The Design, Implementation, and Performance Evaluation of Secure Socket SCTP 2.0

Nicklas Hasselström, Gunnar Hjern, Richard Hoorn, Marcus Hult, Johan Häger, Jens Syrén, Stefan Alfredsson & Stefan Lindskog
The Design, Implementation, and Performance Evaluation of Secure Socket SCTP 2.0

Nicklas Hasselström, Gunnar Hjern, Richard Hoorn, Marcus Hult, Johan Häger, Jens Syrén, Stefan Alfredsson and Stefan Lindskog

Department of Mathematics and Computer Science
Karlstad University, Sweden

Abstract

The Stream Control Transmission Protocol (SCTP) is a comparatively new transport protocol that presents some advanced features compared to other standardized transport protocols. However, there are currently no standardized end-to-end security solutions suited for SCTP. One proposal for end-to-end encryption is the Secure Socket SCTP (S²-SCTP) protocol, developed by researchers at Karlstad University. The security solution for SCTP described in this report uses key agreement for obtaining keys to be able to provide data confidentiality by encryption. The protocol is based on the S²-SCTP protocol, with smaller changes, and an overlaying management protocol has been designed and implemented. The management protocol is used to enable encryption and TLS authentication, to give a secure communication library over existing Berkeley Sockets. The performance evaluation of S²-SCTP compared to the already standardized end-to-end security solutions, i.e., TLS over SCTP and DTLS over SCTP, shows that S²-SCTP achieves a higher throughput while still maintaining most of the advantages of SCTP.

Keywords: S²-SCTP, SCTP, security, peer authentication, key agreement, data protection, TLS, DTLS, performance evaluation, throughput
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 SCTP</td>
<td>3</td>
</tr>
<tr>
<td>3 Existing End-to-End Security Solutions for SCTP</td>
<td>7</td>
</tr>
<tr>
<td>3.1 TLS over SCTP</td>
<td>7</td>
</tr>
<tr>
<td>3.2 DTLS over SCTP</td>
<td>7</td>
</tr>
<tr>
<td>3.3 SCTP over IPsec</td>
<td>8</td>
</tr>
<tr>
<td>3.4 Other Solutions</td>
<td>8</td>
</tr>
<tr>
<td>4 S²-SCTP</td>
<td>9</td>
</tr>
<tr>
<td>4.1 Peer Authentication</td>
<td>10</td>
</tr>
<tr>
<td>4.1.1 TLS Handshake</td>
<td>10</td>
</tr>
<tr>
<td>4.1.2 Management Protocol</td>
<td>12</td>
</tr>
<tr>
<td>4.2 Data Protection</td>
<td>12</td>
</tr>
<tr>
<td>4.2.1 Message Authentication</td>
<td>13</td>
</tr>
<tr>
<td>4.2.2 Message Data Layout for Key, Algorithm, IV, Data and Padding</td>
<td>13</td>
</tr>
<tr>
<td>4.2.3 Encrypted Data and Padding</td>
<td>14</td>
</tr>
<tr>
<td>4.2.4 Message Overhead</td>
<td>15</td>
</tr>
<tr>
<td>5 S²-SCTP Implementation</td>
<td>19</td>
</tr>
<tr>
<td>5.1 The Object Based Scheme</td>
<td>19</td>
</tr>
<tr>
<td>5.2 Peer Authentication Implementation</td>
<td>19</td>
</tr>
<tr>
<td>5.3 Central Functions – Send and Receive</td>
<td>22</td>
</tr>
<tr>
<td>5.3.1 The Receive Function</td>
<td>22</td>
</tr>
<tr>
<td>5.3.2 The Sending Function</td>
<td>23</td>
</tr>
<tr>
<td>5.3.3 The S²-SCTP Management Protocol Sending Function</td>
<td>23</td>
</tr>
<tr>
<td>6 Performance Evaluation</td>
<td>25</td>
</tr>
<tr>
<td>6.1 Experimental Setup</td>
<td>25</td>
</tr>
<tr>
<td>6.2 Test Cases</td>
<td>26</td>
</tr>
<tr>
<td>6.3 Test Script</td>
<td>26</td>
</tr>
<tr>
<td>6.4 Results</td>
<td>27</td>
</tr>
<tr>
<td>6.4.1 Loss Free Test</td>
<td>27</td>
</tr>
<tr>
<td>6.4.2 1% Package Loss Test</td>
<td>28</td>
</tr>
</tbody>
</table>
7 Future Work

7.1 Multi-homing .................................................. 31
7.2 Fragmented Messages ........................................... 31
7.3 One-to-many Socket Model ..................................... 31
7.4 Stored Sessions .................................................. 31
7.5 Thread Safety and Other Improvements ....................... 32
7.6 Renewed Key Exchange, Reduced IV Overhead and Counter Mode AES .................................................. 32

8 Concluding Remarks .............................................. 33

References .......................................................... 35

A Abbreviations ....................................................... 37

B Implementation Details ............................................ 39

B.1 Project Limitations ............................................. 39
B.2 Language and Compilers ....................................... 39
B.3 Compiler Directives ............................................. 39
B.4 Optimizations ................................................... 40
B.5 Source File Structure ........................................... 40
B.6 Current Status ................................................... 40
B.7 Strategy .......................................................... 40
B.8 Object Model .................................................... 41
B.9 Object-based Scheme and Object Model Strategy ........ 43
List of Figures

1. SCTP, similar to TCP and UDP, runs over the IP layer. ........................................... 3
2. SCTP 4-way handshake. .......................................................................................... 3
3. A generic chunk header. ......................................................................................... 4
4. SCTP data chunk. .................................................................................................... 5
5. SCTP message with IP header. ................................................................................ 5
6. SCTP common header. ............................................................................................ 5
7. The layer structure of \( S^2 \)-SCTP. ........................................................................... 9
8. A complete IP packet with an \( S^2 \)-SCTP packet including two data chunks of different encryption types. .......................................................... 10
9. TLS handshake messages. ..................................................................................... 11
10. LHV format design of the \( S^2 \)-SCTP management messages. ............................ 12
11. Structure of an encrypted message. ..................................................................... 14
12. The absolute efficiency of bundled normal SCTP and \( S^2 \)-SCTP data chunks. ..... 16
13. The quote of efficiency of bundled normal SCTP and \( S^2 \)-SCTP messages compared to bundled unencrypted messages. ........................................... 17
14. The SSL structure of OpenSSL does not allow selection of streams. ................. 20
15. Using buffers allows selection of streams. ............................................................. 20
16. State machine for authentication solution. ............................................................ 21
17. Comparison of OpenSSL over TCP (a) and OpenSSL with \( S^2 \)-SCTP (b). ........ 22
18. The setup for the performance experiment, with consisting of a client machine, a server machine and a machine that emulates the “Internet”. ....................... 25
19. A comparison between the three security solutions and regular SCTP as a capacity reference. ................................................................. 27
20. The graphs shows the individual value of all 30 plots for each user message size to show the result with a 95% confidence interval. ............................... 28
21. A comparison between base SCTP, \( S^2 \)-SCTP and DTLS over SCTP with 1% package loss. ................................................................. 29
22. The UML Scheme for the \( S^2 \)-SCTP implementation. ........................................... 42

List of Tables

1. Brief descriptions of some of the different chunk types used in SCTP [14]. .......... 4
2. Standardized SCTP extensions. ............................................................................... 6
3. Standardized end-to-end security solutions for SCTP. ......................................... 7
1 Introduction

This report was written as part of the examination requirements of the DVAE08 Engineering Project course at Karlstad University. The project itself was about designing, implementing and evaluating a new security solution for the Stream Control Transmission Protocol (SCTP) [14]. SCTP was originally designed to transport public switched telephone network (PSTN) data over IP networks, but has developed into a more extensive transport protocol. Since many existing security solutions are designed for other transport protocols, mainly TCP and UDP, a new security solution designed for SCTP is required. One suggestion for security adapted for SCTP is the Secure Socket SCTP (S2-SCTP). This report describes the design, implementation and evaluation of S2-SCTP.

There has already been work on end-to-end security solutions for SCTP. Mainly two research teams have recently been developing two individual end-to-end security solutions suited for SCTP. One of the research teams were located in Germany and the other in Sweden. The German teams contribution to end-to-end security solutions SCTP was primarily focused on adapting DTLS for SCTP. Their work has resulted in a proposed standard for DTLS over SCTP, described in RFC 6083 [17]. RFC 6083 solves some of the problems that comes with using DTLS over SCTP, but not all. For instance, a DTLS peer cannot perform the SCTP-AUTH key management.

The Swedish team designed and implemented a new end-to-end security solution for SCTP [7]. Their suggested end-to-end security solution for SCTP was called S2-SCTP and could be seen as the 1.0 version. Since this work laid the foundation for the end-to-end security solution described in this report, the end-to-end security solution described in this report could potentially be named S2-SCTP 2.0.

The remainder of the report is organized as follows. In Section 2 a brief overview of SCTP is provided. Section 3 contains information of already existing end-to-end security solutions for SCTP, such as TLS, DTLS and IPsec. The design of S2-SCTP is presented in Section 4. A description of the implementation of S2-SCTP is detailed in Section 5. Section 5 also includes peer authentication. Section 6 contains research on the performance evaluation of S2-SCTP compared to the existing end-to-end security solutions. Future work and concluding remarks are in Sections 7 and 8 respectively.
2 SCTP

SCTP is a message oriented transport protocol that runs over the IP layer, see Figure 1. SCTP differs from TCP and UDP in a number of ways. While TCP establishes connections, SCTP establishes associations. An association in SCTP is a protocol relationship between two endpoints and can be viewed as a set of uni-directional streams, where each stream has their own Stream Identifier (SID).

![TCP UDP SCTP](image)

Figure 1: SCTP, similar to TCP and UDP, runs over the IP layer.

Traditional TCP connections are bound to one IP address while an association in SCTP can consist of multiple IP addresses. Multiple interfaces can be part of an association and this is known as multi-homing. One advantage of multi-homing is that an end-host is not dependent on a single interface. This implies that if an interface were to fail the association could still live on if alternative interfaces is present in the association.

Another advantage of SCTP is that it uses a four-way handshake, instead of a three-way handshake, to initiate an association. A three-way handshake is vulnerable to denial of service (DoS) attacks, whereas a four-way handshake is not. The four way handshake is illustrated in Figure 2.

![SCTP 4-way handshake](image)

Figure 2: SCTP 4-way handshake.
SCTP also supports unordered delivery. In contrast to TCP, which requires an immediate retransmission when a packet is lost or delayed, SCTP stores the message in a queue until the message is reordered. A message consists not only of data, but also necessary control information. A large message may be fragmented into several packets. An SCTP packet may consist of multiple chunks that could be bundled together. A chunk is a fragment of information with header and data part. Each chunk header in SCTP consists of fields for chunk type, flags and length. The generic chunk header in SCTP is illustrated in Figure 3. A brief description of the individual chunks can be found in Table 1. For more in-depth descriptions, see [14].

Table 1: Brief descriptions of some of the different chunk types used in SCTP [14].

<table>
<thead>
<tr>
<th>Chunk Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>Contains the payload data for a packet. The data chunk is illustrated in Figure 4.</td>
</tr>
<tr>
<td>INIT</td>
<td>Is used for initiating an SCTP association.</td>
</tr>
<tr>
<td>INIT ACK</td>
<td>Acknowledges the initiation.</td>
</tr>
<tr>
<td>SACK</td>
<td>Acknowledges the reception of data chunks and lets the sender know if a data chunk has been lost.</td>
</tr>
<tr>
<td>HEARTBEAT</td>
<td>Confirms that the association is still active.</td>
</tr>
<tr>
<td>HEARTBEAT ACK</td>
<td>Acknowledges the confirmation of the active association.</td>
</tr>
<tr>
<td>ABORT</td>
<td>Ends the association abruptly.</td>
</tr>
<tr>
<td>SHUTDOWN</td>
<td>Is used to end the association more gracefully than the abort chunk.</td>
</tr>
<tr>
<td>SHUTDOWN ACK</td>
<td>Acknowledges the shutdown notification.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Notifies of any error within the association.</td>
</tr>
<tr>
<td>AUTH</td>
<td>Can be used for data integrity, but does not require confidentiality.</td>
</tr>
</tbody>
</table>

Figure 3: A generic chunk header.
In SCTP, user messages are contained in a data chunk whereas a control chunk consists of a protocol message, see Figure 4. Control chunks, together with data chunks, a common header and an IP header form an SCTP message, as seen in Figure 5. The common header consist of the source port, the destination port, a verification tag and a checksum as seen in Figure 6.

![Figure 4: SCTP data chunk.](image)

In SCTP, user messages are contained in a data chunk whereas a control chunk consists of a protocol message, see Figure 4. Control chunks, together with data chunks, a common header and an IP header form an SCTP message, as seen in Figure 5. The common header consist of the source port, the destination port, a verification tag and a checksum as seen in Figure 6.

![Figure 5: SCTP message with IP header.](image)

There are several standardized protocol extensions for SCTP. A list of the individual RFCs for each of the following standardized protocol extensions for SCTP is provided in Table 2.

![Figure 6: SCTP common header.](image)
Table 2: Standardized SCTP extensions.

<table>
<thead>
<tr>
<th>Title</th>
<th>Number</th>
<th>Date</th>
</tr>
</thead>
</table>

- The **Partial Reliability** extension gives, with the help of a new chunk type consisting of a cumulative ACK, SCTP the ability to deliver messages reliably by using sequence numbers for messages.

- The **Padding Chunk and Padding Parameter** extension is used in order to adapt the length of an SCTP message so that its length ends up being a multiple of 4 bytes.

- The **Authentication Chunk** extension is used to prove that an SCTP chunk was not sent by an attacker but rather by the original peer that started the association. This is done by placing the result of a Hashed Message Authentication Code (HMAC) computation before the data that is being covered by the computation.

- The **Dynamic Address Reconfiguration** extension handles the available interfaces of an end-host in order to give SCTP the ability to dynamically add and remove IP addresses from an association.

- The **Stream Reconfiguration** extension makes it possible to reset the Stream Sequence Number (SSN). Without this extension the SSN would only increase whenever an old stream is being reused with a different purpose.
3 Existing End-to-End Security Solutions for SCTP

Although SCTP might be a relatively new transmission protocol there already exists some standardized end-to-end security solutions. These were, however, not designed specifically for SCTP but rather for other transmission protocols such as TCP and UDP. As such some of the advanced techniques in SCTP are not kept intact when these standardized end-to-end security solutions are being used. A summary of standardized end-to-end security solutions for SCTP are listed in Table 3, and briefly described in the following subsections. In subsection 3.4, a non-standardized solution is presented.

Table 3: Standardized end-to-end security solutions for SCTP.

<table>
<thead>
<tr>
<th>Title</th>
<th>Number</th>
<th>Date</th>
</tr>
</thead>
</table>

3.1 TLS over SCTP

TLS [2] was mainly designed as an end-to-end security solution for TCP. TLS is designed for an ordered delivery of messages within a byte-stream oriented transport protocol. In order to maintain end-to-end security using TLS over SCTP [5] the unordered message delivery feature in SCTP must not be used, since TLS assumes all messages are delivered. In other words, the partial reliability extension for SCTP must not be used when using TLS as the end-to-end security solution for SCTP. TLS also requires the same number of streams in both directions and one TLS connection for every bidirectional stream.

3.2 DTLS over SCTP

DTLS [12] for SCTP [17] does not require ordered message delivery. In other words DTLS over SCTP supports the partial reliability extension for SCTP. DTLS over SCTP also supports a large number of unidirectional and bidirectional streams. Since DTLS supports unordered messages DTLS also provides message fragmentation as well as retransmission timers. These reliability measures only applies for the handshake and not for the application data. As such DTLS may drop packets containing application data.
3.3 SCTP over IPsec

IPsec [6] differs from DTLS and TLS when used with SCTP. The reason for this is that TLS and DTLS is used over SCTP meanwhile IPsec is used below SCTP. This means that SCTP should be treated just as any transport layer protocol on top of the IP layer. IPsec uses Security Association (SA) for integrity and confidentiality protection. Internet Key Exchange (IKE) can be used to handle this dynamically. The problem with SCTP over IPsec is that IPsec needs to create a new SA for each new IP address that is dynamically added, so the application needs to configure the SA and negotiate a new key exchange [4].

3.4 Other Solutions

The solutions described above are proposed end-to-end solutions for SCTP that work with SCTP to some extent. However, in order to take full advantage of the advanced techniques that SCTP offers a end-to-end solution designed for SCTP is needed.

Secure SCTP (S-SCTP) [3] is a proposed end-to-end security solution specifically designed for SCTP. First SCTP establishes an association then the secure session of S-SCTP is initialized. The basic concept behind S-SCTP is that an association may consist of multiple streams. Multiple streams have only one common secure session, instead of one secure session for each stream (as is the case with for instance TCP), since S-SCTP operates at the same level as SCTP. However, the performance of S-SCTP decreases with long messages. S-SCTP, in contrast to TLS, secures messages after they may have been fragmented. Thus, S-SCTP must secure each fragment separately.
4 S²-SCTP

The objective of this project was to make a secure socket communication library for SCTP together with the SCTP AUTH extension [19]. In contrast to the standardized solutions described in Section 3, S²-SCTP provides full security of user messages while still preserving all the advanced techniques of the SCTP protocol. S²-SCTP was initially proposed in [8]. The project described in this report is regarded to be version 2.0 of S²-SCTP, and contains some modifications regarding the transfer of cryptographic info and initialization vectors. The target platform for this project was FreeBSD. The software layer structure for S²-SCTP is shown in Figure 7.

Figure 7: The layer structure of S²-SCTP.

The basis of S²-SCTP is that two peers, referred to as client and server, should have a secure communication regarding message authenticity and confidentiality. Authenticity is provided through the SCTP AUTH-extension, and relevant chunks are protected with an HMAC (SHA-256). Confidentiality is provided using data encryption, and AES in Cipher Block Chaining (CBC) mode are the proposed algorithm.

The design for the S²-SCTP protocol specifies that a cryptography key number and an algorithm number, each 16 bits long, are to be sent with each SCTP message. It is also specified that the PPID-field of the message is used for this cryptographic key and algorithm info, “crinfo”. The placement of the crinfo-field has in version 2.0 been changed and is now incorporated in the data part of the data chunk, as the first 4 bytes of data. Furthermore, the Initialization Vector (IV) for the encryption is now sent explicitly with each message as the following bytes after the crinfo field, and before the encrypted data. More about the basic S²-SCTP protocol in Subsection 4.2.

The design is based upon an S²-SCTP message data protocol to govern how key and algorithm numbers are mediated, how initialization vectors are given, and how padding is done. Key numbers are temporary key identifiers for each association, but may survive in stored sessions and algorithm numbers are globally defined. The layout of the different parts of a single message data section is shown in Figure 8.

To handle peer authentication and other management tasks in S²-SCTP, a simple but fully working Length Header Value (LHV) format, and a handshake protocol were designed. This is
described in Section 4.1.2. These management tasks are mediated by SCTP messages sent at stream 0, and they are handled internally in the receiving function. The management protocol is described in detail in Section 4.1.2, while peer authentication with certificate and key exchange are described in Section 4.1.

Figure 8: A complete IP packet with an S^2-SCTP packet including two data chunks of different encryption types.

4.1 Peer Authentication

This section describes the authentication component in S^2-SCTP. The requirements from the authentication component is to authenticate the peers and also to provide keys that can be used for further communication. The authentication component used is based on TLS. A central part of an end-to-end security solution is to authenticate the counterpart. Even if this step is not needed to establish a key with a counterpart it is sometimes desirable to be able to know whom you are actually communicating with. In the solution developed in this project the peer’s identities are evaluated using x.509 certificates. TLS also offers key agreement through a variety of methods including Diffie-Hellman key agreement and Elliptic Curve. The Diffie-Hellman key agreement protocol is used in S^2-SCTP.

4.1.1 TLS Handshake

The TLS handshake is a four-way handshake that takes place when two peers that wants to set up a TLS connection with each other. A TLS handshake is shown in detail in Figure 9.
Figure 9: TLS handshake messages.

All information in these messages are, however, not used by the component developed for S²-SCTP. The TLS handshake is further described below.

1. The Client Hello is the first message in the handshake and is sent from the client to the server to initiate the handshake. It also contains a list of ciphers supported by the client.

2. When the Client Hello has reached the server. The Server Hello message is sent back to the client. It contains the server's certificate as well as the server's Diffie-Hellman parameters. The server decides on a cipher supported by the client and notifies the client which cipher has been chosen.

3. The client now sends its parameters for the Diffie-Hellman key agreement. Also an acknowledgment of the cipher chosen by the server is sent, letting the server know that any further communication will be protected.

4. The server sends a message, signaling that the handshake is completed. This message also contains a session ticket.

Since no key is available during the key agreement, the key agreement will take place in plain text. Diffie-Hellman can be performed in the open and an observing attacker will still not be able to eavesdrop on any further communication encrypted with the key established.

To prevent an attacker from performing a so called man-in-the-middle attack, or hijacking the remaining handshake after the server certificate is sent, the server signs its Diffie-Hellman parameter with its private key. This allows the client to verify that the parameter is not sent from an attacker. This ability is well desired since no pre sharing of keys is necessary.
4.1.2 Management Protocol

In order to provide peer authentication an overlay management protocol has been designed. A rather simple LHV format is used for this kind of control information. See Figure 10 for an illustration. This was made possible by both the fact that these messages are sent on a separate stream (stream 0) and that the LHV format is constrained to messages on this stream only. The fact that message length is known beforehand made it possible to have arbitrary binary data in the data part.

The peer authentication messages, as well as some other protocol management messages, will have to go unencrypted. These should be completely separated from all other traffic in each association, and are hence sent on stream 0, which in S²-SCTP is reserved for these kind of messages (a protocol design decision in S²-SCTP). Since the AUTH mechanism is applied to all DATA chunks they are integrity protected with an HMAC. The crinfo field is constantly 4 bytes and this length does not depend on the algorithm.

The header value in the LHV is an ASCII string that identifies each different type of management protocol message type. The message data that follows can have any format. The first byte in the LHV, which is also the first byte in each message, is an offset that indicates the start of the data area in the message as shown in Figure 10.

As each protocol message is carried in one control chunk, which length is known beforehand, the length of the data part could be calculated by its message length and the offset number:

\[ \text{Len(data)} = \text{Len(msg)} - \text{offset} \]

Example above is a Version message with the S²-SCTP-version number "1.3.14 build 2438".

Figure 10: LHV format design of the S²-SCTP management messages.

4.2 Data Protection

To obtain data protection, both data integrity and data confidentiality are of great importance. This means that S²-SCTP has to implement both. The data integrity is, however, already taken
care of by the SCTP AUTH extension [19]. This extension makes it possible to choose which chunk types that are to be protected by AUTH, and also to manage shared AUTH keys to give full protection against man-in-the-middle attacks. What is left for S²-SCTP to do is to make sure that this extension is activated for the right chunks, and manage the shared AUTH keys.

Data confidentiality is obtained mainly via AES encryption, although the old standard DES is supported for backward compatibility. AES is used in CBC mode and with configurable key lengths of 128, 192 and 256 bit key lengths. AES (and DES) in CBC mode require that shared keys are managed, and that an IV is supplied for each message. They further require that data is of a length matching a full number of AES blocks (16 bytes per block), or DES blocks (8 byte per block), so padding is needed for each message that does not fully fill the last block of a message. All this had to be taken into consideration when constructing the S²-SCTP basic protocol.

In [8], a solution on how to fulfill these requirements for AUTH and data encryption is provided. This solution has been adopted for this implementation, except for two details: The key and algorithm numbers are moved from the PPID field into the data area, and an IV is now sent explicitly with each SCTP message. With explicit IVs, the partial reliable extension of SCTP will be supported.

4.2.1 Message Authentication

The SCTP AUTH extension is well suited for providing data integrity by means of authentication in the S²-SCTP implementation. SCTP AUTH is a socket internal mechanism that provides authentication via an HMAC on a per SCTP packet basis. The chunk types that will be protected with the HMAC can be specified by the peers. The authentication in the SCTP AUTH extension is initially based upon a zero key (and the random numbers and parameters sent with the INIT/INIT-ACK packets), but before performing encrypted communication the peers are expected to add a shared AUTH key to the socket internal AUTH key ring.

4.2.2 Message Data Layout for Key, Algorithm, IV, Data and Padding

The key and algorithm numbers are placed in the first four bytes of each message, as shown in Figure 11. This is new compared to the design of S²-SCTP 1.0 as they where originally placed in the PPID field. Both key number and algorithm number are 16 bit unsigned integers in network byte order. The crinfo field exists for all messages, even unencrypted, except for protocol messages on stream 0. This is further explained in Section 4.1.2. Even unencrypted messages must include a key and an algorithm number. For unencrypted messages, key and algorithm numbers are set to zero.

Cryptographic algorithms typically need an IV in order not to be vulnerable for different types of cipher attacks. To support SCTP unordered messages as well as the partial reliable ex-
tensions, the IV is now given explicit for each message. This IV is of different length depending on the algorithm in use, but it is usually the same as the block length. In other words, the length of the IV is 64 bits for DES and 3DES, whereas AES has a 128 bit long IV. Thus, the current S²-SCTP implementation allows the length of the IV to be specified for each algorithm. The sending function creates an IV, if the algorithm requires one and copies it into the message buffer right after the crinfo field. At the receiving side, after that the crinfo is read and key and algorithm numbers are retrieved, a number of bytes will be read on the position after the crinfo and feed to the decrypting function as an IV. The length of the IV is constant with regards to the algorithm number and is thus available through the key in the key ring. If a message is sent unencrypted, or if the algorithm in use does not have an IV, this length will be zero. Figure 11 shows the message data field with all the S²-SCTP basic protocol fields for an AES encrypted message.

The cost of sending the IV along with each message is 16 bytes per message for AES CBC. The advantage of explicit IVs is the ability to handle out of order messages, which cannot be accepted if the IV is implicit. See Section 4.2.4 for further details.

![Figure 11: Structure of an encrypted message.](image)

### 4.2.3 Encrypted Data and Padding

The encrypted data is appended directly after the IV field, and if the algorithm in use is a block encryption algorithm, data will be forced to fill an integer number of blocks. The area that is left is filled with pseudo random data, except for the last byte, which is used to tell the number
of padding bytes needed. This implies that when using message padding there must always be at least one extra byte as the padding length also explicitly must be allowed to be zero. If the message goes unencrypted, or if the encryption algorithm is of stream type, block length is set to one byte and no padding is applied. The block length parameter is tied to the algorithm in the same manner as the IV length, see Figure 11.

4.2.4 Message Overhead

Although the overhead is different for different kinds of encryption algorithms, we can now calculate the average overhead for the standard AES CBC and DES CBS encryption, see Figure 11. Padding length is given by block + 1 byte extra for padding:

\[
OH(AES) = crinfo + IV + \frac{block + 1}{2}
\]

\[
= 4 + 16 + \frac{16 + 1}{2}
\]

\[
= 28.5 \text{ bytes per message on average}
\]

The crinfo length is 4 bytes. The IV and the block lengths are variables that are given by the algorithm number. For unencrypted messages the overhead per message is only 4 bytes, since no IV or padding are needed. The overhead of DES and 3DES algorithms can be calculated as follows:

\[
OH(DES) = crinfo + IV + \frac{block + 1}{2}
\]

\[
= 4 + 8 + \frac{8 + 1}{2}
\]

\[
= 16.5 \text{ bytes per message on average}
\]

We can also calculate the minimum message length for empty or very small message lengths. Let the message length be less than the block length for some encryption method:

AES (message length < 16 bytes):

\[
min - Len(AES) = crinfo + IV + block
\]

\[
= 4 + 16 + 16
\]

\[
= 36 \text{ bytes}
\]

DES and 3DES (message length < 8 bytes):

\[
min - Len(DES) = crinfo + IV + block
\]

\[
= 4 + 8 + 8
\]

\[
= 20 \text{ bytes}
\]
Unencrypted (message length == 0 bytes):

\[ \text{min} - \text{Len}(\text{none}) = \text{crinfo} + 0 + 0 \]
\[ = 4 \text{ bytes} \]

\( S^2\)-SCTP encrypts messages on a strict per message basis. The per message overhead for bundled messages in the same SCTP packet is the same as the message overhead for packet with a single message. The advantage is that only one SCTP header and only one AUTH chunk is needed. An AUTH chunk is 40 bytes long, and this should be taken into account when calculating the efficiency.

Figure 12: The absolute efficiency of bundled normal SCTP and \( S^2\)-SCTP data chunks.

Calculating efficiency in situations with bundled messages tends to be very complex as fewer \( S^2\)-SCTP messages can be bundled together compared to non-encrypted SCTP messages, given a constant Maximum Transmission Unit (MTU). We want to compare the total efficiency degradation when using \( S^2\)-SCTP compared to base SCTP.

Absolute efficiency was first calculated for different message sizes between 4 and 1372 bytes, and assuming the SCTP connection has an MTU of 1500. To get the available space for the data...
Figure 13: The quote of efficiency of bundled normal SCTP and $S^2$-SCTP messages compared to bundled unencrypted messages.

chunks we subtract the IP header of 20 bytes, the SCTP header of 12 bytes, and assume that there is a SACK chunk of 16 bytes and an AUTH chunk of 40 bytes, which also are subtracted. A simulation of sending 1000 message of each size was done. We then get the absolute efficiency of $S^2$-SCTP messages compared to total data sent. See Figure 12 for the two simulations. In this figure it can be seen that for $S^2$-SCTP messages with sizes up to 652 bytes might be bundled with two or more messages per SCTP packet, while messages larger than 652 bytes require an SCTP packet of their own. At this transition size we see a big jump in efficiency, while smaller jumps can be seen at 428 byte messages, and 300 byte messages, where the transitions from two to three, and three to four bundled messages per packet occur. The smaller sawtooth pattern of the curve is due to the 16 byte block size of AES.

We made another simulation to get the same absolute efficiency of normal SCTP, this time without the overhead of encryption, and without AUTH-chunks. Now we can calculate the relative efficiency of $S^2$-SCTP as follows:

$$\text{relative efficiency} = \frac{\text{abs.eff}(S^2 \text{-SCTP})}{\text{abs.eff}(SCTP)}$$
This calculated relative efficiency is shown in Figure 13. Here we see how $S^2$-SCTP compares to normal SCTP and that when using bundled messages the efficiency of $S^2$-SCTP is over 80% for all message sizes over 148 bytes, and over 90% for most message sizes of 360 bytes and above. The dips at 432–468 bytes and 656–708 bytes is due to that the available space in the $S^2$-SCTP packet differs to that of normal SCTP, and the overhead of encrypted packets also differs, which gives different transition sizes when one more message can be bundled in an SCTP packet.
5 S²-SCTP Implementation

As part of the project, an implementation of S²-SCTP was developed. The target platform for the implementation was FreeBSD, and the library was implemented in the C programming language as a layer on top of the Berkeley Sockets API with the use of the OpenSSL TLS API. Apart from the existing SCTP AUTH extension used for data integrity, S²-SCTP runs entirely in user space without modifications to the sockets API. The layered structure is shown in Figure 7. The S²-SCTP library is meant to make it fairly easy for application developers to make use of a secure SCTP connection. This implies that peer authentication and data protection processes must be well encapsulated. The receiving functions in Berkeley Sockets returns fragments of messages as SCTP event notifications. Due to this and the complexity of the one-to-many SCTP model, we found no other solution than to implement a different behavior and interface of our own receiving functions, compared to that of Berkeley Sockets. We found that keeping the handling of notifications, and S²-SCTP management protocol as internal functionality, in the receiving function would solve this challenging situation. We have limited the project to support only the one-to-one socket model of SCTP, and without support for message fragmentation at the receiving side. This because of a more complex task of dealing with multiple associations and message fragments in the inner loop of the receiving function.

5.1 The Object Based Scheme

The implementation is made as a simple object based model in C, without inheritance and polymorphism. We have used a rather simple object based scheme where methods are regular functions, all with a reference to the corresponding object as the first parameter. Classes are type definitions of structures and the constructor allocates heap space, do the initializations needed and return a reference to the object.

5.2 Peer Authentication Implementation

S²-SCTP is dependent on either key sharing or key agreement to be able to obtain data confidentiality by encrypting messages. The authentication component is implemented using the OpenSSL library. The basic idea is to set up a TLS connection, since this will provide both authentication and key agreement. Once the TLS handshake has been completed, the TLS connection is closed.

In the TLS handshake, the Diffie-Hellman key agreement is used by the peers to mutually agree on a key. This key is called the master key. At least two keys has to be created, one or more keys for authentication of data and one or more keys used for encrypting the communication itself. These keys are derived from the master key.

In S²-SCTP, the TLS handshake messages are supposed to be transmitted over stream 0. When creating a TLS connection, using the standard API, a socket is provided for OpenSSL.
The socket is then used by OpenSSL for the TLS communication. The problem with this method is that when OpenSSL sends messages through the provided socket, there is no way to affect on which stream the communication is to be transmitted over, as illustrated in Figure 14. OpenSSL is designed to be used mainly over TCP connections where this is not a problem. The solution to this problem in S²-SCTP is to obtain the handshake messages from OpenSSL and manually transport them over stream 0.

![Figure 14: The SSL structure of OpenSSL does not allow selection of streams.](image1.png)

Instead of providing OpenSSL with a socket to read and write from, it is possible to provide a read and a write buffer. OpenSSL will write all messages to the write buffer, and read all messages placed in the read buffer. Using these buffers, the messages can be forwarded to the other peer using any arbitrary transport protocol. This property makes this solution ideal for S²-SCTP since it provides full control when sending the handshake messages. With that control it is possible to direct the messages to stream 0, as illustrated in Figure 15.

![Figure 15: Using buffers allows selection of streams.](image2.png)

Since full control of the transportation of the messages is desired, i.e., sending the messages using S²-SCTP and on specific streams, it was necessary to perform the handshake manually, managing the handshake step by step instead of using the usual `SSL_accept` and `SSL_connect`. If these functions were to be used there would be no possible way to decide on which stream the handshake messages would be transported on. This resulted in a state machine that is described in Figure 16.
Figure 16: State machine for authentication solution.

The numbered state transitions shown in Figure 16 is explained below.

1. When the client wants to initiate the TLS handshake, the handshake initialization function is called and the client’s current state changes to wait for Server Hello.

2. The Server Hello message is received by the client who then moves to the new state and waits for Server keyex.

3. The server stands in the idle state and is waiting for a Client Hello message. When the message is received the server changes state to wait for Client keyex.

4. After the handshake is completed both ends go to the finished state.

A comparison between OpenSSL over TCP and OpenSSL with $S^2$-SCTP reveals that all information in a normal OpenSSL message transmitted over TCP is also represented in the corresponding $S^2$-SCTP message, as illustrated in Figure 17.
The packets were captured using the Wireshark [20] packet analyzer. The packet captured from TCP could be decoded as TLS messages by Wireshark. This was unfortunately not the case for SCTP packets. As a result, the SCTP packet could not be decoded in the same way as in TCP and only displays the package content as “Data”. The OpenSSL data expected to be equal in both Figure 17(a) and Figure 17(b) are marked in black. The area separating the black areas, in both figures, is an OpenSSL parameter marked “Random” that should yield different results each session.

5.3 Central Functions – Send and Receive

The most central functions in the implementation are the sending and receiving S²-SCTP functions, especially the latter as it coordinates much of the work of managing the S²-SCTP protocol. The following subsections briefly describe the receiving and sending functions.

5.3.1 The Receive Function

The receive function called s2sctp_recv2msgobj is the central part of the S²-SCTP library. This is mainly due to the fact that this function has to receive messages, notifications and signals. The receive function has to encapsulate the whole process of receiving the messages and decrypt it before delivering it to the upper part of the application. It also has to handle notifications, and
S²-SCTP protocol messages on stream 0.

To manage the encryption/decryption process in an environment where notifications can interfere, and where fragments of messages from different streams and different associations can intermingle, the high encapsulation level of \texttt{s2sctp\_recv2msgObj} seemed necessary. Adoption and tweaking of the properties of the receive function can be done via callback methods.

The structure of the receiving function consists of two nested loops.

- The inner loop handles parts of messages, and the reception of SCTP event notifications, and that loop is exited only when receiving a complete SCTP message.

- The outer loop handles complete messages. It redirects stream 0 protocol messages to the protocol dispatcher and determines if the inner loop should be reentered. The outer loop also extracts crinfo and IV, and calls the decryption function dictated by crinfo. The outer loop is exited upon a decrypted message and returns a message object.

5.3.2 The Sending Function

The \texttt{s2sctp\_sendFromMsgobj} is the send function to send a message in S²-SCTP. It takes the assocObj as arguments for the association, and the msgObj to send. In addition to this, it also takes as parameters which SCTP stream to send on (the stream must have a stream number greater than 0), the key number for the encryption key (0 for unencrypted), and the optflags to use when sending the SCTP messages.

The sending function sends one message at a time. As long as the sockets sending buffer is not full, the sending function will not block. The fact that the sending function does not block implies that even given that the sending succeeded, the application does not have any confirmation of whether the message has reached the other application, or even reached the other host at all. This will not be indicated before the buffer has been full, or an error state has occurred.

5.3.3 The S²-SCTP Management Protocol Sending Function

It is not allowed for the \texttt{s2sctp\_sendFromMsgobj} to send messages over stream 0, since stream 0 is reserved for protocol messages. Thus, another sending function for stream 0 protocol messages is needed. To alleviate for the programmer we have created the \texttt{s2sctp\_sendS2sctpProtocol} function, and equipped it with all necessary code to create the LHV format.
6 Performance Evaluation

In order to evaluate the performance of S²-SCTP, it was compared with other existing security solutions for SCTP (TLS over SCTP and DTLS over SCTP). The performance evaluation measured the mean throughput for a number of different message sizes. This section explains how the machines were set up during the experiments, how the performance evaluation was carried out, and what the results were.

6.1 Experimental Setup

As seen in Figure 18, the setup contained three similar machines (Dell Optiplex GX270). These machines all ran the same operating system (FreeBSD 10.0). The reason for using FreeBSD was that the SCTP implementation existed for the system. Two of the machines acted as a client and as a server. The third machine was setup to act as the “Internet”. The “Internet” was emulated using Dummynet. Pipes were set up to control the flow and the pipes were taken in effect when the client sent data to the server and vice versa. In this experiment the pipes were used only to limit bandwidth and add packet loss rate. During this experiment the bandwidth was set to 100 Mbps. The reason for using the 192.168.63.X network was to have a “management network”. In other words the reason for using the 192.168.63.X network was so that the entire experiment could be monitored and run from the one single computer, in this case the client computer.

Figure 18: The setup for the performance experiment, with consisting of a client machine, a server machine and a machine that emulates the “Internet”.

![Diagram of the setup](image-url)
6.2 Test Cases

The protocols being evaluated were:

- Base SCTP
- Secure Socket SCTP
- TLS over SCTP
- DTLS over SCTP

While the code for $S^2$-SCTP was written as part of this project the other security solutions used in the performance evaluation were not. These were instead open source code only slightly modified in order to perform the same tasks described in the next paragraph. The TLS code was written by Rescorla in 2002 [11] and as such needed some modification in terms of updated certificates in order to run properly. It also required modification as to the type of connection it used, since it originally ran on a TCP connection. The SCTP code used was based on the code from [9]. Their echo_client and echo_server was also used for performance evaluation of SCTP without any end-to-end security, in order to measure the maximum limit for SCTP. The modified TLS code was also used to test an authentication only transmission, where the SSL security was removed when messages were being sent. The DTLS code was implemented by Seggelmann and Tuexen [13].

All test cases consisted of a client and a server. The server started to send unlimited amounts of messages as soon as the client connected to the server. Each message was of the size specified prior to establishing a connection. The client kept track of time in terms of how long it had been active. When the timer reached the specified limit the client disconnected from the server. The client then reported the test results through standard output. The information given was the exact time elapsed (since the client might have read messages at the point when the time limit was reached it would not always disconnect exactly at the time limit given), the amount of messages received, the message size and the calculated speed. This information was then used in order to plot the graphs seen in Figures 19, 20 and 21.

6.3 Test Script

Several scripts were written in order to complete the tests. In this project, the script language Perl was used. All the scripts ran on the client computer and the server was initiated via an SSH connection. Scripts were necessary to be able to run through all the tests with increasing user message sizes and making sure that the server was not occupied.
6.4 Results

The results were be measured in terms of mean throughput (Mbps) for different user message sizes that varies from 50 bytes to 4000 bytes. The message size was increased by 50 bytes at a time. The throughput was measured for a 10 second transmission for each message size and each measurement was repeated 30 times for statistical significance. The encryption algorithm was AES-256 with SHA-1 for authentication. Two different tests were conducted:

- A loss free test
- A test with 1% package losses

6.4.1 Loss Free Test

In Figure 19, the results of the three security solutions, and SCTP without security is plotted. The reason for this was to give a rough estimate of how S²-SCTP performs against the other security solutions, i.e., DTLS over SCTP and TLS over SCTP.

![Figure 19: A comparison between the three security solutions and regular SCTP as a capacity reference.](image)

The behavior of the TLS over SCTP solution is hard to explain. Presumably the TLS over SCTP implementation was not optimized and was written some time ago. However, other trends in the graph that should be noted is at 1300–1400 bytes, where we no longer fit the data into one packet and need to fragment, there is quite a drop. The same thing happens at 2600–2700 bytes. There is also a relatively high increase at 500 bytes, that could possibly be because the AES-256 algorithm change mode after a certain user message size. The reason for the sawtooth behavior that can be seen in between these drops mentioned at 1300 and 2600 bytes will be explained in Figure 20.
To show how stable the security solutions were, Figure 20 shows not only the mean throughput, but also all individual user message results, 30 for each message size. This is to show that the mean values are trustworthy. As Figure 20 shows, the different security solutions stayed relatively stable throughout despite a few abnormal values.

![Graphs showing throughput for different security solutions.](a) DTLS over SCTP  
(b) S²-SCTP  
(c) TLS over SCTP

Figure 20: The graphs shows the individual value of all 30 plots for each user message size to show the result with a 95% confidence interval.

The sawtooth behavior mentioned above could for instance come as a result of the fact that, at certain points in time, other processes in the computer needed more CPU resources, leaving the test machines with less computational resources. This is displayed by the single crosses that at certain points differs a lot from the other results. This could be resolved by shutting off everything on the computer that is not needed.

### 6.4.2 1% Package Loss Test

Figure 21 follows the same trend as Figure 19, but with obvious throughput drops. However, of all the different security solution S²-SCTP still produce the highest mean throughput. TLS
over SCTP was not added in this graph because of the irrational behavior, seen in Figure 19 and Figure 20 (c). In Figure 21 the overhead can be estimated by looking at the point where the drop occurs (around 1500 bytes and again at 3000 bytes), as it occurs at a greater message size for base SCTP compared to $S^2$-SCTP and DTLS over SCTP. The difference in bytes is mostly made up by the extra overhead added for the end-to-end security solutions.

![Graph comparing SCTP, S^2-SCTP, and DTLS over SCTP with 1% package loss.](image)

Figure 21: A comparison between base SCTP, $S^2$-SCTP and DTLS over SCTP with 1% package loss.
7 Future Work

The future development of the S²-SCTP-library aims at making it versatile as well as mature enough for critical applications. Below a set of areas for future work is presented.

7.1 Multi-homing

Full support for the multi-homing feature of SCTP should be provided. This will involve some minor changes to the base object, s2sctpObj, and the association object, but should pose no bigger rework of the implementation.

7.2 Fragmented Messages

The next challenge is to provide support for the fragmented message mode, where fragments of messages from different streams can be received intermingled with each other and with notifications at the output from the native recvmsg() function. This would imply that a collection of non-ready messages has to be maintained. This collection has its natural home in the assocObj, and for each fragment received, either a new message object has to be created and inserted in this collection, or a current object in the collection be updated with the new data. Eventually a fragment reception will result in one of the messages to be completed and handled by the decryption mechanism, or the protocol dispatch.

7.3 One-to-many Socket Model

This is a somewhat larger challenge in that that we now have to maintain and search a collection of associations for each message (fragment) that arrives. Some sort of container has to be implemented, and maybe the ListArray library could be efficient if the number of concurrent associations stays well under 1000. This collection should be sorted on the assocId property, as that will be the property we will get when a new message is received. This one-to-many mode will carry with itself some major changes to the inner loop of the receive function and its parameter list, as we now are interested in the association collection that will be housed in the base object, s2sctpObj. It is our opinion that fragmented messages should be fully supported first, before trying on the one-to-many level.

7.4 Stored Sessions

Since early on in the project we have been perceiving a mechanism to store and retrieve peer authentication sessions to avoid the rather long and CPU heavy processing of doing a full peer authentication handshake. The object model is well prepared for this, and so is the S²-SCTP
management protocol, but it has to be carefully implemented to avoid vulnerabilities in the library. In the long run it will perhaps be better to implement a new hash-map container than to use the existing ListArray or AVL tree containers.

7.5 Thread Safety and Other Improvements

A future goal is to make S²-SCTP thread-safe. Examples of this are the message pools, the keyRings, the assocPool, and probably also the assocObjs. The importance of a thread-safe implementation is emphasized in bigger servers with lots of casual clients connecting, which implies that the protocol internal key exchange needs to be run on different cores and thereby in different threads. When stream 0 messages arrive, they are handled by the s2_protocol_dispatcher function, and the action functions called by this dispatcher. Those action functions are not meant to involve network traffic or any other function with heavy work or unforeseeable delays, like file access. It seems that perhaps a simple inter-thread message queuing mechanism also may be needed for the S²-SCTP key exchange protocol messages.

7.6 Renewed Key Exchange, Reduced IV Overhead and Counter Mode AES

Encryption should be externalized by the means of a dispatcher and callback methods. This is to be able to use other encryption libraries than OpenSSL. The same method that is used to create the first set of working keys should be able to be used at regular intervals when called by one of the peers, and thereby create a new set of keys. The IV sent explicitly with each message is now of full length, that is 128 bits for each AES-encrypted message. By shrinking the length, two advantages can be achieved. A smaller data volume has to be sent with each message reducing the data overhead. Also the IV applied is not known to an adversary trying to crack the cipher by known plain text attacks. To even further strengthen the encryption the Counter Mode AES could be implemented.
8 Concluding Remarks

The objectives of the S²-SCTP project was reached. We now have implemented all parts necessary for encrypted communication using pre-shared keys and the SCTP AUTH extension, all this in the one-to-one SCTP socket model. Currently the AES and DES encryption algorithms of OpenSSL are supported, and the AES encryption algorithm is recommended rather than the older DES encryption algorithm.

The use of TLS was well suited for the requirement of providing S²-SCTP with key agreement and peer authentication. The common way of using OpenSSL is with its listen and connect functions, for the server and client respectively. They are sufficiently documented by OpenSSL and easy to grasp. However, when OpenSSL is used as it has been used in S²-SCTP there is not much documentation to be found. This, and the fact that much of the TLS protocol mechanisms happens behind the API, turned the implementation into trial and error programming. This was not the case for the context setup and other initializations. The context setup and other initializations were more easily implemented despite insufficient documentation due to its more self-explanatory API.

As a reaction to the results presented on the performance, S²-SCTP seems to be a viable solution in terms of throughput. Since only throughput was tested, and seeing how strange the TLS over SCTP solution performed, the result seems a bit unfair. Other aspects that could have been evaluated in terms of performance includes jitter, latency, and the amount of CPU operations required. However, these tests are in no way an absolute statement that S²-SCTP is the best solution, but what is interesting to see is that it has potential. Also, what is missing should not affect the performance greatly. What was interesting in the resulting graphs was to see that S²-SCTP and DTLS over SCTP behaved in similar ways and the overhead for every package was roughly the same.

The basic encryption protocol of S²-SCTP was changed to not make use of the PPID field, and to support unordered SCTP messages, by placing the crinfo field and the IV in the data part. A management protocol was designed to handle peer authentication, key exchange, and stored sessions. These protocol changes and additions have worked out quite well, and the extra overhead has not degraded the performance of S²-SCTP that much. The normal functionality or interface of Berkeley Sockets receive functions could not be preserved, due to the complex situations when receiving intermingled messages parts, and notifications, where messages cannot be decrypted until received in their entirety.

The SCTP associations, the key rings and their keys, and the messages sent and received are all objects. This simple object model performs well, and not much changes has been done to this model during the implementation. Adaptability is provided by providing callback routines to execute in different situations. All this resulted in a solution with a relatively shallow learning curve, and a coding efficiency where only necessary actions need to be taken when using the library.
References


A Abbreviations

All abbreviations used in the report are listed below.

AES  Advanced Encryption Standard
ALDR  Application Layer Delivery Reliability
algo  Algorithm (cryptographic)
API  Application Programming Interface
assocObj  Association Object (in our object model)
CA  Certification Authority
CBC  Cipher Block Chaining
CPU  Central Processing Unit
crinfo  Cryptographic key and algorithm info
DDoS  Distributed Denial of Service
DES  Data Encryption Standard
DoS  Denial of Service
DTLS  Datagram Transport Layer Security
GCC  GNU Compiler Collection
HMAC  Hashed Message Authentication Code
IKE  Internet Key Exchange
IP  Internet Protocol
IPsec  Internet Protocol Security
IV  Initialization Vector
keyEx  Key Exchange
LAMP  Linux Apache MySQL PHP (webserver concept)
LHV  Length Header Value
msg  Message
msgObj  Message Object (in our object model)
OH  OverHead
OO  Object Orientation
MTU  Maximum Transfer Unit
PPID  Payload Protocol IDentifier
RTT  Round Trip Time
S²-SCTP  Secure Socket SCTP
SA  Security Association
SCTP  Stream Control Transmission Protocol
SHA-1  (160 bits) Secure Hash Algorithm
SID  Stream Identifier
SSL  Secure Socket Layer
SSN  Stream Sequence Number
TCP  Transmission Control Protocol
TLS  Transport Layer Security
UML  Unified Modeling Language
UDP  User Datagram Protocol
B Implementation Details

This appendix will treat some common aspects of the implementation, i.e., the limitations, status, structure, compiling, etc. For the object based scheme and strategy, see Appendix B.9, and for the object model strategy, see Appendix B.7

B.1 Project Limitations

We have limited this project to support only the one-to-one model of SCTP (aka the TCP model), and without support for message fragmentation at the receiving side. This because of the large extra overhead of having to deal with multiple message fragments that must be stored until messages are complete and decryption can be done. We would also have had to implement an association pool where the right association and the right keyRing had to be found, before each message fragment could be stored, and before each complete message could be decrypted. An extra pool of message objects would also be necessary. We hope that we will have the possibility to continue this project to make the implementation support all major modes of the SCTP protocol.

B.2 Language and Compilers

The language standard has been set to C99, and currently without the use of “GNU-extensions”. The latter might change in the future if GNU-extensions seem very appropriate for the task. The following compilers have been used:

- FreeBSD: clang version 3.3 (tags/RELEASE_33-final 183502) 20130610
- Linux: gcc (Ubuntu 4.8.2-19ubuntu1) 4.8.2

B.3 Compiler Directives

Compiling switches has always been -Wall and in some test scripts also -std:c99, the latter to ensure adherence to the C99-standard\(^1\). The code has frequently been compiled both with CLANG in FreeBSD and with GCC in Linux (both 32 and 64 bit Linux), and will hopefully compile without warnings, as we always try to eliminate all warnings at an early stage. No efforts have yet been made to compile to other than X86-compatible platforms, e.g., the mostly big-endian ARM, POWER, or MIPS architectures. No efforts whatsoever have been made to support Microsoft(R) Windows(TM), nor have we tried to compile and run in the “Cygwin” environment.

\(^{1}\)When compiling in Linux this implies that also the switch -D_GNU_SOURCE (setting the macro _GNU_SOURCE) is used in order to not to inactivate the needed headers when using -std:c99.
B.4 Optimizations

The implementation has not been compiled with any optimization switches, and not re-factored to minimize memcpy’s or bzero resets of different memory areas. The latter has been done frequently to maximize security. A message pool and a storage of the encryption/decryption key development in the key object is, however, done, and that definitely showed up as greatly enhanced performance at smaller messages, i.e., messages under 250 bytes.

B.5 Source File Structure

The structure of the C files and headers are such that most “classes” have their own C files, and all C file has their own header where structs are declared. All headers include specifically one common header—the s2sctp oo.h where common definitions and all typedefs of objects, i.e., structs, are made. There is also another common header file—s2sctp_configs.h—where user modifiable constants lives. The typedefing of the structs makes it possible for objects to have references to each other without creating very dependent and fragile order of includes, hamper development and cause unnecessary aggravation.

After #defines and typedefs, all other header files are included, so that all function prototypes are fully declared and could be used wherever and whenever needed. This also implies that the structs them self are visible throughout the library and to users, but we strongly believe in “encapsulation by responsibility” rather than by obscurity.

B.6 Current Status

The implementation performs well, seems to be stable when run in laboratory environments, and shows no distinguishable instabilities at the few real life Internet testing that have been done. The software does not show any signs of memory leaks. The source code is still decently malleable, but it could benefit from some re-factorization, and needs a lot of cleaning from unused parts, comments, and different debugging mechanisms. Because of this we will not publish the software with this report, but instead send it personally—for example by e-mail—on direct request only.

B.7 Strategy

The S²-SCTP library is meant to make it fairly easy for application developers to use secure SCTP connections. This implies that the whole encryption and peer authentication processes must be well encapsulated. However, the Berkeley Socket implementation of the one-to-many socket model implies that messages from many different clients will intermingle at the receiving
side, and that messages may even be delivered in fragments\(^2\). To handle these situations and still maintaining a fairly nice and easily used API, the receiving function has to encapsulate the whole decryption process, i.e., from the receiving of, or parts of, raw encrypted message fragments, and to the delivery of ready decrypted messages to the upper layer of the application. Further, to be able to support the one-to-many socket model implementation, the decryption mechanism must have access to the key ring for the association to whom the message belongs, i.e., the keyRings of all active associations.

We could not find appropriate support in the SCTP socket implementation to carry the needed extra information, i.e., a keyRing, or some key/value container of assocIDs, and their corresponding keyrings. Thus we could not keep the original interface of the sockets function \texttt{sendmsg()} and \texttt{recvmsg()}, and we had to construct a new API interface. The size and complexity of the task, and the need for high reliability of the library, called for an object oriented approach. We found that this would benefit all parts, i.e., both developers of the library, users of the library, and further development and maintenance of the library.

### B.8 Object Model

To summarize the object model, it could be mentioned that it centers around the association objects, the message-objects, and the keyRing objects. An assocObj naturally corresponds to one SCTP association, and has a lifetime that is approximately the same, although it may survive as long as it has active message objects in circulation not yet returned to it’s messagePool. The association object knows the keyRing for that user, and it owns all message objects that are created.

The message objects each correspond to one SCTP message. On the sending side, the application programmer creates them, when a message needs to be sent, and the sending function “consumes” them as soon as the message content is encrypted and sent, i.e., they are handed back to the messagePool. On the receiving side the receive function creates message objects, i.e., fetches from the messages pool as soon as a new message is received and is decrypted. Messages are then handed over to the application, either referenced in the return value from the receive function call, or as a reference parameter in a callback function. The application programmer then has the responsibility to return used message object to the message pool.

\(^2\)Messages may be delivered as fragments both due to the transfer of the messages—some can be regulated via SCTP socket options—and due to that the message was too large for the current user supplied buffer; the latter happens fairly often.
The keyRing is meant to be tightly associated with each user object. This is on the application level and it is the application programmer's responsibility. The keyRing object may survive the assocObj, as a so-called “stored session”. It may be re-connected to a new assocObj, if it has not been deleted before the user reconnects to the same server. If no stored keyRing are found when a new user connects, a new keyRing is created, with only the zero-key present.

We found that a TLS peer authentication introduce latency and require computer resources. Therefore, we paved the way to save the keyrings for use in a later reconnection from the client. This will require some mechanism to identify a previous user, and will be implemented via a shared cookie, and a hash based collection of keyrings mapped on these cookies. This will hopefully result in that only a simple two-way handshake will be needed, and without the network and
CPU overhead of a new TLS certificate and keyExchange session. This stored sessions mechanism is one of the items for future work.

Although authentication, key exchange, stored sessions, and some other necessary tasks, e.g., version control takes place over a separate stream (stream 0), they will require another protocol to distinguish different kind of tasks. For this reason we implemented a rather simple LHV format, which was described further in Section 4.1.2.

B.9 Object-based Scheme and Object Model Strategy

As we have not needed any inheritance or polymorphism so far in the project, the object-based scheme we have used to this point is a very simple one without any interface classes, V-tables, or even any method members (function pointers) in the classes. Classes are structs, or rather typedef's of the structs to aid aggregation, and objects are heap areas allocated to fit the structs in question. Object references are pointers declared to be of the types of those structs. The methods are normal functions, which all take an object reference to an allocated heap area, with the type of some class-struct as its first parameter. We say that the function is a method of certain class, if the object whose reference is the first of its parameter list belongs to this class.

The UML-diagram in Figure 22 depicts the object model, and as seen it very much centers around the assocObj and msgObj. The keyRing_t object naturally plays an important role as encryption and HMAC authentications is used. The container class “ListArray” which we for example use for keyRings and message pools, is also a class, but it is generic in the sense that the objects it stores are not typed, i.e., they are declared as generic pointers. This adds greatly to simplicity, generality, and ease of use. It gives for example the possibility to have lists of lists, but it also places a strong responsibility on the programmer, who has to be very watchful about what types of objects he/she stores in a certain list. Either all objects in a list must be of the same type, or the programmer himself has to implement polymorphism to handle the set of classes stored in the list. The generic properties of the ListArray also stems from the fact that the user must make his own comparison functions, and optionally destruction function, and send them with the call to the constructor of the ListArray.