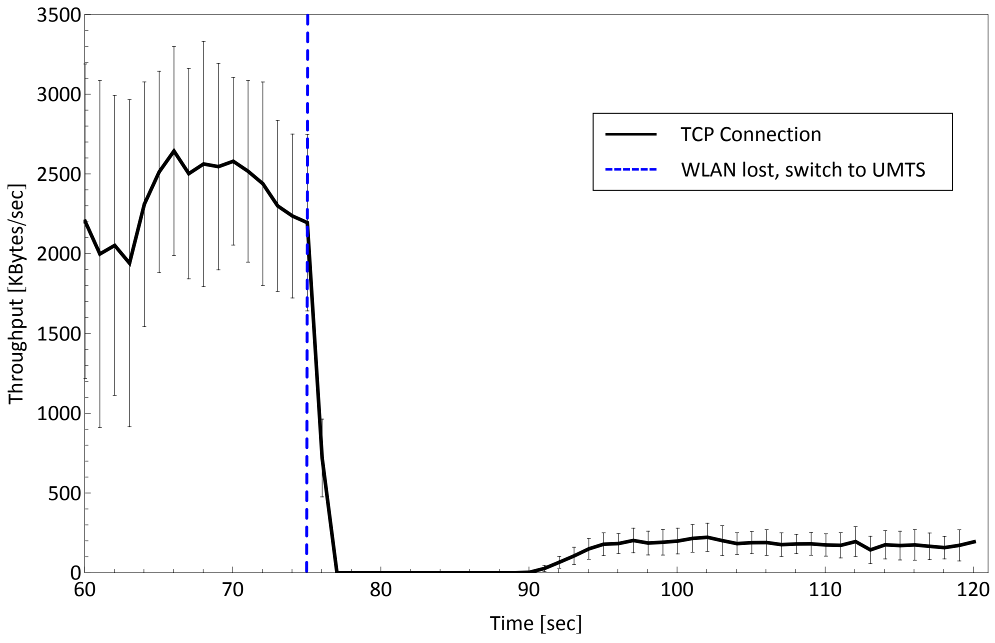


Figure 7.14: Throughput of the TCP Connection, WLAN to UMTS Change

7.7 Conclusion

In this chapter new extensions for DTLS have been suggested to improve to protocol by adding new features. The DTLS specification requires performing a Path MTU Discovery because of UDP's message-orientation. Unfortunately, neither UDP nor DTLS provide any mechanism to determine the maximum message size. Furthermore, checking the availability of the peer is necessary because UDP is connectionless, but DTLS has no mechanism other than a costly renegotiation for this task. These issues can be solved with the Heartbeat extension and its request and response protocol, which can be used for keep-alive checks and also to perform a Path MTU Discovery. The extension has been implemented for OpenSSL and has been integrated in stable release 1.0.1 [61]. The TLS working group at the IETF has accepted it for standardization and has published it in RFC 6520 [93].

The second proposed extension is DTLS Mobility, to add mobility features for DTLS connections. With this extension, connections can be continued even if the used IP addresses change, which allows a host to move between multiple networks without reconnecting. The mobility features can be used for direct connections or transport tunnels realized with DTLS. The functionality has been evaluated in Section 7.6 for several scenarios, with a prototype implementation for OpenSSL, as described in Section 7.5. This has shown that, after a new physical link has been established, DTLS handled the address change and resumed data transfer within a single RTT. Connection-oriented and reliable protocols, like TCP, can also benefit from the mobility features with tunneling. After an address change, the data transfer continues immediately with the first successful retransmission, which is more efficient than reconnecting after detecting the connection was lost.

Chapter 8

SSH Tunneling

Another method to secure security-agnostic applications is to use application protocol tunneling, which is commonly called forwarding, and has been discussed in Section 5.2.3. It has the advantage that no support by the operating system is necessary. Secure Shell (SSH) is an often-used solution to realize such a forwarding for TCP, but does not support SCTP yet. The necessary modifications to the SSH protocol for SCTP support are presented, and possible benefits by running SSH over SCTP are examined. The adaptations for SSH suggested in this chapter have been published in [91].

8.1 SSH Forwarding of SCTP

The connection-forwarding feature of SSH is only specified for TCP, but it can basically also be used with SCTP associations. The corresponding protocol stack can be found in Figure 5.4. The SSH endpoints have to be modified to accept and establish SCTP associations rather than TCP connections. However, since only the payload is forwarded, any information about the transport protocol will be lost. That prevents SCTP and TCP from being used concurrently, because the endpoint establishing the connection to the destination host cannot know which protocol has to be used for each forwarded connection. Furthermore, additional information of the transport protocol will be lost, too. This includes the information for multi-streaming, multi-homing, unordered transfer and SCTP's Payload Protocol Identifier, indicating which upper protocol is used. Therefore, the SCTP association would be limited to a single address and a single ordered stream, that is basically the same features a TCP connection has.

8.1.1 SSH Modifications

To overcome these limitations, the additional information has to be retained. The extension of the SSH protocol by adding new channel types for SCTP forwarding is suggested, such as *direct-sctpip* for local and *forward-sctpip* for remote forwarding,

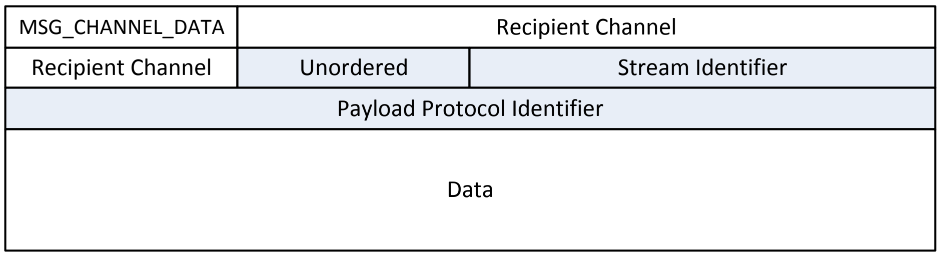


Figure 8.1: SSH Data Message Extended for SCTP

to follow the naming of the existing types for TCP [112]. Multi-streaming, unordered transfer and the Payload Protocol Identifier can be supported by prepending this information at the beginning of the SSH data messages carrying the payload, as depicted in Figure 8.1.

SCTP supports using multiple addresses of a multi-homed destination when establishing an association, as a failover in case one of them is currently unreachable. This is part of SCTP's multi-homing features for increased reliability. With SSH, only a single destination address can be provided, which is transmitted when requesting the forwarding service. To support multiple destination addresses, the

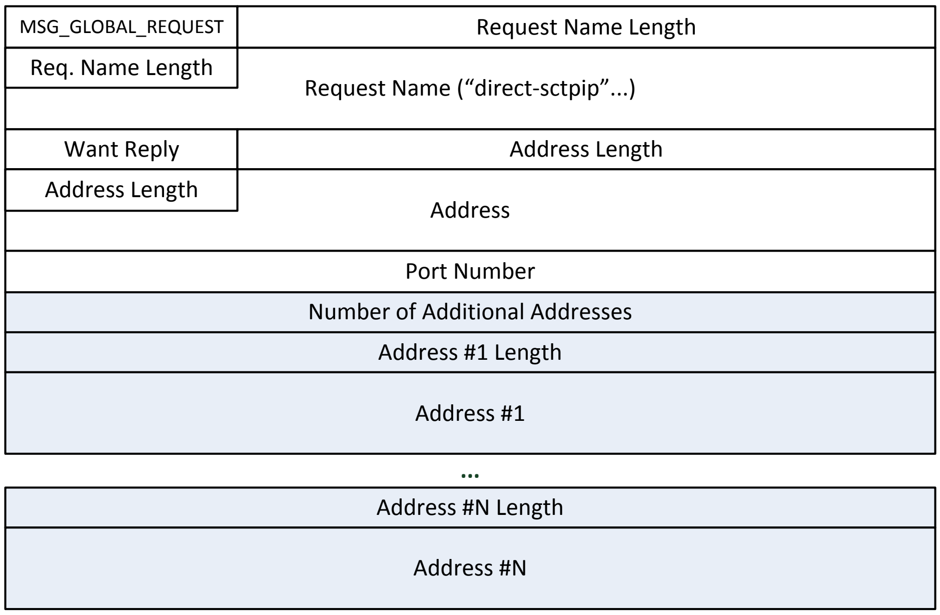


Figure 8.2: SSH Request Message Extended for SCTP

service request message also has to be extended, as depicted in Figure 8.2. The SSH endpoint can then use theses addresses when connecting to the destination.

8.1.2 Limitations

Despite SCTP's multi-homing support and the support for SSH to use multiple addresses for the destination, the multi-homing support with SSH forwarding is still limited. The initial SCTP association to the first SSH endpoint may use multi-homing, as well as the final SCTP association from the other SSH endpoint to the destination, compare Figure 8.3. The forwarding SSH connection, however, uses TCP and is therefore single homed. This can result in a single point of failure, despite the additional reliability of the SCTP associations.

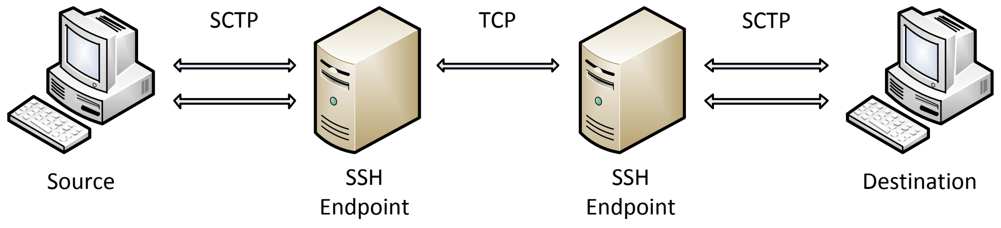


Figure 8.3: SCTP Forwarding over SSH

The extension PR-SCTP is also only partially supported. To enable it, both endpoints must announce their support during the association establishment. With SSH forwarding, the SCTP associations from the source and to the destination both have an SSH endpoint as their direct peer, as seen in Figure 8.3 Hence, these SSH endpoints must be configured to enable PR-SCTP, in order to use it. Both source and destination hosts can use arbitrary PR-SCTP policies when sending to their forwarding SSH endpoint, but the SSH endpoints will always send reliable, unless manually configured otherwise. The forwarding SSH connection itself uses TCP and is therefore always reliable. This can be detrimental if PR-SCTP is used to avoid retransmissions to reduce the message delay.

8.2 Forwarding over SCTP

Although TCP is commonly used, the specification of the SSH Transport Layer only requires a transport protocol that protects against transmission errors (lost messages are repeated) [114] and preserves the order of the messages. SSH uses multiple channels to realize its services, but they are multiplexed for the secured connection provided by the SSH Transport Layer. As a result, the protocol expects a single ordered transport channel for its SSH Transport Layer from the transport protocol. This corresponds to a single-homed SCTP association with a single stream, which behaves just like TCP.

8.2.1 Usage of Multi-Homing

The immediate benefit of using SCTP is the multi-homing support that increases the reliability or bandwidth of SSH connections. The necessary modifications are limited to changing the sockets to SCTP and extending the configuration, to allow specifying multiple addresses for multi-homing support. With its message-orientation, every SSH packet will be sent in an SCTP message. Although this reveals the message boundaries, that is not a security issue. Message boundaries are visible for interactive remote shells anyway, due to the disabled Nagle's algorithm. Every input is sent immediately, instead of awaiting enough data to fill a packet. And for bulk transfer, always the maximum message size will be used, which is 32 KBytes in the current OpenSSH implementation. The ADD-IP extension for SCTP [101] that allows adding and removing addresses of already established connections, can even be used to realize mobility as described in [80].

8.2.2 Usage of Multi-Streaming

The multi-streaming feature of SCTP is a concept similar to the channels of an SSH connection, both are used to separate logically independent data. The SCTP streams are not only used to mitigate the impact of head-of-line blocking on lossy links. With specialized stream schedulers a fair distribution of the available bandwidth or the prioritization of specific streams, like those carrying interactive remote shells, can also be achieved, and will be discussed in Section 8.2.3.

Although the message order across multiple SSH channels is arbitrary, they are multiplexed before being carried by the SSH Transport Layer, which prevents any further reordering. This is shown in Figure 8.4. Since SCTP does not preserve the order of messages across multiple streams, the reordering of messages of multiple channels must be possible at any time to allow the mapping. Therefore, the SSH

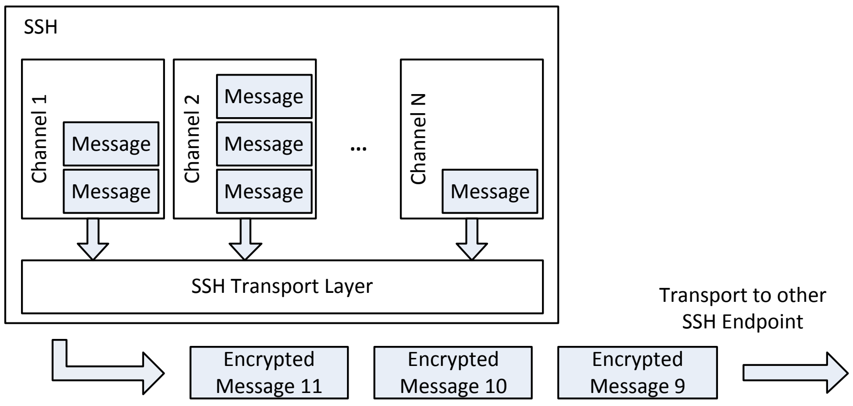


Figure 8.4: SSH Channel to SSH Transport Layer Multiplexing

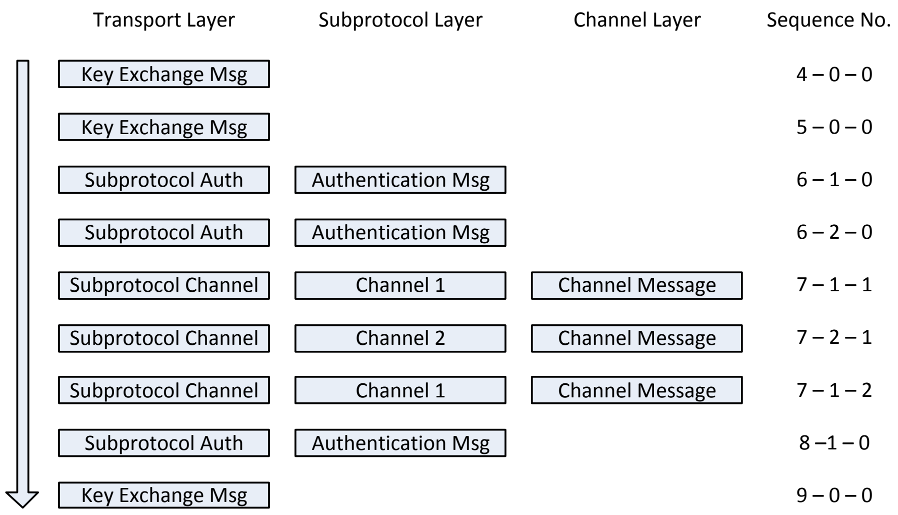


Figure 8.5: SSH Sequence Numbers with SCTP

Transport Layer needs to be adapted to tolerate reordering across multiple channels, while still requiring the messages within a single channel and of the other sub-protocols to stay in order.

SSH uses a sequence number for each message for the calculation of the Message Authentication Code (MAC) to ensure integrity. This sequence number is not transmitted; both endpoints keep track of it instead. If a message arrives out of order, the assumed sequence number is different, and the verification of the MAC and therefore the entire connection fails. To prevent this, the message header of the SSH Transport Layer has to be extended with the sequence number to transfer it explicitly, like it is done with DTLS or the Authentication Header of IPsec. Contrary to these protocols, the entire SSH header is encrypted, so the sequence number will not be revealed. Additionally, the sequence number has to be extended to differentiate between messages that have to arrive in order or that may be reordered. As a possible solution it is suggested to use multiple sequence numbers, one for the SSH Transport Layer, one representing the sub-protocol or channel, and one for the messages within a channel. Figure 8.5 illustrates how three sequence numbers can be used to ensure the order of the messages with the new constraints. The first sequence number is increased for every SSH Transport Layer message, such as key exchange messages. Every time a different sub-protocol is used, the first sequence number is also increased but stays the same for all messages of the sub-protocol, and the second sequence number is used for the messages of the sub-protocol. In case of channels, every channel has its own second sequence number, and the third number is used for each of its messages. The order can be verified as follows:

- If only the first sequence number is set, they have to be in order.
- If the first and second sequence numbers are set, the second number has to be in order for each first number.
- If all three sequence numbers are set, the third number has to be in order for each second number.

Each sequence number has to have a size of 32 bit, like the original sequence number, so it is also ensured that a wrap around and thus a number reuse still only occurs after at least 2^{32} messages.

A similar issue is caused by cipher suites that are depending on the message order, for instance because a counter (counter mode) or data from the previous message (cipher-block chaining) is used as the Initialization Vector (IV) for the encryption. Generating random IVs, which are then prepended to the SSH Transport Layer message, can solve the dependency, like it has been done for DTLS. The SSH Transport Layer message format with the extensions is illustrated in Figure 8.6.

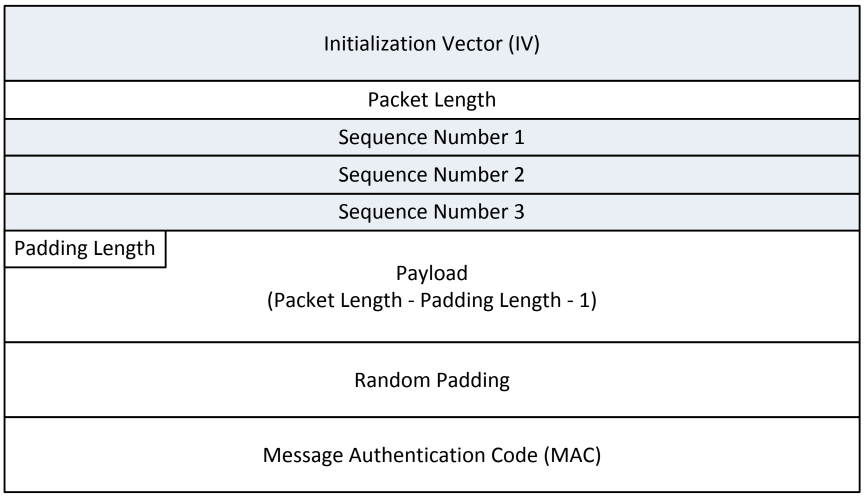


Figure 8.6: Extended SSH Transport Layer

The SSH connection is established with a key exchange of the SSH Transport Layer. This negotiation can be repeated at any time to refresh the key material of an existing connection. No other messages are allowed during the negotiation, and the message sequence of the key exchange has to be preserved. This is the same issue that had to be dealt with for SCTP-aware DTLS. Sending all corresponding messages on the same SCTP stream can retain the order. When mapping channels onto streams, however, channel messages on other streams may arrive during the exchange if they arrive too late or too early, respectively. To prevent this, the message order has to be enforced across all streams while a key exchange is performed.

This requires to drain the transfer on all streams except the one the key exchange is performed on. Hence, before sending the Key Exchange Init and before resuming the regular data transmission after the New Keys message (compare Figure 3.11), the reception of all previous messages has to be awaited before continuing. The same is necessary before shutting down the SSH connection. The SENDER_DRY event of SCTP can be used to realize this, similar to DTLS.

The packet size of SSH channel messages should be limited to the Maximum Transmission Unit (MTU), the maximum possible message size on the path between the peers. The default packet size of SSH may be larger and therefore messages have to be fragmented by SCTP, which limits the effect of stream scheduling, because all fragments have to be transmitted before the scheduler can select the next stream.

8.2.2.1 Security Considerations

The mapping of channels onto streams, however, causes a confidentiality issue that has to be considered. The header of SCTP data messages is not encrypted. Therefore, an attacker knows how many streams are used and thus can estimate the number of channels. SSH supports 2^{32} channels, but SCTP only 2^{16} streams. So there may be more channels used than streams are available, and multiple channels have to be mapped onto each stream. This would avoid the disclosure of the exact number of channels, but the use of more than 2^{16} channels should occur rarely. Hence, revealing the number of channels can hardly be avoided.

8.2.2.2 SCTP Data Chunk Header Encryption

Using encryption for the transport protocol to protect the header information can solve this issue. Since there is no such solution available, a new extension called the SCTP Encryption Chunk is recommended. With the Encryption Chunk extension, each peer can announce which encryption algorithms it supports and which types of chunks it will only accept encrypted. This is done during the handshake for the connection establishment, similar to SCTP-AUTH. The relevant chunks will then be encapsulated in a new encrypted chunk, rather than being sent as plain text. The receiver decrypts the content of the encryption chunk and can process the resulting original chunks as usual.

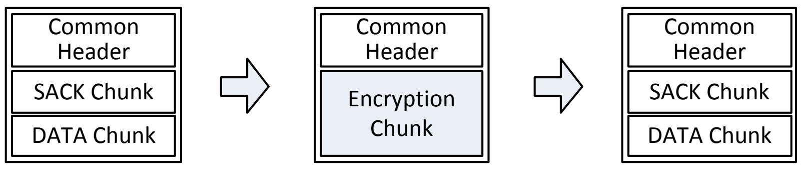


Figure 8.7: SCTP Encryption Chunk

This process is shown in Figure 8.7. The necessary shared secret has to be provided by the user, and can be derived from the secret that is negotiated for the SSH connection. This concept is similar to Secure SCTP introduced in [22], although this extension requires rather small changes, since it resembles SCTP-AUTH in many aspects. Additionally, it only provides additional confidentiality and can therefore be considered optional.

However, with the use of the encryption chunk, the already encrypted SSH packet will be encrypted again, which most likely results in a rather poor performance. Therefore, the encryption of SSH should be disabled. This may even increase the performance for small messages, common with remote shells, because the encryption is done only once for all chunks bundled into a packet, while SSH would encrypt every message separately.

8.2.3 Stream Scheduling Considerations

Multiple streams are defined in the SCTP specification [96], but it is not mentioned how the scheduling should be done. Different implementations are using different strategies, for example FreeBSD is using a round-robin algorithm, while Linux and Solaris are using first-come, first-served. From this it follows that there are several degrees of freedom, which can be used for optimization with specialized scheduling approaches. In the following sections, several common scheduling algorithms [66] are discussed. This discussion of the topic has also been published in [89].

8.2.3.1 First-Come, First-Served Scheduling

The first-come, first-served scheduling is the simplest stream scheduling algorithm. Messages are processed in the order supplied by the application or their arrival. Therefore, the application can decide on the order messages are sent.

However, this influence is limited since the application cannot modify queues or change the way bundling is done. This makes it impossible for the application to realize scheduling strategies that prefer messages of a certain kind or any preemptive algorithms that require queue modification.

Because behavior of this scheduler depends on the application, which can for example send messages in a round-robin fashion, there is no unique characteristic.

8.2.3.2 Round-Robin Scheduling

For its simplicity, the round-robin algorithm is one of the most used scheduling algorithms. Its concept is just to cycle through all available queues, that is streams in the case of SCTP, and always choose a single message per stream. This kind of scheduling is always fair regarding the number of messages sent per stream, but there is no consideration of other aspects like the length of the messages. This can lead to an unfair distribution of the available bandwidth, if messages sent on one stream are significantly larger than on another.

8.2.3.3 Fair Bandwidth Scheduling

When forwarding multiple connections, a fair distribution of the available bandwidth is usually desired. The round-robin approach cannot always guarantee this, so when the message size differs, another solution is necessary to achieve a fair treatment of all streams. This can be realized with the fair bandwidth algorithm, which has been introduced in [55].

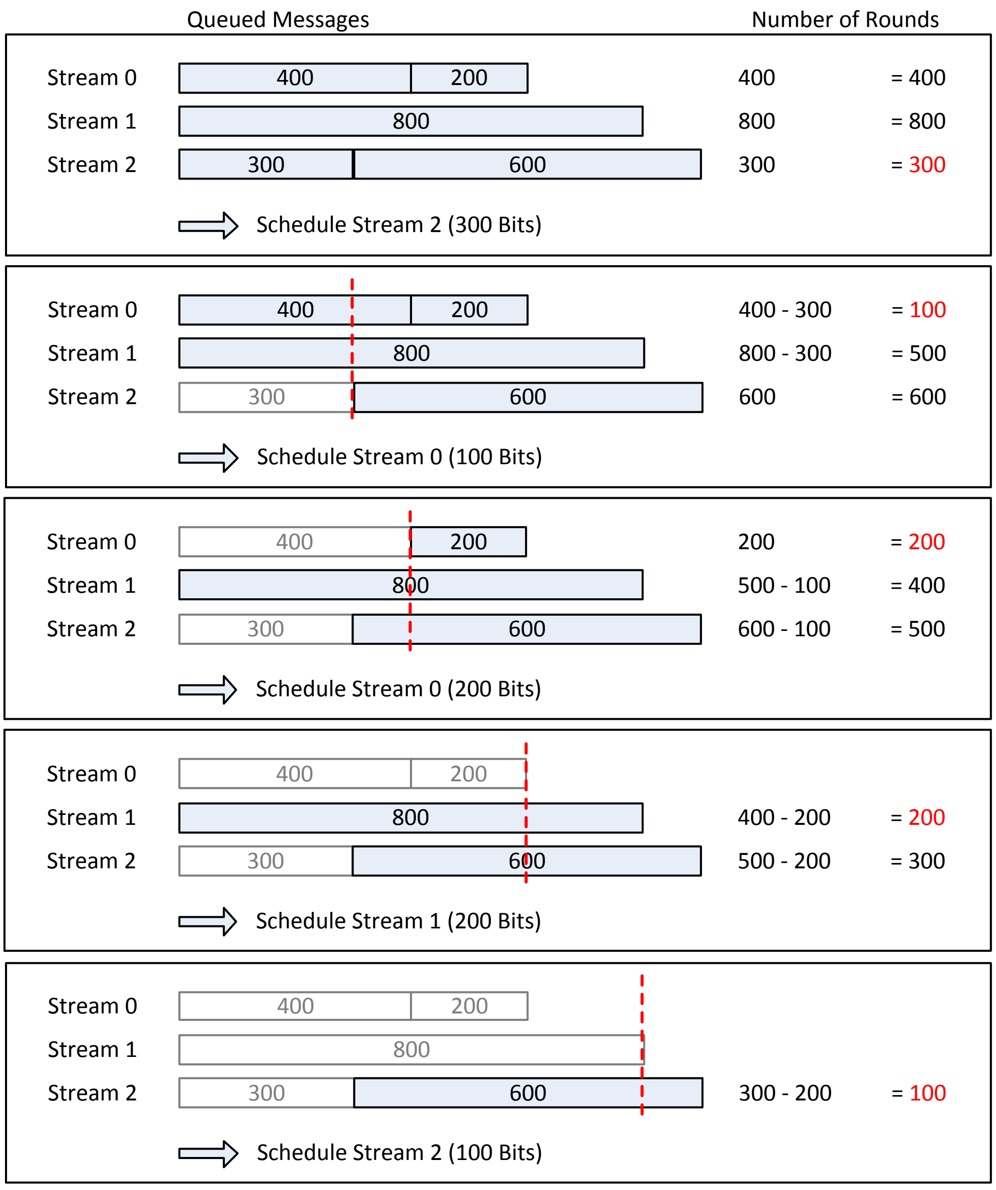


Figure 8.8: Fair Bandwidth Scheduling Implementation

A fair bandwidth algorithm can be implemented for SCTP based on the formal definition in [13]. It is based on the idea of a bit-by-bit round-robin algorithm. A single bit of a stream is sent before moving on to the next stream. This is impracticable, so the cycling through the streams will just be realized as a packet- or message-based algorithm. The selection of the message to be sent next is done by determining the message that can be finished in the least number of virtual bit-by-bit rounds.

Figure 8.8 illustrates how the determination of the next stream is done. In the first step no data has been sent yet. Thus, the number of rounds of bits necessary to finish a message is the message length itself. The lengths of the first message in the send queue of each stream are considered. In this case the smallest message demands the least number of rounds to be done and its stream is scheduled.

The number of virtual rounds already done is moved to the end of the message just sent. In the figure the red dashed line represents this. The rounds necessary to finish one of the other messages are decreased. For the just scheduled stream the next message in the queue will now be taken into consideration. This time the message that needs the least number of rounds is not necessarily the smallest one. The following steps in the figure depict how the algorithm will process the messages in the queues by maintaining the number of rounds of bits relative to the virtual amount of rounds already done.

This results in a fair bandwidth at any time, since this implementation is time independent. Idle streams will be ignored until there are messages to consider again. Simple bandwidth counting per stream, on the other hand, becomes unfair when there are idle streams. Idle streams have no messages to increase their counters. If a stream that has been idle for a certain time has messages to send again, it will be chosen again and again, because its counter value is far lower than of those streams that had messages continuously. So the previously idle stream will be preferred until its counter is evened again with the other streams, even if there are messages of other streams waiting to be sent, too.

8.2.3.4 Priority Scheduling

In some cases fairness is less important, rather not wanted at all to prefer a specific kind of data. An example would be to assign interactive services of SSH, like remote shells or X11 forwarding, a higher priority than simple data transfer that may be done over the same connection.

This can be realized with priority scheduling [7] as described in [29], where each stream gets a priority ID. The higher this ID is, the less important is the data. An ID of 0 has therefore the highest priority and is the most preferred. Per default all streams have the highest priority, so a lower priority has to be assigned explicitly. This is to maintain the normal behavior when the feature is not used.

The scheduling algorithm is simple. It always sends from the stream with the highest priority, that is the one with the lowest ID first. When multiple streams have the same priority, the default scheduling should be used for them, that is either first-come, first-served or round-robin. There also has to be a part of all relevant queues reserved for higher priorities, otherwise a queue can be filled with low priority data, and high priority data cannot be accepted anymore and has to be delayed.

8.2.3.5 Per Packet Scheduling

Sending small messages with a round-robin scheduling has the disadvantage that data of multiple streams is bundled into a single packet. This results in head-of-line blocking for all bundled streams when the packet is lost, which in worst case can be all used streams. Hence, the concept of streams to prevent head-of-line blocking is made ineffective.

As a solution for this issue it is suggested to bundle only messages of the same stream in a single packet, so if it gets lost only one stream is affected and therefore the delivery of its messages delayed. This is expected to improve the average end-to-end delay on a lossy link. As discussed in [89], this scheduler indeed improved the average delay with a saturated sender. A disadvantage, however, is that it may be worse than standard round-robin if there are not enough messages on a single stream to fill an entire packet, which leads to delays until there is additional data or the packet is sent with the smaller size.

8.2.3.6 Preemptive Scheduling

Priority scheduling can still delay important messages if a large and maybe even fragmented low priority message has to be finished first. In very time-critical environments, even these usually small delays can be unacceptable. This can be solved with preemptive priority scheduling [54]. It allows the implementation to abort the sending of any lower priority data instantly to pass through high priority messages. The aborted messages have to be dropped with PR-SCTP, because otherwise there would be a new message within the fragments, which is impossible for the peer to recognize. If the peer doesn't support partial delivery or can abort it, the message can just be counted as lost and retransmitted later. The behavior is the same as with priority scheduling but without any delay of any kind for important data.

8.2.4 Congestion Control Considerations

SSH allows the forwarding of multiple connections over a single encrypted connection. This has the advantage that an attacker cannot know to how many connections are forwarded. On the other hand, this has the disadvantage that the SSH connection only gets the amount of bandwidth for a single connection, if there are competing connections. The Congestion Control always maintains the fairness with other connections in the same network. This is regardless of how many connections are actually forwarded, although the multiple forwarded connections may allow SSH to claim a larger part of the available bandwidth.

By default, TCP and SCTP use a similar Congestion Control for a fair behavior of multiple connections sharing a limited bandwidth on a link. To optimize the forwarding, an alternative algorithm can be used, which allows to increase the bandwidth claimed by the SSH connection by the number of forwarded connections. Such an algorithm has been described in [11]. This allows to significantly improve the performance of forwarding, if several connections are forwarded simultaneously.

8.2.5 Flow Control Considerations

SSH maintains a Flow Control per channel to slow down a fast sender if the receiver reads the incoming data too slow and the buffers might run full. As described in [73], the receiver window size in older implementations was only 128 KBytes, a remnant from the times before window scaling was used for TCP and thus the maximum TCP window size was only 64 Kbytes. With modern networking infrastructure and the available computing power, however, this is a severe bottleneck resulting in a far lower throughput than expected. The authors of [73] suggest a window size that is at least 1000 times larger to make use of the capabilities of modern hardware. The default window size has been raised to at least 2 MBytes in current versions of OpenSSH to address this issue, so no further considerations are necessary.

8.3 Forwarding other Transport Protocols

The specification of SSH only describes the forwarding of TCP, but as already suggested in Section 8.1, SCTP can be forwarded as well. With SCTP as the transport protocol for SSH, more features of forwarded SCTP associations can be supported. Beyond that, the forwarding of unreliable transport protocols should be possible as well. However, forwarding them over a reliable TCP connection will increase the average message delay because of retransmissions and the preservation of the message order. An effect, which was to be avoided by using an unreliable protocol in the first place. This issue can be mitigated by using SCTP with its unreliabil-

	TCP	**SCTP**	**UDP**	**DCCP**
Protocol information needs to be preserved	-	x	-	-
Ordered transfer	x	x	-	-
Reliable transfer	x	x	-	-
Unordered transfer	-	x	x	x
Unreliable transfer	-	x	x	x
Connectionless	-	-	x	-

Table 8.1: Forwarding Characteristics of Transport Protocols

ity features for the forwarding SSH connection. Table 8.1 lists the characteristics of several transport protocols, which are relevant for forwarding. In the following sections it is discussed how SSH can be modified to support the forwarding of various transport protocols with their specific characteristics. These modifications have been implemented in a prototype based on OpenSSH, except for DCCP forwarding, because of its scarce deployment and availability. The implementation is described in Section 8.5.

8.3.1 SCTP Forwarding Characteristics

Characteristics of SCTP relevant for forwarding that have not been discussed in Section 8.1 are its multi-streaming features and unordered transfer. Furthermore, unreliability with PR-SCTP was only partially supported. By default, a separate channel is used for each forwarded connection. With SSH over SCTP, the channels can be mapped onto the streams of SCTP, as described in Section 8.2.2, to mitigate the impact of head-of-line blocking. When an SCTP association is forwarded, however, the entire association with all its streams would be forwarded over a single channel and thus mapped onto a stream of the underlying association. To improve the mitigation, the streams of forwarded SCTP associations can be mapped onto the streams of the SCTP association of SSH, as illustrated in Figure 8.9. With this approach more streams can be used, which makes less data affected by delays in case of losses. Additionally, an attacker cannot guess the number of forwarded connections anymore, because the number of streams used by the SCTP association of SSH only reveals how many streams all forwarded SCTP associations combined are using.

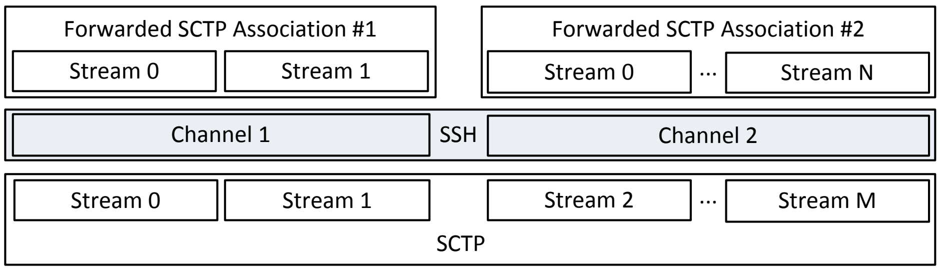


Figure 8.9: Streams and Channel Mapping for SCTP Forwarding

SCTP also supports unordered transfer, so the order is not retained even within streams. This is done to avoid delays, but when forwarding SCTP over SSH with TCP, the order of the messages will always be restored. With SSH over SCTP, however, unordered messages can also be forwarded unordered, so there will be no additional delays. Unreliability with PR-SCTP can be configured for the SSH endpoints, and with SSH over SCTP not only for the associations to the source and destination, respectively, but also for the forwarding association.

8.3.2 UDP Forwarding Characteristics

The User Datagram Protocol (UDP) is unreliable and connectionless. New channel service types have to be added, but no information has to be preserved. It can be forwarded with unordered transfer and deactivated retransmissions (PR-SCTP). However, the protocol is connectionless, so a host can send messages to arbitrary destinations at any time. To successfully forward the protocol, a channel has to be opened for every pair of source and destination addresses and ports. Without connection-orientation, there is also no shutdown mechanism, therefore no signal that the forwarding channel can be closed again. A possible solution is to use a timer that closes UDP forwarding channels automatically if they have been idle for a certain time. This does not affect the forwarding, because if further messages arrive, a new channel would be opened immediately. The channel opening for every pair of source and destination addresses, however, is a major security risk and prone to Denial-of-Service attacks. An attacker can easily send as many messages as necessary to open the maximum number of channels, so no other messages can be forwarded and the forwarding service is basically not available anymore. To reduce the risk, this kind of forwarding should only be allowed in trusted environments unless there is an additional mechanism to authorize the messages to be forwarded before opening a channel.

8.3.3 DCCP Forwarding Characteristics

The Datagram Congestion Control Protocol (DCCP) is connection-oriented, but unreliable and features a Congestion Control. DCCP can be forwarded by simply adding appropriate service types for the forwarding channel, there is no additional information that has to be preserved. For an efficient transfer, the messages of the corresponding channels can be sent unordered and without any retransmissions.

8.4 Implementing Forwarding in OMNeT++/INET

To evaluate the possible benefits of SSH over SCTP, measurements made in a simulation are helpful. They allow to test whether the assumed improvements actually ensue, without dealing with possible side effects or influences that competing processes and network traffic in a real environment may have. The INET framework already contains a working SCTP implementation including extensions, such as PR-SCTP and CMT. One advantage of SCTP's multi-streaming feature is that different scheduling algorithms can be used. Since the INET framework only had a default scheduler, which could not be changed, it had to be extended to allow the selection of multiple algorithms. To measure delays and throughputs of forwarded connections, an application behaving exactly like that had to be added as well.

8.4.1 Stream Scheduling

To allow the user to select a certain algorithm for the scheduler, the scheduling had to be made pluggable at first, that is the implementation of the scheduling method had to be separated and realized as a module that can be exchanged. Each algorithm was then implemented as such a module, and the user could choose in the configuration which one is used. In this way, the algorithms described in Section 8.2.3 could be used for measurements.

8.4.2 Forwarding Application

To simulate forwarding, an application with the corresponding behavior has been added. A client and a server were required, which communicate with multiple channels over a single transport connection. The channels had to be opened and closed arbitrarily and were used for data transfer. The client sent messages over open channels, while the server recorded delays and the achieved throughput. The transport protocol of the main connection was either SCTP or TCP. In case of SCTP, the channels were mapped onto streams and multi-homing was possible. Due to deficiencies of the TCP implementation in the simulation, the forwarding TCP connection has been realized with a single homed, single streamed SCTP association with parameters set to the default values of TCP. This is acceptable because SCTP and TCP use the same algorithms to be fair on shared links.

8.5 Implementing SSH over SCTP

To evaluate SSH over SCTP in a real environment, a prototype had to be implemented. Commonly used on most UNIX-like systems is OpenSSH, which was therefore the basis. The implementation of SSH over SCTP could be separated into three parts. The first was just using SCTP as the transport protocol without modifications of the SSH protocol. The second, and more extensive, was to add mapping of channels onto streams, which required adaptations of the SSH Transport Layer. The third was to add forwarding support for other transport protocols than TCP.

8.5.1 Adding Support for SCTP

To use SCTP as the transport protocol, the socket creation had to be modified. For backward compatibility, the use of SCTP should be additional to TCP and configurable. OpenSSH uses multiple sockets one for each listening address, so adding SCTP sockets for the same addresses was rather straightforward. When using multi-homing, however, multiple addresses have to be bound to a single socket. Therefore, the configuration had to be extended to allow the differentiation between multiple addresses bound to separate sockets or to a single one. The handling of TCP-specific socket options had to be changed appropriately, if an SCTP socket is used. With these comparably small modifications, SSH can already be used over SCTP including its multi-homing features.

8.5.2 Adding Support for Multi-Streaming

The mapping of channels onto streams required tracking a socket's protocol if TCP and SCTP should be supported concurrently. In case SCTP is used, the appropriate stream had to be set according to the current channel, and the modified SSH Transport Layer had to be used. The latter includes the different handling of sequence numbers, as well as the generation of a random IV for the encryption instead of using data of the previous message to allow the reordering of channel messages, as described in Section 8.2.2. The message order had to be enforced during key exchanges, so before sending a Key Exchange Init and before resuming normal transmission after a New Keys message, the reception of all data still in flight had to be awaited. The same needed to be done before shutting down the SSH connection. So instead of sending the Key Exchange Init message or the next channel data, the SENDER_DRY event had to be activated with a socket option. When the SCTP stack sent a notification that no data was outstanding anymore, the actual action could be performed. Finally, the maximum packet size had to be limited to avoid fragmentation, which is set in the channel opening messages. After these modifications, SSH can already benefit from being less susceptible to head-of-line blocking. A specialized stream scheduler, like prioritization, can be activated with a single socket option.

8.5.3 Forwarding Other Transport Protocols

To forward other transport protocols than TCP, new channel service types had to be defined for each additional protocol. The sockets used to accept incoming connections and to establish connections to the destination have to be created with the appropriate transport protocol. In case of UDP, connected sockets have been used to simplify the integration, since they can be used similar to TCP sockets. To support the forwarding of different protocols simultaneously, each protocol required its own configuration parameters. To address UDP's connectionlessness, a timer was added to close sockets that have been idle for some time. For SCTP forwarding the data messages had to be adapted to retain the streaming information and the PPI.

8.5.4 Pluggable Schedulers in the Kernel

The FreeBSD kernel, as well as the Network Kernel Extension (NKE) of Mac OS X, use a round-robin like stream scheduler by default. To measure the impact of different scheduling methods, the kernel (extension) had to be extended to add multiple algorithms. This was done similar to the approach for the simulation by realizing the scheduler as an exchangeable module. Additional configuration parameters allow setting the default scheduler system wide for all, or by an application for each of its associations.

8.6 Performance Evaluation

The use of SCTP as the transport layer for SSH promises several performance improvements, like smaller delays, increased throughput and failover features. To examine the possible benefits, measurements had to be done. At first with a simulation, to evaluate the maximum improvements that can be expected in a controlled environment, then with real systems to assess the influences of factors like hardware limitations on the expected results.

8.6.1 Simulation and Real Setup

For comparable results, the measurement setup within the simulation and with the real systems had to be identical. Two hosts have been used that were connected with four links having configurable bandwidths, delays and packet loss rates. Each real host had two quad-core 2.4 GHz Xeon CPUs. They were also equipped with four 1000 MBit/s interfaces and running FreeBSD 9.0 (Beta2). Dummynet [82, 21] was used to limit the bandwidth, increase the delays or add packet loss. A forwarding SSH connection was set up between the hosts and one or multiple TCP connections were forwarded over it from one host to the other. This setup is depicted in Figure 8.10. For simplicity and to avoid additional delays, the senders and the SSH client were running on one host, and the SSH server and the receiver on the other. Because each host had eight CPU cores and all applications were single-

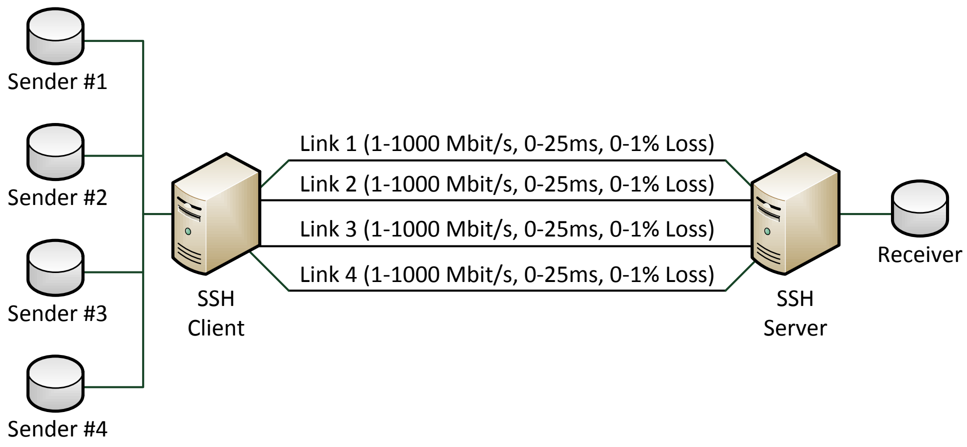

Sender #1
Sender #2
Sender #3
Sender #4
SSH Client
Link 1 (1-1000 Mbit/s, 0-25ms, 0-1% Loss)
Link 2 (1-1000 Mbit/s, 0-25ms, 0-1% Loss)
Link 3 (1-1000 Mbit/s, 0-25ms, 0-1% Loss)
Link 4 (1-1000 Mbit/s, 0-25ms, 0-1% Loss)
SSH Server
Receiver

Figure 8.10: SSH Measurement Setup

threaded, this did not affect the performance. The delays and achieved throughput of each forwarded connection were measured.

Multiple configurations have been used for the series of measurements to cover the scenarios in which SCTP behaves different to TCP. Each measurement has been repeated 25 times to calculate average values and 95% confidence intervals.

8.6.2 Reliability with Multi-Homing

SCTP offers increased reliability by using multiple network links for a single association. If the primary link fails, another one can be used to continue the association. To compare this to TCP, two links were used, limited to 100 MBit/s to avoid side effects from a maximum load of the CPUs on the real systems. One host tried to send 100 MB of data to the other, and 5 seconds after the transfer was started the primary link failed, which was achieved by increasing the packet loss rate to 100%. The duration of the outage before the link was available again was increased during the measurement. While SSH over TCP can only use a single link, SSH over SCTP is dual homed, so a fallback is possible.

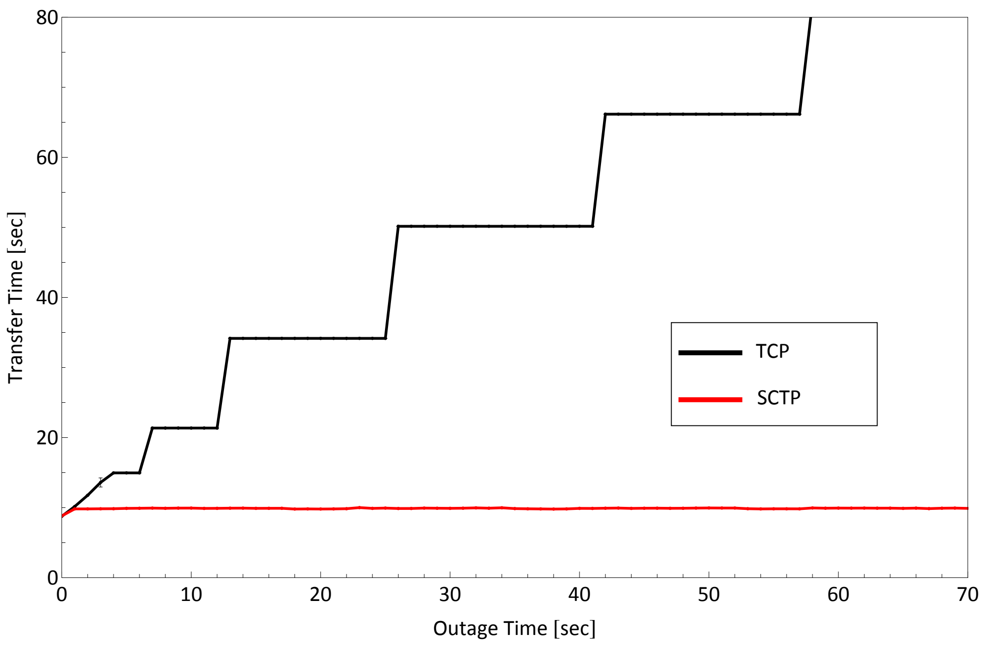


Figure 8.11: Simulation of Transfer with Link Outage

The results of the simulation of this scenario are shown in Figure 8.11. Without any network failures, the transmission lasted about 9 seconds with both TCP and SCTP. If there was an outage, TCP retransmitted the last packet until the link was available again, so the transfer time was prolonged by at least the duration of the outage. However, since TCP's timers are increased exponentially, the prolongation was not linear, because even when the network was already available again, the time until the next retransmission still had to be awaited. This caused the graph to increase in steps, because as long as the duration of the outage did not require an additional retransmission, the time until the data transfer resumed stayed the same. In the simulation the initial RTO is 200 ms and doubled for every retransmission, as stated by the specification. This can be described with $(0.2 * 2^{n-1})$ s, where n is the number of retransmissions. Therefore, the overall time necessary until retransmission n can be described with Equation 8.1.

$$\sum_{x=1}^{n} 0.2 * 2^{x-1}\ s \tag{8.1}$$

So if the link outage lasts 4 seconds for example, the first packet arrives after 5 retransmissions or 6.2 seconds. After 60 seconds of network outage, the SSH connection over TCP was terminated because of too many unsuccessful retransmissions.

SCTP was measured with its Quick Failover extension [58] activated, so the failure of the primary network link was assumed after a single Retransmission Timeout (RTO). For SCTP, this is by default one second compared to only 200 ms for TCP. Therefore, the time to complete the transfer was only prolonged by the time necessary to detect the failure and change to a fallback link, which was done within a second, no matter how long the primary link was actually unavailable.

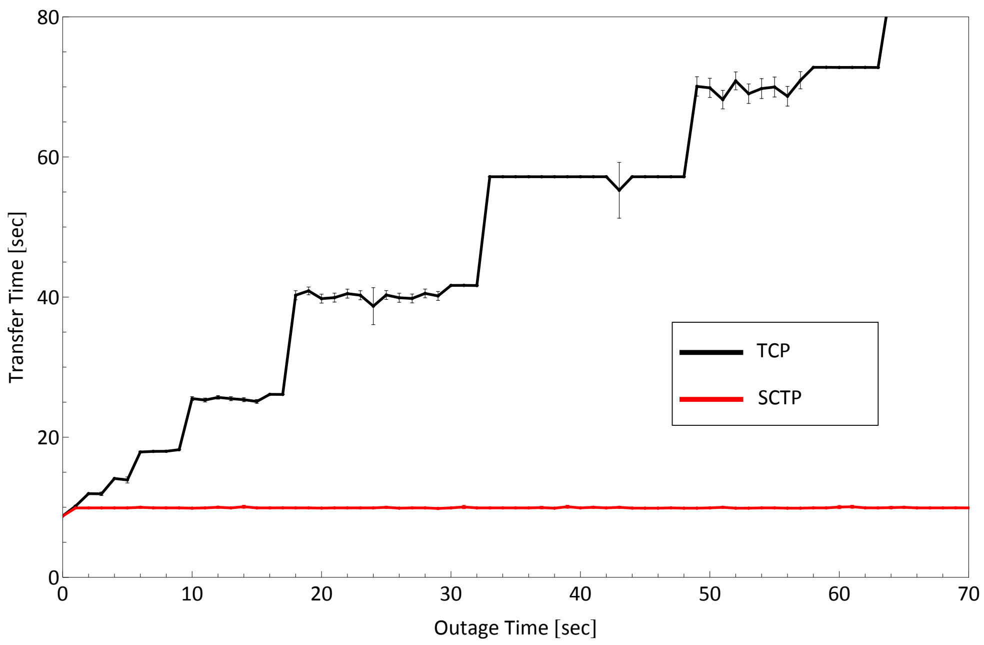


Figure 8.12: Real System Measurement of Transfer with Link Outage

The same measurement has also been done on real systems and the results are shown in Figure 8.12. While the behavior of SCTP is exactly the same as in the simulation, the time necessary to complete the transfer with TCP was slightly better. This was because FreeBSD uses a different retransmission strategy to lower the time between the retransmissions. By default, the initial value is 30 ms and is doubled for every retransmission, but in addition 200 ms is added, which can be described as $0.2 + (0.03 * 2^{n-1})$ s. The overall time necessary until retransmission n can be described with Equation 8.2.

$$\sum_{x=1}^{n} 0.2 + (0.03 * 2^{x-1})\ s \tag{8.2}$$

In Table 8.2 this approach is compared to that of the specification. So if the link outage lasts again 4 seconds, with the approach of FreeBSD the transfer resumes after 8 retransmissions or 5.31 seconds. Hence, FreeBSD is more time efficient in resuming the transfer after an outage, but uses significantly more retransmissions, which can worsen the situation in case the losses are caused by a network congestion.

Retransmission	**# 1**	**# 2**	**# 3**	**# 4**	**# 5**	**# 6**	**# 7**	**# 8**
TCP Specification	0.200s	0.400s	0.800s	1.600s	3.200s	6.400s	12.800s	25.600s
TCP in FreeBSD	0.230s	0.260s	0.320s	0.440s	0.680s	1.160s	2.220s	4.040s

Table 8.2: Retransmission Timings

8.6.3 Throughput with Multi-Homing and CMT

The second possible benefit of multi-homing is to increase the throughput by using multiple network links simultaneously with the CMT extension [36] for SCTP. Only the basic features of the extension were used, since maintaining fairness was not an issue in this scenario. To evaluate how the throughput increases with multiple links, a saturated TCP connection was forwarded over an SSH connection using multiple links with SCTP and a single one with TCP. The configured bandwidth of the links was increased from 1 to 200 MBit/s during the measurement and was always the same for all links.

The simulation results in Figure 8.13 show that as TCP can only make use of a single link, the achieved bandwidth increases linearly with the configured bandwidth. This is identical to SCTP with a single link, but with multiple links, SCTP uses all available links with CMT simultaneously, so the throughput is the configured bandwidth times the number of links used, up to almost 800 MBit/s with four links at 200 Mbit/s.

The comparison with the real systems, displayed in Figure 8.14, basically shows the same results, although the hardware limitation becomes visible. The possible

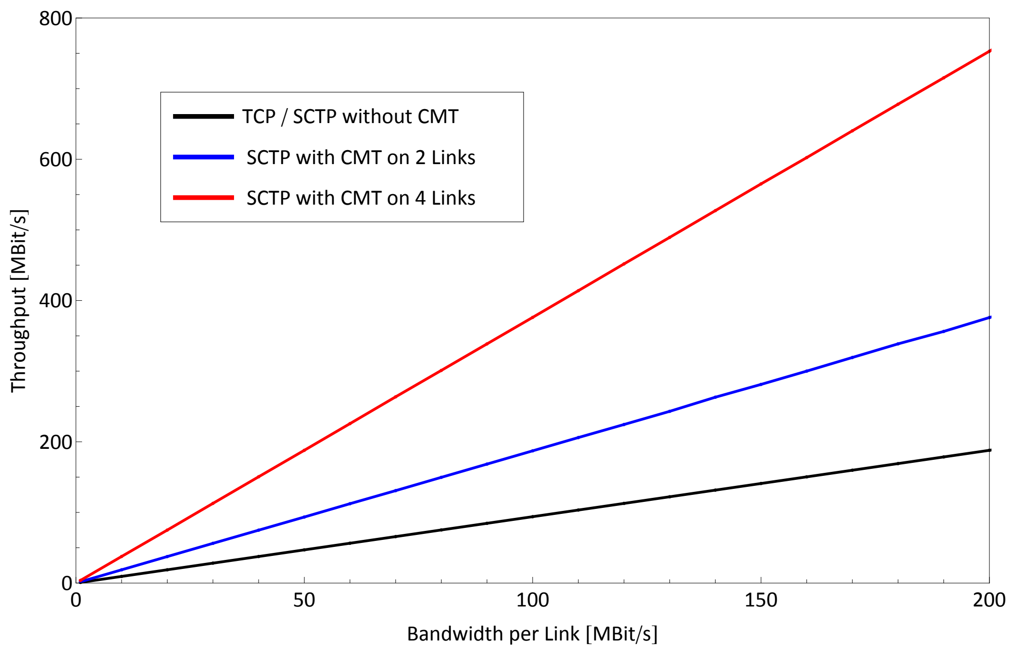


Figure 8.13: Simulation of Throughput with Multiple Links

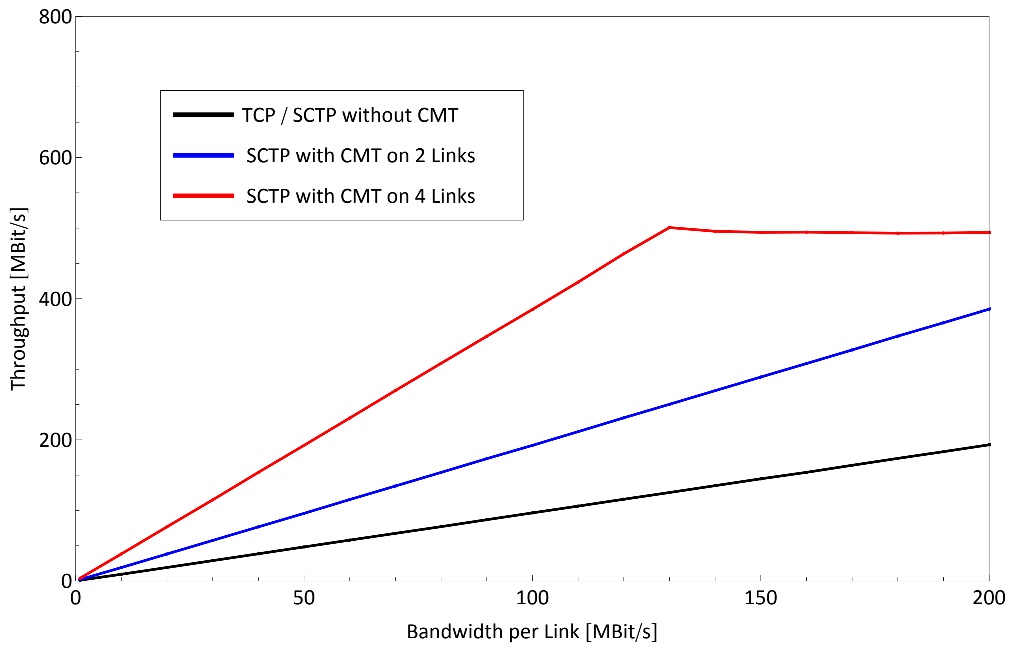


Figure 8.14: Real System Measurement of Throughput with Multiple Links

throughputs only increase until the CPU cannot perform the encryption and decryption fast enough. This limit is reached at about 550 MBit/s, slightly more without CMT and slightly less with CMT. This is because CMT requires more computing power and the implementation is still experimental, so optimizations have not been done yet.

The measurement shows that until a hardware limitation is reached, which can be mitigated by using multiple CPU cores for the cryptographic calculations of SSH as described in [73], the possible throughput can be increased with additional network links, which is especially effective on low bandwidth links, such as Internet connections.

8.6.4 Delay with Mapping Channels onto Streams

SCTP's multi-streaming mitigates the impact of head-of-line blocking. To allow SSH to benefit from this feature, its channels can be mapped onto the streams of SCTP. To examine how much the delay improves, four saturated TCP connections were forwarded over SSH with a single link that had a delay of 25 ms and a packet loss rate of 1% configured. These values were chosen to represent a 3G Internet link with High Speed Downlink Packet Access (HSDPA) [32], which is common nowadays. The delay of each message of the forwarded connections, from being sent to its reception, was measured. Figure 8.15 shows the distribution function of the message delays for the simulation.

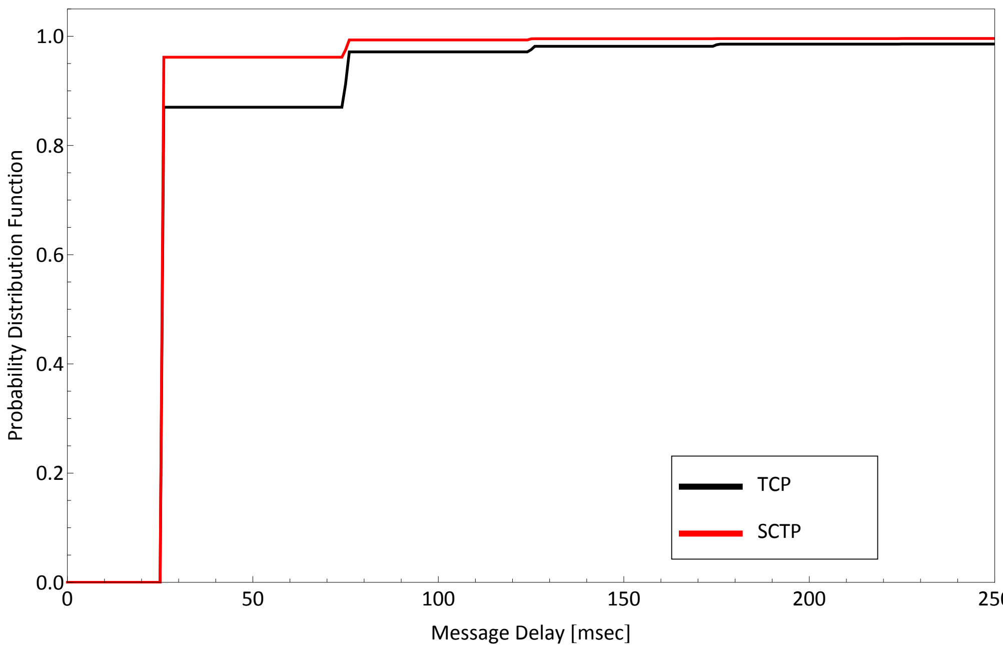


Figure 8.15: Simulation of Message Delays

With the configured delay and packet loss rate, 1% of the packets are expected to be lost and therefore delayed until their retransmission, while the others should arrive after the link latency of 25 ms. With TCP, however, almost 15% of the messages were delayed, because the order of the messages had to be preserved. Therefore, already received messages could not be delivered until previous ones had been retransmitted. The steps of the graph represent the retransmissions performed after each Round-Trip Time (RTT).

This issue is mitigated by SCTP's multi-streaming, which only requires the delay of messages of the same stream in case of a loss. Four connections were forwarded in this scenario, so four streams were used. This resulted in a delay of only about 5% of the messages on the same link.

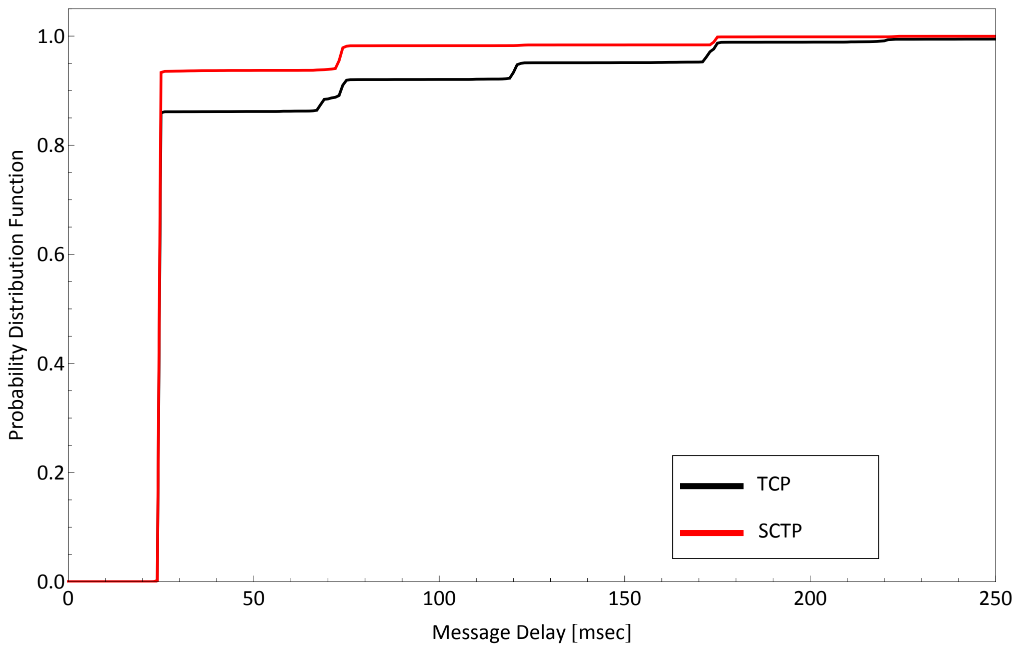


Figure 8.16: Real System Measurement of Message Delays

These results were confirmed with the real systems in Figure 8.16. The measurement therefore shows that SCTP can improve the message delay on lossy links, which is a benefit for time critical applications, such as interactive remote shells or real-time monitoring traffic.

8.6.5 Delay with Different Stream Schedulers

Multi-streaming also allows using specialized stream schedulers. SCTP supports the selection of a specific scheduler for each association and as already published in [89], all standard scheduling algorithms can be used. Scenarios in which this can be beneficial is priority scheduling to prefer time critical data, like remote shells or X11 forwarding, for instance. Such a scenario was measured by forwarding four TCP connections, three of which were saturated and one was only sending a message every 500 ms. This simulates using a remote shell while transferring files or forwarding other connections with the same SSH connection. For the measurement, a single link at 10 MBit/s was used without delay or packet loss, forming a bottleneck for the SSH connection. Hence, all forwarded connections had to share the reduced bandwidth. TCP was measured as a reference and compared to SCTP with first-come, first-served (FCFS), round-robin and priority scheduling. The message delay of the unsaturated "interactive" connection was measured.

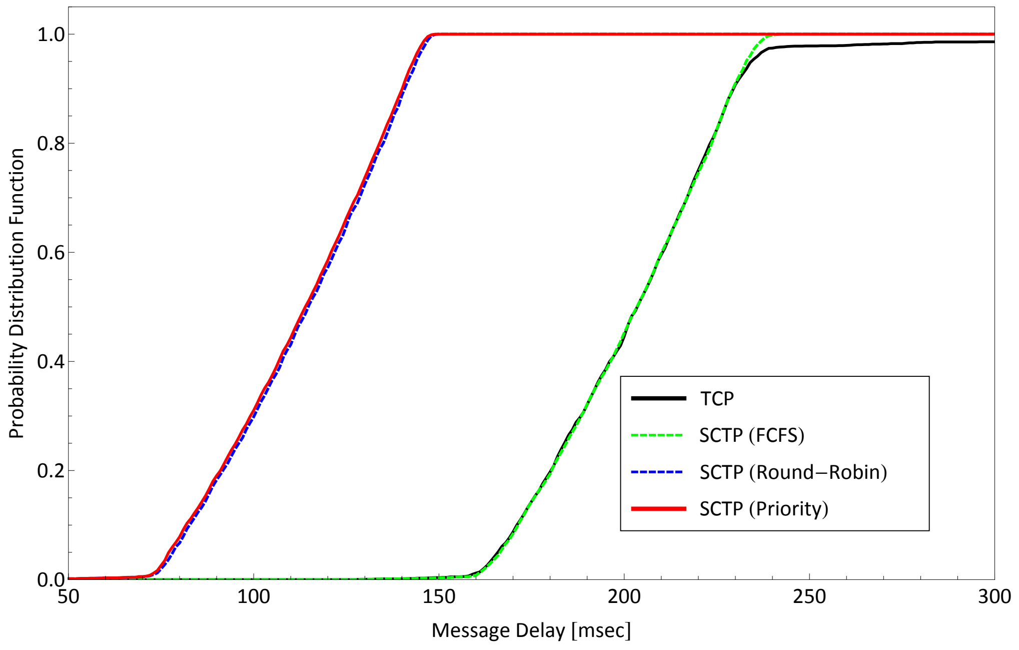


Figure 8.17: Simulation of Stream Scheduling Benefits

Figure 8.17 illustrates the results of the simulation. In this scenario all messages were delayed because of the bottleneck, so they remained in the send buffer until they could be transmitted. With FCFS scheduling, all messages were processed in the order in which they arrived from the application. The same was true for TCP, which does not support multi-streaming at all. Because the majority of the messages on the SSH connection belonged to the saturated connections, which filled up the send buffer, the interactive messages had to wait a considerably long time until they could be sent, resulting in the large delay of about 200 ms for TCP and SCTP with FCFS scheduling.

A round-robin scheduler on the other hand cycles through all available streams when choosing the next message that will be sent. In this case there were four streams, so the stream with the interactive messages was chosen every fourth time. Whenever an interactive message was available, there were never more than three messages that were sent first, so the delay was about 110 ms and so significantly less than with FCFS scheduling or with TCP. With priority scheduling, a higher priority can be assigned to the streams with interactive service channels. Even if there are many other messages already in the send buffer, a high priority message will always be sent first. This results in the smallest possible delay, which is especially effective when many streams are used, or the transmission of a single message takes a long time because of large delays or a very low bandwidth. In this case only four streams have been used, so the difference to round-robin scheduling was still very small.

The same behavior could be observed with the real systems with only small deviations, as shown in Figure 8.18.

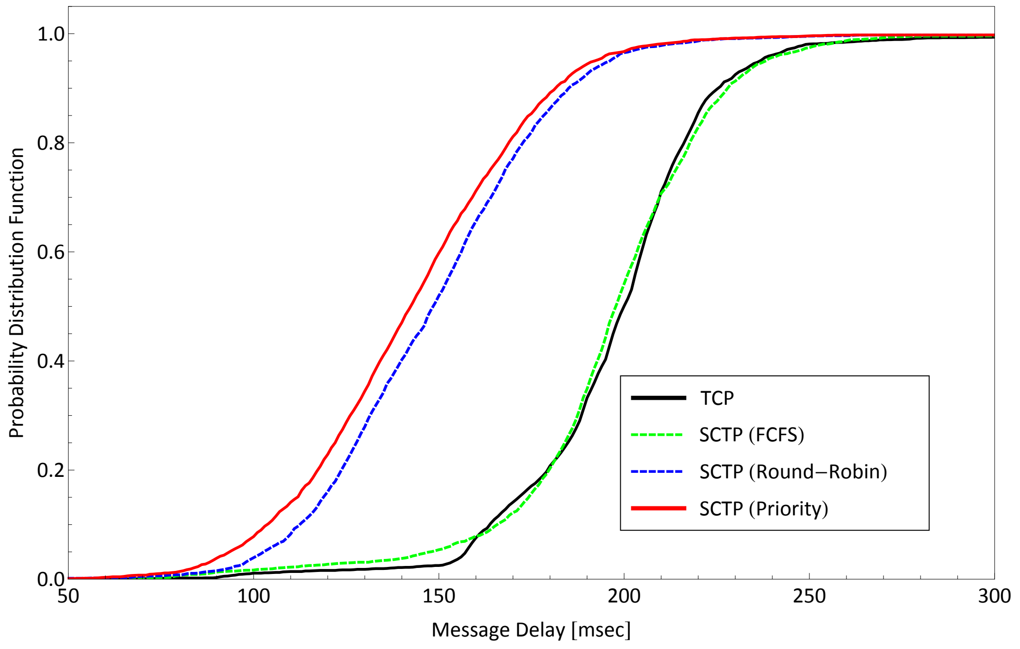


Figure 8.18: Real System Measurement of Stream Scheduling Benefits

8.7 Conclusion

SSH is the most common protocol for application protocol tunneling or forwarding. It is only specified for TCP, so extending it for SCTP support has been suggested in Section 8.1. Only minor modifications were necessary to prevent SCTP-specific information from getting lost during the forwarding.

Since the SSH specification does not limit it to TCP as its transport protocol, using SSH over SCTP has been proposed in Section 8.2. Without further modifications, this allows to use SCTP's multi-homing features for SSH connections to increase the reliability, or to increase the throughput with the CMT extension.

SCTP supports multi-streaming to mitigate the impact of head-of-line blocking and by mapping SSH's channels onto the streams, SSH can benefit from it. This also allows the use of specialized stream schedulers. By extending the OMNeT++ simulation and OpenSSH with prototype implementations, these benefits have been proven with measurements in Section 8.6. Additionally, SCTP's unreliability features can be used to support the forwarding of unreliable transport protocols, as suggested in Section 8.3.

Chapter 9

Conclusion

In this final chapter, the previously achieved results on the stated goal to analyze strategies and propose solutions for securing end-to-end connections with SCTP are recalled. Additionally, an outlook is given on possible future work and research that can still be done on this topic.

9.1 Achieved Results

The analysis of possible security strategies for SCTP in Chapter 5 resulted in two categories: applications can either be security-aware and make direct use of security features, or security-agnostic, and security has to be provided via tunneling mechanisms.

A common approach for security-aware applications is to use external libraries, to avoid starting all over again for every new application. Libraries, such as OpenSSL, can provide TLS or DTLS security for TCP and UDP, respectively. Unfortunately, these security protocols have not been developed with SCTP in mind, so they can only be used with several limitations. Therefore, an adaptation of DTLS for SCTP has been introduced in Chapter 6 that is able to secure SCTP associations without limiting the features that can be used, and can still be realized with a library. Performance measurements in Section 6.4 have shown that with optimizations, this approach can even be a viable alternative to TLS over TCP, despite the additional computing power necessary for SCTP. The adaptations have been contributed to the IETF for standardization, which has accepted and published them in RFC 6083 [105]. The implementation described in Section 6.3 has been accepted by the maintainers of OpenSSL and is included since stable release 1.0.1 [61].

Security-agnostic applications are usually secured by tunneling their connections, which avoids modifications to both the application and its protocols. Tunneling can be realized on different layers, either network, transport, or application layer. Network layer tunneling is often called VPN and common examples are IPsec or the DTLS-based Cisco VPN software. This approach supports arbitrary transport protocols, which includes SCTP. The same applies to transport protocol tunneling, which can also be done with DTLS.

DTLS-based tunneling, however, can still be improved by extending the DTLS protocol, which has been done in Chapter 7. Since DTLS is used with the message-oriented UDP, a Path MTU Discovery is necessary and also required by the specification, although UDP does not provide such a feature, and also DTLS itself does not have any suitable mechanisms. Additionally, UDP is connectionless, but there is no mechanism other than a costly renegotiation to check the availability of the peer. Both issues can be solved with the Heartbeat extension proposed in Section 7.2. It adds a request and response protocol to DTLS that can be used for keep-alive checks and also to perform a Path MTU Discovery, because it allows to vary the message size. We have presented this extension to the TLS working group at the IETF. It has been accepted for standardization and is published in RFC 6520 [93]. The implementation has been accepted by the OpenSSL maintainers and is included since stable release 1.0.1 [61]. Another extension proposed in Section 7.4, which makes use of the Heartbeats, is DTLS Mobility, to add mobility features for DTLS connections. This allows continuing the connection regardless of changes of the used IP addresses and therefore enables a mobile host to move between networks, for example Wi-Fi and 3G, without reconnecting. This can be used for mobile direct connections or also mobile network or transport tunnels. Functionality evaluations in Section 7.6 for several scenarios with the prototype implementation for OpenSSL, described in Section 7.5, have proven that the connection stays alive even with multiple address changes, and is restored within a single RTT after a new physical link has been established. With tunneling it is also possible to add mobility to connection-oriented and reliable protocols like TCP. If the address has changed and the link has been reestablished, the data transfer continues immediately with the first successful retransmission. This is far more efficient than to reconnect to the new address after waiting until the loss of connectivity is detected because the connection is given up when the maximum number of retransmissions is reached.

Application protocol tunneling, or forwarding, requires explicit support of the transport protocol used by the forwarded connection. The most common application tunneling protocol SSH, however, is only specified for TCP-based connections. The use of SSH for and with SCTP has been considered in Chapter 8. An extension to the protocol for SCTP forwarding support has been suggested in Section 8.1, which requires only minor modifications to retain SCTP-specific parameters, like multi-streaming information, during the forwarding.

The SSH specification does not explicitly require TCP to be used as the transport protocol for SSH connections, only a reliable service and message order preservation is necessary. This can also be provided by SCTP, so the use of SCTP has been proposed as an alternative in Section 8.2, with the immediate advantage that its multi-homing features become available. This enables not only to realize an increased reliability with additional links for failover, but also increased throughput with the CMT extension for SCTP, using all available links simultaneously.

With manageable modifications, SSH's channels can be mapped onto the streams of SCTP, which reduces the impact of head-of-line blocking and therefore leads to smaller delays. Additionally, this allows to make use of specialized stream schedulers as described in Section 8.2.2, like fair bandwidth scheduling, to distribute the available bandwidth equally to every forwarded connection, or priority scheduling, to prefer interactive remote shells over plain data transfer. These expected benefits with SCTP have been proven with measurements in simulations as well as with real systems in Section 8.6. To achieve this, the OMNeT++ simulation environment and the open source tool OpenSSH have been extended as described in Sections 8.4 and 8.5. Lastly, the use of SCTP with its unordered transfer and PR-SCTP extension also supports the forwarding of unreliable transport protocols, like DCCP or UDP, as suggested in Section 8.3.

9.2 Future Work

The Heartbeat extension for DTLS has already been standardized and included in OpenSSL. The Path MTU Discovery based on Heartbeats is available as a prototype implementation, but has not yet been provided to the OpenSSL maintainers for inclusion in an upcoming release. The same is true for the Mobility extension, which first has to be introduced to the IETF for standardization, so the Internet Assigned Numbers Authority (IANA) will assign the necessary type values for the new parameters. The modifications for OpenSSH to support SCTP-based SSH connections have to be introduced to the project maintainers, so they can be included in an upcoming release. However, further research is still required for the forwarding of UDP communication before a deployment is possible.

The suggested SCTP Encryption Chunk needs to be realized and its impact on the performance of SCTP associations analyzed. Possible security issues also have to be discussed, to determine whether there are possible attacks and if countermeasures are necessary, for example by adding random values. Furthermore, the benefits of a specialized Congestion Control for SCTP with multi-channel protocols, such as SSH, also still need to be examined. Claiming a larger share of the bandwidth for forwarding connections is a possible optimization, but there may be also other algorithms that can be beneficial in specific scenarios.

The work done for this thesis will also be the basis for our next project, which is real-time communication between web browsers. The IETF working group RTCWeb is currently discussing the standardization of the necessary protocols. The proposed architecture includes generic data transfer with SCTP over a transport protocol tunnel based on DTLS over UDP. Hence, the gathered expertise in using these protocols is not only helpful in supporting the standardization activities, there are also several open questions where further research is necessary. Since this is the first application for this tunneling approach, a significant amount of work is still necessary to examine the impact of this decision. No prototype implementations are available yet, and it is not known if there may be possible performance as well as security issues that have to be addressed. Additionally, it would be interesting to investigate the possibilities of using this approach to security in other scenarios after it becomes available.

List of Figures

2.1 The Internet Model 5
2.2 SCTP Packet Format with Common Header and Chunks 7
2.3 SCTP Association Establishment 8
2.4 SCTP Association Teardown 9
2.5 SCTP Data Chunk Format 10
2.6 SCTP Sender Stream Scheduling 11
2.7 SCTP Receiver Stream Scheduling 12
2.8 SCTP Multi-homing 13

3.1 IPsec Transport and Tunnel Modes 18
3.2 TLS Protocol Structure 19
3.3 TLS Record Header 19
3.4 TLS Handshake Message Header 20
3.5 TLS Handshake 21
3.6 TLS Handshake for Session Resumption 22
3.7 DTLS Record Header 23
3.8 DTLS Handshake Message Header 24
3.9 DTLS Handshake 25
3.10 SSH Protocol Structure 26
3.11 SSH Key Exchange Message Flow 27
3.12 SSH Forwarding 29

4.1 OMNeT++ Graphical User Interface 32
4.2 OpenSSL Architecture 33

5.1 External Security 39
5.2 Network Protocol Tunneling 40
5.3 Transport Protocol Tunneling 41
5.4 Application Protocol Tunneling 42

6.1 Switching TSNs for Reordering Attack 48
6.2 Message Drop Attack with PR-SCTP 52
6.3 Modifications to OpenSSL to support DTLS-aware SCTP 53
6.4 Read SCTP Message Flow Chart 54

6.5 Measurement Setup . . . 55
6.6 SCTP-aware DTLS and TLS / TCP (Single Core) . . . 56
6.7 SCTP-aware DTLS and TLS / TCP (Dual Core) . . . 57
6.8 Expected Throughput with Renegotiations . . . 59
6.9 Measured Throughput with Renegotiations . . . 60
6.10 DTLS/SCTP Handshake with SACK Immediately Optimization . . 61
6.11 SCTP-aware DTLS with HMAC Optimization (Single Core) . . . 63
6.12 SCTP-aware DTLS with HMAC Optimization (Dual Core) . . . 63

7.1 Heartbeat Message . . . 66
7.2 Heartbeat Hello-Extension . . . 67
7.3 Path MTU Discovery Flow Chart . . . 69
7.4 DTLS Record Header with Added Connection Identifier . . . 72
7.5 Mobility Hello-Extension . . . 72
7.6 Implicit Address Change with Verification . . . 73
7.7 Modifications to OpenSSL to support Heartbeats and DTLS Mobility 75
7.8 DTLS Mobility Flow Chart . . . 76
7.9 Laptop Connected over a NAT Middlebox . . . 78
7.10 Laptop Moving to Another Wi-Fi Network . . . 78
7.11 Cell Phone Moving to Another Wi-Fi Network with 3G . . . 79
7.12 TCP Connection over a Mobile DTLS Tunnel . . . 80
7.13 Throughput of the TCP Connection, Cable to WLAN Change . . . 82
7.14 Throughput of the TCP Connection, WLAN to UMTS Change . . . 82

8.1 SSH Data Message Extended for SCTP . . . 86
8.2 SSH Request Message Extended for SCTP . . . 86
8.3 SCTP Forwarding over SSH . . . 87
8.4 SSH Channel to SSH Transport Layer Multiplexing . . . 88
8.5 SSH Sequence Numbers with SCTP . . . 89
8.6 Extended SSH Transport Layer . . . 90
8.7 SCTP Encryption Chunk . . . 91
8.8 Fair Bandwidth Scheduling Implementation . . . 93
8.9 Streams and Channel Mapping for SCTP Forwarding . . . 97
8.10 SSH Measurement Setup . . . 101
8.11 Simulation of Transfer with Link Outage . . . 102
8.12 Real System Measurement of Transfer with Link Outage . . . 103
8.13 Simulation of Throughput with Multiple Links . . . 105
8.14 Real System Measurement of Throughput with Multiple Links . . . 105
8.15 Simulation of Message Delays . . . 106
8.16 Real System Measurement of Message Delays . . . 107
8.17 Simulation of Stream Scheduling Benefits . . . 108
8.18 Real System Measurement of Stream Scheduling Benefits . . . 109

List of Tables

5.1 Comparison of Security Solutions 43

6.1 DTLS Handshake Timing 58

7.1 Some Possible MTU Values 68
7.2 Comparison of Different Approaches to Mobility 71
7.3 Mobility Delays 79
7.4 Mobility Delays with a DTLS Tunnel 81

8.1 Forwarding Characteristics of Transport Protocols 96
8.2 Retransmission Timings 104

List of Abbreviations

ADD-IP	Dynamic Address Reconfiguration Extension for SCTP
AES	Advanced Encryption Standard
AH	Authentication Header
API	Application Programming Interface
BIO	Basic Input/Output
CID	Connection Identifier
CMT	Concurrent Multipath Transfer
CTX	Context
DCCP	Datagram Congestion Control Protocol
DFG	Deutsche Forschungsgemeinschaft
DOS	Denial-of-Service attack
DSL	Digital Subscriber Line
DTLS	Datagram Transport Layer Security
ECN	Explicit Congestion Notification
ESP	Encapsulating Security Payload
FCFS	First-Come, First-Served
FTP	File Transfer Protocol
GUI	Graphical User Interface
HMAC	Hash-based Message Authentication Code
HSDPA	High Speed Downlink Packet Access
HTTP	Hypertext Transfer Protocol

IANA	Internet Assigned Numbers Authority
ICE	Interactive Connectivity Establishment
ICMP	Internet Control Message Protocol
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IMAP	Internet Message Access Protocol
IP, IPv4	Internet Protocol
IPFIX	IP Flow Information Export
IPsec	Internet Protocol Security
IPv6	Internet Protocol Version 6
IV	Initialization Vector
KPA	Known-Plaintext Attack
MAC	Message Authentication Code
MBP	MacBook Pro
MP-TCP	Multipath-TCP
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NKE	Network Kernel Extension
NS-2	Network Simulator 2
PPI	Payload Protocol Identifier
PPP	Point-to-Point Protocol
PR-SCTP	Partial Reliability Extension for SCTP
RFC	Request for Comments
RSerPool	Reliable Server Pooling
RTCWeb	Real-Time Communication in Web Browsers
RTO	Retransmission Timeout
RTT	Round-Trip Time

SA	Security Association
SACK	Selective Acknowledgment
SCTP	Stream Control Transmission Protocol
SCTP-AUTH	Authenticated Chunks for SCTP Extension
SHA	Secure Hash Algorithm
SID	Stream Identifier
SMTP	Simple Mail Transfer Protocol
SPD	Security Policy Database
SRTP	Secure Real-time Transport Protocol
SS7	Signaling System No. 7
SSH	Secure Shell
SSL	Secure Sockets Layer
SSN	Stream Sequence Number
TCP	Transmission Control Protocol
TCP-R	TCP Redirection
TLS	Transport Layer Security
TLV	Type-Length-Value
TSN	Transmission Sequence Number
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunications System
VNC	Virtual Network Computing
VPN	Virtual Private Network
VoIP	Voice over IP
Wi-Fi	Wireless Fidelity (IEEE 802.11)
X11	X Window System (Version 11)

Bibliography

[1] Apple. Xgrid: High Performance Computing for the Rest of Us. *http://developer.apple.com/hardwaredrivers/hpc/xgrid_intro.html*, Retrieved February, 6th 2012.

[2] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). *RFC 3711*, March 2004.

[3] M. Bellare, R. Canetti, and H. Krawczyk. Keying Hash Functions for Message Authentication. *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '96)*, 1109:1–15, 1996.

[4] S. Bellovin, J. Ioannidis, A. Keromytis, and R. Stewart. On the Use of Stream Control Transmission Protocol (SCTP) with IPsec. *RFC 3554*, July 2003.

[5] T. Berson. Skype Security Evaluation. *IEEE Transactions on Neural Networks*, 10, October 2005.

[6] Cisco. Release Notes for Cisco AnyConnect VPN Client. *http://www.cisco.com/en/US/docs/security/vpn_client/anyconnect/anyconnect24/release/notes/anyconnect24rn.html*, Retrieved July, 12th 2011.

[7] Computer Systems Research Group and K. Sevcik. *Priority Scheduling Disciplines in Queueing Network Models of Computer Systems.* University of Toronto, 1977.

[8] A. Conta, S. Deering, and M. Gupta. Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. *RFC 4443*, March 2006.

[9] D. Converse, J. Gettys, A. Mento, and R. Scheifler. *X Window System: Core and Extension Protocols.* Digital Press, 1997.

[10] M. Crispin. Internet Message Access Protocol – Verison 4rev1. *RFC 3501*, March 2003.

[11] J. Crowcroft and P. Oechslin. Differentiated End-to-End Internet Services using a Weighted Proportional Fair Sharing TCP. *SIGCOMM Comput. Commun. Rev.*, 28:53–69, July 1998.

[12] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. *RFC 2460*, December 1998.

[13] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, 1989.

[14] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. *RFC 5246*, August 2008.

[15] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644 – 654, November 1976.

[16] N. Doraswamy and D. Harkins. *IPSec (2nd Edition)*. Prentice Hall, 2003.

[17] T. Dreibholz. *Reliable Server Pooling – Evaluation, Optimization and Extension of a Novel IETF Architecture*. PhD thesis, University of Duisburg-Essen, Faculty of Economics, Institute for Computer Science and Business Information Systems, March 2007.

[18] T. Dreibholz, M. Becke, E. P. Rathgeb, and M. Tüxen. On the Use of Concurrent Multipath Transfer over Asymmetric Paths. *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, December 2010.

[19] T. Dreibholz, I. Rüngeler, R. Seggelmann, M. Tüxen, E. P. Rathgeb, and R. Stewart. Stream Control Transmission Protocol: Past, Current, and Future Standardization Activities. *IEEE Communications Magazine*, 49(4):82–88, April 2011.

[20] L. Dryburgh and J. Hewett. *Signaling System No. 7 (SS7/C7): Protocol, Architecture, and Services (Networking Technology)*. Cisco Press, 2004.

[21] Dummynet. Traffic Shaper, Bandwidth Manager and Delay Emulator. *http://info.iet.unipi.it/~luigi/ip_dummynet/*, Retrieved April, 16th 2012.

[22] U. Esbold, E. Rathgeb, and A. Jungmaier. Secure SCTP: A versatile secure transport protocol. *Telecommunication Systems*, 27(2–4):273–296, 2004.

[23] FIPS 180-2. Secure Hash Standard. *National Institute of Standards and Technology (NIST)*, August 2002.

[24] FIPS 197. Advanced Encryption Standard (AES). *National Institute of Standards and Technology (NIST)*, November 2001.

[25] A. Ford, C. Raiciu, M. Handley, S. Barre, and J. Iyengar. Architectural Guidelines for Multipath TCP Development. *RFC 6182*, March 2011.

[26] P. Ford-Hutchinson. Securing FTP with TLS. *RFC 4217*, October 2005.

[27] B. A. Forouzan. *TCP/IP Protocol Suite*. McGraw-Hill Medical Publishing, 2010.

[28] D. Funato, K. Yasuda, and H. Tokuda. TCP-R: TCP mobility support for continuous operation. *Proceedings of the International Conference on Network Protocols (ICNP)*, pages 229–236, October 1997.

[29] G. Heinz II and P. Amer. Priorities in Stream Transmission Control Protocol (SCTP) Multistreaming. Master's thesis, Protocol Engineering Laboratory, University of Delaware, January 2003.

[30] P. Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. *RFC 3207*, February 2002.

[31] C. Hohendorf, E. P. Rathgeb, E. Unurkhaan, and M. Tüxen. Secure end-to-end transport over SCTP. *JCP*, 2(4):31–40, 2007.

[32] H. Holma and A. Toskala. *HSDPA/HSUPA for UMTS: High Speed Radio Access for Mobile Communications*. Wiley, 2006.

[33] I. Rüngeler. SCTP – Evaluating, Improving and Extending the Protocol for Broader Deployment . *Dissertation Universität Duisburg-Essen*, December 2009.

[34] INET Framework. Networks simulation package. `http://inet.omnetpp.org`, Retrieved February, 3rd 2012.

[35] International Organization for Standardization. Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. *ISO/IEC 7498-1*, November 1994.

[36] J. R. Iyengar, R. D. Amer, and R. Stewart. Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths. *IEEE/ACM Transactions on Networking*, 14(5):95–964, 2006.

[37] V. J, M. Messier, and P. Chandra. *Network Security with OpenSSL*. O'Reilly Media, 2002.

[38] D. Johnson, C. Perkins, and J. Arkko. Mobility Support in IPv6. *RFC 3775*, June 2004.

[39] A. Jungmaier, E. Rescorla, and M. Tüxen. Transport Layer Security over Stream Control Transmission Protocol. *RFC 3436*, December 2002.

[40] C. Kaufman. Internet Key Exchange (IKEv2) Protocol. *RFC 4306*, December 2005.

[41] S. Ken. IP Authentication Header. *RFC 4302*, December 2005.

[42] S. Ken. IP Encapsulating Security Payload (ESP). *RFC 4303*, December 2005.

[43] S. Kent and K. Seo. Security Architecture for the Internet Protocol. *RFC 4301*, December 2005.

[44] R. Khare and S. Lawrence. Upgrading to TLS Within HTTP/1.1. *RFC 2817*, May 2000.

[45] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). *RFC 4340*, March 2006.

[46] S. Lindskog and A. Brunstrom. An End-to-End Security Solution for SCTP. *Proceedings of the 2008 Third International Conference on Availability, Reliability and Security (ARES)*, pages 526–531, 2008.

[47] D. Maltz and P. Bhagwat. MSOCKS: An Architecture for Transport Layer Mobility. *Proceedings of the Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies*, 3:1037–1045, 1998.

[48] M. Mathis and J. Heffner. Packetization Layer Path MTU Discovery. *RFC 4821*, March 2007.

[49] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. *RFC 2018*, October 1996.

[50] Microsoft Skype Division. Skype. *http://www.skype.com*, Retrieved June, 14th 2012.

[51] N. Modadugu and E. Rescorla. The Design and Implementation of Datagram TLS. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2004.

[52] J. Mogul and S. Deering. Path MTU Discovery. *RFC 1191*, November 1990.

[53] Münster University of Applied Sciences. DTLS bugfixes and sample code. *http://sctp.fh-muenster.de*, Retrieved November, 17th 2011.

[54] R. R. Muntz and E. G. Coffman Jr. Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems. *Journal of the ACM*, 17(2):324–338, April 1970.

[55] J. Nagle. On Packet Switches with Infinite Storage. *Communications, IEEE Transactions on*, 35(4):435–438, April 1987.

[56] Network Simulator 2 (NS-2). Discrete event simulator. `http://www.isi.edu/nsnam/ns`, Retrieved February, 3rd 2012.

[57] C. Newman. Using TLS with IMAP, POP3 and ACAP. *RFC 2595*, June 1999.

[58] Y. Nishida and P. Natarajan. Quick Failover Algorithm in SCTP. *IETF draft-nishida-tsvwg-sctp-failover-05 (work in progress)*, March 2012.

[59] OpenSSH Project. Project History and Credits. `http://openssh.com/history.html`, Retrieved February, 3rd 2012.

[60] OpenSSL Project. The Open Source toolkit for SSL/TLS. `http://openssl.org`, Retrieved January, 25th 2011.

[61] OpenSSL Project. ChangeLog. `http://openssl.org/news/changelog.html`, Retrieved November, 17th 2011.

[62] OpenVPN Technologies Inc. OpenVPN. `http://www.openvpn.net`, Retrieved June, 14th 2012.

[63] OPNET Technologies. OPNET Modeler. `http://www.opnet.com/solutions/network_rd/modeler.html`, Retrieved February, 3rd 2012.

[64] C. Patrikakis, M. Masikos, and O. Zouraraki. Distributed Denial of Service Attacks. *Internet Protocol Journal, Cisco Systems*, 7(4):13–35, July 2004.

[65] C. Perkins. IP Mobility Support for IPv4. *RFC 3344*, August 2002.

[66] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Springer, 2012.

[67] J. Postel. User Datagram Protocol. *RFC 768*, August 1980.

[68] J. Postel. Internet Control Message Protocol. *RFC 792*, September 1981.

[69] J. Postel. Internet Protocol. *RFC 791*, September 1981.

[70] J. Postel. Transmission Control Protocol. *RFC 793*, September 1981.

[71] J. Postel and J. Reynolds. Telnet Protocol Specification. *RFC 854*, May 1983.

[72] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. *RFC 3168*, September 2001.

[73] C. Rapier and B. Bennett. High speed bulk data transfer using the SSH protocol. *Proceedings of the 15th ACM Mardi Gras conference (MG '08)*, pages 11:1–11:7, 2008.

[74] J. Reardon. Improving Tor using a TCP-over-DTLS Tunnel. *Master's thesis, University of Waterloo*, September 2008.

[75] E. Rescorla. RTCWEB Security Architecture. *IETF draft-ietf-rtcweb-security-arch-00 (Work in Progress)*, January 2012.

[76] E. Rescorla. HTTP Over TLS. *RFC 2818*, May 2000.

[77] E. Rescorla. Keying Material Exporters for Transport Layer Security (TLS). *RFC 5705*, October 2009.

[78] E. Rescorla and N. Modadugu. Datagram Transport Layer Security Version 1.2. *RFC 6347*, January 2012.

[79] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper. Virtual network computing. *Internet Computing, IEEE*, 2(1):33–38, January 1998.

[80] M. Riegel and M. Tüxen. Mobile SCTP — Transport Layer Mobility Management for the Internet. *Proceedings of the SoftCOM 2002, International Conference on Software, Telecommunications and Computer Networks*, pages 305 – 309, 2002.

[81] R. L. Rivest, A. Shamir, and L. M. Adleman. Cryptographic communications system and method. *Patent US 4405829*, September 1983.

[82] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[83] S. Robinson. *Simulation: The Practice of Model Development and Use*. Wiley, 2004.

[84] J. Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols. *RFC 5245*, April 2010.

[85] A. Samukic. UMTS universal mobile telecommunications system: development of standards for the third generation. *IEEE Transactions on Vehicular Technology*, 47(4):1099 –1104, November 1998.

[86] Scalable Networks. QualNet. *http://www.scalable-networks.com*, Retrieved April, 10th 2012.

[87] R. Seggelmann, I. Rüngeler, M. Tüxen, and E. P. Rathgeb. Parallelizing OMNeT++ simulations using Xgrid. *Proceedings of the 2nd International Conference on Simulation Tools and Techniques (Simutools '09)*, pages 69:1–69:8, 2009.

[88] R. Seggelmann, M. Tüxen, and E. P. Rathgeb. Design and Implementation of SCTP-aware DTLS. *Proceedings of the International Conference on Telecommunication and Multimedia (TEMU 2010)*, July 2010.

[89] R. Seggelmann, M. Tüxen, and E. P. Rathgeb. Stream Scheduling Considerations for SCTP. *Proceedings of the 18th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, Sept. 2010.

[90] R. Seggelmann, M. Tüxen, and E. P. Rathgeb. DTLS Mobility. *Proceedings of the 13th International Conference on Distributed Computing and Networking (ICDCN 2012)*, January 2012.

[91] R. Seggelmann, M. Tüxen, and E. P. Rathgeb. SSH Over SCTP – Optimizing a Multi-Channel Protocol by Adapting It to SCTP. *Proceedings of the 8th International Symposium on Communication Systems, Networks and Digital Signal Processing (CSNDSP 2012)*, July 2012.

[92] R. Seggelmann, M. Tüxen, and E. P. Rathgeb. Strategies to Secure End-to-End Communication – And Their Application to SCTP-Based Communication. *PIK – Praxis der Informationsverarbeitung und Kommunikation*, 34(4), December 2012.

[93] R. Seggelmann, M. Tüxen, and M. Williams. Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension. *RFC 6520*, February 2012.

[94] W. Stallings. *Cryptography and Network Security: Principles and Practice (5th Edition)*. Prentice Hall, 2010.

[95] W. R. Stevens. *TCP/IP Illustrated, Vol. 1: The Protocols*. Addison-Wesley Professional, 1994.

[96] R. Stewart. Stream Control Transmission Protocol. *RFC 4960*, September 2007.

[97] R. Stewart, P. Lei, and M. Tüxen. Stream Control Transmission Protocol (SCTP) Stream Reset). *IETF draft-ietf-tsvwg-sctp-strrst-13 (Work in Progress)*, December 2011.

[98] R. Stewart, M. Ramalho, Q. Xie, M. Tüxen, and P. Conrad. Stream control transmission protocol (SCTP) Partial Reliability Extension. *RFC 3758*, May 2004.

[99] R. Stewart, M. Tuexen, K. Poon, P. Lei, and V. Yasevich. Sockets API Extensions for the Stream Control Transmission Protocol (SCTP). *RFC 6458*, December 2011.

[100] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. *RFC 2960*, October 2000.

[101] R. Stewart, Q. Xie, M. Tüxen, S. Maruyama, and M. Kozuka. Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration. *RFC 5061*, September 2007.

[102] R. R. Stewart and Q. Xie. *Stream Control Transmission Protocol (SCTP): A Reference Guide.* Addison-Wesley Professional, 2001.

[103] B. Trammell and E. Boschi. An introduction to IP flow information export (IPFIX). *Communications Magazine, IEEE*, 49(4):89–95, April 2011.

[104] M. Tüxen, I. Rüngeler, and R. Stewart. SACK-IMMEDIATELY extension for the Stream Control Transmission Protocol. *IETF draft-tuexen-tsvwg-sctp-sack-immediately-02 (work in progress)*, July 2009.

[105] M. Tüxen, R. Seggelmann, and E. Rescorla. Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP). *RFC 6083*, January 2011.

[106] M. Tüxen, R. Stewart, P. Lei, and E. Rescorla. Authenticated Chunks for the Stream Control Transmission Protocol (SCTP). *RFC 4895*, August 2007.

[107] A. Varga. The OMNeT++ discrete event simulation system. *Proceedings of the European Simulation Multiconference (ESM 2001)*, 2001.

[108] M. Welzl. *Network Congestion Control: Managing Internet Traffic.* John Wiley & Sons, 2005.

[109] M. Williams and J. Barrett. Mobile DTLS. *IETF draft-barrett-mobile-dtls-00 (work in progress)*, March 2009.

[110] Wireshark. Network protocol analyzer. *http://www.wireshark.org*, Retrieved February, 15th 2012.

[111] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. *RFC 4252*, January 2006.

[112] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. *RFC 4254*, January 2006.

[113] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. *RFC 4251*, January 2006.

[114] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Transport Layer Protocol. *RFC 4253*, January 2006.

[115] B. P. Zeigler, H. Praehofer, and T. G. Kim. *Theory of Modeling and Simulation, Second Edition.* Academic Press, 2000.

Curriculum Vitae

Name	Robin Seggelmann
29.07.1982	born in Oelde, Germany
08.1993 – 06.2002	High School Graduation Gymnasium Laurentianum, Warendorf
12.2002 – 09.2003	Alternative civilian service
09.2003 – 10.2006	B.Sc. in Applied Computer Science Münster University of Applied Sciences
01.2006 – 05.2006	Exchange Student Juniata College, Huntingdon, USA
10.2006 – 08.2008	M.Sc. in Information Technology Münster University of Applied Sciences
since 09.2008	Research Associate Network Programming Lab of the Department of Electrical Engineering and Computer Science Münster University of Applied Sciences
since 09.2009	Ph.D. Student University of Duisburg-Essen

Selbstständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und nur die angegebene Literatur und Hilfsmittel verwendet zu haben.

Robin Seggelmann
22. Oktober 2012