**NTNU – Trondheim**
Norwegian University of
Science and Technology

# Authenticated uplink for the small, low-orbit student satellite NUTS

**Bram Bezem & Per Kristian J. Fjellby**

Submission date: December 2012
Responsible professor: Stig F. Mjølsnes, ITEM
Supervisor: Roger Birkeland, Project Manager NTNU Test Satellite

Norwegian University of Science and Technology
Department of Telematics

## Abstract

CubeSat projects are low-budget, small-scale builds that brings space research to the masses. We want to focus on the communication security aspect of such pico-satellites, especially message authentication and replay attacks. Without any means to meet such attacks, malfeasors with the right equipment can record and replay messages to the satellite. If the command set is known, either through reverse engineering or other means, malfeasors can even generate and send legitimate commands at will. The NTNU Test Satellite (NUTS) project has chosen the CubeSat Space Protocol (CSP) as main communication protocol. This protocol lacks a replay attack prevention mechanism. We propose a scheme to improve CSP with such a mechanism. A sequence number together with a hash-based message authentication code (HMAC) is used to provide resistance against message replay attacks. A resynchronization mechanism is also implemented to account for any drifts in the sequence numbers between the satellite and ground station. We have made test-applications based on the publicly available CSP source. Using these applications, the design has been tested on a realistic, full-duplex radio link between two PCs. The tests showed that the functionality added to CSP worked as designed both when no faults were introduced and when either of the nodes were reset to force a resynchronization.

## Sammendrag

CubeSat er lavbudsjetts, småskala utviklingsprosjekt som bringer romforskning til massene. Vi ønsker å bringe fokus på kommunikasjonssikkerhet i forbindelse med slike pico-satellitter, spesielt meldings-autentisering og replay angrep. Uten noe til å motstå slike angrep kan angripere med det rette utstyret ta opp og spille av meldinger til satellitten. Hvis kommandosettet er kjent, enten via reverse engineering eller på andre måter, kan angripere generere og sende gyldige kommandoer til satellitten når de selv måtte ønske. NTNU Test Satellite (NUTS) har valgt CubeSat Space Protocol (CSP) som kommunikasjonsprotokoll. Denne mangler en mekanisme for å beskytte mot replay angrep. Vi spesifiserer her en forbedring av CSP med en slik mekanisme. Sekvensnummer sammen med en hash-based message authentication code (HMAC) blir brukt for å motstå replay angrep. En resynkroniseringsmekanisme er også implementert for å justere eventuell drift mellom sekvensnummerene på henholdsvis satellitten og bakkestasjonen. Vi har laget testapplikasjoner basert på den offentlig tilgjengelige CSP kildekoden. Gjennom bruk av disse applikasjonene har designet blitt testet over en realistisk, to-veis radiolink mellom to PCer. Testene viste at funksjonaliteten vi har lagt til CSP virket som tiltenkt, både når applikasjonene kjørte uten uten intervensjon og når en av av nodene ble startet på nytt for å fremprovosere en resynkronisering av sekvensnummerene.

# Preface

This report is written as a part of our autumn project at Norwegian University of Science and Technology (NTNU) 2012. The project has been a joint effort where we have been cooperating on all aspects of the project. We both have a similar background coming from the 5-year Master of Science programme in Communication Technology and both specializing in information security.

Both of us joined the NTNU Test Satellite (NUTS) project group via the compulsory 4th year subject Experts in Teamwork (EiT). We developed an interest for radio and radiocommunications and we ended up earning our amateur radio licences around Easter 2012. We both wanted to combine our academic background with our new found interest in radiocommunications and saw the NUTS uplink project as the perfect opportunity. The project has at times been very challenging, especially when you have to make decisions that impact the whole project.

As a result of Roger Birkeland's continuous exhortations toward the NUTS group to attend conferences we will write an abstract for the 2nd IAA Conference on University Satellites Missions and Cubesat Winter Workshop in Rome. The conference is to be held in the beginning of February 2013. The abstract will rely mostly on this project work. If accepted we are expected to write a paper followed by a presentation.

We would like to extend our gratitude to Knut Magnus Kvamtrø, LA3DPA, for his help, lending us test equipment and for sharing his knowledge through fruitful discussions. We would also like to thank Roger Birkeland and Stig F. Mjølsnes for guidance and support through the semester.

Bram Bezem, LA6WTA
Per Kristian J. Fjellby, LA6XTA

Trondheim, December 2012

# Contents

# List of Figures

# List of Tables

# List of Acronyms

**AAU** University of Aalborg.

**ACK** Acknowledgement.

**AFSK** Audio Frequency-shift Keying.

**CCSDS** Consultative Committee for Space Data Systems.

**CRC** Cyclic redundancy check.

**CSP** CubeSat Space Protocol.

**EiT** Experts in Teamwork.

**FSK** Frequency-shift Keying.

**GPS** Global Positioning System.

**HMAC** Hash-based Message Authentication Code.

**LGPL** GNU Lesser General Public Licence.

**MAC** Message Authentication Code.

**NTNU** Norwegian University of Science and Technology.

**NUTS** NTNU Test Satellite.

**OBC** On-Board Computer.

**OSI** Open Systems Interconnection.

**PTT** Push-to-talk.

**RDP** Reliable Datagram Protocol.

**RX** receiving.

**SHA-1** Secure Hash Algorithm 1.

**TX** transmitting.

**UDP** User Datagram Protocol.

**UHF** Ultra High Frequency.

**VHF** Very High Frequency.

**VOX** Voice-operated switch.

**XTEA** eXtended Tiny Encryption Algorithm.

# Chapter 1
# Introduction

NUTS is a project run by the Department for Electronics and Telecommunications at NTNU. The goal is for students to design, build and launch a CubeSat by 2014 [3]. Most of the work is done by students in project- or thesis work, but it is also possible for volunteers to contribute to the project. The satellite will be a double CubeSat measuring 10x10x20 cm (LxWxH). Its primary function will be to capture images of the earth with an infrared camera. These images will be used for atmospheric observations [3].

Building a satellite is an interdisciplinary task. The NUTS group consist of people with backgrounds from disciplines such as electronics, mechanics, computer science, cybernetics, communications, and space technology. What this means for us is that everything we do will in some way impact other team members. There is also a goal to develop as much as possible of the satellite in-house. This means that work is maturing over time and presumptions that are valid one week may not longer hold the next week. This is one of the drawbacks of building as much as possible ourselves, as opposed to buying finished modules which are ready to go.

When we joined the project, the CubeSat Space Protocol (CSP) was already proposed as the communication protocol to be used on top of the physical layer radio link. CSP lacks a replay protection mechanism, and we wanted to add that to CSP.

We had three major goals in this project:

1. Study how to use CSP to create an authenticated uplink channel.
2. Enhance the CSP specifications with a replay detection mechanism
3. Set up an experimental implementation for the enhanced CSP, aiming to validate the functionality of the authentication properties of the uplink communication under realistic conditions.

We will start with an elaboration on the structure of CSP and related protocols in order to provide background on how these systems are used and how they can be modified to increase uplink security. Continuing, we will explain how we can add a replay detection mechanism and how we implemented this in CSP. Furthermore, we will discuss the choices made during the design and an analysis of the security provided by our extension. Finally, we will demonstrate the functionality of our modification through a series of tests.

# Chapter 2
# Background

A satellite communication system is set up using a radio link. Frequencies in the Very High Frequency (VHF) and Ultra High Frequency (UHF) ranges can be used. Even though our ground station uses directional antennas [3], directing most of the radio waves toward the satellite and not the surroundings, recording the communication between the ground station and satellite from a location close to the antenna is feasible. Suppose that this communication includes a message with a command scheduling the satellite to take five pictures. The recorded message can be *replayed* at a later time, and if the satellite is within reach it will happily take five more pictures. This can be repeated until the battery is drained. It is clearly beneficial to prevent these replays and to establish some understanding of who the communicating parties are, thereby restricting access to legitimate parties.

Earlier thesis work done for the project indicate that previous CubeSat projects have focused more on redundancy checks and error correction as opposed to security and authentication [4]. Our perception is that many run AX.25 protocols modified for their project, and try to keep their control commands a secret, a so-called *security by obscurity* approach. Our starting point is the CubeSat Space Protocol, which has been chosen as the backbone protocol for communications in the NUTS mission. Ultimately, everything we propose has to fit nicely into this software suite which will be running on both the satellite and the ground station.

## 2.1 CubeSat Space Protocol

### 2.1.1 Overview of the protocol

The CSP is a "small network-layer delivery protocol designed for CubeSats .." [5]. The idea stems from the University of Aalborg (AAU) but is today a part of GomSpace, a spin-off company founded by some of the entrepreneurs behind the AAU-CubeSat [6]. The source-code is made public under the GNU Lesser General Public Licence (LGPL) and can be found on `GitHub` under `gomspace/libcsp`. According to [5], it

is just the core that is made available on `GitHub`, and the extensions added after the creation of GomSpace is only available through their products.

CSP is based on a 32 bit header. Figure 2.1 shows the layering structure of CSP and the header together with the standard Open Systems Interconnection (OSI) model. The figure tries to illustrate how the various parts of CSP fit into the OSI abstraction layers.

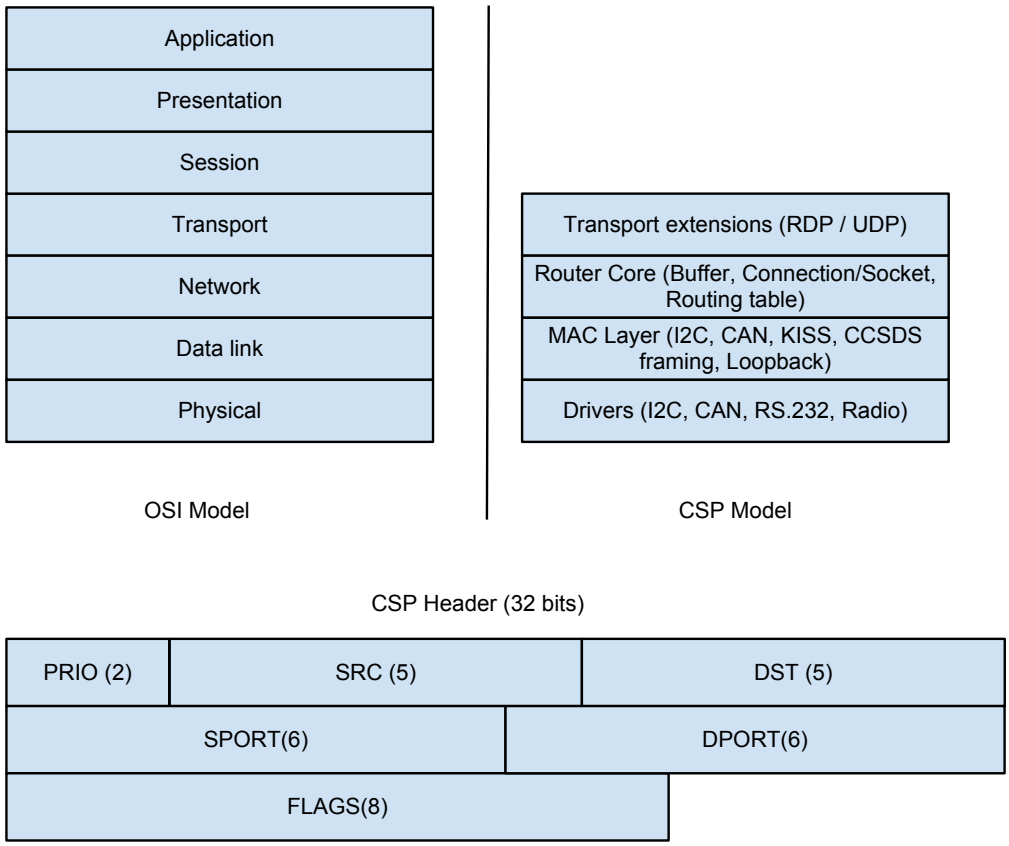


**Figure 2.1:** CSP layering and header.

Layer one, or the driver-layer, contains interfaces for many physical interfaces. The second layer, MAC layer, contains framing options for the physical media. As a GomSpace-extension, i.e. not available under LGPL, the space-link is set up with "Consultative Committee for Space Data Systems (CCSDS) framing format with

forward error correction, scrambling and a 32 bit sync marker" [5]. Layer three, the router core, handles all the routing based on the CSP header and buffer allocation. No route discovery is implemented so routing tables are static one-to-one maps of destinations and next-hop-interface [5]. Layer four, transport extensions, contains an implementation of the connection-less User Datagram Protocol (UDP). There is also an implementation of the Reliable Datagram Protocol (RDP) which adds retransmission and re-ordering on top of UDP.

The CSP header is made up of the following fields:

- `PRIO`: 2 bit field used to assign a priority to a packet. Possible values are
  `CSP_PRIO_CRITICAL` = 0
  `CSP_PRIO_HIGH` = 1
  `CSP_PRIO_NORM` = 2
  `CSP_PRIO_LOW` = 3
- `SRC`: 5 bit host address used to identify the source of the packet.
- `DST`: 5 bit host address used to identify the destination of the packet.
- `SPORT`: 6 bit number used to identify the process at the source.
- `DPORT`: 6 bit number used to identify the process at the destination.
- `FLAGS`: 8 bit number used to enable extensions. The available flags are

```
CSP_FCRC32 = 0x01
CSP_FRDP = 0x02
CSP_XTEA = 0x04
CSP_FHMAC = 0x08
CSP_FRES4 = 0x10
CSP_FRES3 = 0x20
CSP_FRES2 = 0x40
CSP_FRES1 = 0x80
```

  The four last flags are reserved for future use.

### 2.1.2 Cryptographic libraries and state of authentication

CSP is bundled with implementations of Hash-based Message Authentication Code (HMAC), Secure Hash Algorithm 1 (SHA-1), and eXtended Tiny Encryption Algorithm (XTEA). The HMAC and SHA-1 implementations are based on their respective `libtom` implementations, but adopted to CSP by the authors. The XTEA source file in CSP is not credited to any one. Enabling these primitives in CSP is a matter

of specifying the right *socket options* when sending or receiving a packet. Packets with flags not matching the socket options are dropped, i.e. if a packet with the HMAC flag set is received on a CSP socket without the HMAC socket option, and conversely, if HMAC is enabled on the socket but the flag is not set in the packet header, the packet is dropped.

CSP lacks any sort of replay-protection mechanism. Extending existing libraries and structures to enable such a feature has been one of our main goals. Message Authentication Codes (MACs) and hash functions fit nicely into what we wish to achieve, as the next sections will highlight.

## 2.2 Message Authentication Codes

MAC is a mechanism used to ensure that a piece of data is received exactly as the sender sent it (integrity) and that the message is from a legitimate source (authentic). More formally it means that the message can not be modified in flight, i.e. deletion or insertion of a chunk of data, without the receiving party finding out.

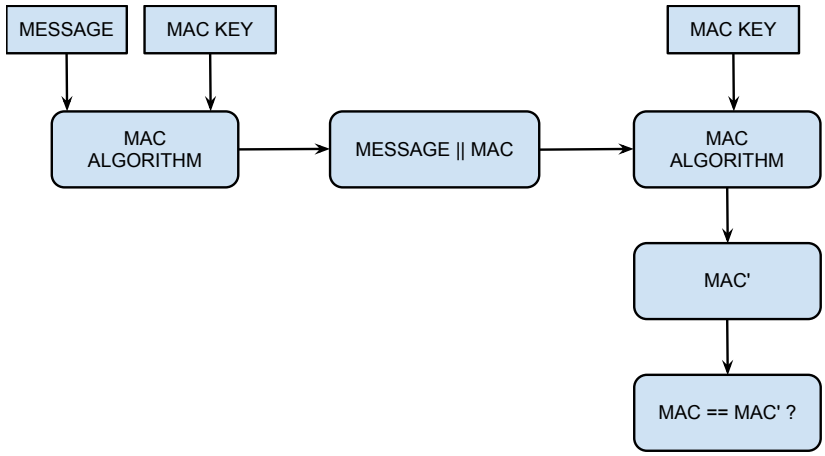


**Figure 2.2:** MAC setup. Adopted from [1].

Figure 2.2 shows a MAC setup. The message along with the MAC secret key is fed to a MAC algorithm. The algorithm returns the message and a MAC checksum/tag, which is appended and sent along with the message. When the message is received at the other end, the message, along with the MAC key, is fed to the MAC algorithm. The MAC is computed over the same data as on the other end, meaning that the MAC appended to the data is first removed and stored, then the receiving-side MAC is computed. If the received MAC and the locally computed MAC (MAC' in the

| Input | Output |
|---|---|
| 1 | 1 |
| 1000 | 1 |
| 100000 | 1 |
| 100001 | 2 |
| ... | |

**Table 2.1:** The use of $f(x) = x \bmod 3$ as a hash function

picture) are found to be equal, the MAC verification is successful. It they are not equal, the message should be discarded.

### 2.2.1 MAC generation

Let us now focus on the MAC generation or *MAC algorithm*. There exist numerous functions that can be used as a basis in the MAC scheme. Two examples are hash functions and block ciphers. Since CSP already supports generating MACs with hash functions, and we will be using this scheme, our focus is put here. Before we cover the Hash-based MAC scheme, we think it is justified to elaborate on the difference between hash functions and MAC.

**Hash functions** In essence, a hash function is a mapping of variable length input to a fixed set of output states. An example is doing the mathematical operation *modulo*. Table 2.1 shows input and output values using modulo 3 operations. If we limit the legal input to the natural numbers, $\mathbb{N} = \{0, 1, 2, \ldots\}$, we see that we have a way of mapping arbitrary sized input into a fixed size output. This approach is to simple to be directly adopted for use in cryptography and MACs, but captures the essence of a hash function.

**Cryptographic Hash Functions** A cryptographic hash is built up so that a small change in the input value generates a completely different output value. This property is referred to as the *avalanche effect* [7, p. 241]. The purpose is to make it difficult to select an input-string which produces a chosen hash. It should also be difficult to modify the input without changing the hash, and equally difficult to find two input values that result in the same hash. At the same time it should be computationally easy to compute the hash given the input. This set of properties makes up the general requirements for a cryptographic hash function, but the same properties are also sought for a MAC scheme based on hash functions [8].

**Collisions** Due to the finite-length output of hash functions *collisions* are bound to exist. By collisions we mean that two different messages, $m_1 \neq m_2$ will have the

same hash-value, $H(m_1) = H(m_2)$. In cryptographic terms, if $H(x) = y$, $x$ is often referred to as the *preimage* of $y$. The following requirements are often made for a cryptographic hash function [2, p. 360]:

1. First preimage property: It should be easy to compute $H(x) = y$, but given $y$ it should be *computationally infeasible*, meaning that if you throw an overwhelmingly large amount of computer resources at the problem, you will not be able to find an $x$ that gives $H(x) = y$ within a reasonable time frame.

2. Second preimage property: You are given $x$ and can compute $H(x) = X$. It should be computationally infeasible to find $y$ such that $H(y) = Y = X$ , $y \neq x$.

3. Collision property: It should be computationally infeasible to find pairs $(x, y)$ which satisfy $H(x) = H(y)$, and obviously $x \neq y$ to rule out the trivial case.

### 2.2.2 Hash-based Message Authentication Codes

What ultimately separates HMACs from cryptographic hash functions is the introduction of a secret key. RFC 2104 states the following goals for HMAC [8]:

- use of available hash functions
- easy replacement of hash function
- preserve performance of the hash function
- simple key handling
- thorough and well understood cryptographic analysis

Figure 2.3 shows the HMAC setup. $K$ is the shared secret key. The hash function operates on chunks of data called *blocks*. If the key is smaller than a block, it will be padded with 0s, hence $K^+$. If the key is longer than a block, it can be run through the hash function to produce a hash that is smaller than the blocksize, and be padded out to size afterwards. **ipad** and **opad** are constants, and named in such a way according to when they are used - the *inner* padding is applied first, then the *outer* padding is applied. $IV$ is the initialization vector for the hash function. More compact this set of operations can be written, for the key $K$ and the message $M$:

$$\text{HMAC}(K, M) = H[\ (K^+ \oplus \text{opad}) \parallel H[(K^+ \oplus \text{ipad}) \parallel M]\ ] \tag{2.1}$$

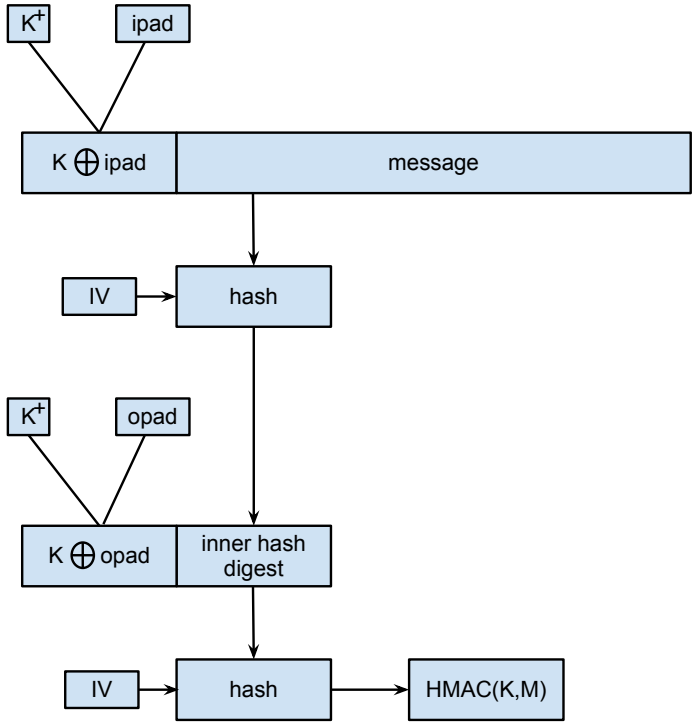


**Figure 2.3:** HMAC setup. Adopted from [2].

**Security** HMAC has gained widespread acceptance, much due to the superior performance in software of hash functions compared to MACs generated using block ciphers [2, p. 376]. The authors of HMAC stress that the security depends heavily on the underlying hash function [9]. They make a point of HMAC using the hash function as a black box, which can easily be swapped should the function show signs of weaknesses or be broken.

## 2.3 Secure Hash Algorithm (SHA)

The Secure Hash Algorithm is a National Institute of Standards and Technology (NIST) initiative. Developed by NIST it was published as a federal information processing standard (FIPS 180) in 1993 [2]. The SHA family is made up of algorithms known as SHA-0, SHA-1, SHA-2, and SHA-3. SHA-0 is the initial implementation from 1993 which in 1995 was modified due to discovered weaknesses. This revision became known as SHA-1 (FIPS 180-1). In 2005, SHA-2 was released by NIST (FIPS 180-3). SHA-2 is actually three hash functions with different output lengths. Lengths

of 256, 384, and 512 bits are available. These new versions have the same structure and operations as SHA-1 [2].

## 2.4 Replay mechanism and MAC

By introducing a MAC you protect the integrity of the data and make the transmission dependent upon a secret key shared by the communicating entities. If an attacker is able to record a packet and replay it over the communications channel, the integrity and authenticity provided by MAC is less trustworthy. Some form of time-dependent variable is needed in the data to protect the system from being prone to replay attacks. If this is added to the packet before the MAC is computed, thereby protecting it under the MAC secret key, an attacker is not able to modify the time-dependent variable without the MAC-check failing. Since only the parties with knowledge about the secret key can compute the correct MAC, the attacker is at best left with a way to disturb the communication. Two examples of such a time-dependent variables are sequence numbers and time stamps.

### 2.4.1 Sequence Number approach

Every data packet contains a sequence number and both communicating parties keep a local counter representing this number. When a party sends a data packet, the sequence number is added to the packet and the counter is increased. At the receiver end, the number is checked against the locally stored number. The requirement is that the received sequence number must be greater or equal than the one stored locally. If it is not, the packet is considered a replay and is discarded. If for some reason the two sequence numbers should become unsynchronized, a mechanism to put them back in order is needed.

### 2.4.2 Timestamp approach

Using timestamps the communicating entities rely on synchronized clocks to decide on the timeliness of a packet. Each packet contains a timestamp. If a packet is received within a predetermined threshold, the verification is successful - otherwise the packet is dropped. This require maintaining a strict synchronization between the clocks. A possible approach is to use the clocks of the Global Positioning System (GPS).

## 2.5 Experimental setup

A part of our project is validating our work by testing the protocol under realistic conditions. This involves running a CSP satellite process and a ground station process

on separate computers and connect them through radios. This interconnection involves several protocols which we introduce over the next sections.

### 2.5.1 AX.25

AX.25 is the a link-layer protocol published by the American Radio Relay League (ARRL) and the Tuscon Amateur Packet Radio Corporation (TAPR) [10]. What it gives us is the protocol framework for connecting our two amateur radio stations. Implementations in software exists, i.e. the ham radio tool `soundmodem` under `linux` provides such an implementation. AX.25 can also implemented in stand-alone boxes known as Terminal Node Controllers (TNC). With the latter setup, communications between the TNC and the radio is often run over a serial link using the `KISS` framing protocol.

### 2.5.2 KISS

KISS provides a standardized framing format for connecting a TNC to a radio. For example, `soundmodem` creates a virtual serial device over which the KISS frames are sent.

### 2.5.3 Modulation

For sending data packets over a radio link, many modulation techniques exist. The one we introduce here is Frequency-shift Keying (FSK). FSK is the general term used to describe modulation that uses changes in frequency of a carrier wave to represent data. Audio Frequency-shift Keying (AFSK) is a specific implementation where the frequency shits are done on an audible tone. The frequency of the tone is alternated between two frequencies to modulate the binary data. No modulation scheme is yet decided for the NUTS project. In order to test our implementation, a communications link has to be set up with a modulation scheme.

### 2.5.4 A note on link-speed

For satellite operations, the speed of the communications link is always a hot topic. One of the main tasks will be to download images from the satellite, and the usable time of a satellite pass is limited. A higher throughput will yield more satellite images per pass but sets stricter requirements on the equipment, including the choice of modulation. For testing purposes the link speed is not of great concern, and limited focus has been put on achieving speeds of 9600 bps and beyond.

# Chapter 3
# Methods

The following chapter describes the overall approach we used in designing our solution to the replay attack vulnerability.

## 3.1 Sequence numbers

It should be possible to attach a sequence number to packets between two nodes. This will ensure that every message is unique and therefore any HMAC-digest will be unique (excluding collisions). The system can also recognize and reject any previously accepted message so that the retransmission of recorded authentic messages is not possible. The presence of such a sequence number can be indicated by setting a flag in the header, just as it is done with HMAC, XTEA and CRC32 in the current implementation of CSP. Each direction of travel needs their own sequence number meaning each node will need to maintain a list of two sequence numbers for all possible hosts. One is used when receiving packets from the host in question and the other when sending to that host. Since CSP supports up to $2^5 = 32$ hosts, at most a list of 64 sequence numbers needs to be maintained by each node.

### 3.1.1 Shortcomings of using only one sequence number

The reason for introducing two sequence numbers has to do with the distributed nature of satellite communications. If we used one sequence number and the satellite and the ground station transmitted a message at the same time, thereby increasing their local sequence number counters, the message received would fail the replay check. An example with numbers makes this even more clear.

Imagine that the ground station and the satellite both have the number 12 stored locally as the current sequence number. The ground station starts to send a large set of commands to the satellite. The set is so large that it has to be fragmented into multiple CSP packets. The first packet sent from the ground station has sequence number 12. When the first packet arrives at the satellite it increases the stored

sequence number to 13. An Acknowledgement (ACK) packet is sent back to the ground station. The sequence number in this packet will be 13. Meanwhile, at the ground station, the second packet of the fragmented set of commands has left for the satellite with the sequence number 13. After the packet is sent from the ground station the sequence number is increased to 14. When the ACK packet from the satellite arrives at the ground station, it will be discarded because the sequence number is too low (13 in the packet from the satellite and the ground station is expecting 14). This is the reason for introducing different sequence numbers for each direction.

### 3.1.2 Introduction of two sequence numbers

We propose a setup where two sequence numbers are used. Both the ground station and the satellite have to store two sequence numbers. One for receiving (RX) and one for transmitting (TX). Lets revisit the same example as above, but now with the new setup. Assume that the TX and RX sequence numbers on both sides are 12 before we start. The ground station sends one packet, this packet has TX sequence number 12. After the packet is sent, the TX sequence number stored on the ground station is increased to $12 + 1 = 13$. The packet is received by the satellite which checks the sequence number in the packet (ground station TX sequence number) against its RX sequence number. Both are 12 so the check verifies. The satellite send an ACK packet with the satellite TX sequence number of 12. After the ACK is sent, the satellite TX sequence number is increased to $12 + 1 = 13$. Meanwhile, the ground station has sent another packet to the satellite with the ground station TX sequence number 13 and increased its TX sequence number to 14. When the ACK from the satellite arrives the TX sequence number from the satellite, which was 12 is compared to the ground station sequence number, which is 12. The verification is successful, and the ground station RX sequence number is increased to 13. And so the scheme continues.

## 3.2 Resynchronization

In order to make the system as reliable as possible there must be a way for them to resynchronize the sequence numbers. The two copies of the sequence number can become unsynchronized for several reasons. The satellite's orbit height makes it more prone to influence from cosmic and solar radiation. This influence might cause restarts or corrupt data in memory. In this case we do not want to start at zero again as that would allow messages from the previous sequence to be replayed. We will require that at least one of the nodes is able to store the sequence number in a reliable way. Using the resynchronization protocol it can share that number with the other node. It is automatically initiated when a threshold for errors is met, so that is does not require user intervention. The packets are sent using the same link

as the main data connections, but on a separate socket using a reserved port. This makes them distinguishable from regular packets. The packets are simple in structure merely containing the two sequence numbers concerning the hosts in question. Figure 3.1 shows the packets exchanged during a successful resynchronization sequence.

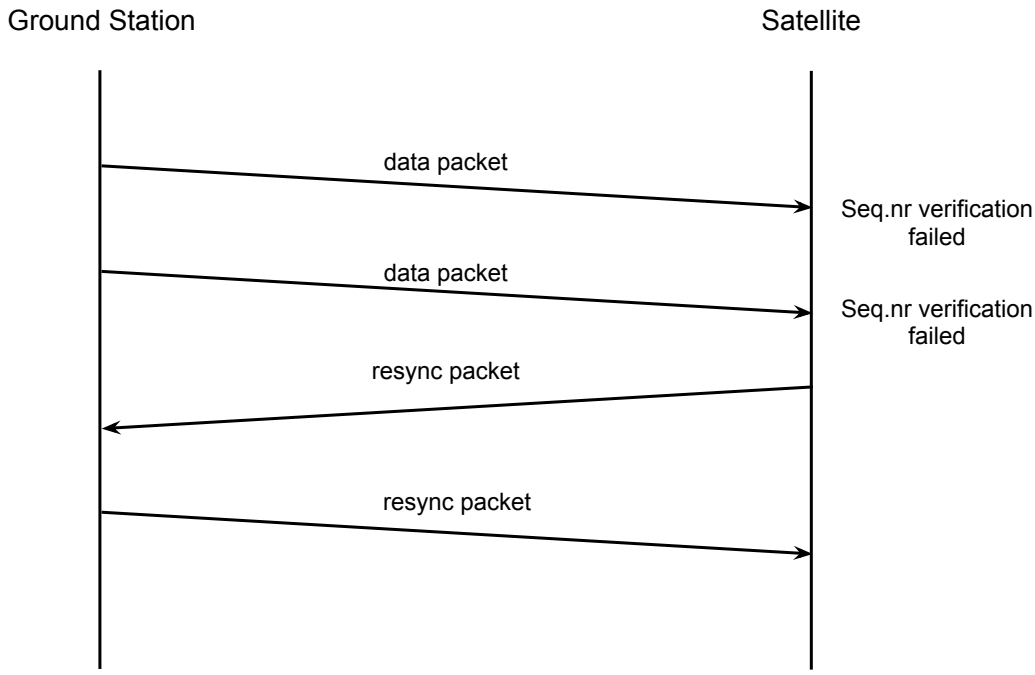


**Figure 3.1:** CSP resynchronization exchange

## 3.3 Testing

One of the goals stated in our project description was that we should test whether our solution worked in a practical test scenario. The NUTS project is currently aiming at getting an engineering model ready. A lot of theoretical work has been done as project and master thesis work, but we are lacking implementations due to the high turnover rate amongst students. Students leave the project before they implement their work. To avoid this, we wanted to test our solution over a radio-link.

The first of our two test setups was a RS.232 null-modem connection between two dedicated computers using their integrated COM-ports.

The radio setup was slightly more complicated. One side consisted of a computer and an ICOM IC-9100 radio. The computer runs `soundmodem`, a software package that uses the computer's sound-card to send AX.25 packets to a radio. `soundmodem` can use a variety of modulation types including AFSK which is what we wanted to use. The IC-9100 has an integrated USB interface that presents itself as a sound-card with an input and an output channel, plus a virtual COM-port for controlling the radio. This is well suited for use with `soundmodem`, as the latter can be configured to use the sound-card provided by the radio itself. The `soundmodem` application can listen to any audio signals as long as the radio is in receive mode, and when it is in transmit mode, it will transmit the audio received via its sound-card. However, for unknown reasons, the protocol used to control the radio over the virtual COM-port does not support Push-to-talk (PTT), therefore we ran in difficulties trying to find a way to have the radio switch to transmit mode. The solution we found was to build our own PTT circuit consisting of a transistor that connected the PTT-pin on the accessory (ACC) socket of the radio to ground. This transistor could then be activated by the Request-to-send pin of a physical COM-port on the computer. `soundmodem` can be configured to use the RTS-pin to control the radio.

On the other side, the hardware consisted of a computer, a Signalink USB and an ICOM IC-910H radio. The Signalink is a usb sound-card interface, that similarly to the IC-9100 radio, presents itself as a sound-card that the `soundmodem` software can use for input and output. It provides galvanic isolation and a Voice-operated switch (VOX), meaning it will trigger a PTT to the radio as soon as it detects an audio signal on its input port. It connects to the radio using its DATA port which has input, output, PTT and ground pins.

`Soundmodem` allows both half-duplex mode, where the two nodes share the medium and transmit and receive on the same frequency, and full-duplex, which requires different frequencies for the directions of traffic. Both radios have a function called Satellite mode. This will have them send and receive on different bands, each with their own antenna (for a total of four). True full-duplex was therefore possible and we decided to try both full- and half-duplex.

These setups would then be used to transmit packets over a channel protected by HMAC and sequence numbers, using simple test applications written by us. One was termed the 'server', it would merely listen to incoming connections and receive packets, similar to the satellite waiting for and receiving commands. The other was termed the 'client', it would attempt to connect, if successful send several packets of dummy data and disconnect. It would continue until stopped by user-intervention. Both programs would start using sequence number 0 every time they are stopped and restarted.

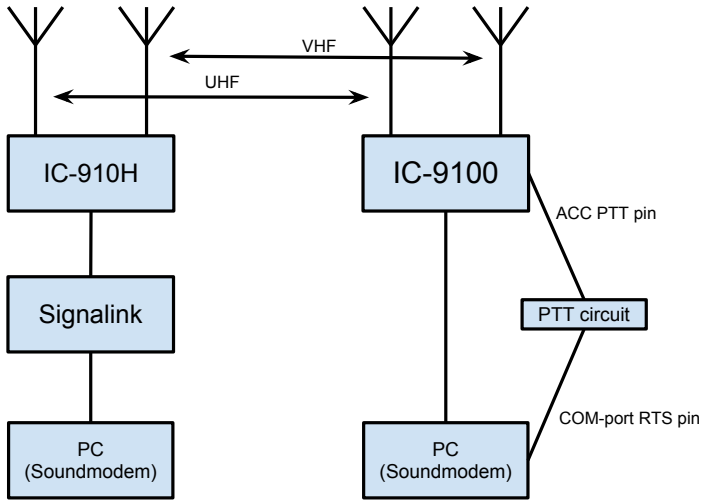


**Figure 3.2:** An illustration showing the interconnection of various devices used in the radio test setup

We had four different scenarios we wished to test over three different link types; serial, half-duplex radio and full-duplex radio.

1. Normal use of sequence numbers, by running both programs over a period of time
2. Resynchronization by letting them run and then restart the 'client' so it suddenly loses its sequence number
3. Resynchronization by letting them run and then restart the 'server' so it suddenly loses its sequence number
4. Resynchronization when both sides lose the sequence number.

# Chapter 4
# Results

Our solution consists of two main components. The addition of sequence numbers as an option for connections and packets, and a resynchronization algorithm so we can ensure both sides are expecting and using the same values.

## 4.1 Sequence numbers

The first step for adding the required sequence numbers was renaming one of the 'Reserved for future' use flags `CSP_FRES1` and use it to indicate whether a packet has a sequence number added or not by renaming it to `CSP_SEQNR`. We also added socket and connection options so one could specify which connections should use sequence numbers. These are called `CSP_SO_SEQNR` and `CSP_O_SEQNR`. In order to keep a maximal amount of bandwidth available for data, the sequence number was not added as a part of the `csp_packet_t`, the packet type definition. This would mean that even packets without sequence numbers would have space reserved for the sequence number. Instead it is appended to the data and its presence is indicated by the `CSP_SEQNR` flag. Figure 4.1 show the location of the sequence number is the CSP packet. We chose the sequence number to be represented by a unsigned 16 bit integer, we will elaborate upon that choice in section 5.2

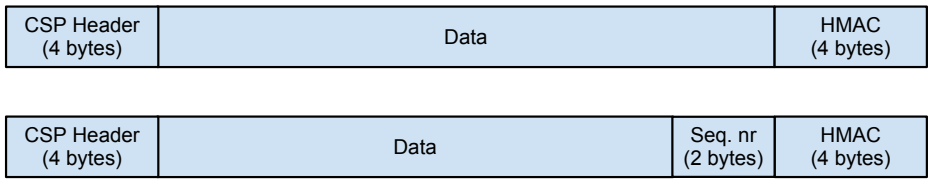


**Figure 4.1:** The overall structure of CSP packets, the original from CSP v1.1 on top and the modified one below

Any packet sent on a connection or via a socket that requires sequence numbers will automatically have a sequence number appended. Upon reception of a packet with a sequence number CSP will check that the connection or socket has sequence number support enabled. If so, CSP will try to verify the sequence number. If the verification is successful, the sequence number will be removed from the packet and the remaining data is delivered to the connection or socket. If the verification fails, the system will check if the number of sequential failures exceeds a configurable limit of `CSP_SEQUENCE_RESYNC_THRESHOLD` and initiate a resynchronization process if it does.

## 4.2 Resynchronization

As part of our solution to the replay attack vulnerability we needed a way to resynchronize the two copies of the sequence number. This resynchronization is initiated by the verification process after a set number messages (`CSP_SEQUENCE_RESYNC_THRESHOLD`) from a specific host fail to pass sequence number verification. A new thread will be started which will connect to the other communicating party on the `CSP_RESYNC`-port and send its copies of the sequence numbers. Due to this design choice, any party wishing to send packets with sequence numbers must listen to the `CSP_RESYNC`-port so the receiver of its packets can notify the sender when the sequence numbers do not match. The resynchronization process is a simple transaction where the initiating party sends over their sequence numbers and the responding party decides on which sequence numbers should be the shared ones. In our case we choose the highest of the local and remote numbers for both directions in order to avoid any reuse. This chosen value is then sent in return and the receiver verifies they are higher than its local numbers.

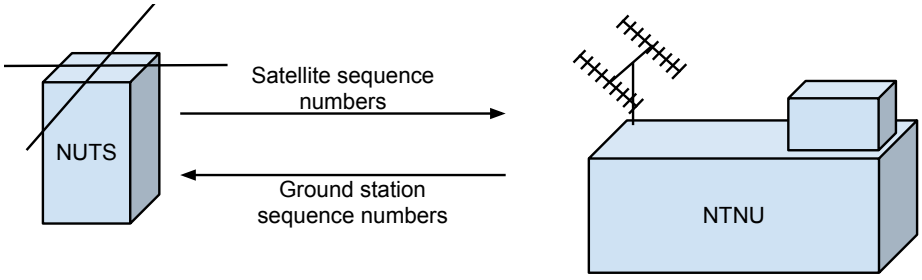


**Figure 4.2:** An overview of the messages sent during resynchronization

Figure 4.2 shows the flow of packets during resynchronization. These are not protected by HMAC, due to the design of CSP which uses one HMAC key for all packets, we will discuss this in more detail in section 6.3. This opens up for a vulnerability described in section 5.5.2

## 4.3 Testing

We ran the four different scenarios described in section 3.3 on three different types of links

### 4.3.1 Serial

Since the RS.232 serial link has dedicated wires for transmitting and receiving, it is a full-duplex link by nature. It handles all scenarios without problems other than the fact that in the final scenario, the system re-uses sequence numbers.

```
bob@bob:~/workspace/project$ ../../csp/bin/server
Initialising CSP
Using '/dev/ttyS0' as the serial port
Starting resync_listen_task
Starting server task
Listening
Verifying sequence number: PACKET = 0x00, COUNTER = 0x00
Verifying sequence number: PACKET = 0x01, COUNTER = 0x01
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x02, COUNTER = 0x02
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x03, COUNTER = 0x03
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x04, COUNTER = 0x04
Verifying sequence number: PACKET = 0x05, COUNTER = 0x05
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x06, COUNTER = 0x06
Verifying sequence number: PACKET = 0x07, COUNTER = 0x07
Verifying sequence number: PACKET = 0x08, COUNTER = 0x08
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x09, COUNTER = 0x09
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x0A, COUNTER = 0x0A
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x0B, COUNTER = 0x0B
Verifying sequence number: PACKET = 0x0C, COUNTER = 0x0C
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x0D, COUNTER = 0x0D
^C
bob@bob:~/workspace/project$
```

**Figure 4.3:** The output from the server when both nodes are working as expected

```
bram@office2:~$ ./client
Initialising CSP
Using '/dev/ttyS0' as the serial port
Starting resync_listen_task
Starting send_hello_world_task
Verifying sequence number: PACKET = 0x00, COUNTER = 0x00
Sending a packet: Length = 11, DATA = Hello world
Sending a packet: Length = 11, DATA = Hello world
Sending a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x01, COUNTER = 0x01
Sending a packet: Length = 11, DATA = Hello world
Sleeping 6000 ms
Verifying sequence number: PACKET = 0x02, COUNTER = 0x02
Done sleeping
Verifying sequence number: PACKET = 0x03, COUNTER = 0x03
Sending a packet: Length = 11, DATA = Hello world
Sending a packet: Length = 11, DATA = Hello world
Sending a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x04, COUNTER = 0x04
Sending a packet: Length = 11, DATA = Hello world
Sleeping 6000 ms
Verifying sequence number: PACKET = 0x05, COUNTER = 0x05
^C
bram@office2:~$
```

**Figure 4.4:** The output from the client when both nodes are working as expected

Figures 4.3 and 4.4 show the output from the server and client respectively when we tested scenario 1 over the serial link. The output consist of some initial information explaining the start-up process for both, but our main interest lies in the output from the sequence number verification and the part of the packet to the application layer. Since the verification happens before the packet is handed to the application layer, the check right before the confirmation of reception is the one that verified that packet.

Some other notable observations that can be made are that RDP is enabled, as shown by the fact that there is a verification of a transport-layer packet before any data packet, indicating the presence of a SYN-packet used for connection setup. Any verification done on the client side, is the result of signalling packets. These are the ACK-packets confirming the reception of the data-packets the client sent. Interestingly, there is only one ACK for every two data-packets.

```
bob@bob:~/workspace/project$ ../../csp/bin/server
Initialising CSP
Using '/dev/ttyS0' as the serial port
Starting resync_listen_task
Starting server task
Listening
Verifying sequence number: PACKET = 0x00, COUNTER = 0x00
Verifying sequence number: PACKET = 0x01, COUNTER = 0x01
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x02, COUNTER = 0x02
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x03, COUNTER = 0x03
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x04, COUNTER = 0x04
Verifying sequence number: PACKET = 0x05, COUNTER = 0x05
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x06, COUNTER = 0x06
Verifying sequence number: PACKET = 0x00, COUNTER = 0x07
[01] csp_route.c:167 SEQNR verification failed
Verifying sequence number: PACKET = 0x01, COUNTER = 0x07
[01] csp_route.c:167 SEQNR verification failed
Starting resynchronization
Received a RESYNC RESPONSE packet: Length = 4, RAW = 0x00070003
Verifying sequence number: PACKET = 0x07, COUNTER = 0x07
Verifying sequence number: PACKET = 0x08, COUNTER = 0x08
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x09, COUNTER = 0x09
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x0A, COUNTER = 0x0A
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x0B, COUNTER = 0x0B
Verifying sequence number: PACKET = 0x0C, COUNTER = 0x0C
Received a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x0D, COUNTER = 0x0D
^C
bob@bob:~/workspace/project$
```

**Figure 4.5:** The output from the server when the client loses track of the sequence numbers

Figures 4.5 and 4.6 on the other hand, show the output from the system when the client's sequence numbers are reset between to rounds of data. In this case we see the resynchronization behaviour required for resuming normal operation. The

```
bram@office2:~$ ./client
Initialising CSP
Using '/dev/ttyS0' as the serial port
Starting resync_listen_task
Starting send_hello_world_task
Verifying sequence number: PACKET = 0x00, COUNTER = 0x00
Sending a packet: Length = 11, DATA = Hello world
Sending a packet: Length = 11, DATA = Hello world
Sending a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x01, COUNTER = 0x01
Sending a packet: Length = 11, DATA = Hello world
Sleeping 6000 ms
Verifying sequence number: PACKET = 0x02, COUNTER = 0x02
^C
bram@office2:~$ ./client
Initialising CSP
Using '/dev/ttyS0' as the serial port
Starting resync_listen_task
Starting send_hello_world_task
Processed in the service handler
Preparing to send RESYNC RESPONSE: 3 7
Sending a RESYNC RESPONSE packet: Length = 4, RAW = 0x00070003
Verifying sequence number: PACKET = 0x03, COUNTER = 0x03
Sending a packet: Length = 11, DATA = Hello world
Sending a packet: Length = 11, DATA = Hello world
Sending a packet: Length = 11, DATA = Hello world
Verifying sequence number: PACKET = 0x04, COUNTER = 0x04
Sending a packet: Length = 11, DATA = Hello world
Sleeping 6000 ms
Verifying sequence number: PACKET = 0x05, COUNTER = 0x05
^C
bram@office2:~$
```

**Figure 4.6:** The output from the client when the client loses track of the sequence numbers

light-blue lines show the failed verifications and from the lines directly above we see the server received a packet with sequence number 0 while expecting 7. These numbers do not match, and a verification error is triggered. After the second failure, the `CSP_SEQNR_RESYNC_THRESHOLD` of two is exceeded and a resynchronization is initiated. This process results in an agreement to use 7 and 3 as the receiving and transmitting sequence numbers respectively for the server. These will be reversed for the client. Once resynchronization is successful, normal operation resumes.

### 4.3.2 Radio (half-duplex)

This setup works well with UDP, the packets are sent, verified and if there are any unsynchronized sequence number, the system can resynchronize without problems. However enabling RDP turned out to be disastrous. Even though `soundmodem` has several configuration options to allow both nodes access on the shared medium, the client would use the channel continuously without allowing the server to acknowledge. This then leads to it thinking the packets were not received and retransmitting them until it assumes the connection is lost and gives up. This lead us to change to full-duplex radio communication using the satellite mode offered by the radios.

### 4.3.3 Radio (full-duplex)

This configuration was introduced since the half-duplex demonstrated problems with all four scenarios. UDP did not have any problems in this case either. All four scenarios worked as designed. However we did discover some issues once RDP was introduced. In particular, the RDP parameters were not suited for 1200 baud operation. The client expected an ACK even before it was done transmitting its own packet, since this timer runs from when the packet is handed off to the link layer. Once we changed the `csp_rdp_packet_timeout` and `csp_rdp_ack_timeout` parameter to 3000 and 3000/4 respectively from 1000 and 1000/4, the transfer flowed smoothly. The output from the test applications was identical compared to the output when using the physical RS.232 link.

# Chapter 5
# Discussion

## 5.1 Sequence number or time stamp

As indicated previously, either sequence numbers or time stamps can be used to introduce uniqueness to each packet. We present here our rationale for choosing sequence numbers over time stamps.

GPS clocks could be used to provide accurate time to be used in our scheme. Even without the use of GPS, we expect there to be some type of time reference in the satellite to be used in picture timestamping. For our purpose, the clocks in the satellite and on the ground station would need to be closely synchronized. It is a well known fact that clocks drift. To account for this, a scheme has to be put in place to synchronize satellite and ground station clocks. We did not want to introduce this requirement for our solution, so we discarded the idea of using our own clocks.

GPS is known to have very accurate clocks, in fact the entire GPS scheme is reliant upon the accuracy of clocks in the satellites and the receiver [11]. Putting a GPS receiver in our satellite and one on the ground station could give us the accuracy we need. The problem is that GPS modules intended for civilian use have height and speed restrictions. Typical figures we have seen indicate an upper bound on the height and speed of 18 km and a 515 m/s [12]. It varies how some vendors enforce these requirements. Some deploy an AND requirement, meaning that both the speed and height restrictions have to be exceeded. Others use an OR requirement, meaning that only one violation is enough. The requirements are anyway to strict for most space related activities. Applying for a military-grade GPS receiver and obtaining the proper export permissions seemed like a too daunting task.

## 5.2 Sequence number length

Previous work on the NUTS project [4] lead to a suggested size of 16 bits which we deemed adequate, taking into consideration the approximate rate of sent packets and

the expected lifetime of the satellite. A size of 16 bits allows 65536 unique packets to be sent. The sequence numbers will only be used on the uplink when sending commands, and on the acknowledgement packets for those command-packets. A scheduling system allowing several commands per packet, each with a time-stamp for when they are to be executed, is planned. Several commands can be then be sent per packet. Because of these factors we do not believe the average number of packets sent per pass will exceed 10 over the lifetime of the satellite. Calculating with an average of ten packets per pass and five passes per day, this gives us over three and a half years of operation. The lifetime is estimated to be between six months and two years, thus we consider a length of 16 bits enough with a decent safety margin.

Another factor we took into consideration is the overhead our scheme would introduce. By overhead we mean metadata that must be transferred together with user data, thus negatively impacting the net throughput. Due to our design choice to not add it to the header, the sequence number will only add overhead when it is enabled on a connection. We expect for it to be used only on the uplink where the user data consists of application layer data with commands and their arguments destined for components in the satellite. There data throughput is not a big concern, while authentication is. The downlink is where the pictures from the satellite will be transmitted, our recommendation would be to not use sequence numbers on this connection because it will negatively impact bandwidth, which already is a major concern. Already a lot of effort is being put into data compression and image averaging [13] in order to minimize the amount of data that has to be sent down to earth. The large data volumes would also mean that a large number of packets would be sent, thus requiring a longer sequence number. The next step up in unsigned integers would be 32 bit allowing over four billion unique packets, but having a twice as large impact on throughput.

## 5.3 Resynchronization selection function

As part of the resynchronization process the parties are notified when they are unsynchronized. The initiating party sends their sequence numbers. The responding party makes the decision of what both parties' sequence numbers should be set to. It chooses the highest of the received number and its own copy, to ensure that the sequence number always is increasing. The initiating party also checks that the number it receives in the response is larger than the number it has stored at that time. This means a sequence number is never used twice, but presents a problem with bit flips in the memory of the satellite that can cause the sequence number to increase radically. For example, a bit flip in the most significant bit in the sequence number would lead to a jump of $2^{15} = 32768$ which is half the possible space.

## 5.4 Choice of layer

Any time one desires to implement security features in a protocol stack, which layer it is added to is an important consideration. For this project we mainly considered four different options the link layer: the network layer, the transport layer and the application layer.

### 5.4.1 The link layer

The lowest of the four options is the link layer, also known as the KISS layer in CSP when used in conjunction with AX.25. Implementing security would secure the radio-link and nothing else. While limited, this would not pose a problem as the rest of signalling path consists of wired connections. These connections are physically part of either the satellite or on the ground-station, and we consider physical access for an attacker to be unlikely. For the satellite there is the challenge of its location in space. For the ground-station, physical access would mean they also could get control of the secret keys, rendering all security features based on those useless.

The major objection to adding security on this layer were the fact that this layer is specific to KISS/AX.25. The implementation would either lock the satellite project to using KISS/AX.25 or we could risk that our work is discarded if an alternative is chosen. Another issue is that the AX.25 implementation is part of the official linux-kernel and the KISS interface is rather simple. In order to add security features like HMAC with a sequence number and resynchronization protocol for KISS, the entire addition would essentially be a security layer between the network and link layers. Rather than doing that we thought it would be more prudent to integrate it with the existing network layer.

### 5.4.2 The network layer

The network layer is the layer we ended up using for our replay-protection implementation. The advantages being that it already contains a working HMAC based authentication and integrity protection scheme, thereby reducing the amount of work that needed to be done. In addition, when computing a hash you want as much information as possible protected by this hash. On the network layer this includes the CSP-header, which contains information about final destination, original source host, ports and flags which are used to indicate the security level of a packet. By adding security on this layer, any packets would be secure from the source host to its destination providing end-to-end authentication. Since the satellite will consist of at least two CSP nodes (the radio and the On-Board Computer (OBC)), this allows us to address both independently and securely

### 5.4.3 The transport layer

The transport layer in CSP is where the RDP and UDP implementations reside. Our initial thoughts was that while the RDP implementation contains a sequence number, it starts from zero on every new connection. Unlike the network layer it does not contain any information about hosts so captured valid packets can be used on any new connection to any host, unless a different key is used for each of these. However such a session-key scheme poses problems of its own, which we look at in section 6.2

### 5.4.4 The application layer

Putting the authentication on the application layer was something we also considered. All security would then be handled without intervention by CSP. Again, if using HMAC with a fixed key, there must be something separating two identical commands from each-other. There are at least two nodes capable of receiving commands from the ground station (the radio and the OBC). These will be running the same software. Just like the network layer one could maintain a sequence number which allows the receiver to recognize replay attempts, but one would also need an identifier for which node is being addressed so that a command to one node cannot be replayed to the other. A scheme like the one described requires duplication of information already present on the network layer. It would also complicate the application layer software, which would oblivious in case of a network layer implementation. The software can only see which socket options are enabled, not how the network layer implements them.

## 5.5 Attack analysis

### 5.5.1 MAC

Here we identify and analyse ways to attack our design. We look at the following scenarios:

1. Modify a previously sent message to run a command of his choosing.
2. Replay a message previously sent from the ground station to the satellite.

**#1** Here an attacker is able to record a message and tries to modify it at bit level to insert his own command. The source code will be made public and no proprietary codecs are used on the radio, so modification should be feasible. We see two major obstacles. If one alters the message contents by which the HMAC calculation is based upon, the subsequent verification at the satellite will fail. For it not to fail, the attacker would have to find an input message that produces the same HMAC tag as the original message. This is known as a collision, and if the MAC is strong,

finding such collisions is computationally infeasible. If one should succeed in finding such a pair, the likelihood if it resembling the command structure of a CSP packet is highly unlikely.

**#2** This attack is just a replay of a previous message, meaning that no modification to the packet is done. The packet will have a valid HMAC tag, so the sequence number check will decide if this is a replay or not. Since the satellite will require an increase in the sequence number, if replayed the packet will be discarded. The attacker could however delay the transmission, i.e. by jamming the satellite so that no commands are received and thereby no increase to the satellite sequence number is made. If the ground station sends messages while the satellite is jammed and an attacker records these messages, they can be replayed after the jamming is stopped. The commands are authentic since they are sent by the ground station, but the delay could cause potential problems, i.e. that the camera is triggered when it should not be, etc. Using some form of scheduling of the commands could mitigate this problem. Say that the command instructs the satellite to take 5 pictures at a given time. Then you have limited the possibility of issuing untimely commands. If a valid command is received after it is intended to be executed it should be discarded by the scheduler.

**Concluding remarks** If an attacker were to get access to the keys used in the authentication scheme, the whole system is compromised. An attacker can then create messages with a sequence number that is large enough and recompute the HMAC with the key. Protecting the keys is therefore vital.

### 5.5.2 Sequence number

**Pre-analysis** As mentioned earlier, CSP sets connection options when the connection is first created. If the sequence numbers on the ground station and satellite become skewed, packets are discarded. If we were to send the resynchronization packets over the same connection, these packets would also be dropped. The way we solved this was to make both nodes listen on a port we reserved for sequence number resynchronization. When a node initiates a resynchronization, a new connection is established on the resynchronization port with the socket option of sequence number disabled. In essence this means that resynchronization packets does not contain any means to support replay detection, so previously captured resynchronization packets may be recorded and resent at a later time. The goal of this section is to go through different attack scenarios to see what an attacker has to work with in order to compromise our system through the resynchronization procedure.

**Interfering with the resynchronization** We assume for this section that the attacker is in possession of a valid resynchronization request packet. This packet could be obtained by recording a transmission of such a packet from either the

satellite or the ground station. It is also possible to construct such a packet bit for bit. Let us first examine the case where an attacker initiates a resynchronization by transmitting a valid resynchronization packet.

**Attacker initiates resynchronization request** The resynchronization request packet could be sent to either the ground station or the satellite. If the packet is valid, where valid in this sense means that it is a correctly structured CSP resynchronization packet, the receiving side will process the packet. The receiver will then select the sequence number with the highest numerical value as the new sequence number, based on a comparison of the sequence number received in the resynchronization request packet and the number stored locally. A response will then be sent containing the sequence number to be used. Upon receiving this message, the attacker can make the same check as the receiver and determine the new sequence number. The sequence number is now synchronized between the attacker and either the satellite or the ground station. Since the attacker does not know the shared secret key used for HMAC he is not able to send packets that will pass the message authentication check on the authenticated channel, following the same reasoning as the previous section. The fact that the attacker has interfered with the sequence number mechanism will have implications for legitimate operations when the satellite comes within radio distance of the ground station - if the sequence numbers have been altered a new resynchronization is required in order to resume operations on the authenticated channel.

**Effect of attack** If the attacker simply replays a resynchronization request he has heard before, the damage done is limited. The fact that the packet is a replay means that the sequence number included in the resynchronization request packet is, with high probability, used before. Therefore the new chosen sequence number will remain unchanged from the ground station/satellite perspective, and no resynchronization is required for legitimate communications to resume.

Things get more interesting if the attacker has the possibility of generating his own resynchronization request, and especially inserting a sequence number of his choosing. Remember that the resynchronization scheme is based on selecting the highest sequence number as the new sequence number. If an attacker is able to introduce large jumps, the sequence number space will be used up fast.

One possible solution is to disallow very large jumps in the sequence number. The problem is where to draw the line. Ultimately you could end up with a situation where the sequence numbers differ by so much that such a large jump is required in order to resynchronize, and you effectively end up with a *livelock* situation. The best solution we have come up with for this problem is to apply HMAC on the resynchronization link as well. An attacker could still send replays of legitimate

resynchronization sequences, but not create custom requests. The replays will have no effect since the system selects the highest of the received sequence number and its own, thereby rejecting the replayed attempt. The attacker is not able to create his own request because they will fail under the HMAC verification. We have not focused on key generation and key handling, but we believe that the use of different keys for message authentication and resynchronization authentication is good practice.

# Chapter 6
# Future work

## 6.1 Secure storage of sequence numbers

One important consideration when using sequence numbers is how it is stored locally. RAM is susceptible to power losses so a more persistent location should be chosen. The loss of a sequence number can result in a security breach since it might reset the sequence number to zero. An attacker can impersonate a node and start replaying old packets, at least until the next resynchronization. However, if both sides lose track of the sequence number, no such resynchronization will take place. The nodes will start using the latest sequence number used by the attacker. Because of this issue, our solution requires that at least one node is able to keep track of the sequence number. There might still be a window of attack between the loss of a sequence number and until a resynchronization has taken place, but it can be mitigated by having both nodes attempt to securely store the sequence number. In this case, security refers to security from data loss, not authentication security. Since the satellite will be approximately 600 kilometres above the earth's surface it will be vulnerable to cosmic radiation. As of the writing of this report, there is work ongoing in providing the satellite with a secure memory protected from bit-flips and other undefined behaviour using data duplication and checksums. Once completed this can provide us with a persistent way to store keys and sequence numbers.

## 6.2 Sessions keys

As a possible alternative to the solution presented in this report, we considered using the sequence number that RDP uses for providing reliability. However, HMAC will produce the same output when used with the same key and input data. We either need some uniqueness in the data, like sequence number, or we must use a different key. RDP will start using zero as it's sequence number on every connection, in order to assure a unique HMAC for every packet even on different connections, a new key must be used for each connection. These keys must then be generated and exchanged

in a secure manner. RDP uses a three-way handshake for connection setup, this might be a suitable candidate for such an exchange. Yet, CSP uses HMAC on the network/routing core layer, where it can include hosts and ports to the input for the hash. This is independent of the transport protocol used, and therefore it requires a key even for the handshake packets. For this solution to work, either the HMAC operation can be moved to another layer, or a key-exchange can take place outside the authenticated channel.

## 6.3 Adding an authentication key for resynchronization

As discussed in section 5.5.2, our solution is vulnerable to an attacker performing resynchronizations with both the ground station and satellite, leading to both sides running out of room in the 16-bit sequence number. However, it is possible to prevent such a attack by adding a separate authentication key for the resynchronization channel. This can then be used to calculate an HMAC over the resynchronization packets. While there is no particular resynchronization sequence number, this solution is not vulnerable to replay as any replayed packet at worst will result in a resynchronization to the same number as the current.

The addition of another HMAC-key for this purpose would require a change in the way CSP stores keys. Currently it is only capable of storing a single key used for all authentication related processes like adding and verifying HMAC if enabled on the socket/connection. Thus the amount of keys stored must be changed and logic added so the system can decide which key it should use in a particular situation.

# Chapter 7
# Conclusion

In this report we have described and discussed the various options for an modification to the current CSP implementation which adds replay protection. We have defined a specification for both the addition of a sequence number to packets sent on selected connections. In addition, we have defined behaviour for the verification of those sequence numbers, and a resynchronization protocol in case the different nodes disagree on the correct sequence number. This protocol is initiated automatically once the number of sequential failures exceeds a set threshold.

Furthermore, we implemented our specification as an extension of the latest version of CSP published on `GitHub`. This extension includes a pair of applications that make use of the added functionality while sending dummy data over an authenticated connection.

Finally we set up a realistic test environment consisting of two radios, two computers and various supporting equipment, in order to run these application over a radio-link and verify the functionality provided by the extension. These tests showed that the software functioned as designed and provided additional security features.

We feel that we have reached all three goals we set for ourselves for this project.

# References


[1] K. Martin, *Everyday Cryptography:Fundamental Principles and Applications*. OUP Oxford, 2012.

[2] W. Stallings, *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 2011.

[3] R. Birkeland, "Nuts-1 mission statement," 2011. http://nuts.cubesat.no/upload/2012/01/20/nuts-1_mission.pdf.

[4] S. Prasai, "Access control of nuts uplink," Master's thesis, Norwegian University of Science and Technology, July 2012.

[5] Gomspace, "Network-layer delivery protocol for cubesats and embedded systems.." http://www.gomspace.com/documents/GS-CSP-1.1.pdf.

[6] Gomspace, "Company profile." http://gomspace.com/index.php?p=profile.

[7] H. Williams, *Advances in Cryptology: Proceedings of CRYPTO '85*. Lecture Notes in Computer Science, Springer, 1986.

[8] H. Krawczyk, M. Bellare, and R. Canetti, "Rfc 2104," *HMAC: Keyed-Hashing for Message Authentication*, February 1997.

[9] H. Krawczyk, M. Bellare, and R. Canetti, "Advances in cryptology. crypto 96 proceedings," *Keying Hash Functions for Message Authentication*, June 1996.

[10] W. A. Beech, D. E. Nielsen, and J. Taylor, "Ax.25 link access protocol for amateur packet radio," *Version 2.2*, July 1998.

[11] G. Xu, *Gps: Theory, Algorithms and Applications*. Springer Verlag, 2003.

[12] M. D'Errico, *Distributed Space Missions for Earth System Monitoring*. Space technology library, Springer, 2013.

[13] M. Bakken, "Signal processing for communicating gravity wave images from the ntnu test satellite," Master's thesis, Norwegian University of Science and Technology, July 2012.

Appendix A

# CSP sequence number specification

This document is meant to be a quick reference for our CSP sequence number implementation.

## A.1 CSP data packet


**Figure A.1:** CSP data packet

Figure A.1 shows the CSP data packet with HMAC enabled. Other options such as XTEA, Cyclic redundancy check (CRC) is not shown.

## A.2 CSP resynchronization packet

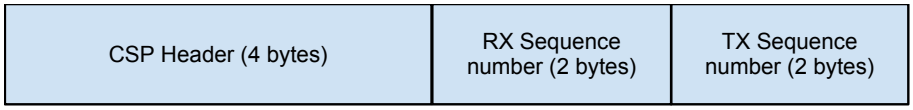


**Figure A.2:** CSP sequence number resynchronization packet

Figure A.2 shows the structure of a resynchronization packet. The CSP header, consisting of four bytes and two sequence numbers consisting of two bytes each.

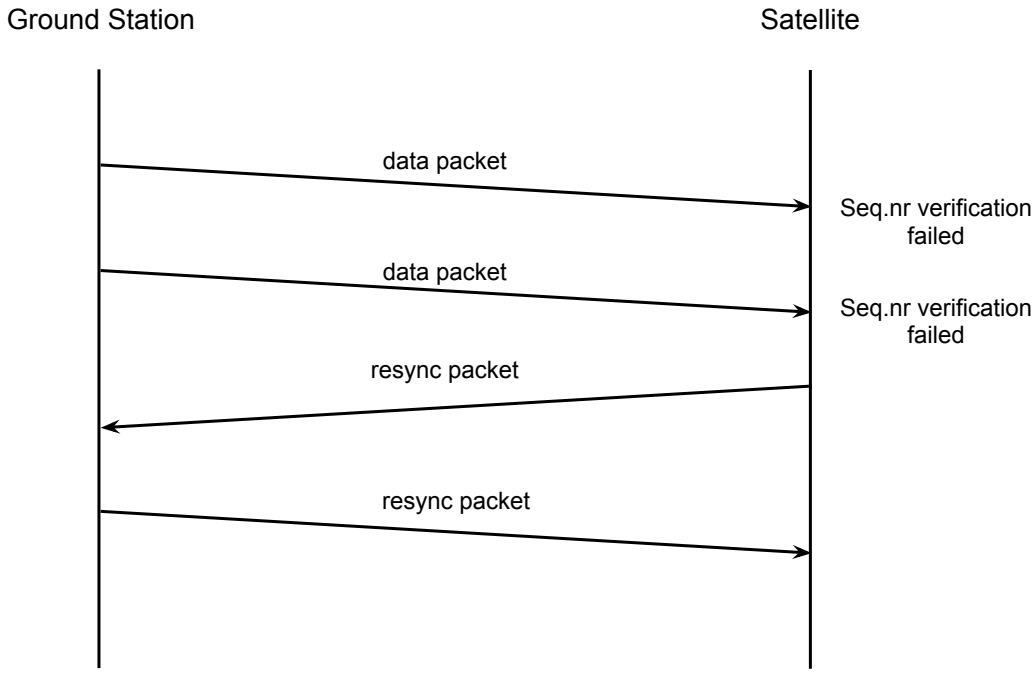


**Figure A.3:** CSP resynchronization exchange

## A.3 CSP resynchronization exchange

### A.3.1 `CSP_SEQUENCE_RESYNC_THRESHOLD`

The number of failed messages received before the resynchronization exchange is initiated is controlled with a parameter called `CSP_SEQUENCE_RESYNC_THRESHOLD`. In figure A.3 this parameter is set to 2.

### A.3.2 `CSP_SEQUENCE_THRESHOLD`

This parameter controls the maximum jump allowed in the sequence number. I.e. if the local sequence number is 12 and `CSP_SEQUENCE_THRESHOLD` is set to 3, 12, 13, and 14 are valid. The check is implemented to disallow arbitrary large jumps in the sequence number to avoid rapid exhaustion.

```
if stored_seqnr < received_seqnr + CSP_SEQUENCE_THRESHOLD
```

# Appendix B
# Our code

In this appendix, we have included code we have written for this project. A copy will also be made available via the project's `git` repository.

## B.1 csp_seqnr.h

```
1  #ifndef _CSP_HMAC_H_
2  #define _CSP_HMAC_H_
3
4  #ifdef ___cplusplus
5  extern "C" {
6  #endif
7
8  #include <stdint.h>
9
10 #include "../arch/csp_thread.h"
11
12 #define CSP_SEQUENCE_NUMBER_LENGTH 2 /* DO NOT CHANGE THIS ↩
       WITHOUT CHANGING TO SUITABLE VARIABLE TYPES */
13 #define CSP_SEQUENCE_THRESHOLD 1
14 #define CSP_SEQUENCE_RESYNC_THRESHOLD 2
15
16 /**
17  * Initialize the seqnr environment
18  */
19 void csp_seqnr_init();
20
21 /**
22  * Set/get the sequence number to a specific value
23  *
24  * @param the host for set it for
25  * @param the value to set it to
```

```
26  */
27 void csp_seqnr_set_rx(uint8_t host, uint16_t new_seqnr);
28 void csp_seqnr_set_tx(uint8_t host, uint16_t new_seqnr);
29
30 uint16_t csp_seqnr_get_rx(uint8_t host);
31 uint16_t csp_seqnr_get_tx(uint8_t host);
32
33
34 /**
35  * Increase sequence number for a specific host
36  *
37  * @param the host
38  */
39 void csp_seqnr_increase_rx(uint8_t host);
40 void csp_seqnr_increase_tx(uint8_t host);
41
42 /**
43  * Append sequence number to the packet
44  *
45  *@param packet Pointer to packet
46  *@return + on success, -1 on failure
47  */
48 int csp_seqnr_append(csp_packet_t * packet);
49
50 /**
51  * Verify that the sequence number actually is increasing.
52  *
53  *
54  * @param packet Pointer to packet
55  * @return 0 if the sequence number is within the ↩
        CSP_SEQUENCE_NUMBER_JUMP threshold. -1 if failure
56  */
57 int csp_seqnr_verify(csp_packet_t * packet);
58
59 /**
60 * If the sewuence number gets out of synch on either the groundstation (GS) or in ↩
        the satellite (S), this methow will bring it back to a consistent state.
61 * @param csp_conn_t the connection used to send the resynch-commands over.
62 * @return 0 if the opreration is successfull, != 0 otherwise.
63 */
64 csp_thread_handle_t csp_seqnr_resync_init(csp_packet_t * invalid_packet);
65
66 int csp_seqnr_resync_respond(csp_conn_t * resync_conn, csp_packet_t * ↩
        resync_packet);
67
```

```
68
69 #endif // _CSP_HMAC_H_
```

## B.2 csp_seqnr.c

```
1  #include <stdint.h>
2  #include <string.h>
3
4  /* CSP includes */
5  #include <csp/csp.h>
6  #include <csp/csp_endian.h>
7  #include "csp_seqnr.h"
8
9  #include "../arch/csp_thread.h"
10 #include "../arch/csp_semaphore.h"
11
12 #ifdef CSP_USE_SEQNR
13
14 /* Global seqnr. counter */
15 /* Initialize large enough so that we don't wrap arround in the forseable lifetime of the satellite . */
16 uint16_t csp_seqnr_rx[CSP_ID_HOST_MAX];
17 csp_mutex_t csp_seqnr_mutex_rx;
18
19 uint16_t csp_seqnr_tx[CSP_ID_HOST_MAX];
20 csp_mutex_t csp_seqnr_mutex_tx;
21
22 uint8_t csp_seqnr_errorcount[CSP_ID_HOST_MAX];
23
24 uint16_t csp_seqnr_get_rx(uint8_t host){
25     return csp_seqnr_rx[host];
26 }
27
28 uint16_t csp_seqnr_get_tx(uint8_t host){
29     return csp_seqnr_tx[host];
30 }
31
32 void csp_seqnr_set_rx(uint8_t host, uint16_t new_seqnr){
33     if(csp_mutex_lock(&csp_seqnr_mutex_rx, 1000) == CSP_MUTEX_OK){
34   csp_seqnr_rx[host] = new_seqnr;
35   csp_mutex_unlock(&csp_seqnr_mutex_rx);
36     }
37 }
38
39 void csp_seqnr_set_tx(uint8_t host, uint16_t new_seqnr){
40     if(csp_mutex_lock(&csp_seqnr_mutex_tx, 1000) == CSP_MUTEX_OK){
41         csp_seqnr_tx[host] = new_seqnr;
```

```
42          csp_mutex_unlock(&csp_seqnr_mutex_tx);
43      }
44  }
45
46  void csp_seqnr_increase_rx(uint8_t host) {
47      if(csp_mutex_lock(&csp_seqnr_mutex_rx, 1000) == CSP_MUTEX_OK){
48    csp_seqnr_rx[host] +=1;
49          csp_mutex_unlock(&csp_seqnr_mutex_rx);
50      }
51  }
52
53  void csp_seqnr_increase_tx(uint8_t host) {
54      if(csp_mutex_lock(&csp_seqnr_mutex_tx, 1000) == CSP_MUTEX_OK){
55          csp_seqnr_tx[host] +=1;
56          csp_mutex_unlock(&csp_seqnr_mutex_tx);
57      }
58  }
59
60  int csp_seqnr_append(csp_packet_t * packet){
61
62    if (packet == NULL)
63      return CSP_ERR_INVAL;
64
65    /* Make room for the sequence number (dst, src, len) */
66    uint16_t hton_seqnr = csp_hton16(csp_seqnr_get_tx(packet->id.dst));
67
68    memcpy(&packet->data[packet->length], &hton_seqnr, ↩
        CSP_SEQUENCE_NUMBER_LENGTH);
69    packet->length += CSP_SEQUENCE_NUMBER_LENGTH;
70
71    csp_seqnr_increase_tx(packet->id.dst);
72
73    return CSP_ERR_NONE;
74  }
75
76  /* Read seqnr from packet and check if > than global_counter + ↩
       CSP_SEQUENCE_THRESHOLD */
77  int csp_seqnr_verify(csp_packet_t * packet){
78
79      uint16_t sequence_number;
80      memcpy(&sequence_number, &packet->data[packet->length - ↩
        CSP_SEQUENCE_NUMBER_LENGTH], CSP_SEQUENCE_NUMBER_LENGTH);
81      sequence_number = csp_ntoh16(sequence_number);
82      printf("Verifying sequence number: PACKET = 0x%02X, COUNTER = 0x%02X \r↩
        \n", sequence_number, csp_seqnr_get_rx(packet->id.src));
```

```
83
84      // Strip seqnr
85      packet->length -= CSP_SEQUENCE_NUMBER_LENGTH;
86
87      if(sequence_number >= csp_seqnr_get_rx(packet->id.src) && ↩
         sequence_number < csp_seqnr_get_rx(packet->id.src) + ↩
         CSP_SEQUENCE_THRESHOLD){
88    /* Successful verification , reset error counter */
89    csp_seqnr_errorcount[packet->id.src] = 0;
90    csp_seqnr_increase_rx(packet->id.src);
91    return CSP_ERR_NONE;
92      }
93
94      /* Verification failed , checking to see if resync required or just increase ↩
         counter */
95      csp_seqnr_errorcount[packet->id.src] += 1;
96      if(csp_seqnr_errorcount[packet->id.src] >= ↩
         CSP_SEQUENCE_RESYNC_THRESHOLD){
97    csp_seqnr_resync_init(packet);
98    csp_seqnr_errorcount[packet->id.src] = 0;
99      }
100
101     return CSP_ERR_SEQNR;
102 }
103
104 CSP_DEFINE_TASK(resync_thread){
105
106     printf("Starting resynchronization\r\n");
107     csp_packet_t * invalid_packet = param;
108
109     // Trying to connect unathenticated to notify other party of HMAC/SEQNR ↩
         verification errors
110     csp_conn_t * resync_conn = csp_connect(CSP_PRIO_HIGH, invalid_packet->id↩
         .src , CSP_SEQNR_RESYNC, 1000, CSP_O_NONE);
111     if(resync_conn == NULL){
112         csp_log_error("Could not connect to resync port\r\n");
113   //  return CSP_ERR_RESYNC;
114   csp_thread_exit();
115     }
116
117     // Creating a packet to send
118     csp_packet_t * packet = csp_buffer_get(100);
119     if(packet == NULL){
120   csp_log_error("Could not get buffer for resync packet\r\n");
121   csp_close(resync_conn);
```

```
122   //  return CSP_ERR_RESYNC;
123   csp_thread_exit();
124     }
125
126     // Setting  local  sequence numbers as data
127     uint16_t hton_rx_seqnr = csp_hton16(csp_seqnr_get_rx(invalid_packet->id.src));
128     uint16_t hton_tx_seqnr = csp_hton16(csp_seqnr_get_tx(invalid_packet->id.src));
129     memcpy(&packet->data[0], &hton_rx_seqnr, CSP_SEQUENCE_NUMBER_LENGTH);
130     memcpy(&packet->data[CSP_SEQUENCE_NUMBER_LENGTH], &hton_tx_seqnr, CSP_SEQUENCE_NUMBER_LENGTH);
131     packet->length = 2 * CSP_SEQUENCE_NUMBER_LENGTH;
132
133     // Sending with HIGH priority
134     if(csp_send_prio(CSP_PRIO_HIGH, resync_conn, packet, 1000) != 1){
135   csp_log_error("Could not send resync packet\r\n");
136   csp_buffer_free(packet);
137   csp_close(resync_conn);
138   //  return CSP_ERR_RESYNC;
139   csp_thread_exit();
140     }
141     csp_buffer_free(packet);
142
143     // Wait for response
144     csp_packet_t * response_packet = csp_buffer_get(100);
145     if(response_packet == NULL){
146         csp_log_error("Could not get buffer for resync response packet\r\n");
147         csp_close(resync_conn);
148   //         return CSP_ERR_RESYNC;
149   csp_thread_exit();
150     }
151
152     response_packet = csp_read(resync_conn, 100000);
153     if(response_packet == NULL){
154   csp_log_error("No response to resync packet\r\n");
155   csp_buffer_free(response_packet);
156         csp_close(resync_conn);
157   //  return CSP_ERR_RESYNC;
158   csp_thread_exit();
159     }
160
161     printf("Received a RESYNC RESPONSE packet: Length = %u, RAW = 0x", response_packet->length);
```

```
162     int i;
163     for(i = 0; i < response_packet->length; i++){
164   printf("%02X", response_packet->data[i]);
165     }
166     printf("\r\n");
167 
168     uint16_t rx_response;
169     uint16_t tx_response;
170     memcpy(&rx_response, &response_packet->data[0], ↩
        CSP_SEQUENCE_NUMBER_LENGTH);
171     memcpy(&tx_response, &response_packet->data[CSP_SEQUENCE_NUMBER_LENGTH↩
        ], CSP_SEQUENCE_NUMBER_LENGTH);
172     rx_response = csp_ntoh16(rx_response);
173     tx_response = csp_ntoh16(tx_response);
174 
175     // Set local seqnr to value in response ONLY if it is smaller, to prevent ↩
        resetting by malicious mallards
176     if(rx_response >= csp_seqnr_get_rx(response_packet->id.src)){
177   csp_seqnr_set_rx(response_packet->id.src, rx_response);
178     }
179     if(tx_response >= csp_seqnr_get_tx(response_packet->id.src)){
180   csp_seqnr_set_tx(response_packet->id.src, tx_response);
181     }
182 
183     // Cleaning up
184     csp_buffer_free(response_packet);
185     csp_close(resync_conn);
186 
187     //     return CSP_ERR_NONE;
188     csp_thread_exit();
189 }
190 
191 csp_thread_handle_t csp_seqnr_resync_init(csp_packet_t * invalid_packet){
192 
193   csp_thread_handle_t resync_thread_handle;
194   csp_thread_create(resync_thread, (signed char *) "Resynch thread", 1000, ↩
       invalid_packet, 0, &resync_thread_handle);
195   return resync_thread_handle;
196 
197 }
198 
199 int csp_seqnr_resync_respond(csp_conn_t * resync_conn, csp_packet_t * ↩
       resync_packet){
200 
201     uint16_t remote_rx_sequence_number;
```

```
202     uint16_t remote_tx_sequence_number;
203     memcpy(&remote_rx_sequence_number, &resync_packet->data[0], ↩
         CSP_SEQUENCE_NUMBER_LENGTH);
204     memcpy(&remote_tx_sequence_number, &resync_packet->data[↩
         CSP_SEQUENCE_NUMBER_LENGTH], CSP_SEQUENCE_NUMBER_LENGTH);
205     remote_rx_sequence_number = csp_ntoh16(remote_rx_sequence_number);
206     remote_tx_sequence_number = csp_ntoh16(remote_tx_sequence_number);
207
208     uint16_t rx_sequence_number_to_send = (remote_rx_sequence_number > ↩
         csp_seqnr_get_tx(resync_packet->id.src)) ? remote_rx_sequence_number : ↩
         csp_seqnr_get_tx(resync_packet->id.src);
209     uint16_t tx_sequence_number_to_send = (remote_tx_sequence_number > ↩
         csp_seqnr_get_rx(resync_packet->id.src)) ? remote_tx_sequence_number : ↩
         csp_seqnr_get_rx(resync_packet->id.src);
210     csp_seqnr_set_rx(resync_packet->id.src, tx_sequence_number_to_send);
211     csp_seqnr_set_tx(resync_packet->id.src, rx_sequence_number_to_send);
212
213     printf("Preparing to send RESYNC RESPONSE: %u %u\r\n", ↩
         tx_sequence_number_to_send, rx_sequence_number_to_send);
214
215     // Creating a packet to send
216     csp_packet_t * packet = csp_buffer_get(100);
217     if(packet == NULL){
218         csp_log_error("Could not get buffer for resync response packet\r\n")↩
         ;
219         csp_close(resync_conn);
220   return CSP_ERR_RESYNC;
221     }
222
223     // Setting highest sequence number as data
224     rx_sequence_number_to_send = csp_hton16(rx_sequence_number_to_send);
225     tx_sequence_number_to_send = csp_hton16(tx_sequence_number_to_send);
226     memcpy(&packet->data[0], &rx_sequence_number_to_send, ↩
         CSP_SEQUENCE_NUMBER_LENGTH);
227     memcpy(&packet->data[CSP_SEQUENCE_NUMBER_LENGTH], &↩
         tx_sequence_number_to_send, CSP_SEQUENCE_NUMBER_LENGTH);
228     packet->length = 2 * CSP_SEQUENCE_NUMBER_LENGTH;
229
230     printf("Sending a RESYNC RESPONSE packet: Length = %u, RAW = 0x", packet↩
         ->length);
231     int i;
232     for(i = 0; i < packet->length; i++){
233         printf("%02X", packet->data[i]);
234     }
235     printf("\r\n");
```

```
236
237     if(csp_send_prio(CSP_PRIO_HIGH, resync_conn, packet, 1000) != 1){
238         csp_log_error("Could not send resync response packet\r\n");
239   csp_buffer_free(packet);
240         csp_close(resync_conn);
241   return CSP_ERR_RESYNC;
242     }
243
244     // Cleaning up
245     csp_buffer_free(packet);
246     csp_close(resync_conn);
247
248     return CSP_ERR_NONE;
249 }
250
251 CSP_DEFINE_TASK(resync_listen_task){
252     // Need to listen  with no options on CSP_SEQNR_RESYNC incase other wants↩
        to tell me we're out of sync
253     csp_socket_t *sock = csp_socket(CSP_SO_NONE);
254     csp_bind(sock, CSP_SEQNR_RESYNC);
255     csp_listen(sock, 10);
256
257     csp_conn_t * conn;
258     csp_packet_t * packet;
259
260     while(true){
261         if((conn = csp_accept(sock, 10000)) != NULL){
262             while ((packet = csp_read(conn, 100)) != NULL) {
263                 switch (csp_conn_dport(conn)) {
264                 case CSP_SEQNR_RESYNC:
265         csp_service_handler(conn, packet);
266                     break;
267                 default:
268         csp_buffer_free(packet);
269                     break;
270                 }
271             }
272             csp_close(conn);
273         }
274     }
275
276     return CSP_TASK_RETURN;
277 }
278
279 void csp_seqnr_init(){
```

```
280     csp_mutex_create(&csp_seqnr_mutex_rx);
281     csp_mutex_create(&csp_seqnr_mutex_tx);
282
283     printf("Starting resync_listen_task\r\n");
284     csp_thread_handle_t resync_handle;
285     csp_thread_create(resync_listen_task, (signed char *) "Resync", 1000, NULL, ↩
         0, &resync_handle);
286 }
287
288 #endif // CSP_USE_SEQNR
```

## B.3 csp__io__(excerpt).c

```
1   if (idout.flags & CSP_SEQNR) {
2 #ifdef CSP_USE_SEQNR
3   /*Append the gloal seqnr. to the packet */
4     if(csp_seqnr_append(packet) != 0){
5       /* SEQNR append failed */
6       csp_log_warn("SEQNR append failed !\r\n");
7       goto tx_err;
8     }
9 #else
10    csp_log_warn("Attempt to send packet with SEQNR, but csp was compiled ↩
       without SEQNR support. Discarding packet\r\n");
11    goto tx_err;
12 #endif
13  }
```

## B.4 csp_route_(excerpt).c

```
1   /*SEQNR enabled packet */
2   if(packet->id.flags & CSP_SEQNR) {
3
4 #ifdef CSP_USE_SEQNR
5
6         if(csp_seqnr_verify(packet) != 0){
7               /*Fail*/
8               csp_log_error("SEQNR verification failed \r\n");
9               interface->autherr++;
10              return CSP_ERR_SEQNR;
11        }
12  } else if (security_opts & CSP_SO_SEQNR) {
13        csp_log_warn("SEQNR enabled but received packet without SEQNR. ↩
      Discarding packet \r\n");
14        interface->autherr++;
15        return CSP_ERR_SEQNR;
16
17 #else
18
19    csp_log_error("Received packet with SEQNR, but CSP was compiled without↩
        SEQNR support. Discarding packet \r\n");
20    interface->autherr++;
21    return CSP_ERR_NOTSUP;
22
23 #endif
24
25  }
```

## B.5 server.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4
5  #include <csp/csp.h>
6  #include <csp/drivers/usart.h>
7  #include <csp/interfaces/csp_if_kiss.h>
8
9  #include "../../src/arch/csp_thread.h"
10
11 #define MY_ADDRESS  1     // Address of local CSP node
12 #define REMOTE_ADDRESS 2
13 #define MY_PORT 10       // Port to send test traffic to
14
15 CSP_DEFINE_TASK(server) {
16
17   /* HMAC SETUP */
18   csp_hmac_set_key("Testkey", 7);
19   /*HMAC SETUP */
20
21   csp_socket_t *sock = csp_socket(CSP_SO_RDPREQ | CSP_SO_SEQNR | ↩
       CSP_SO_HMACREQ);
22
23   csp_bind(sock, MY_PORT);
24   csp_listen(sock, 200);
25
26   printf("Listening \r\n");
27
28   csp_conn_t *conn;
29   csp_packet_t *packet;
30
31   while (1) {
32
33     if ((conn = csp_accept(sock, 10000)) == NULL)
34       continue;
35
36     while ((packet = csp_read(conn, 1000)) != NULL) {
37       switch (csp_conn_dport(conn)) {
38       case MY_PORT:
39         /* Process packet here */
40           printf("Received a packet: Length = %u, DATA = %.*s \r\n", packet↩
      ->length, packet->length, packet->data);
```

```
41          csp_buffer_free(packet);
42          break;
43        default:
44          csp_service_handler(conn, packet);
45          printf("Service request received (%s) \r\n");
46          break;
47        }
48      }
49      csp_close(conn);
50    }
51    return;
52  }
53
54  int main(int argc, char * argv[]) {
55
56    printf("Initialising CSP\r\n");
57    csp_buffer_init(100, 3100);
58    csp_init(MY_ADDRESS);
59
60    struct usart_conf conf;
61    conf.device = "/dev/ttyS0";
62    printf("Using '%s' as the serial port\r\n", conf.device);
63          conf.baudrate = 1200;
64          usart_init(&conf);
65          csp_kiss_init(usart_putstr, usart_insert);
66          usart_set_callback(csp_kiss_rx);
67
68          csp_route_set(REMOTE_ADDRESS, &csp_if_kiss, CSP_NODE_MAC);
69    csp_route_start_task(500, 1);
70    csp_seqnr_init(0);
71
72    if ((argc > 1) && (strcmp(argv[1], "-v") == 0)) {
73      printf("Debug enabed\r\n");
74      csp_debug_toggle_level(4);
75
76      printf("Conn table\r\n");
77      csp_conn_print_table();
78
79      printf("Route table\r\n");
80      csp_route_print_table();
81
82      printf("Interfaces\r\n");
83      csp_route_print_interfaces();
84    }
85
```

```
86    printf("Starting server task\r\n");
87          csp_thread_handle_t server_handle;
88          csp_thread_create(server, (signed char *) "Server", 1000, NULL, 0, &↩
        server_handle);
89
90    while(1) {
91      csp_sleep_ms(100000);
92    }
93
94    return 0;
95  }
```

## B.6 client.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <unistd.h>
4
5  #include <csp/csp.h>
6  #include <csp/drivers/usart.h>
7  #include <csp/interfaces/csp_if_kiss.h>
8
9  #include "../../src/arch/csp_thread.h"
10
11 #define MY_ADDRESS  2     // Address of local CSP node
12 #define REMOTE_ADDRESS  1
13 #define MY_PORT   10      // Port to send test traffic to
14
15 CSP_DEFINE_TASK(send_hello_world_task) {
16
17     csp_packet_t * packet;
18     csp_conn_t * conn;
19     csp_hmac_set_key("Testkey", 7);
20
21     while (true) {
22
23   /* Get packet buffer for data */
24   packet = csp_buffer_get(300);
25   if (packet == NULL) {
26       printf("Failed to get buffer element\n");
27       return;
28   }
29
30   /* Connect to host HOST, port PORT with regular UDP-like protocol and 1000 ↩
       ms timeout */
31   conn = csp_connect(CSP_PRIO_NORM, REMOTE_ADDRESS, MY_PORT, 1000, CSP_O_SEQNR ↩
       | CSP_O_HMAC | CSP_O_RDP);
32   if (conn == NULL) {
33       printf("Connection failed\n");
34       csp_buffer_free(packet);
35       continue;
36   }
37
38   for(int k = 0; k < 4; k++){
39     /* Copy dummy data to packet */
```

```
40      //      char *msg = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. ↩
        Proin at ipsum lorem, at malesuada urna. Aliquam et laoreet purus. Maecenas ↩
        pellentesque, arcu sit amet posuere tristique, quam nisi ornare massa, vel ↩
        tempus magna enim quis volutpat.";
41        char *msg = "Hello world";
42      strcpy((char *) packet->data, msg);
43      packet->length = strlen(msg);
44
45      /* Send packet */
46      printf("Sending a packet: Length = %u, DATA = %s \r\n", packet->length, ↩
        (char *) packet->data);
47
48      if (!csp_send(conn, packet, 1000)) {
49        /* Send failed */
50        printf("Send failed\n");
51        csp_buffer_free(packet);
52      }
53      csp_sleep_ms(300);
54
55    }
56    /* Close connection */
57    csp_close(conn);
58
59    printf("Sleeping 6000 ms \r\n");
60    csp_sleep_ms(6000);
61    printf("Done sleeping\r\n");
62      }
63
64      return CSP_TASK_RETURN;
65  }
66
67  int main(int argc, char * argv[]) {
68
69    printf("Initialising CSP\r\n");
70    csp_buffer_init(100, 1000);
71    csp_init(MY_ADDRESS);
72
73    struct usart_conf conf;
74    conf.device = "/dev/ttyS0";
75    conf.baudrate = 500000;
76    printf("Using '%s' as the serial port \r\n", conf.device);
77    usart_init(&conf);
78    csp_kiss_init(usart_putstr, usart_insert);
79    usart_set_callback(csp_kiss_rx);
80
```

```
81    csp_route_set(REMOTE_ADDRESS, &csp_if_kiss, CSP_NODE_MAC);
82    csp_route_start_task(500, 1);
83    csp_seqnr_init(0);
84
85    if ((argc > 1) && (strcmp(argv[1], "-v") == 0)) {
86      printf("Debug enabed\r\n");
87      csp_debug_toggle_level(4);
88
89      printf("Conn table\r\n");
90      csp_conn_print_table();
91
92      printf("Route table\r\n");
93      csp_route_print_table();
94
95      printf("Interfaces\r\n");
96      csp_route_print_interfaces();
97    }
98
99    printf("Starting send_hello_world_task\r\n");
100         csp_thread_handle_t send_hello_world_handle;
101         csp_thread_create(send_hello_world_task, (signed char *) "Send hello ↩
        world", 1000, NULL, 0, &send_hello_world_handle);
102
103   while(1) {
104     csp_sleep_ms(100000);
105   }
106
107   return 0;
108 }
```