Integration and verification of a keyed-hash message authentication scheme based on broadcast timestamps for NUTS

Marius Muench

Master of Telematics - Communication Networks and Networked Services (2
Submission date: July 2014
Supervisor: Stig Frode Mjølsnes, ITEM
Co-supervisor: Roger Birkeland, IET

Norwegian University of Science and Technology
Department of Telematics

Title: "Integration and verification of a keyed-hash message authentication scheme based on broadcast timestamps for NUTS"

Student: Marius Münch

Problem description:

NUTS is NTNU's student satellite project aiming to launch a double CubeSat into low earth orbit in 2015.

The main objective is to take pictures of a phenomen originating in the earth's atmosphere. These pictures need to be relayed back to earth. The satellite itself will be equipped with VHF and UHF radios and a self-developed protocol will be used for the communication between the ground segment and the satellite. A radio uplink authentication mechanism should be used to prevent unauthorized commands and hijacking of the satellite.

Previous work has shown that authentication based on keyed-hash message authentication codes in combination with broadcast timestamps is appropriate for the NUTS project. The design proposal and a proof of concept implementation have been carried out in previous project work.

The task of the present work is to integrate the proposal with the satellite system using customized software. Close coordination with the development of the satellite communication system must be observed. A test and validation procedure for the authentication functionality and performance in the launched satellite should be worked out. Moreover, the protocol security properties should be analyzed and assured by appropriate formal methods.
If time allows, the candidate may propose a solution for the key management of the authentication scheme.

Responsible professor: Stig Frode Mjølsnes, ITEM

Supervisor: Roger Birkeland, IET

Abstract

The NTNU Test Satellite (NUTS) is a small satellite developed by students of the Norwegian University of Science and Technology (NTNU). The satellite follows the CubeSat specification and the development started in 2010, while a launch is planned for 2015. One goal of the NUTS project is to build the satellite entirely from scratch in terms of both hard- and software.
Another objective is to provide an effective security mechanism for the operational uplink. The traditional approach of using encryption on the satellite links in order to prevent a takeover is not realizable for NUTS and a variety of other CubeSat programs. The reason for this is that the satellite is operated via amateur radio frequencies which regulations are not allowing encrypted traffic. Thus, a demand for alternative solutions providing uplink security does exist.
Previous work inside the NUTS project has pointed out, that an authentication scheme based on keyed-hash message authentication codes in combination with timestamps embodies an alternative to an encrypted uplink and a specific scheme has been proposed recently.
This thesis specifies the proposed scheme in detail in order to establish its correctness to a large extend with methods of formal verification. Additionally, the scheme is implemented on hardware having similar computational restrictions compared to the NUTS satellite. This implementation is carried out in a way which guarantees an easy integration into the finalized satellite software. Accompanying this, a conceptual integration to the hard- and software of NUTS is provided.
The implemented authentication scheme is selected as security solution for the NUTS satellite in space. Therefore, an in-space evaluation of the scheme can be accomplished as soon as the satellite is launched. In preparation for this evaluation, a test suite is developed and presented in this thesis in order to verify the space suitability of the scheme by experimental results later on.
Furthermore, the existence of minor flaws in the authentication scheme could be shown and their impacts are discussed in order to demonstrate their negligibility.
Summarized, this thesis demonstrates that an authentication scheme based on HMACs and broadcast timestamps provides is reasonable secure for the operational uplink of NUTS and elaborates a specific implementation of the scheme which is ready for an integration to the satellites' software.

Preface

This master's thesis is a continuation of my project work "Development of a Security Module for the Uplink of the NUTS Student Satellite" which I composed in the third semester of my studies in the Norwegian University of Science and Technology.
I spent one year inside the NUTS project and had the possibility to combine two of my passions: information security and space technology. When I arrived two years ago in Trondheim, I would never have dreamed about this possibility.

Therefore, I want to thank the NUTS project staff, especially Roger Birkeland and Amund Gjersvik for providing a platform for students to work with satellite technology. They have guided me in the past year and helped to establish a profound knowledge basis for space technology.

Furthermore, I would like to express my special appreciation to my supervising professor Stig Frode Mjølsnes. He has been on hand with help and advice to all times. He introduced me to the concepts of cryptographic protocols and their verification and opened my eyes for an interesting branch of research I was not familiar with. Additionally, he helped me to shape my academic future which is going far beyond my expectations of the outcome of this thesis.

Besides that, I want to state my gratitude for professor Colin Boyd who supervised the preceding project work and therefore helped to lay the foundation for this master's thesis.
Last but not least, I want to extend my sincere thanks to my family and friends who supported me throughout the two years of studies at the Norwegian University of Science and Technology. Without their help I would not be where I am today.

Contents

List of Figures vii

List of Tables ix

List of Algorithms xi

List of Symbols xiii

List of Acronyms xv

Glossary xvii

1 Introduction 1

1.1 Structure of this Thesis . . . 2

1.2 Terminology . . . 2

2 Background 3

2.1 The NTNU Test Satellite Project . . . 3

2.2 Uplink Security . . . 4

2.3 Previous Work . . . 5

3 Authentication Scheme 7

3.1 Goal . . . 7

3.2 Environment Assumptions . . . 8

3.2.1 Clock Reliability . . . 8

3.2.2 Beacon Reliability . . . 9

3.2.3 Satellite as Trusted Source . . . 9

3.2.4 Correctness of Transmission . . . 10

3.3 Keyed-Hash Message Authentication Code . . . 10

3.4 Broadcast Timestamps . . . 10

3.4.1 Clock Synchronization . . . 11

3.4.2 Format . . . 11

3.4.3 Tolerance . . . 12

3.5 The Authentication Protocol 13
3.5.1 Informal Description 14
3.5.2 Abstractions 14
3.5.3 Basic NUTS authentication Protocol 15
3.5.4 Extended NUTS authentication Protocol 15

4 Formal Verification of the NUTS Authentication Protocol 17
4.1 Scyther 17
4.1.1 Features 18
4.1.2 Input Language 19
4.1.3 Verifiable Security Properties 19
4.2 Modeling the protocol 21
4.2.1 Trust Assumption 21
4.2.2 Timestamps 22
4.2.3 Resulting Scyther Models 23
4.3 Verification Results 24
4.3.1 Analysis of Characterization 25
4.3.2 Analysis of Reported Attacks 25
4.3.3 Analysis of the Protocols' Flaws 27

5 Implementation 31
5.1 Development Environment 31
5.1.1 Hardware 31
5.1.2 Software 33
5.2 Communication Protocol 35
5.2.1 Hash Function for HMAC Construction 35
5.2.2 Protocol Header 35
5.2.3 Flexibility 36
5.3 Satellite 37
5.3.1 Operation Modes 37
5.3.2 Validation 38
5.3.3 Response 38
5.3.4 Interface 40
5.4 Base Station 41
5.4.1 Creation of the NUTS Authentication Protocol Header 41
5.4.2 Obtaining of HMACS 43

6 Integration 45
6.1 Hardware Design 45
6.2 Software Design 46
6.3 Protocol Stack 47

7 Test Suite **49**
7.1 Laboratory Test Environment 49
7.2 Functional Testing 51
7.3 Non-Functional Testing 55

8 Discussion **59**
8.1 Effectivity of Replay Protection 59
8.2 Results of the formal verification 60
8.3 Scope of the Integration 61

9 Future Work **63**
9.1 Integration to the Final System 63
9.2 Radiation Hardness Testing 63
9.3 Evaluation of Performance in Space 64
9.4 Key Management 64

10 Conclusions **65**

References **67**

Appendices

A SDPL Files **I**

B Satellite Code **V**

C Base Station **IX**

D Testing Code **XI**

List of Figures

2.1 Schematic view of the communication structure, first appeared in [Mü14] 4

3.1 Authentication goal in regard to the communication channels 8
3.2 An informal description of the authentication protocol 14

4.1 Example attack on non-injective synchronization for S with B 28
4.2 Example attack on non-injective synchronization for B with S 29

5.1 The development environment 32
5.2 Hierarchy of operation modes 38
5.3 Conceptual base station program flow for the extended NUTS authentication protocol 43

6.1 Protection circuit for the RTC (figure provided by the NUTS hardware group) 46
6.2 Overview of the NUTS Radio Software Design (figure provided by the NUTS software group) 47
6.3 NUTS protocol stack 47

7.1 Schematic overview over the test environment 50
7.2 Test results for the validation of correct fingerprint construction 54
7.3 Test results for evaluating the performance of the implementation . . . 56
7.4 Test results for the validation of the effectivity of the replay protection . 57

List of Tables

4.1 Number of found trace patterns for the NUTS authentication protocol . 25
4.2 Found attacks for the basic NUTS authentication protocol 26
4.3 Found attacks for the extended NUTS authentication protocol 26
4.4 Found attacks for the extended NUTS authentication protocol with precise timestamps 30

5.1 The communication protocol header 36
5.2 Expected parameter for the interface function 41

List of Algorithms

4.1 A simple secrecy protocol for Scyther 20
4.2 Scyther model for the basic NUTS authentication protocol 24
5.1 Example of a FreeROTS task 34
5.2 Definition of Macros for the Protocol Header 36
5.3 Function for validating received messages for the satellite 39
5.4 Function for expanding the response 40
5.5 Perlscript for constructing uplink packets 42

List of Symbols

Symbol	**Definition**	**Unit**
τ	Satellite lifetime	days
ε_{sr}	Bit error rate caused by space radiation	error(s)/day
c	Speed of light	m/s
d	Physical link length	m
D_{prop}	Propagation delay	s
D_{tot}	Total delay	s
D_{trans}	Transmission delay	s
$D_{M_{Up}}$	Total delay for the message uplink	s
$D_{T_{Down}}$	Total delay for the timestamp downlink	s
L	Packet length	bit(s)
M	Morse code speed	words/minute
N_{bits}	Number of bits	-
$N_{letters}$	Number of letters	-
s	Signal propagation speed	m/s
R	Bandwidth	bit(s)/s
T	Tolerance	s
W	Number of letters per word	1/words

List of Acronyms

ADCS	Attitude Determination and Control System.
AFSK	Audio Frequency Shift Keying.
CSP	CubeSat Space Protocol.
EOF	END OF FILE, a non-printable character.
EPS	Electronic Power System.
HMAC	Keyed-Hash Message Authentication Codes.
I/O	Input/Output.
IDE	Integrated Development Environment.
I^2C	Inter-Integrated Circuit.
ITU	International Telecommunication Union.
JTAG	Joint Test Action Group.
MCU	Microcontroller Unit.
NAP	NUTS Authentication Protocol.
NTNU	Norwegian University of Science and Technology.
NUTS	NTNU Test Satellite.
OBC	On-Board Computer.

RTC	Real-Time Clock.
RTTY	Radioteletype.
SDPL	Security Protocol Description Language.
SRAM	Static Random Access Memory.
UHF	Ultra High Frequency.
USART	Universal Asynchronous Receiver/Transmitter.
USB	Universal Data Bus.
VHF	Very High Frequency.

Glossary

Beacon	Transmitter continously broadcasting a trackable signal.
Broadcast Channel	A communication channel where an ingoing message is sent to all participants of the communication.
	.
CubeSat	A nano satellite with standardized dimensions.
Digest	Output of a hash function.
Identity	A unique discription for an entity in a cryptographic scheme.
Nonce	A number used only once in the lifetime of a system.
Timestamp	Encoded temporal information in order to associate an event to a specific time.
Transceiver	A device providing both transmitting and receiving functionalities.

Chapter 1
Introduction

> "Experience without theory is blind, but theory without experience is mere intellectual play." - Immanuel Kant

This masters' thesis is a part of continuous research in the NTNU Test Satellite (NUTS) project. Over the last three years students have been working on possibilities for a secured communication with the CubeSat of the Norwegian University of Science and Technology (NTNU). The reason for these ambitions is not far to seek; it is desired that the satellite is protected against malicious users.
By now, several proposals for a security solution have been made, each of them with different assets and drawbacks. The most recent proposal suggests the usage of an authentication scheme based on Keyed-Hash Message Authentication Codes (HMAC) and broadcast timestamps. It has been selected to be implemented on the NUTS satellite. This is the starting point for this thesis.
While previous work focused on conceptual implementations and testing of different schemes, this thesis shall provide a solid theoretical justification for the usage of the proposed scheme. In order to prevent that this gets stuck in intellectual play, an implementation for the scheme shall be provided and integrated to the satellite. Since the scheme will be used on the satellite after its launch, special experiences for in space operation can be gathered. For this reason, this thesis aims also to establish a test suite for the evaluation of the scheme in space.
In order to fulfill the demands on the thesis, the following tasks have to be fulfilled:

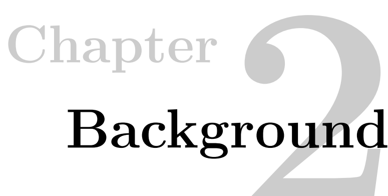
1. Elaborate the authentication scheme in detail
2. Verify the scheme with methods of formal verification
3. Provide a working implementation
4. Integrate the implementation to satellite and base station
5. Establish a test suite for the implementation

1.1 Structure of this Thesis

This thesis is structured in nine further chapters besides this introduction. The second chapter shall provide the background and context for this thesis. The third chapter describes the authentication scheme which will be verified with formal methods in section four. The fifth section is providing information about a specific implementation of the authentication scheme for the NUTS satellite. This implementation is tried to be integrated to the actual satellite in chapter six and tested with the test suite elaborated in chapter seven. The eighth chapter discusses the results established in the thesis while the ninth chapter shows how the research of this thesis can be continued. The thesis is closed with concluding remarks provided in the last chapter.

1.2 Terminology

In order to understand the theoretical part of this thesis, knowledge over the meaning of specific terms is required. Therefore, a small terminology is introduced in this section for establishing a common knowledge basis.
A *cryptographic scheme* is a system of one or more computers utilizing cryptographic elements in order to reach one or more *security goals*. Such *goals* are security related attributes of data or *entities* in the system, such as secrecy or authenticity. Achieved *security goals* are denoted as the *security properties* of the scheme. Thus, a *cryptographic scheme* can *claim* that it establishes certain *security properties*.
Cryprographic protocols or *security protocols* are crucial for schemes with more than one *entity* or *party*. They define in which order and under which circumstances messages are exchanged between communicating *parties* in order to reach the *security goals*. An *execution* or *run* of the protocol is one specific instance of the message exchange according to the protcol's rules. Here, the communicating *parties* or *entities* are representing specific *roles*, for instance the *initiator* or *responder*.
The messages in a *cryptographic protcol* are exchanged over *communication channels*. The *channels* are normally assigned specific *channel properties* according to their nature. One *channel property* is the directionality of sent messages which could be one-to-one or broadcast for instance.
Cryptographic schemes are normally assuming the presence of an *adversary* which is a generalized *attacker* who might have control of one or more *entities* participating in the scheme.



Chapter 2 Background

The purpose of this chapter is to enlighten the context of the thesis. First the project for which the authentication scheme is established will be described. Afterwards, a brief summary as to why the scheme is needed is provided in order to conclude with the previous work leading to the scheme presented in this thesis.

2.1 The NTNU Test Satellite Project

The NTNU Test Satellite (NUTS) project aims to built CubeSat entirely from scratch. It was started in 2010 and provides a plattform for the students of the NTNU to work with satellite technology. The launch of the satellite is planned for 2015. It is designed according to the double CubeSat specification and has therefore dimensions of 10 cm x 10 cm x 20 cm.
The planned payload for the satellite is a camera designated to take pictures of the earth. By the time this thesis is authored, most parts of the satellites hardware are already finalized in design and available to use. Thus, the specific subsystems besides the payload are well defined: the On-Board Computer (OBC), Attitude Determination and Control System (ADCS), Electronic Power System (EPS), Real-Time Clock (RTC) and the radio.
The radio subsystem consists of two transceivers, a beacon and a Microcontroller Unit (MCU) to coordinate incoming and outgoing signals. Each transceiver is connected to a different antenna in order to allow communication in both the UHF and VHF spectrum. More specifically, the communication takes place in the 2-meter and 70-centimeter amateur radio bands with a frequency of ca. 145 MHz and 437 MHz, respectively.
One of the transceivers is only designated for downlink controlled by the OBC in its normal mode of operation. The beacon is designed as fail-safe component and shall continuously broadcast status information of the satellite via Morse Code. Figure 2.1 illustrates the NUTS system with focus on the radio subsystem. Between base station and satellite a custom made link layer protocol, optimized for the needs of

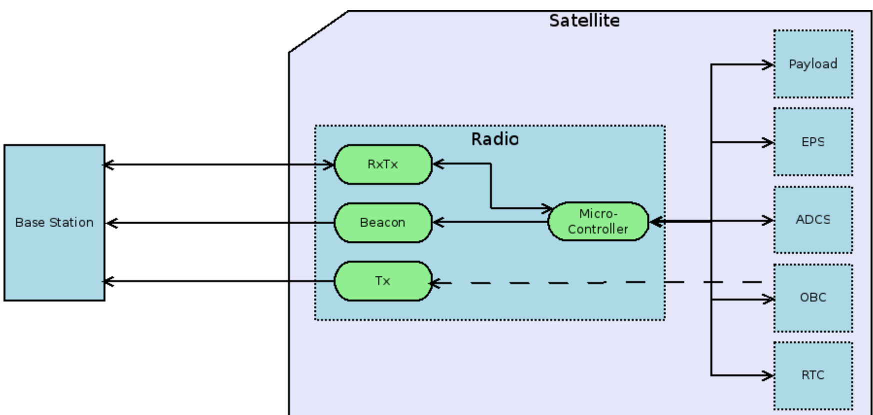


Figure 2.1: Schematic view of the communication structure, first appeared in [Mü14]

NUTS, will be used. The development of this protocol is ongoing and carried out by the NUTS communication group. For addressing the particular components of the satellite, the CubeSat Space Protocol (CSP) is used. This protocol is developed and maintained by GomSpace, a company founded by students of the University of Aalborg [Gom12]. On top of this, a specific developed application protocol will contain the specific commands for the components. Both the integration of the CSP and the development of the NUTS application layer protocol is carried out by the NUTS software group.
The authentication scheme presented in this thesis has been developed for the NUTS project and relies therefore on the existence of the described components.

2.2 Uplink Security

The demand of a secured uplink for satellite communication is reasonable when it comes to the operational uplink. Since the signals are transmitted and received over an open medium, an attacker can easily eavesdrop the communication or introduce signals on his own. Therefore it is desired that the operational uplink of a satellite is secured in order to prevent a takeover.
Traditionally, this is achieved by encrypting the satellite uplink which is the *de facto* standard for existing satellite applications [Sop12]. However, establishing encryption for an CubeSat uplink is in many cases not possible due to legal issues. It is common praxis in the CubeSat community to utilize amateur radio frequencies for satellite communication due to the openness of this frequencies. Therefore, it is not needed to purchase special frequency usage rights. However, common regulations have to be satisfied when communicating on the amateur radio frequency. The most

important regulation in regard to uplink security is article 25.2A of the International Telecommunication Union (ITU) radio regulations [ITU12] which states that:

> "Transmissions (...) shall not be encoded for the purpose of obscuring their meaning, except for control signals exchanged between earth command stations and space stations in the amateur-satellite service."[ITU12, p. 295]

Hence, as long as CubeSats are not part of the amateur-satellite service, encryption of satellite links is not allowed when using amateur radio frequencies.
As a consequence, the CubeSat community has to solve the challenge of uplink security without encryption. This thesis aims to contribute to the solution of this challenge by presenting an scheme for authentication in a space environment.

2.3 Previous Work

The first examination of the demand fpr a secured uplink inside the NUTS project dates back to 2011. The introductory research within the project focused on uplink encryption and key exchange mechanisms [Vis11].
Later on it was shown by Prasai that encryption is not mandatory for NUTS in order to establish a secured uplink [Pra12]. He proposed an authentication scheme based on HMAC functionalities of the CSP. At the same time, it has been shown that an authentication scheme only based on HMAC is vulnerable to replay attacks. Therefore, he proposed to expand the CSP with unique sequence numbers in messages from the base station to the satellite in order to guarantee freshness of every message. Subsequently, this proposal has been improved in order to deal with synchronization problems [BF12]. In this work, one additional sequence number and an extended resynchronization protocol were introduced, both integrated into the CSP.
Recently, alternative elements for guaranteeing the freshness of messages were discussed in a separate report by the author of this thesis [Mü14]. It was shown that the usage of sequence numbers causes unnecessary complexity for the system. Instead of sequence numbers, the usage of broadcast timestamps sent by the satellite has been suggested. Building upon this, an authentication procedure for the satellite which is independent from other protocols could be established. This authentication procedure is the starting point for this thesis.

Chapter 3
Authentication Scheme

The main purpose of the thesis is to verify, implement and integrate the authentication scheme for the NUTS satellite. The presented scheme was elaborated in the preceding report and particular design decisions can be reviewed in [Mü14].
The purpose of this chapter is to provide an extensive exposition of the scheme as basis for the rest of the thesis. Therefore, the goal of the scheme is specified and the special assumptions over the environment are examinated. The scheme is based on two different components, namely HMAC and broadcast timestamps. Thus, background knowledge over both components is provided before the authentication protocol itself is presented.

3.1 Goal

The main goal of the authentication scheme is to prevent unauthorized access to the satellite. More specifically, the usability of the operational uplink shall be restricted to authenticated base stations. Therefore, the main goal of the authentication scheme is to authenticate any legitimate base station to the satellite in order to reject messages and commands from unauthorized ground segments.
In order to provide further clearance of the goal in regard to the different communication channels utilized by the NUTS satellite, figure 3.1 illustrates the desired behavior of the channels. For the NUTS project, the common data uplink can be put on a level with the operational uplink since no services for other users than the NUTS project are planned. Since data downlink is only triggered after a corresponding command, only the legitimate NUTS base station B shall be able to initiate a communication session with the satellite.
On the contrary, the satellites beacon forms a broadcast channel since he is designed to be receivable by a large number of base stations B' in order to allow satellite tracking.

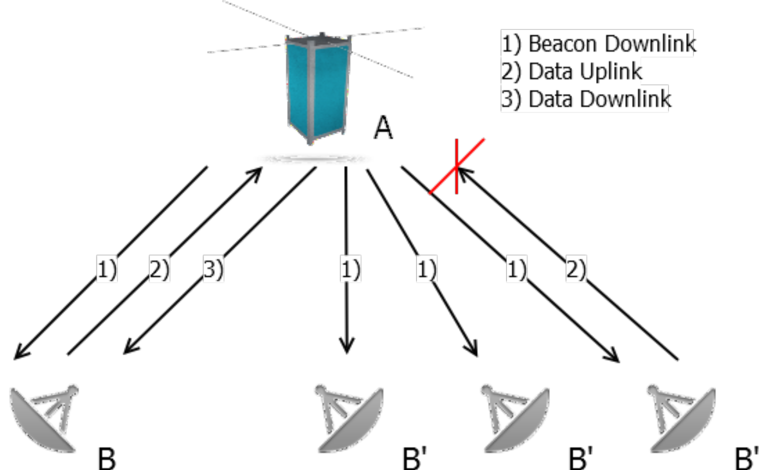


Figure 3.1: Authentication goal in regard to the communication channels

3.2 Environment Assumptions

In order to guarantee the correctness of the authentication scheme, several assumptions over the environment have to be made. In particular, we assume reliability of the clocks in the system and the beacon of the satellite. Additionally, we presume the satellite as a trusted source and require correctness of sent and received messages. In this section, the needs and reasons for these assumptions are presented.

3.2.1 Clock Reliability

The authentication scheme will utilize timestamps as fresh element for establishing authentication. Therefore, the reliability of the clocks has to be guaranteed. This is especially challenging in a space environment since the satellite satellite's on-board RTC is exposed to space radiation effects. Space radiation can cause single bit flips and change the saved time of the on-board RTC.
According to the data sheet of the DS3231 RTC [Max13], the used clock on the satellite, the time is kept in seven 8-bit registers which results into an amount of $N_{bits} = 56$ exposed to space radiation. Assuming a satellite lifetime τ of two years and an bit error rate ε_{sr} of 10^{-5} errors per day caused by space radiation [NAS96] the expected amount of errors in the RTC during the lifetime of the satellite $E(RTC)_{Lifetime}$ can be calculated as shown in equation 3.1.

$$E(RTC)_{Lifetime} = \tau \cdot N_{bits} \cdot \varepsilon_{sr} = 0.409 \, \text{errors} \tag{3.1}$$

Thus, a corresponding error could occur with a probability of 40% during the satellite's lifetime, which is rather high. Thus, we assume the presence of a watchdog for the RTC which detects and corrects errors based on space radiation in order to establish clock reliability for the satellite.
The base station resynchronizes its clock according to the satellite time and hence does not require additional mechanisms to ensure reliability.

3.2.2 Beacon Reliability

The beacon has a substantial role in the authentication scheme. It continuously broadcasts the time which is required for the scheme. Therefore, it has to be ensured that it can not fail and will be available at all times. Inside the NUTS project, this reliability was defined from the beginning since the beacon is designed to transmit signs of life of the satellite even if all other subsystems fail [EB06].
Thus, the reliability of the beacon is achieved in two manners: On one hand, the beacon hardware and software is designed by the NUTS communication group as fail-safe component. On the other hand, one of the radios can be utilized to transmit the beacon signal in the case that unexpected failures occur in the primary beacon.

3.2.3 Satellite as Trusted Source

The authentication scheme has only to authenticate the base station to the satellite because we assume that messages from the satellite are trusted. This means that all messages, which are apparently sent by the satellite, are indeed created and sent by the satellite. Therefore, an adversary can not induce messages in the network which are claiming to be originated by the satellite.
This assumption is based on a physical phenomena called Doppler shift. It describes the change of a signals frequency which can be observed by a receiver if the sender is moving relatively to it while sending. This is caused by the changing signal run time resulting from the movement. Signals from satellites in a low earth orbit are heavily influenced by Doppler shift since these satellites are moving relatively fast to maintain there orbit. For instance, a satellite with an altitude of 500 km over the earth surface moves with a velocity of approximately 7.4 kilometer per second [AADH98].
To correct the doppler shift, the base station filters an incoming signal in order to obtain the received data. Signals with a different characteristic of the Doppler shift would result into unreadable data after filtering.
Additionally, the base station's antenna is directed towards the satellite. Therefore, it is rather hard to construct signals on earth which would be received by the base station.
Thus, the trust assumption relies on the non-reproducibility of a signal with low earth orbit characteristics for the adversary.

However, the trust assumption would break if the adversary have the resources to develop and launch a satellite into an identical low earth orbit. Since the costs for this are high, especially compared with the benefits of compromising the NUTS authentication scheme, this is not considered as a threat for the NUTS project.

3.2.4 Correctness of Transmission

Normally, space links are error-prone because of, inter alia, limited transmit power on the satellite side, radiation effects on on-board components and signal losses on the long communication path. This which would heavily influence the integrity of sent and received data.
At the current state, a specialized link layer protocol is developed by the NUTS communication group. It shall provide error detection and correction in order to deal with transmission errors. Therefore, we assume that transmission errors are filtered out by appropriate mechanisms on the link layer.

3.3 Keyed-Hash Message Authentication Code

This bulk of this section appeared first in [Mü14].

The authentication procedure will mainly rely on Keyed-Hash Message Authentication Codes (HMAC). These are special message authentication codes with the underlying idea to concatenate the message with a secret key and hash the result with a cryptographic hash function. Such hash functions are both non-linear and non-reversible and the secret key is only known by sender and recipient. Therefore, authentication and data integrity is provided at the same time.
However, research has shown that a simple concatenation of the key and message is vulnerable to length extension and collision attacks, depending on the order of the concatenation [PvO95]. The standardized method for creating an HMAC is defined in RFC2104 [HKC97] and has no known vulnerabilities. The creation of a HMAC for a message m with the key k is defined as shown below, where h is the cryptographic hash function, *ipad* and *opad* are distinct padding constants and $||$ denotes the concatenation.

$$\mathrm{HMAC}(k, m) = h(k \,\mathrm{XOR}\, \mathit{opad} \,||\, h(k \,\mathrm{XOR}\, \mathit{ipad} \,||\, m))$$

3.4 Broadcast Timestamps

The usage of HMACs alone is not sufficient enough in order to reach the authentication goal. It has been shown for the NUTS project that the usage of HMACs alone would allow *replay attacks* [Pra12]. In such an attack, the attacker records a sent message to the satellite including the authentication code and replays it later to the satellite.

In order to deal with those attacks, a fresh element is required for every transmitted message from the base station. The NUTS authentication scheme utilizes timestamps as fresh element. The NUTS satellite broadcasts its actual satellite time via its beacon which is receivable by any interested base station. Thus, we speak about *broadcast timestamps* for the authentication scheme.
In order to use timestamps as fresh elements, a few additional particularities have to be discussed. First of all, the clocks of the communication parties have to be synchronized. Additionally, a common format for the timestamps has to be chosen in order to be understandable for the parties.

3.4.1 Clock Synchronization

Systems which are using timestamps as fresh element are required to have loosely synchronized [NS93] clocks, otherwise it can not be guaranteed that a message would be successfully validated. The broadcast of timestamps from the satellite has the purpose to establish this synchronization. Due to the broadcast, the base station is able to adjust its clock according to the satellite time. Hence, the satellites RTC can be seen as master clock inside the system.
This way of achieving clock synchronization has two advantages: An additional clock resynchronization protocol is not required and the time is provided by a trusted source.
The downside of this approach is that the timestamps has to be evaluated as coarse timestamps due to the communication delays which is closer elaborated in section 3.4.3.

3.4.2 Format

The timestamps are transmitted as Unix timestamps, a common standard for digital time keeping [Sin02]. An according timestamp stores the numbers of seconds since the beginning of the Unix epoch which is the January 1, 1970 at 00:00:00 UTC. The timestamp is saved as 32-bit value on the radio MCU.
The usage of this format provides several advantages. First of all, the amount of bytes needed to denote the time is rather low with 4 bytes compared to other methods. The decimal representation of unix timestamps will likewise not exceed the size of 10 characters which is smaller than a notation in the traditional *yyMMddHHmmss* format. Additionally, conversion functions to a human readable format a provided in almost every standard C library making the translation of Unix timestamps extremely easy. Last but not least, the Unix time standard is widely known and broadcasted timestamps can be interpreted by amatuer radio operators around the world which support the capabilities of satellite tracking.
For the sake of completeness the *year 2038 problem* is mentioned. The problem describes that on systems using 32-bit Unix timestamps an integer overflow occur in

2038. Thus, a Unix timestamp will represent a wrong time. However, since NUTS is expected to be launched in 2015 and the lifetime shall not exceed two years, this problem is not of relevance for the NUTS authentication scheme.

3.4.3 Tolerance

In order to use the broadcast timestamps effectively as an element for the authentication, the satellite must be able to validate the correctness of received timestamps. Here, it has to be kept in mind that significant deviations between a received timestamp and the actual satellite time are possible. This relies on the fact that the timestamps are broadcasted from the satellite in the first place and the base station uses them to synchronize its clock. Since the NUTS Satellite has different fall back modes in which other transceivers can act as the beacon, it is not possible to assume a static delay for the timestamp downlink. Thus, the satellite has to deal with both, the delays resulting from broadcasting a timestamp and sending a message to the satellite.
Traditionally, this issue is resolved by validating that the received timestamp is *approximately equal* to the current time [LB92]. This introduces the need for estimating a tolerance in which timestamps should be treated as valid.
Generally speaking, the required tolerance T can be expressed as:

$$T = D_{T_{Down}} + D_{M_{Up}} \tag{3.2}$$

For determining the delay of the timestamp downlink $D_{T_{Down}}$ and the message uplink $D_{M_{Up}}$, it is important to evaluate the links with the largest delays in order to recognize a valid timestamp in any case.
Since the directed radio transmission will unlikely experience congestion and the processing delay is rather negligible, we calculate the total delay D_{tot} as sum of transmission D_{trans} and propagation delay D_{prop} as shown in equation 3.3.

$$D_{tot} = D_{trans} + D_{prop} \tag{3.3}$$

The transmission delay D_{trans} is calculated as quotient of the packet length L and the bandwidth R.

$$D_{trans} = \frac{L}{R} \tag{3.4}$$

Similarly, the propagation delay D_{prop} is the quotient of the physical link length d and the signal propagation speed s.

$$D_{prop} = \frac{d}{s} \tag{3.5}$$

Since the propagation speed of radio waves is equal to the speed of light c, D_{prop} is equal for both $D_{T_{Down}}$ and $D_{M_{Up}}$. The exact altitude of the orbit for NUTS is not

known by now but is assumed to be less than 800 km [Men13]. Thus:

$$D_{prop} = \frac{d}{c} = \frac{1}{375}s \approx 2.7ms \tag{3.6}$$

For $D_{T_{Down}}$ the highest propagation delay can be experienced when using the radio beacon. It is designed to send a signal which can easily be decoded by amateur radio operators. Thus, it utilizes the Radioteletype (RTTY) specifications. Here, Audio Frequency Shift Keying (AFSK) is used in order to transmit Morse Code at a speed of 60 words per minute. The number of letters per word W is defined to be 5 for Morse Code, where special characters and numbers are perceived as two letters. The transmitted timestamps are consisting of 10 digits during the lifetime of the NUTS satellite. Therefore, the assumed number of letters $N_{letters}$ is 20. Hence, the transmission delay for $D_{T_{Down}}$ can be calculated as shown in equation 3.7.

$$D_{trans_T} = \frac{N_{letters}}{W} \cdot \frac{60}{M} = 4s \tag{3.7}$$

Following from this:

$$D_{T_{Down}} = D_{trans_T} + D_{prop} = 4.003s \tag{3.8}$$

The message uplink to the satellite will operate with 9600 baud. Unfortunately, neither coding scheme nor frame structure is defined for the radio link of the NUTS project yet. Thus, we will assume a data rate R_M of 9600 bits per second. Moreover, due to the missing specifications for the uplink, we can freely choose the maximum length of a packet. For this choice two factors were taken into account: the available memory on the satellite radio MCU and the scalability to the transmission delay of the timestamp downlink. The memory of the radio MCU is an important factor because received messages have to be kept in the memory for a fast evaluation. We decided that a maximum packet size L_M of 4 kilobyte is appropriate. Here, only 6.25% of the radio MCU memory is occupied by an incoming message and the transmission delay for a message of this size is comparable to the transmission delay for an timestamp. Hence, the message uplink delay can be calculated as shown in equation 3.9.

$$D_{M_{Up}} = \frac{L_M}{R_M} + D_{prop} = 3.416s \tag{3.9}$$

Following from this, the required tolerance T should be:

$$T = D_{T_{Down}} + D_{M_{Up}} = 7.419s \tag{3.10}$$

Since the timestamp precision is limited to the second range, the tolerance is rounded up to 8 seconds which also allows the uplink data rate to be marginal smaller than 9600 bps.

3.5 The Authentication Protocol

The cryptographic protocol for the authentication is the core of the authentication scheme. This section first presents an informal and intuitive description of the protocol in order to demonstrate the necessary abstractions for notating a cryptographic protocol. Afterwards, the basic authentication protocol is formally presented as basis for a protocol verification. Additionally, an improved version of the protocol is introduced which covers minor flaws in the basic protocol.

3.5.1 Informal Description

The authentication protocol for the NUTS project has been first established in the preceding report [Mü14] without a formal description. Here, the protocol specification was rather implied by the proof of concept implementation. The first protocol description is shown in in figure 3.2. Here, the satellite first sends a timestamp T to the base station. The base station constructs the $HMAC$ of the message M, the received timestamp T and the shared secret key K. Afterwards, the triple out of M, T and $HMAC$ is sent to the satellite. The satellite verifies that the received timestamp T' is equal to its own timestamp and the received $HMAC'$ is equal to a locally constructed HMAC based on the received message M', received timestamp T' and secret key K. If and only if this is the case, a response is sent to the base station.
Furthermore, it is implied that both satellite and base station share the same secret key K and are agreeing on a hash function $H()$, even if this is not directly stated.

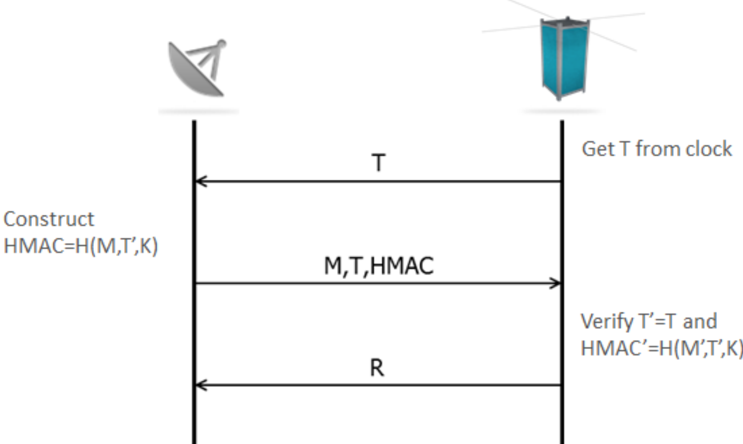


Figure 3.2: An informal description of the authentication protocol

3.5.2 Abstractions

It is easy to see that the informal protocol description as of section 3.5.1 uses several abstractions. First of all, the HMAC construction is notated as $H(M, T, K)$. This abstraction is legitimate under the *perfect cryptographic assumption* which assumes that cryptographic functions are perfect. Thus, length extension and collision attacks are not existing under this assumption. In general, it is advisable to analyze protocols under the perfect cryptographic assumption in order to focus on the algorithmic parts of a security protocol [CLT03].
The second important abstraction is that timestamps are precise and, thus, treated like cryprographic nonces. While we have to deal with coarse timestamps in reality *(cf. section 3.4.3)*, it is common praxis to model timestamps as nonces in a cryptographic protocol in the first place. This is done to reduce the complexity of protocols which is a huge advantage especially for their verification [CDL06].
The third abstraction in the informal protocol description is that timestamps are sent directed to the base segment from the satellite. This will be changed in the formal description and is therefore negligible.

3.5.3 Basic NUTS authentication Protocol

The NUTS authentication protocol, henceforth denoted as NAP, in its basic form requires that all parties have agreed on a common hash function $h()$. Additionally, each pair of communication partners have a shared secret key K. The protocol can be described as follows:

1. $A \rightarrow * : T$
2. $B \rightarrow A : M, T, h(M, T, K_{AB})$
3. $A \rightarrow B : R$

Alice, in the role as satellite, first broadcasts a timestamp T. Bob, who represents a base station, sends his message M together with T and the hash of M, T and K_{AB} to Alice, whereupon Alice sends her response R to Bob.

3.5.4 Extended NUTS authentication Protocol

From a cryptographic point of view, the basic NAP protocol has a major flaw. Intuitively, a response R shall ensure that a message M was received. However, it is not specified by the protocol that a specific response R belongs to a specific message M. Thus, even if Bob receives the third message of the protocol, it is not guaranteed that it corresponds to his recently sent message.
This can be fixed by appending the hash $h(M, T, K_{AB})$ to the third message. This hash can be considered as a fingerprint for a message M and, therefore, Alice signals

that she has received a specific message by including it in the third message of the protocol.
Hence, the extended NAP protocol is denoted as:

1. $A \to * : T$
2. $B \to A : M, T, h(M, T, K_{AB})$
3. $A \to B : R, h(M, T, K_{AB})$

Chapter 4
Formal Verification of the NUTS Authentication Protocol

The aim of formal analysis of security protocols is to assure their correctness. On the example of the Needham-Schroeder Protocol [NS78], it has shown that the classical way of just claiming protocol properties is rather unreliable [Mjø11]. Thus, researchers have elaborated formal methods for the analysis of cryptographic protocols. Hereby, formal methods are techniques based on mathematics and logic which allows to prove if a model of a system can satisfy its requirements [Mea03]. The requirements for cryptographic protocols are the claimed *security properties*, such as secrecy and authentication.

However, it has to be mentioned that the security of a cryptographic protocol is an undecidable problem [EG83, MSDL99, CDL06]. Thus, formal verification can not guarantee the non-existence of weak spots in a protocol. Nevertheless, a protocol is more trustworthy when formal verification can confirm its soundness. [Mjø11].

By now, a huge amount of automated tools for security protocol analysis utilizing formal methods have been developed, such as *AVISPA* [ABB+05], *CryptoVerif* [Bla07] and *Scyther* [Cre08a]. The latter one is used for this work and will be introduced in section 4.1.

In order to establish a formal analysis with Scyther, protocols must be specified in its input language. Here, not all messages can be translated into an one-to-one relationship, thus a model of NAP for the formal verification with Scyther is elaborated in section 4.2. The results of the verification are presented in section 4.3.

4.1 Scyther

Scyther is a tool for automatic analysis of cryptographic protocols. It is developed by Cas Cremers and available for Linux, Windows and Mac OS X [Cre13a]. Besides a command-line and python scripting interface, the tool offers a GUI for the verification of protocols. In order to analyze the protocols introduced in chapter 3.5, Scyther version v1.1.3 for Linux was used.

This section shall serve as a short introduction to Scyther, elaborating its special features, the input language and the verifiable security properties.

4.1.1 Features

One advantage of Scyther in comparison with other tools is the possibility to perfom an *unbounded verification*. Traditionally, tools verified only the claimed security properties for a finite subset of possible behaviors for a given protocol which is called *bounded verification*. Opposing this, unbounded verification aims to demonstrate the soundness of a protocol for all possible behaviors, even in the presence of an adversary [Cre08b]. Here, the adversary is canonically modeled according to the Dolev–Yao model [DY83] which grants the three following properties:

- The adversary can obtain, alter or delete any message sent on the network.
- The adversary is a legitimate user of the network and is therefore allowed to send messages to any other users.
- Moreover, he is allowed to be the legitimate receiver of a message from any other user.

However, Scyther can not guarantee to establish an unbounded verification for any given protocol. Instead, the termination of the verification algorithm is guaranteed. This relies on the fact that the verification algorithm establishes a bounded verification for rather seldom cases where neither an unbounded verification nor a falsification for a given protocol is possible [Cre08a].
A second important feature of Scyther is the *complete characterization* of given protocol roles in the sense of [CV02]. This means that Scyther is able to determine representatives for all possible protocol behaviors including the execution of a specific role. This representatives are denoted as *trace patterns*. The trace patterns are forming a finite set and are designed in a manner allowing the verification of security properties. Following from this, a claimed property can be verified for all patterns. If the property can not be falsified for every pattern, it is holding for all possible protocol behaviors by implication [Cre08b].
Another positive aspect is the efficiency and performance of Scyther. It is one of the fastest tools as of 2009 while still finding attacks efficiently [CLN09].
Lastly, the visualization functionalities of Scyther are worth mentioning. Besides a graphical representation of the trace patterns for characterized roles, Scyther generates attack graphs for found attacks.

4.1.2 Input Language

The input language for describing security protocols in Scyther is called Security Protocol Description Language (SDPL). The syntax for SDPL is to some degree based on C and Java [Cre14] and, therefore, rather intuitively. Due to the nature of cryptographic protocols the language itself is role-based and a protocol must always define its containing roles.
The available data types are derived from those usually found in security protocols, such as nonces, functions or agents. Additionally, the user can define own data types for additional clarification. All variables have to be declared according to their nature with one of the keywords *fresh*, *var* or *const*. The first keyword indicates that the corresponding variable is fresh generated by the role, while *var* is used for variables received from another agent. The last keyword, however, is used to declare global constants known by more than one role.
Furthermore, SDPL provides expressions for cryptographic primitives, such as symmetric and asymmetric encryption or hash functions. The syntax of those expressions here is similar to the notation used in this work. However, in Scyther hash functions have to be defined before they can be used. These definitions are done in the same manner like user type definitions.
The last element of SDPL are events which have to occur within a role definition and are either *send*, *recv* or *claim*. These events are used to express sending and receiving of a message or a claim for a specific security property. Every event is followed by an underscore and an unique label to provide distinguishability. Send and receive events are used for indicating the communication among the different roles. Usually, every send event in one role has a corresponding receive event in another role. Exception to this is the leakage of information to or receiving messages from the adversary. Here, an exclamation mark has to be added after the underscore to symbolize the intention of communicating with the adversary.
The claim events are used to claim the security properties of the specified protocol. Here, one property can only be claimed for the role in which the claim event occurs. The possible security claims are closer elaborated in subsection 4.1.3.
Putting it all together, a commented version of a basic secrecy protocol from the Scyther exercise set [Cre13b] is provided in Algorithm 4.1 in order to visualize the basic syntax.

4.1.3 Verifiable Security Properties

A first security property which can be verified by Scyther was already mentioned in section 4.1.2: The secrecy of specific variables.
Contrary to this, the main aim of the NUTS authentication protocol is to authenticate the ground segment to the satellite. While the concept of authentication itself is comprehensible, the claim that a protocol authenticates an agent A to an agent B is

Algorithm 4.1 A simple secrecy protocol for Scyther

```
1  //declaration of the protocol including the roles
2  protocol protocol0(I,R)
3  {
4    //declaration of the initiator role I
5    role I
6    {
7      //declaration of the nonce ni which is generated by this role
8      fresh ni: Nonce;
9      //I sends {I,ni}pk(R) to R, whereby the term {I,ni}pk(R) denotes
10     //Is identity and the nonce ni encrypted under the public key of R
11     send_1(I,R, {I,ni}pk(R) );
12     //Claim secrecy for the nonce ni
13     claim_i(I,Secret,ni);
14   }
15   //declaration of the responder role R
16   role R
17   {
18     //declaration of the nonce ni which is generated by another role
19     var ni: Nonce;
20     //R receives {I,ni}pk(R) from I
21     recv_1(I,R, {I,ni}pk(R) );
22     //Claim that ni stays secret
23     claim_r(R,Secret,ni);
24   }
25 }
```

rather unspecific. *Aliveness* for instance, the weakest form of authentication, only assures that the intended communication partner B exists and has run the protocol previously. On the other hand, it is often required that both agents are *agreeing* to the exchanged data which is a way stronger form of authentication. However, both forms of authentication could be implied by the phrase "A authenticates itself to B" without a closer examination what the actually meaning of authentication is. Thus, an authentication hierarchy was introduced by Lowe in [Low97] which got extended by Cremers et al. in [CMdV06].
The Scyther tool is able to verify some of this authentication forms, namely *Aliveness*, *Weak Agreement*, *Non-injective agreement* and *Non-injective synchronization*. This verifiable security properties are shortly elaborated at this point in the definitions 4.1 - 4.4. A complete formal description of the authentication hierarchy can be found along with details about Scythers verification algorithm in [CM12].

Definition 4.1. Aliveness, adapted from [Low97]
Whenever an initiator A finishes a run of a protocol and the presumed responder B has previously been running the protocol, aliveness with B is established for A.
B may neither have believed to run the protocol with A, nor is it required that he has run the protocol recently.

Definition 4.2. Weak agreement, adapted from [Low97]
Whenever an initiator A finishes a run of a protocol and the presumed responder B has previously been running the protocol presuming A as initiator, weak agreement with B is established for A.
B may not have been in the role of the responder during his run of the protocol.

Definition 4.3. Non-injective agreement, adapted from [Low97]
Whenever an initiator A finishes a run of a protocol and the presumed responder B has previously been running the protocol presuming A as initiator, non-injective agreement with B over a set of data s is established for A when B was in the role of the responder in his run and both A and B are agreeing on all values of s.

Definition 4.4. Non injective synchronization, adapted from [CMdV06]
Whenever an initiator A finishes a run of a protocol and the presumed responder B has been running the protocol presuming A as initiator, non-injective synchronization with B is established for A when B was in the role of the responder in his run, each run of A corresponds to an unique run of B and all messages are received and sent in the order provided by the protocol.

4.2 Modeling the protocol

It was pointed out in section 3.2 that various assumptions of the environment are made. Some of them are crucial for the proposed protocol. Thus, not only the protocols as shown in section 3.5 have to be modelled for Scyther but also the implications heritating from these assumptions.
Furthermore, the usage of timestamps has to be modeled for Scyther before a complete model can be established.

4.2.1 Trust Assumption

We showed in section 3.2.3 that the satellite is a trusted source due to the physical properties of the communication. This assumption is not directly expressible in Scyther. Thus, an appropriate method to model this assumption has to be provided. Considering the main properties of digital signatures as shown in definition 4.5, similarities to the trust are not undeniable.

Definition 4.5. Main properties of digital signatures
A digital signature from an agent A for a message M provides the following properties:

1. *Authentication* M is guaranteed to be created by A
2. *Non-repudiation* A can not repudiate that he generated M
3. *Integrity* It is guaranteed that M is not altered after signing

Since a message sent from a trusted agent is guaranteed to originate from this agent, property 1 is achieved. Furthermore we are assuming that the underlying link layer protocol is able to detect and correct transmission errors. Combining this with the fact that an attacker is not able to intercept and alter a message, property 3 is given as well. Only property 2 can not be directly guaranteed by the environment for the NUTS authentication scheme. However, it is still justifiable to model the trust assumption according to a digital signature for Scyther because it is a reasonable approximation. Besides that, a repudiation of a message from the satellite affects only the timestamps in the basic authentication scheme as described in section 3.5.3. Since timestamps can be constructed and verified by the ground segment as soon a single valid timestamp from the satellite is received, a repudiation of a former timestamp would not neglect the authentication procedure.
Thus, we model every message originating from the satellite role as a message signed by the satellite.

4.2.2 Timestamps

Scyther does not have an inbuilt data type for timestamps. However, it is possible to reproduce the behavior of a protocol using timestamps and the Scyther manual [Cre14] recommends two ways of modeling timestamps.
The first possibility is to declare the timestamps as nonces and make them public to the adversary before using them in the protocol. Logically, this also seems to be a great way to model the broadcasted nature of the timestamps.
However, this way of modeling timestamps is only valid when the probability of accepting the same timestamp in two diverse runs is low. This is not feasible for NAP since the timestamp acceptance tolerance has to be greater than one second to guarantee a reliable mode of operation due to the expected delay in the beacon downlink *(cf. section 3.4.3)*. Additionally, it might occur that more than one package per second are sent to the satellite but the resolution of the timestamps is limited to the second range.
The second possibility for modeling timestamps in Scyther is catching the probability of coarse timestamps. Here, the timestamps have to be declared as variables and their values are getting assigned by the adversary. By doing so, the property of the timestamps to be broadcasted is not easily seeable anymore. Nevertheless, it can be deduced from the fact that the adversary determining the timestamps is a legitimated member of the network. Thus, the timestamps must be known within the network, which is the implication of broadcasting the timestamps. Therefore, the timestamps can still be considered as broadcasted.
Accordingly, modeling timestamps as variables with values assigned by the adversary is an appropriate way to express NAP in Scyther.

4.2.3 Resulting Scyther Models

Two slightly different Scyther model for NAP have been developed, according to the basic version shown in section 3.5.3 and the extended version shown in section 3.5.4. For a better clarity the defined roles for the protocol are base stations B and satellite S. The formal description of the authentication protocol as modeled for Scyther is:

1. $* \rightarrow S : T$
2. $S \rightarrow B : \{T\}_{sk(S)}$
3. $B \rightarrow S : M, T, h(M, T, K_{BS})$
4. $S \rightarrow B : \{R\}_{sk(S)}$

Message 1 is required to model the timestamp as descriped in subsection 4.2.2. Associated with this is the fact, that message 2 is directed to the B instead of broadcasted. Additionally, messages 2 and 4 are signed by the S which is used to model the trust assumption *(cf. section 3.2.3)*.
This protocol model can easily be transformed in a model for extended NAP, here only message 4 has to be substituted with:

4. $S \rightarrow B : \{r, h(M, T, K_{BS})\}_{sk(S)}$

The resulting SDPL specification is given in algorithm 4.2. Lines 1-3 are containing global definitions for the hashfunction and user types for data messages and timestamps. The commands for uplink and the responses for the downlink are declared as constants in line 7-8. This declarations are made because both commands and possible responses are elements of finite sets which can be obtained by the adversary - for instance through eavesdropping or access to the specification from the NUTS project. The lines 14-16 and 25-28 represent the send and receive events according to the above-presented basic protocol description.
Since the aim of the NAP protocol in its basic form is to authenticate the ground segment to the satellite, claims for the different forms of authentication as defined in section 4.1.3 are made in line 31-34. Here, we are claiming that aliveness, weak agreement, non-injective agreement and non-injective synchronization are established by the protocol for S with A. Additionally, secrecy is claimed in both roles for the shared key since it is crucial for the protocol that this key remains undisclosed.
This protocol specification can easily be transformed to verify the extended NAP protocl as well. Here, only lines 16 and 28 have to be modified and additional claims for authentication have to be added for the base station role.

Algorithm 4.2 Scyther model for the basic NUTS authentication protocol

```
1  hashfunction hash;
2  usertype Msg;
3  usertype T;
4
5  protocol NAP(B,S)
6  {
7    const m: Msg;
8    const r: Msg;
9
10   role B
11   {
12     var t: T;
13
14     recv_1(S,B,{t}sk(S));
15     send_2(B,S,m,t,hash(m,t,k(B,S)));
16     recv_3(S,B,{r}sk(S));
17
18     claim_b5(B,Secret,k(B,S));
19   }
20
21   role S
22   {
23     var t: T;
24
25     recv_!T1(S, S, t);
26     send_1(S,B,{st}sk(S));
27     recv_2(B,S,m,t,hash(m,t,k(B,S)));
28     send_3(S,B,{r}sk(S));
29
30     claim_s1(S,Alive);
31     claim_s2(S,Weakagree);
32     claim_s3(S,Niagree);
33     claim_s4(S,Nisynch);
34     claim_s5(S,Secret,k(B,S));
35   }
36 }
```

4.3 Verification Results

Both specifications have been successful characterized and verified with Scyther. Here, Scyther was run with the additional parameters *--unbounded*, for guaranteeing the unbounded verification, *--all-attacks* and *--one-role-per-agent* which specifies that agents are not able to switch roles within distinct runs of the protocol.
The results obtained by the verification with Scyther are shown and analysed in the following section 4.3.1 and 4.3.2. The resulting flaws are analysed in section 4.3.3 by the example of specific attacks.

Notes on the additional parameters It is urgent to run Scyther with the -*-unbounded* and *--all-attacks* parameter to determine the exact amount of trace

patterns and attacks, otherwise Scyther would just establish lower.
The --*one-role-per-agent*, on the other hand, is logically correct for the NAP protocol but does not influence the verification results for this particular protocol. Thus, the following analysis is also valid when agents are allowed to occupy different roles in different runs.

4.3.1 Analysis of Characterization

Scyther was able to establish a complete characterization for both protocol specifications. The number of found trace patterns for the according roles is provided by table 4.1.
In both cases more than one trace pattern has been found indicating the possibility of attacks. Furthermore, it is easy to see that the extended NAP protocol reduces the number of patterns for the base segment. This results from the fact that guaranteeing the reception of sent data lowers the possibilities for the adversary to trick the base segment. Accordingly, the number of trace patterns for the satellite are the same for both specifications. Thus, we can assume that the basic authentication works in the same manner in both cases and is not dependent on the additional information introduced by the extended NAP protocol.

Table 4.1: Number of found trace patterns for the NUTS authentication protocol

Analysed Specification	Role	
	Base Station	Satellite
Basic authentication protocol	39	8
Extended authentication protocol	18	8

4.3.2 Analysis of Reported Attacks

It was possible to verify both specifications in an unbounded manner. The number of found attacks on the specific claims are shown in table 4.2 for the basic NAP protocol and in table 4.3 for the extended protocol. For the sake of completeness, all verifiable authentication claims were analyzed with Scyther even for the basic authentication protocol.
A found attack for a claim means that the corresponding claim of this role is not holding and can be falsified with this specific attack. To clarify this, we utilize the 13 found attacks for the base station's claim of non-injective synchronization in table 4.3 as example: The found attacks are demonstrating that the base station can not establish weak agreement with the satellite because 13 distinct attacks are falsifying the corresponding claim.
Analysing the results for the basic authentication protocol, it is easy to see that the the shared key K_{BS} stays secret. All authentication claims except non-injective

synchronization are holding for the satellite role which means that the protocol authenticates the base station to the satellite without establishing non-injective synchronization.
On the other hand, it is not surprising that a huge amount of attacks for weak agreement, non-injective agreement and non-injective synchronization were found for the base station role. The protocol is not designed to establish this properties since they would induce that the satellite authenticates itself to the base station. However, the Aliveness of the satellite is guaranteed for the base station which relies on the fact that a signed timestamp has to be received in order to initiate the protocol.

Table 4.2: Found attacks for the basic NUTS authentication protocol

Claimed Property	Role	
	Base Station	Satellite
Secrecy for K_{BS}	-	-
Aliveness	-	-
Weak Agreement	31	-
Non-Injective Agreement	31	-
Non-Injective Synchronization	35	4

In the extended NAP protocol, the weak agreement and non-injective agreement claims are also holding for the base station role and the number of found attacks for non-injective synchronization has been reduced. Thus, the protocol can establish the same authenticity properties for both roles. Additionally, the number of reported attacks on non-injective synchronization for the satellite role has not changed. In fact, these are exactly the same attacks which confirms the assumption that the authenticity of the base station for the satellite does not get changed by using the extended NAP protocol.

Table 4.3: Found attacks for the extended NUTS authentication protocol

Claimed Property	Role	
	Base Station	Satellite
Secrecy for k_{BS}	-	-
Aliveness	-	-
Weak Agreement	-	-
Non-Injective Agreement	-	-
Non-Injective Synchronization	13	4

4.3.3 Analysis of the Protocols' Flaws

In order to be clear about the flaws of the protocols, namely, why non-injective synchronization is not holding for both roles, we analyzed the reported attacks of Scyther. Figure 4.1 is an example attack falsifying the claim that S is synchronized with B. Here, Alice plays the role of the satellite and both Bob and Charlie are playing the role of distinct base stations. The attack graph shows, that in the first run Alice sends a timestamp to Bob which is never received. In the meantime, Alice starts another run of the protocol with Charlie. The adversary intercepts the timestamp which is sent to Charlie and redirect it to Bob instead. Bob is now executing the protocol and sending his constructed message to Alice and the protocol follows its normal execution. It is easy to see that Alice run the protocol two times to establish authentication with Bob. Thus, more than a single run of Alice corresponds to a run of Bob which violates the definition of non-interjective synchronization due to the missing one-one relationship between the runs.
There are two main causes for this attack. The obvious one is, that in the first message from S to B the identity of B is not included. This allows the possibility of redirecting this message as shown in the attack graph. Even if the identity would be included in the protocol, it still can not guarantee non-injective synchronization. Assuming that the S initiates the protocol two times consecutively with B, the adversary can intercept the timestamp from the first run. This would result that B executes his run as result of the second run of S, while S thinks that the received messages from B correspond to the first run. This problem originates from allowance of coarse timestamps: Since a timestamp can be valid in more than one run and no other nonces are included in the protocol, it is obvious that the protocol can not guarantee synchronization for S with B.
If we analyze the extended protocol, we can see that the attack on non-injective synchronization for B with S relies on the fact that one sent timestamp could possibly be received in two distinct runs of B. Thus, it could occur that Alice receives a message from one run with Bob but her response is captured by another run of Bob. It is clearly to see that a one-one relationship between the runs of Bob and Alice can not be established. The reason for this lies in the nature of the protocol: B receives two messages from S in one run of the protocol but only sends one message. Since only one fresh variable, the timestamp, is used over the full protocol run, B can not be assured that the second received message really corresponds to the actual run.

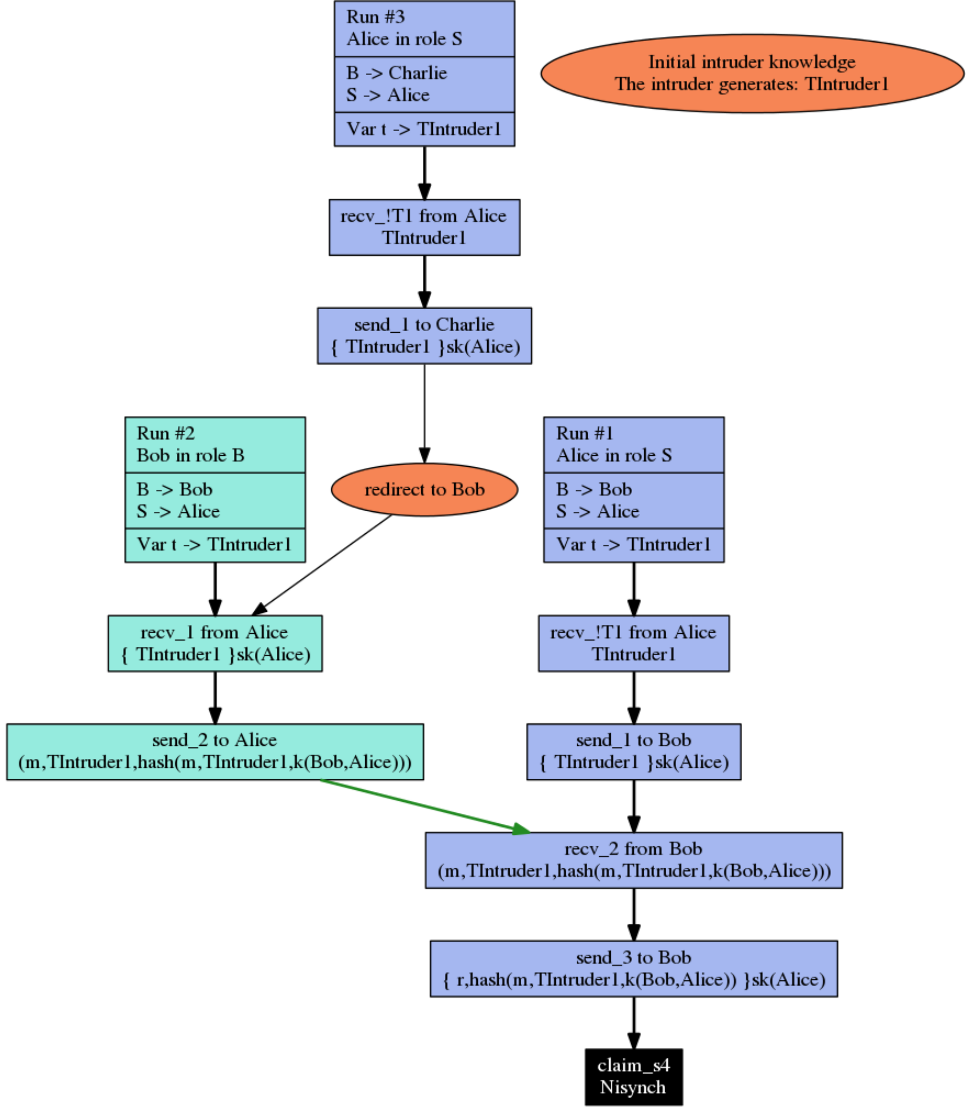


Figure 4.1: Example attack on non-injective synchronization for S with B

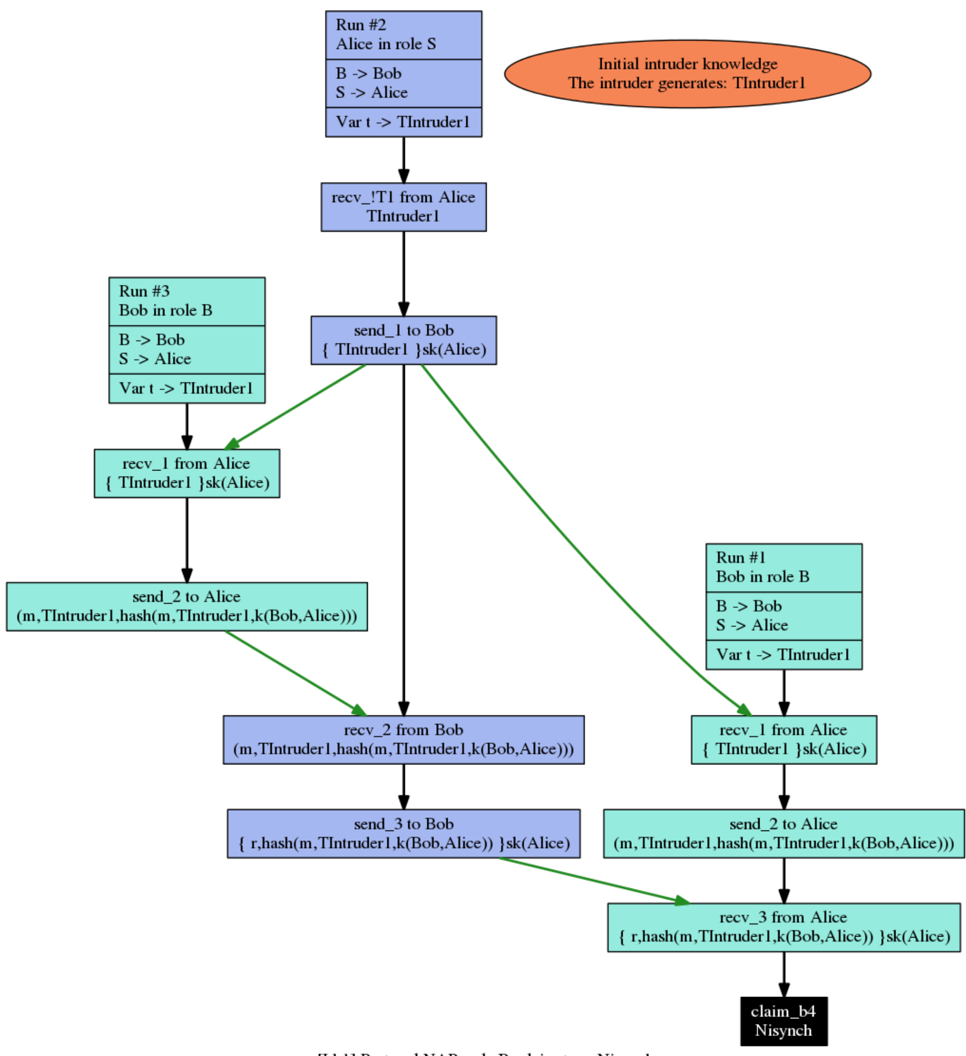
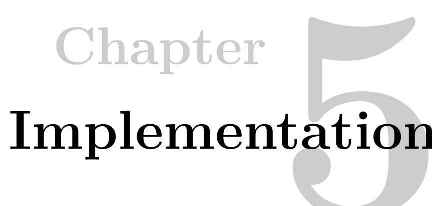


Figure 4.2: Example attack on non-injective synchronization for B with S

To verify our reasoning about the causes of the protocol flaws, we developed additional Scyther models for the protocol utilizing precise timestamps. Like already seen for coarse timestamps, it does not matter for S if NAP would be executed in its basic or extended form. Thus, only the number of found attacks on the specific claims of the extended NAP protocol are shown in table 4.4. It is easy to see, that precise timestamps would establish non-injective synchronization for S with A. Additionally, the number of attacks found on non-injective synchronization for B with A has been reduced. The remaining attacks can be explained with the same argument as mentioned above.

Table 4.4: Found attacks for the extended NUTS authentication protocol with precise timestamps

Claimed Property	Role	
	Base Station	Satellite
Secrecy for k_{BS}	-	-
Aliveness	-	-
Weak Agreement	-	-
Non-Injective Agreement	-	-
Non-Injective Synchronization	4	-



Implementation

Parts of the authentication scheme were implemented as proof of concept module in the preceding project report [Mü14]. In the course of this thesis, the functionalities of this proof of concept have been enhanced in order to establish a usable realization of the NUTS authentication protocol.
In this chapter, the development environment is described first. Afterwards, details f the implementation of the communication protocol are provided which is followed by the specifics of the developed satellite and base station code.

5.1 Development Environment

The development area was to a large extend already established in the preceding report [Mü14]. However, in order to establish a common basis for the implementation details, the development environment is also presented in terms of both hardware and software within this thesis.

5.1.1 Hardware

The bulk of this section appeared first in [Mü14].

The hardware used to implement the authentication scheme for satellite can be divided into four subsystems. The key element of the implementation is the UC3-A3 Xplained evaluation kit for a microcontroller from Atmel. An AVR Dragon is used as programming and debugging interface for this kit. The RTC is a DS3231SN from the company maxim integrated. The communication link between microcontroller and RTC is an Inter-Integrated Circuit (I^2C) bus which had to be deployed separately. Figure 5.1 provides an overview over the development hardware.

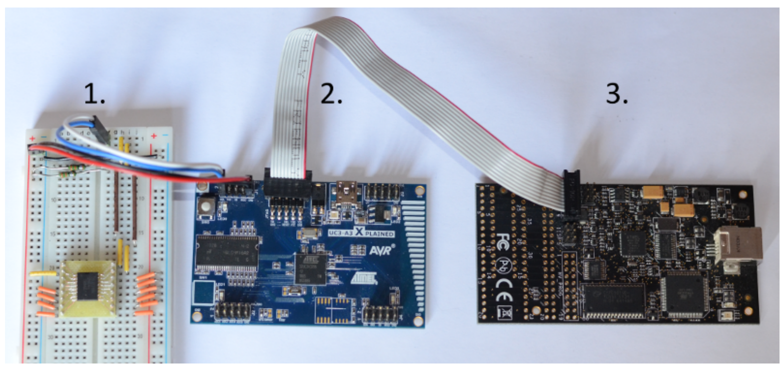


Figure 5.1: The development environment: 1.) I²C/RTC 2.)AVR UC3-A3 3.) AVR Dragon

U3-A3 Xplained

The UC3-A3 Xplained is an evaluation kit for the Atmel AT32UC3A3256 microcontroller which is also used by the satellite. The kit contains a 64Mbit SDRAM, four status LEDs and both one push and one slider button. It provides one USB, one Joint Test Action Group (JTAG) and four Universal Asynchronous Receiver/Transmitter (USART) headers [Atm12]. The JTAG header is utilized as interface to the AVR Dragon. One of the USART interfaces is used for the I²C bus to the RTC.

AVR Dragon

The AVR Dragon is a programmer and debugger for 8-bit and 32-bit AVR microcontroller. A core feature of the Dragon is on chip debugging and it supports up to three hardware and 32 software breakpoints [Coo09]. It offers different interfaces where the JTAG interface is used for this project.

DS3231SN

The used RTC is the industrial implementation of the DS3231 I²C clock. It supports operating temperatures from -45 °C to +85 °C and is therefore suitable for operation inside a satellite in the low earth orbit. The accuracy is denoted with ±3.5 ppm which means that the RTC is satisfyingly precise. The clock has two power inputs including one for a battery which is automatically activated when the normal supply fails. The clock processes the time in seconds, minutes, hours, days, month and years. It contains an internal leap year compensation up to the year 2100 which is by far out ranging the operating time of the NUTS satellite [Max13].

The RTC is already designated to be deployed inside the satellite. A previous project developed customized software for the use of the DS3231 in combination with the UC3-A3 Xplained [JMT+13].

I²C

I²C is a serial data bus for the communication between different peripherals and used as bus between the different components of the satellite [IB03]. It uses two bi-directional connections, the serial data line (SDL) and serial clock (SCL). Both of them are connected to the peripherals and to the power line with pull-up resistors. The resistance value of the resistors used for the I²C implemention is 2.7 kΩ.
The bus is designed as a master/slave system and the UC3-A3 operates as master node in the development environment. This will be changed in a final implementation where the OBC of the satellite will serve as master. The used I²C implementation uses a 7-bit addressing scheme and the address of the RTC (0b1101000) is hardcoded inside the clock itself. Software to access the I²C bus is also provided by [JMT+13].

5.1.2 Software

The bulk of this section appeared first in [Mü14].

This section describes the third party software which was used during the development process. In particular, this is the development software Atmel Studio, the real time operating system FreeRTOS and the cryptographic library PolarSSL.

Atmel Studio

Atmel Studio 6.1 is an Integrated Development Environment (IDE) for Atmel microcontrollers [Atm13]. The IDE is free of charge and runs on Windows. It offers a variety of useful tools like software libraries for different Atmel products, a built-in compiler tool chain and direct device programming. The provided software libraries are all part of the Atmel Software Framework and are therefore customized for Atmel microcontrollers. The studio supports different programming and debugging interfaces like the AVR Dragon and is capable of on-chip debugging functionalities. Additionally, it contains a simulator in order to run and test code without the need of a connected microcontroller. All in all it is a mighty platform and simplifies the development process.

FreeRTOS

FreeRTOS is a small open source operating system for embedded systems. It is available for different microcontrollers including the used AT32UC3A3256. The operating systems provides basic functionalities to enable tasking like a scheduler and semaphores for both mutual exclusion and synchronization. FreeRTOS V7.0.0

is used on all microcontrollers of the NUTS project and the implementation of the security module runs therefore on FreeRTOS.
A task itself is just a C function which never returns. In order to initiate a task the function *xTaskCreate*() has to be called with the pointer to the function and various other parameters like task name, stack size and priority. After all desired tasks are created the scheduler has to be started through the call of *vTaskStartScheduler*(). Algorithm 5.1 shows exemplary C-code in order to demonstrate the task functionalities of FreeRTOS. Here, a simple function which toggles a LED on and off on the AVR UC3-A3 evaluation board is defined in line 1-6. In the corresponding main function, this function is initiated as task in line 10. This task is executed as soon the scheduler is started in line 11.

Algorithm 5.1 Example of a FreeROTS task

```
1  void simpleLEDtask(void) {
2    while(1) {
3      gpio_toggle_pin(AVR32_PIN_PB03);
4      vTaskDelay(1000);
5    }
6  }
7
8  int main(void)
9  {
10   xTaskCreate((TaskFunction_t)simpleLEDtask, "SimpleLEDtask",
       configMINIMAL_STACK_SIZE+512,NULL, 1, NULL);
11   vTaskStartScheduler();
12 }
```

PolarSSL

PolarSSL is used as library for cryptographic hash functions. It is easy to use and provides a direct access to different HMAC functions. Therefore it is only necessary to include the corresponding header file and call the function with its parameters. These parameters are, in order of their occurrence, the HMAC key, the length of the key, the buffer containing the data which shall be evaluated, the length of the data and the buffer in which the calculated HMAC shall be saved.

5.2 Communication Protocol

While the authentication protocol described section 3.5 describes in which order and with which data messages should be exchanged, the communication protocol defines the structure of the messages itself. This is necessary in order to provide a fixed set of rules for the implementation in base station and ground segment to enable successful communication between both entities.

5.2.1 Hash Function for HMAC Construction

The authentication protocol assumes a secure hash function used by both parties. In the implementation, this hash function has to be defined. An analysis of the efficiency and performance of different hash functions has been carried out in the preceding report [Mü14]. The analysis has shown that MD5 is suitable for the NUTS project due to the small digit size, the high performance and the low on-chip size. All of this properties are ideal for an embedded system with low computational power such as NUTS.
However, it is widely known that MD5 is considered as broken and its usage as secure hash should therefore be well considered. More specifically, it has been shown that MD5 is vulnerable to collision attacks and collisions can be generated fairly easy [WY05]. Additionally, preimage attacks are theoretically possible on MD5 with a computational complexity of $2^{123.4}$ [SA09].
Notwithstanding this vulnerabilities, MD5 can still be used as secure hash function in the NUTS authentication protocol. The reason for this is that MD5 is used as hash function for HMAC functionalities and not as standalone hash function. In fact, after the collision attacks for MD5 were found, it was shown that the underlying hash function for the construction of secure HMACs does not have to be collision-resistant [Bel06]. Stallings concludes that it is reasonable to use MD5 in time critical cases as hash function for HMAC on embedded system because an attacker would need 2^{64} HMAC blocks in order to reconstruct the key [Sta11].
NUTS, on the other hand, is expected to not receive more than 2^{16} messages in his full lifetime [Pra12]. Thus, the usage of MD5 for HMAC can achieve reasonable security for the NUTS authentication scheme, while still allowing a fast computation and reduced overhead on the radio link.

5.2.2 Protocol Header

The protocol header is designed to be as easy and short as possible, in order to minimize the overhead for transmitted packets. Thus, the header consists of only 3 fields: the timestamp, the HMAC and the data length. The first two fields are required to provide the authentication functionalities while the latter one is used to specify the length of an packet.

Timestamps are transmitted in the Unix timestamp format and have a size of 4 bytes *(cf. section 3.4.2)*. The length of a HMAC is equal to the digest length of the underlying hash function. Since MD5 is used as hash function, the HMAC header field has a length of 16 bytes. The data length field removes the need for an additional procedure to determine the actual packet size. It also specifies the amount of bytes over which the HMAC was constructed. Since the maximum amount of possible bytes in a packet can be expressed in 2 bytes, the tradeoff between protocol and computational overhead is reasonable.
The result header has a length of 22 bytes and is shown in table 5.1.

Table 5.1: The communication protocol header

Field	Timestamp	HMAC	Data length
Size [bytes]	4	16	2

5.2.3 Flexibility

The communication protocol has been implemented in a flexible manner on the satellite side in order to provide adaptability to changes in the project specification. In this sense, a transition to another hash function is easily realizable. Additionally, this flexibility establishes a reusability for other projects or scenarios.
The flexibility is given by expressing the protocol definition as macros in C. The implementation of the authentication functionalities uses only this macros for the processing of the protocol. Algorithm 5.2 shows the defined macros for the protocol header as defined in section 5.2.2

Algorithm 5.2 Definition of Macros for the Protocol Header

```
1 #define PROTO_POS_TIMESTAMP 0
2 #define PROTO_POS_HMAC 4
3 #define PROTO_POS_DATALEN 20
4 #define PROTO_HEADERLEN 22
5 #define PROTO_HMACLEN 16
```

5.3 Satellite

All required functionalities for the authentication are provided in one C file with an corresponding header file. In order to allow substantial tests for the security functionalities, different modes of operations are possible. The main authentication functionalities are provided in two distinct functions, *security_validate()* and *security_response()*. The first one is designated to verify the correctness of the timestamp and the HMAC correctness of an incoming packet. The latter one is only required when the extended NAP protocol is used since it appends the fingerprint of an received message to the corresponding outgoing response.
The provided interface utilizes callback functions for further data processing and is elaborated separately.

5.3.1 Operation Modes

The need for different operation modes results from the fact, that the authentication scheme is a state-of-the-art feature for CubeSats. Thus, the specific behavior in-space could differ from the expected behaviour. Thus, different operation modes allow unit tests for the individual parts of the authentication.
Besides that, the operation modes are providing fallback functionalities. Should a required subsystem for the authentication fail, it is still possible to operate the satellite with a weaker form of authentication. This is important since it is a worst case scenario that the NUTS base station is not able to communicate with the satellite due to unexpected failures in the authentication procedures. Therefore, the weakest operation mode does not perform any security checks at all. The second weakest mode is designated to run even when the RTC fails. Here, the correctness of the timestamp in a received message is not verified while a received HMAC is still validated. The third mode validates a received message according to the basic authentication protocol while the strongest mode also processes outgoing messages according to the extended authentication protocol.
It is easy to see that this for operation modes are forming a hierachy which is shown in figure 5.2. Here, the name of the operation modes corresponds to their names in the source code. It is remarkable that the different authentication modes can be considered as tradeoff between fail-safeness and strength of authentication.
The interface for changing the operational mode is defined by a seperate function *security_setmode*() which expects the integer representation of the specific modes as input. It is obvious that for a secure system this function should only be called from an authenticated context. This also holds for transitions from `SECURITY_MODE_OFF` to other operation modes since otherwise an attacker could lock out legitimate users by demanding another security mode in this particular case. Thus, as an additional measurement to minimize the resulting damage from such an attack a transition is only possible from one operation mode to the next higher or lower one.

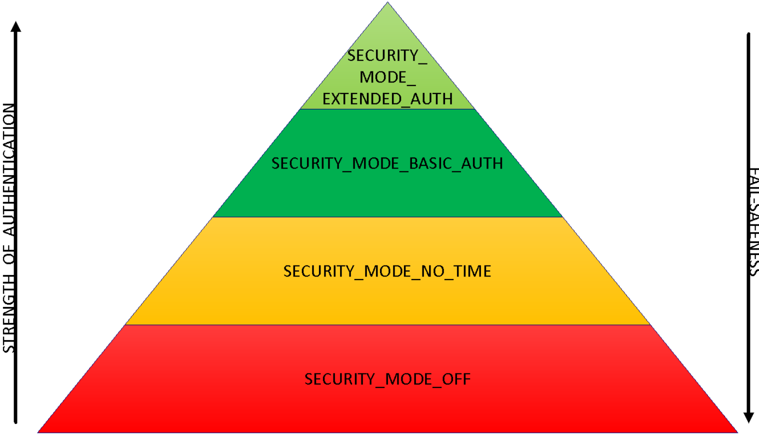


Figure 5.2: Hierarchy of operation modes

5.3.2 Validation

The core of the authentication implementation on the satellite side is the function *security_validate*() which is shown in algorithm 5.3. Its input is a pointer to the buffer containing the received message and the output is an status code according to definitions made in the header file.
The execution flow from the validation itself is dependent from the used operation mode is observable in line 6 and 13. A validity check on the data length field is performed in lines 8-11. This check shall deny the possibility of malicious data length field inducing unexpected behavior. Afterwards, the conditional branch from line 13 to 18 verifies if a received timestamp is within the specified tolerance in the according operational nodes.
Thereafter, the local HMAC is constructed in lines 20-22, where the last line calls the the external HMAC function from PolarSSL. Since the local HMAC construction requires that the HMAC field in the received message is set to zero, the original state of the input buffer is restored in line 23. Lastly, it is checked if the received HMAC matches the local constructed one in lines 24-26. In the case that none of the validation checks fails, the received message is assumed as authenticated and the function returns with the according status code.

5.3.3 Response

The *security_response*() procedure as shown in algorithm 5.4 is only invoked when then extended NAP protocol shall be run which corresponds to the operation mode `SECURITY_MODE_EXTENDED_AUTH`. It expects a pointer to the pointer of the output

Algorithm 5.3 Function for validating received messages for the satellite

```
1  static int security_validate(char *inbuf){
2    unsigned char hmac_recv[PROTO_HMACLEN];
3    unsigned char hmac_calc[PROTO_HMACLEN];
4    unsigned short datalen;
5    int32_t timestamp_recv, time;
6    if(mode > SECURITY_MODE_OFF){
7
8      datalen = security_extract_data_length(inbuf);
9      if(datalen < 1 || datalen > MAX_PACKET_SIZE-PROTO_HEADERLEN){
10       return VALIDATE_INVALID_DATALEN;
11     }
12
13     if (mode>SECURITY_MODE_NO_TIME){
14       memcpy(&timestamp_recv,&inbuf[PROTO_POS_TIMESTAMP], sizeof(
         int32_t));
15       time = rtc_get_timestamp();
16       if( time > timestamp_recv || time+TOLERANCE < timestamp_recv)
17       return VALIDATE_INVALID_TIME;
18     }
19
20     memcpy(&hmac_recv,&inbuf[PROTO_POS_HMAC],PROTO_HMACLEN);
21     memset(&inbuf[PROTO_POS_HMAC],0,PROTO_HMACLEN);
22     md5_hmac((const unsigned char *) &HMAC_KEY, sizeof(HMAC_KEY)-1,
       inbuf, datalen+PROTO_HEADERLEN, hmac_calc);
23     memcpy(&inbuf[PROTO_POS_HMAC],&hmac_recv,PROTO_HMACLEN);
24     if (memcmp(hmac_calc,hmac_recv,PROTO_HMACLEN)){
25       return VALIDATE_INVALID_HMAC;
26     }
27     return VALIDATE_SUCCESS;
28   }
29   return VALIDATE_SECURITY_DISABLED;
30 }
```

buffer, the length of the output and a pointer to the HMAC of the received message. The output buffer will be allocated on the heap, thus the procedure tries to reallocate the output buffer in line 2 in order to create space for the communication header. In the case that this reallocation fails, no further data processing takes place. Upon an successful reallocation, the original contents of the output buffer are moved backwards by the length of communication header in line 4. Afterwards, the communication header is constructed and prepended to the actual output in lines 5-7. Here, the timestamp field is set to a dummy value and the HMAC field is filled with the HMAC of the received message according to the specification of the extended NAP protocol. The data length field is used to indicate the length of the output.
The procedure does not return an error code on purpose. Even if the reallocation fails, the output data should still be sent to the base station because they are still valid and meant to be sent. Since no other kind of expected failures can occur in the function, a special error handling is not necessary.

Algorithm 5.4 Function for expanding the response

```
1  static void security_response(char **outbuf, unsigned short outlen,
       char * hmac){
2    *outbuf = realloc(*outbuf, outlen+PROTO_HEADERLEN);
3    if (*outbuf){
4      memmove(*outbuf+PROTO_HEADERLEN, *outbuf, outlen);
5      memset(*outbuf+PROTO_POS_TIMESTAMP, ".", sizeof(int32_t));
6      memcpy(*outbuf+PROTO_POS_HMAC, hmac, PROTO_HMACLEN);
7      memcpy(*outbuf+PROTO_POS_DATALEN, outlen, sizeof(unsigned short));
8    }
9    //realloc failed, dont change anything
10 }
```

5.3.4 Interface

The interface to the authentication functionalities is provided by the function *security_validate_and_respond*(). This function is designed to be called from a FreeRTOS task responsible for data input and output handling. Since an incoming data packet has not only to be validated but also processed, it provides an interface for further data processing via *callback functions*.
A callback function *cb* is a function which gets passed as argument to another function f in order to be potentially called within f. This allows a function g which calls f to specify which code shall be executed as *cb*. Therefore, it is not required to know the implementation details of *cb* during the development of function f. This allows flexibility regarding the actual implementation of *cb*.
These properties are used by the *security_validate_and_respond*() to provide interfaces for specifying how the received data shall be processed after the validation. In particular, two callback functions are defined and expected to be passed as parameters, *cb_success* and cb_fail. They shall determine the further execution flow after the validation took place. According to the name, the first callback function is executed upon successful validation of an received packet. Thus, it is declared to be of the type *int (*cb_success)(char *, char **, unsigned short)*. The callback function shall return an integer and expect three input parameters. In C, it is only possible to assign the input parameters for a callback function a type but not a name. Thus, it is additionally mentioned what the callback function expects as input: a pointer to the data location, a pointer to the pointer for output buffer and the data length. The second callback function, $int(*cb_fail)(int, char**)$, is invoked upon a failed validation of an incoming packet. Here, the function expects the specific error code and the pointer to the pointer to the output buffer as input. Besides the two callback functions, *security_validate_and_respond*() requires pointer to the input buffer and a pointer to the pointer for the output buffer as parameter. A summary of the expected input for the interface function is provided in table 5.2, where parameter for the callback functions are indented.

The double reference of the output buffer is required to ensure a lifetime of the output data over several functions. Here, it is assumed that inside the callback functions a specific memory region is allocated for the output buffer. Between the specific functions, just the pointer to the pointer of the allocated memory is passed. This allows that a called function, for instance one of the callbacks, can allocate a chunk of memory and the calling function still can determine the location of this memory chunk.

Table 5.2: Expected parameter for the interface function

Parameter	Description
char * inbuf	Pointer to the input buffer
char ** outbuf	Pointer to the pointer for the output buffer
int (*cb_success)	Pointer to the callback function for successful validation
char *	Pointer to the data buffer
char **	Pointer to the pointer for the output buffer
unsigned short	Length of the data
int (*cb_fail)	Pointer to the callback function for failed validation
int	error code
char **	Pointer to the pointer for the output buffer

5.4 Base Station

The NUTS base station will be operated with *LabVIEW* which is a graphical programming language. LabVIEW is developed by National Instruments and is widely used for instrument control systems [TK06]. One advantage of LabVIEW is its ability to easily invoke scripts written in other languages such as *Perl* and *Python* [Ins10]. Here, LabVIEW is possible to evaluate the standard output, the error output and the return code resulting from the execution of a script.
Since base station code is not existing at the time this thesis is being composed, we developed two small Perl scripts to provide the authentication scheme functionalities. This scripts are designed to be as easily integrable into LabVIEW as possible.

5.4.1 Creation of the NUTS Authentication Protocol Header

The first script as shown in algorithm 5.5 is responsible for adding the NAP header to an outgoing packet. The script takes the name of the file representing the packet, the timestamp and the HMAC key as input. It is necessary to save the packets intermediate in files because it is likely that they contain zero bytes. A zero byte is a byte where all bits are set to zero. Those bytes are not printable characters and can therefore not be represented in standard strings. Since LabVIEW invokes scripts via

the command line and the arguments are given as strings, zero bytes could get lost which would result into missing information. Therefore, it is better to save a packet in a file and read its content to preserve zero bytes.
Thus, the Perl script reads the given input file in line 11-16 into a *$data* variable. Afterwards, the NAP header is constructed in line 18-21 in order to calculate and insert the HMAC in line 23-25. Finally, the packet including the security header is saved to the hard disk and the filename is printed for further processing of the packet in LabVIEW. The filename is hereby constructed of the timestamp and the HMAC.

Algorithm 5.5 Perlscript for constructing uplink packets

```
1 #!/usr/bin/perl
2 use warnings;
3 use strict;
4 use Digest::HMAC_MD5 qw(hmac_md5 hmac_md5_hex);
5
6 my $ifilename = shift;
7 my $timestamp_int = shift;
8 my $key = shift;
9
10 my $data;
11 open FILE, "<", $ifilename or die $!;
12 {
13   local $/;
14   $data = <FILE>;
15 }
16 close FILE;
17
18 my $timestamp = pack "N",int($timestamp_int);
19 my $digest = "\0"x16;
20 my $length = pack "n", length($data);
21 my $uplink_data = $timestamp.$digest.$length.$data;
22
23 my $hmac_hex = hmac_md5_hex($uplink_data, $key);
24 $digest = hmac_md5($uplink_data, $key);
25 $uplink_data = $timestamp.$digest.$length.$data;
26
27 my $ofilename = "secured_out/$timestamp_int\_$hmac_hex";
28 open FILE, "+>", $ofilename or die $!;
29 print FILE $uplink_data;
30 close FILE;
31
32 print $ofilename;
```

5.4.2 Obtaining of HMACS

The second base station script, *get_nap_hmac.pl* is responsible to extract the HMAC of existing packets and print them into a human readable format. This is a simple task since only bytes at a static offsets must been extracted. Nevertheless, the script is important for the extended authentication protocol since the HMAC of an outgoing packet must match the HMAC of a corresponding incoming packet in order to establish mutual authentication.
Figure 5.3 shows schematic the flow of a specific message which shall be sent to the satellite. First, the file containing the message is opened and the authentication header is added. The HMAC of this message is extracted and the message itself is sent to the satellite. As soon the satellite sends a response, this is saved in an output file in order to extract the HMAC of the response. If both extracted HMAC are matching each other, the extended authentication protocol was executed successful.

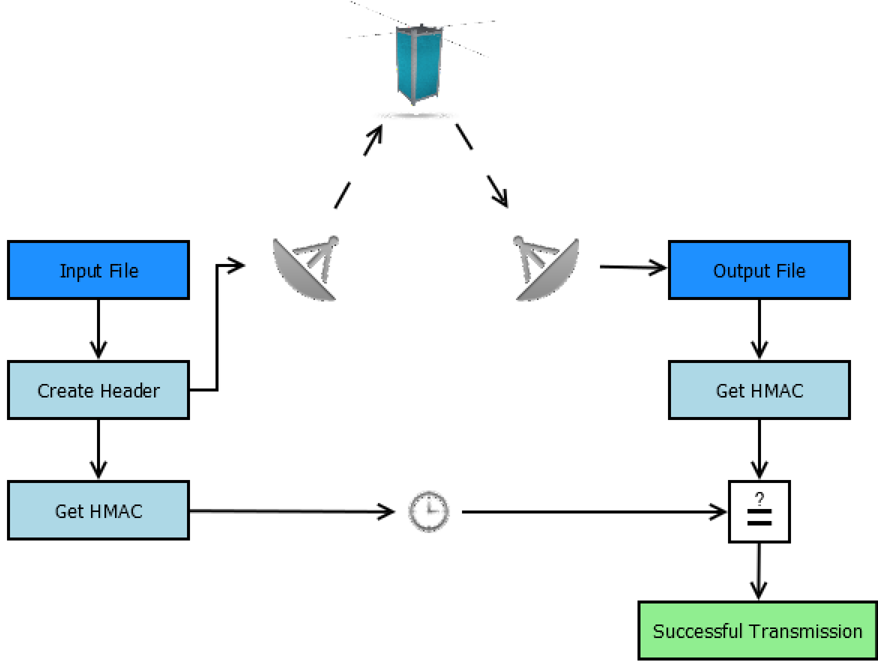


Figure 5.3: Conceptual base station program flow for the extended NUTS authentication protocol

Chapter 6
Integration

The authentication scheme must be integrated in both satellite and base station. Unfortunately, several parts of satellite and base station were not in an usable state at the time this thesis was written. Therefore, an integration beyond the design is not possible.
Nevertheless, the integration in the design had to be fulfilled in close cooperation with the different working groups inside the NUTS project. This chapter shows in which manner the authentication scheme is integrated in hardware design, software design and the protocol stack.

6.1 Hardware Design

Special hardware for the NUTS authentication scheme is only necessary for the satellite and the required elements are already integrated in the satellites hardware design. Namely, these elements are the RTC, the I²C bus, the radio MCU, the beacon and transceivers.
The only additional circuit which was designed for the hardware due to the authentication scheme is a fallback power circuit for the RTC. The RTC would loose the current time as soon its not powered. Since the satellite is expected to do a power cycle at least once per day, this additional circuit is needed in order to achieve clock reliability.
The designed circuit is shown in figure 6.1. Here, the DS3231SN component is the RTC. The other three components are dual ideal diodes with adjustable current limit of type LTC4415. For these components, OUT1 is enabled when EN1's voltage is higher than a certain treshhold and OUT2 is enabled when EN2's voltage is lower than this treshhold [Lin12]. Thus, as soon the regular power supply VCC is switched off, $VBAT$ is provided for the RTC. Additionally, the output of the I²C data and clock connections, SDA and SCL, is blocked by the diodes as soon VCC is switched off. The reason for this is that during a power cycle no current on other components than the RTC is desirable. Thus, a feedback of the SDA or SCL signal to the

main circuit during the power cycle can cause unexpected behavior and is therefore circumvented.

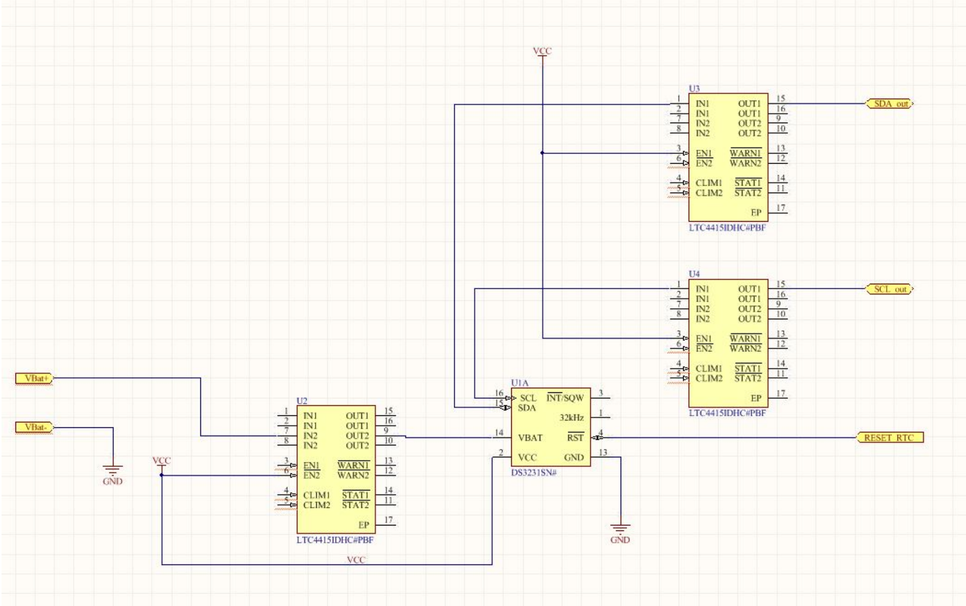


Figure 6.1: Protection circuit for the RTC (figure provided by the NUTS hardware group)

6.2 Software Design

Since the software design for the base station is in an early stage with ongoing changes, a finalized integration of the authentication scheme could only be established for the satellite software design as shown in figure 6.2. Here, the authentication functionalities are located in the radio MCU in form of an independent module. Logically, the authentication scheme requires access to the RTC and the I²C bus. Correspondingly, the authentication module is able to utilize the RTC driver which includes the I²C driver. Below the authentication scheme, the radio encode and decode functionalities for received and sent signals are located in order to provide access to the radio link. The authentication module itself must provide an interface for the radio interface which takes care of further processing of the data. This interface is provided in the form of callback functions as described in section 5.3.4.

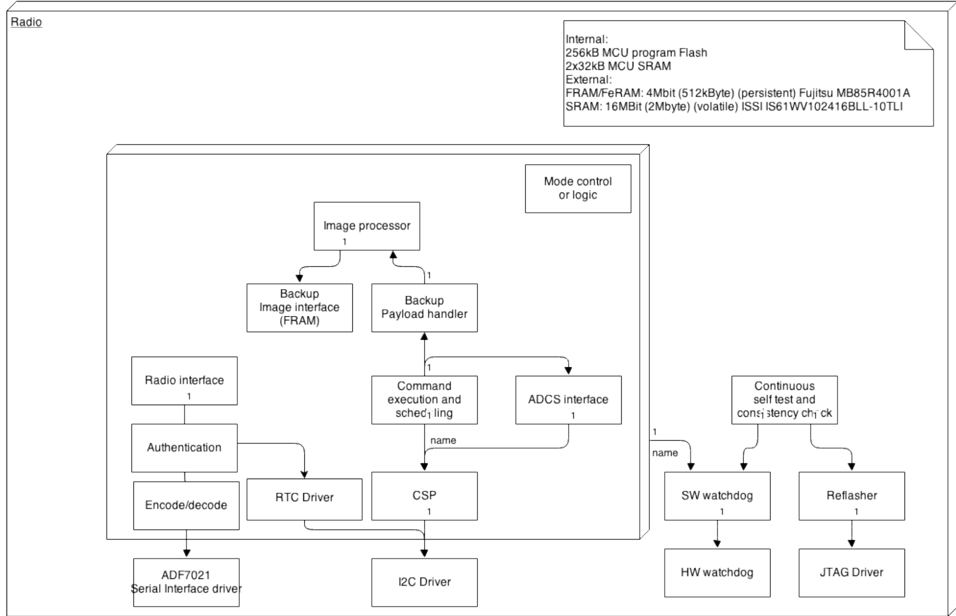


Figure 6.2: Overview of the NUTS Radio Software Design (figure provided by the NUTS software group)

6.3 Protocol Stack

The placement of the communication protocol required for the NAP is well defined and intuitively comprehensible.
The actual validation of received messages takes place before messages are forwarded to other satellite components. Thus, the protocol is directly positioned above the link layer protocol. Together with the CubeSat Space Protocol (CSP) and the NUTS application layer protocol, the protocol stack for the NUTS satellite can be visualized like shown in figure 6.3.

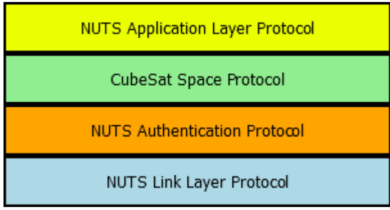


Figure 6.3: NUTS protocol stack

Chapter 7

Test Suite

The authentication scheme aims to establish a secure satellite uplink. Thus, its applicability should also demonstrated in the space environment. This is not possible at the time this thesis is written but can be done as soon the NUTS satellite is launched. Therefore, a test suite is elaborated and presented in this chapter. Hence, a specific set of test cases is established so that the capabilities of the scheme's implementation can be evaluated easily.
Additionally, a laboratory test environment is presented which is used to run the specific tests before the satellite is launched. This fulfills two purposes: First, it can be evaluated if the implementation of the scheme is working in general. Second, reference values for an in-space evaluation are established.

7.1 Laboratory Test Environment

The established laboratory test environment consists of the AVR UC3-A3 evaluation board and two PCs. The UC3-A3 contains the implementation of the authentication scheme for the satellite. Additionally, two FreeRTOS tasks have been developed in order to provide necessary input and output functionalities for testing. The first task, *simulation_beacon_task*() , is frequently checking the time of the RTC and outputs the timestamp. The second task, $simulation_{l}ink_{t}ask()$ listens for incoming messages and calls the *validate_and_respond*() function of the authentication scheme's implementation. The provided callback functions are just writing the result of the validation to the output buffer which is afterwards printed. Additionally, within the task for the link simulation useful debugging information are printed.
The input and output functions for the AVR UC3-A3 are provided by the *ATMEL Universal Data Bus (USB) Standard Input/Output (I/O) driver*. This driver emulates a virtual serial connection via USB and redirect the standard C I/O functions to this connection. Thus, a computer can establish a serial connection via USB to the evaluation kit in order to access its input and output.
For the laboratory test environment, this is done by the first PC. The purpose of

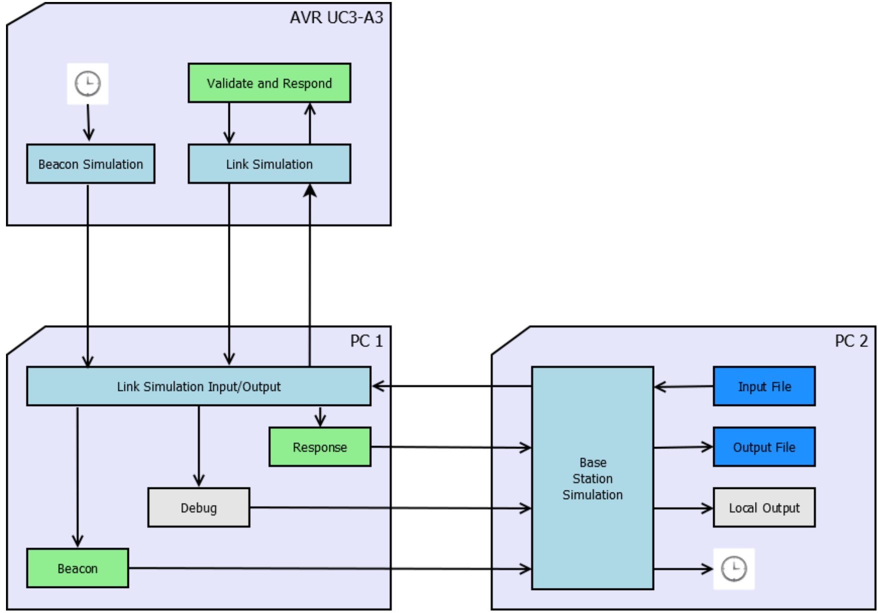


Figure 7.1: Schematic overview over the test environment

this PC is to provide a middleware between the link simulation and the base station simulation which is carried out by the *nap_middleware.pl* Perl script. Additional middleware is required in order to emulate the behavior of the different communication channels. Thus, the middleware establishes a virtual serial connection to the evaluation board and listens on three different ports representing the data uplink, data downlink and beacon downlink channel. As soon a connection is present on every port, the middleware starts to receive and forward messages to their destination without any further processing of the data. Debug output of the *simulation_link_task*() is treated like downlink data if the reception of the debug information is desired.
The second PC simulates a base station with the *nap_base_station_simulation.pl* Perl script. The script establishes connections to the different middleware ports and keeps track of the time received on the beacon downlink channel. Furthermore, it invokes the developed base station scripts in order to send authenticated packets and validate the fingerprint of received packets. Figure 7.1 illustrates the interaction of the components in the laboratory test environment.
In order to do quantified testing with a lot of packets containing different data, an additional *nap_packetgenerator.pl* Perl script is separately provided. The purpose of the script is to generate an amount of n packages with the length L containing random data.

7.2 Functional Testing

The purpose of functional testing is the validation that specific functions are fulfilling their purpose. This is a form of black-box testing and done by feeding the particular functions with input and evaluating the resulting output [KFN00].
For testing the implementation of the authentication scheme in space, the complete authentication procedure is abstracted to one function. The reason for this is that during in-space operation only the interface for the authentication functionalities should be accessed due to the security reason. Additional, it can be said that the authentication procedure is one specific function of the NUTS satellite.
The first test case F1 shall demonstrate, that the implementation of the authentication scheme recognizes authenticated packets. Correspondingly, packets which are not fulfilling the authentication requirements have to be correctly identified by the implementation. Thus, test case F2 evaluates that packets with an incorrect HMAC are detected correctly, while test case F3 examines the outcome of packets with incorrect timestamps. Additionally, it is important to verify that the data length field is correctly evaluated. To demonstrate the need of this evaluation, CVE-2009-2415 [UC09] is taken as example. Here, an attacker could achieve remote code execution on servers running *memcached* version 1.1.12 and 1.2.2 by introducing a malicious value for the data length field which results into an heap-based buffer overflows. Thus, test case F4 determines if the data length field is evaluated correctly.
Last but not least, it has to be verified that the extended authentication scheme is also working which is done in test case F5.

Test Case F1 - Validation of authenticated packets

Input Specification

For every operation mode, packets with following components have to be provided:

1. A timestamp within the tolerance window.
2. A HMAC constructed over the full message with the correct key.
3. A correct specified data length.
4. Arbitrary data.

Expected Output

- A response packet indicating that the authenticity of the sent packet was successful validated.

Laboratory results

The output received under laboratory conditions is matching the expected output.

Test Case F2 - Validation of packets with an incorrect HMAC

Input Specification

For every operation mode, packets with following components have to be provided:

1. A timestamp within the tolerance window.
2. An arbitrary invalid HMAC.
3. A correct specified data length.
4. Arbitrary data.

Expected Output

- For the operation mode `SECURITY_MODE_OFF`, a response packet indicating that the authentication functionality is turned off.
- For any other other mode, a response packet indicating that the authenticity of the packet could not be validated due to an incorrect HMAC.

Laboratory results

The output received under laboratory conditions is matching the expected output.

Test Case F3 - Validation of packets with an incorrect timestamp

Input Specification

For every operation mode, packets with following components have to be provided:

1. A timestamp outside the tolerance window.
2. A valid or invalid HMAC
3. A correct specified data length.
4. Arbitrary data.

Expected Output

- For the operation mode `SECURITY_MODE_OFF`, a response packet indicating that the authentication functionality is turned off.
- For the operation mode `SECURITY_MODE_NO_TIME` and a correct constructed HMAC, a response packet indicating that the authenticity of the sent packet was successful validated.
- For the operation mode `SECURITY_MODE_NO_TIME` and an invalid HMAC, a response packet indicating that the authenticity of the packet could not be validated due to an incorrect HMAC.
- For any other other mode, a response packet indicating that the authenticity of the packet could not be validated due to an incorrect timestamp.

Laboratory results

The output received under laboratory conditions is matching the expected output.

Test Case F4 - Validation of packets with an incorrect data length value

Input Specification

For every operation mode, packets with following components have to be provided:

1. A timestamp within the tolerance window.
2. A HMAC constructed over the full message with the correct key.
3. A incorrect specified data length.
4. Arbitrary data.

Expected Output

– For the operation mode `SECURITY_MODE_OFF`, a response packet indicating that the authentication functionality is turned off.

– For any other operation mode and an incorrect data length value between 1 and the maximal allowed data length, a response packet indicating that the authenticity of the packet could not be validated due to an incorrect HMAC.

– For an incorrect data length value outside this range, a response packet indicating that the specified data length value is incorrect.

Laboratory results

The output received under laboratory conditions is matching the expected output.

Test Case F5 - Validation of correct fingerprint construction

Input Specification

For the operation mode `SECURITY_MODE_EXTENDED_AUTH`, packets with following components have to be provided:

1. A timestamp within the tolerance window.
2. A HMAC constructed over the full message with the correct key.
3. A correct specified data length.
4. Arbitrary data.

Expected Output

– A response packet containing the fingerprint of the sent message and indicating that the authenticity of the packet was successful.

Laboratory results

The output received under laboratory conditions is not matching the expected output in all cases. The number of matching fingerprints is shown in fig 7.2 for 10 distinct runs of the test with 100 sent packages each.

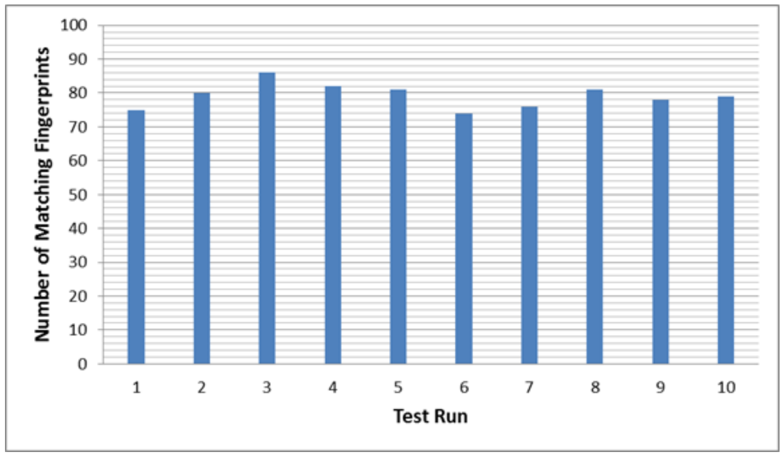


Figure 7.2: Test results for the validation of correct fingerprint construction

Discussion of laboratory results

The reason for the laboratory results is easy explainable. The *simulation_link_task()* utilizes $printf()$ in order to provide a response as string to the middleware. The $printf()$ function stops the printing of a string as soon a zerobyte is reached. The middleware sends a response to the base station simulation when a newline is read and the base station simulation stops printing to a file as soon an EOF character is read.

This sums up to three distinct bytes which would inhibit that a fingerprint is saved correctly to the hard disk. Since the fingerprint is 16 bytes long, the possibility that one of the disruptive bytes occur amounts to 18.75% which matches the results of figure 7.2.

It is not likely that these bytes are causing errors during in-space operation due to the specialized link layer and the functionality of *LabVIEW* to write a bytestream to a file. Both of them will be able to circumvent the errors observed in the laboratory environment for this test case.

7.3 Non-Functional Testing

Contrary to functional testing, non-functional testing does not aim to evaluate a specific function. Instead, the main purpose of non-functional testing is to determine different quality factors of the system.
Test case P is designated to show the performance of the authentication scheme. Hereby, the actual execution time is evaluated. This time might be noticeable higher during in-space operation compared to the times achieved under laboratory conditions. This relies on the fact, that under laboratory conditions no other tasks are concurrently executed while the MCU's workload will be significantly higher during in-space operation.
The last test case R evaluates the effectivity of the replay protection provided by broadcast timestamp. This evaluation is necessary in order to show the influence of the chosen timestamp tolerance window to the authentication scheme.

Test Case P - Performance Evaluation

Input Specification

For every operation mode, an amount N of authenticated packets with different lengths in the range from minimum packet size to maximum packet size.

Expected Output

- Different characteristic curves for the authentication modes.
- Longer evaluation time for higher operation modes in the operation mode hierarchy *(cf. figure 5.2)*.
- Longer evaluation time for larger packets.

Laboratory results

The output received under laboratory conditions is matching the expected output in almost all cases. The results are visualized in figure 7.3 where the packet validation time inside the satelliteis set into context with the packet length for different operation modes.

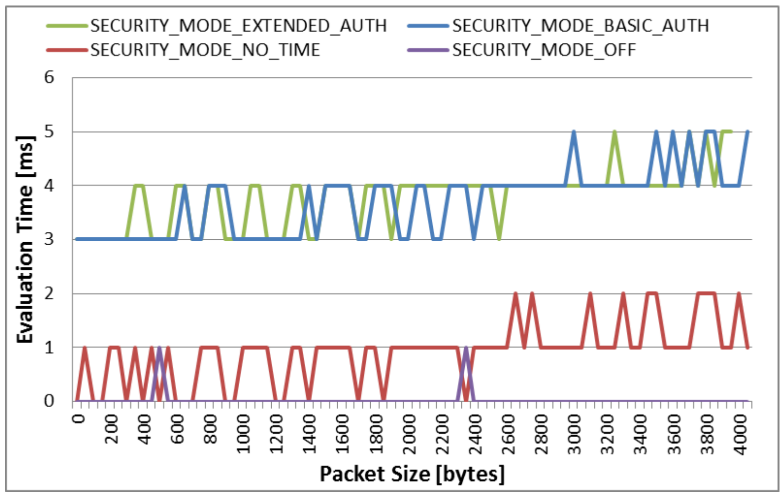


Figure 7.3: Test results for evaluating the performance of the implementation

Discussion of laboratory results

In general, the validation of a received packet does not take longer than 6 ms. This is satisfyable low, compared with the delay times for up and downlink. The validation of a packet is in fact longer for larger packet sizes when a authenticity valuation is executed.

It is easy to see that the curves for the basic and extended authentication mode are within the same range. Thus, the extended authentication scheme does not require significantly more validation time than the basic authentication scheme. This contradicts the expected results but is easy explainable. In fact, the computational overhead for the extended scheme is very low since no expensive computation has to be done. Instead, only the memory overhead is increasing minimally because an additional copy of the fingerprint has to be kept in memory and appended to a outgoing packet.

Another interesting observation established by the test is the almost 2 ms difference between `SECURITY_MODE_NO_TIME` and the two higher authentication mode. This difference is obviously resulting from the need to access the RTC. Therefore, it can be concluded that the access time to the clock is a bottleneck in the implementation.

Test Case R - Effectivity of the replay protection

Input Specification
For SECURITY_MODE_BASIC_AUTH or SECURITY_MODE_EXTENDED_AUTH an authenticated packet with length L is sent to the satellite and replayed until the satellite reject the packet as invalid. The packet length L has to be in the range from minimum packet size to maximum packet size.

Expected Output

- A huge amount of successful replays for packets with a small length L which is decreasing with increasing values for L

Laboratory results
The output received under laboratory conditions is matching the expected output. The results are visualized in figure 7.4 and will be discussed in section 8.1.

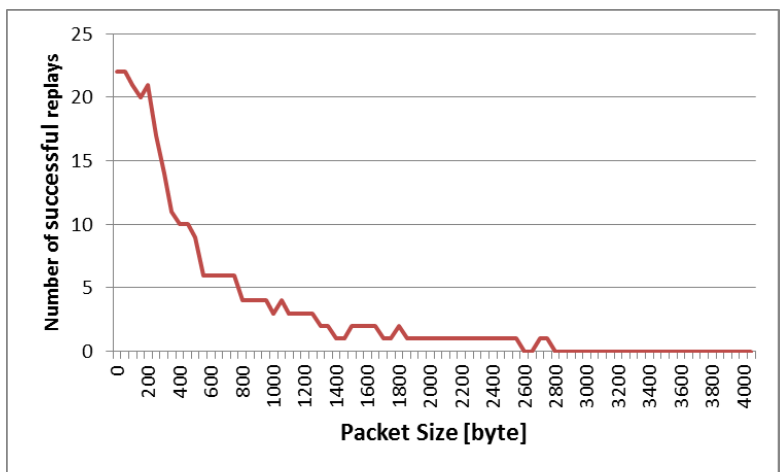


Figure 7.4: Test results for the validation of the effectivity of the replay protection

Discussion

The authentition scheme was successfully verified, implemented and tested. During this process, it was shown, that the authentication scheme has minor formal flaws and a small vulnerability window for replay attacks does exist. Therefore, the impacts of this weaknesses are discussed in this chapter in order to show whether a application of the scheme is advisable or not.
Additionally, the scope of the integration is elaborated since the initially planned integration on the satellite software could not be established.

8.1 Effectivity of Replay Protection

The main reason for using timestamps in the first place is to guarantee replay prevention. Hence, it should not be possible to have two valid messages with the same timestamp. Contrary to this, it has been shown in section 3.4.3 that the tolerance in which a timestamp is considered fresh has to be rather large with 8 seconds. Therefore, an attack window for replay attacks remains which could be verified with the test case reftc:r.
The authentication scheme is not able to detect a replay attack when an attacker is able to intercept and replay a certain packet p within the tolerance window. This allows the attacker to force additional executions of the command contained in p as long the packet length of p is smaller than 2800 bytes. This is not a threat as long the command does not change the state of the satellite upon repeated execution. However, in some cases the re-execution of a certain command can compromise the satellite state. To clarify this, the following example is given:

Example 8.1: Assuming a variable v with an initial value of 0 which can be changed by an function f. Upon receiving a command c, the function f is executed and v may be compromised depending of the implementation of f. The safe implementation of f changes v only according to the passed parameter to f. An exemplary definition

of such a function is $f(X) = X$. This would set $v = 1$ upon calling $f(1)$ which gets not changed upon re-execution of f.
A vulnerable implementation of f would change the value of v according to the passed parameter to f and the current state of v. An exemplary definition of such an function is $f(X) = v + X$. This would set $v = 1$ upon calling $f(1)$ and $v = 2$ upon re-execution etc.

In an complex software system the likelihood that an according vulnerable function is invoked upon receiving a command is rather large. Thus, an attack on the authentication scheme does exist due to temporal side effects introduced by the usage of timestamps in a system with long delays. The impact of the attack depends on the specific vulnerable functions and can therefore not be predicted as for now.
The risk that such an attack is executed is rather low because several conditions have to apply at the same time. First of all, a vulnerable function must exist and a packet with a corresponding command sent. Additionally, the packet has to be short since the reception of it is a blocking process on the satellite. If the packet exceeds a certain length, its reception requires enough time to make a replay of the packet within the same tolerance window impossible. Furthermore, the attacker has to have access to a base station in geographical proximity to a legitimate base station since the attack window could not be utilized otherwise. Thus, it can be concluded that the application of the presented authentication scheme is still reasonable secure.
Additionally, it has to be mentioned that the vulnerability can be fixed by introducing a required minimum packet size. The minimum packet size must establish that a reception of two packets with minimum size is not possible within the tolerance window minus the timestamp downlink delay. Unfortunately, this fix can not be applied for this thesis since the specification of the NUTS link layer protocol is still under development.

8.2 Results of the formal verification

The formal verification has shown that specific flaws are existing in the protocol. However, even with these flaws the NAP protocols are still fulfilling their main purpose.
The basic authentication protocol guarantees that the base station authenticates itself to the satellite in such a way, that the sent data can not be altered by the adversary *(non-injective agreement)*. Nevertheless, it has been confirmed that a received response from the satellite does not necessarily corresponds to a base station's run of the protocol *(weak agreement)* for the basic NAP Protocol. Therefore, a response can also not be resolved to one specific message *(non-injective agreement)*. These issues must be addressed on a higher communication layer when the basic NAP protocol

is used, since it is necessary for the base station to recognize to which message a response belongs to.
The extended NAP protocol establishes non-injective agreement for both roles and is therefore fulfilling its purpose. Unfortunately, it increases the communicational overhead on the downlink. Since the downlink capacities are rather restricted, the extended protocol might not be utilized for NUTS.
The major flaw which has been found by formal verification is the lack of *synchronization* between base station and satellite. This is, however, not a threat for the application of the protocol, since even if data are exchanged in a non-synchronized manner, it can be guaranteed that the data received on the satellite are originated from an authenticated source. This is the main aim of the NUTS authentication protcol and the verification of this security properties makes the application of the protocol more reasonable. However, it should not be forgotten that we only verified a model of the protocol and that the security problem is an undecidable question. Thus, the formal verification does not guarantee the absolute security of the protocol but helped to determine and understand its boundaries. Last but not least, it has to be mentioned that the verification relies completely on the environment assumptions of section 3.2. Without these assumptions, the elaborated models would be wrong and the established verification therefore invalid.

8.3 Scope of the Integration

The authentication scheme could be successfully integrated in the design of the NUTS satellite. Unfortunately, it was neither possible to integrate the authentication scheme in the actual base station nor satellite code. The reason for this is that broad parts of both satellite and base station software were missing while this thesis was carried out. Additionally, the radio links were not ready to use and the NUTS link layer protocol was not finalized.
Therefore, an integration beyond the conceptual design was not possible because functionalities for both delivering messages of the authentication scheme and processing validated messages were lacking.
However, even if the NUTS satellite is not ready yet, the authentication scheme has been implemented on hardware similar to the satellite's hardware. Additional, link delays and properties could be simulated in a laboratory environment. It has also be mentioned that the established interfaces for an integration are not limited to the NUTS project but rather usable for any CubeSat with a RTC and customized radio software.
Thus, this thesis provides all in all a contribution towards a security solution for the operational uplink of small satellites communicating in the amateur radio band.

Chapter 9 Future Work

This thesis established an authentication scheme for the NUTS project along with a formal verification, an implementation ready for integration and a test suite for in-space operation.
Nevertheless, additional work is required which is beyond the scope of this thesis in order to finalize the evaluation of the scheme's applicability. The required future work for this is elaborated in this chapter.

9.1 Integration to the Final System

As pointed out in section 8.3 an integration of the implementation was only possible in a conceptual manner. Thus, a complete integration for the satellite and base station is still required and must be carried out before the satellite is launched.
It is advisable to re-run the test suite on the finalized integration in order to discover flaws resulting from the interaction of different software components and the presence of a realistic physical link layer.

9.2 Radiation Hardness Testing

The proposed implementation of the authentication scheme has only been tested in regard to its functionality and its performance on earth. For operation in space it has to be assured that the scheme is also working while being exposed to space radiation effects. This was not studied by this thesis since underlying hard- and software of the satellite will provide measures against these effects. Nevertheless, it is advisable to test the radiation hardness of the scheme under laboratory conditions as soon it is integrated to the satellite software.

9.3 Evaluation of Performance in Space

The provided test suite aims to provide a framework for evaluating the authentication scheme in space. Logically, the specific tests have to be run and analyzed after the launch of the satellite. By now, the correctness of the authentication scheme can only be assured theoretically. Therefore, running the tests in a space environment is beneficial for evaluating the practical applicability of the scheme and its application to future missions

9.4 Key Management

The HMAC construction and verification relies on a shared key for both satellite and base station. Since bit flips can occur in the satellite's memory due to space radiation effects, the possibility that a saved key is changed does exist. Thus, additional key management is required as basis for the authentication scheme. The key management must in particular establish safety for the stored keys.
Additionally, a secured re-keying functionality could be considered in order to deal with a compromised key.

Chapter 10
Conclusions

The main motivation for this thesis was to integrate and verify an authentication scheme usable for the NUTS satellite. In the first chapter, specific tasks were defined in order to satisfy this motivation. The great majority of the defined tasks could be fulfilled.

The authentication scheme was prior to this work only loosely defined which is changed by this thesis. The third chapter provides a detailed elaboration of the specific authentication scheme used for NUTS.

A formal verification could be established for the underlying cryptographic protocol in chapter three. The verification showed that the protocol is not perfect but fulfills its purpose entirely. Thus, it was implemented on hardware with a similar architecture and computational restraints compared to the satellite.

This implementation has been integrated to the design process of the NUTS satellite. Unfortunately, an actual integration of the developed satellite and base station software was not realizable. This relies on the fact that other subsystems providing necessary functionalities for the authentication scheme were not in a usable state at the point this thesis was written. Therefore, the implementation of the authentication scheme was carried out in such a way that it can easily be integrated onto the base station and satellite as soon as the required functionalities are available. The hereby established necessity of easy usable interfaces comes along with another beneficial fact: the specific implementation is not only restricted to the NUTS satellite. Instead, it can also be easily integrated by other CubeSat projects.

Nevertheless, experiences with an in-space operation of the authentication scheme should be gathered first. Therefore, a test suite for the scheme has been elaborated in chapter seven. This tests are developed to be run once the satellite is in space in order to evaluate the scheme. The complete test suite had also been run against the existing implementation to verify that the implementation is working under laboratory conditions.

All in all it has been shown that reasonable security for satellite uplinks can be established even without the traditional usage of encryption. Nothing stands in the way of integrating the presented authentication scheme to the NUTS satellite. Therefore, the theoretical established security of the scheme might be supported with practical results as soon the satellite is launched.

References

[AADH98] I. Ali, N. Al-Dhahir, and J.E. Hershey. Doppler characterization for leo satellites. *Communications, IEEE Transactions on*, 46(3):309–313, 1998.

[ABB+05] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, others, J. Cuéllar, P. H. Drielsma, P. Héam, O. Kouchnarenko, J. Mantovani, et al. The avispa tool for the automated validation of internet security protocols and applications. In *Computer Aided Verification*, pages 281–285. Springer, 2005.

[Atm12] Atmel Cooperation, San Jose. *AT32UC3A3/A4 Series Complete*, 2012.

[Atm13] Atmel Cooperation, San Jose. *Release Notes - Atmel Studio 6.1*, 2013.

[Bel06] M. Bellare. New proofs for nmac and hmac: Security without collision-resistance. In *Advances in Cryptology-CRYPTO 2006*, pages 602–619. Springer, 2006.

[BF12] B. Bezem and P. K. J. Fjellby. Authenticated uplink for the small, low orbit student satellite NUTS. Project work, Department of Telematics, Trondheim, 2012.

[Bla07] B. Blanchet. CryptoVerif: A computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied"*, October 2007.

[CDL06] V. Cortier, S. Delaune, and P. Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.

[CLN09] C. J. F. Cremers, P. Lafourcade, and P. Nadeau. Comparing state spaces in automatic protocol analysis. In *Formal to Practical Security*, volume 5458/2009 of *Lecture Notes in Computer Science*, pages 70–94. Springer Berlin / Heidelberg, 2009.

[CLT03] Hubert Comon-Lundh and Ralf Treinen. Easy intruder deductions. In *Verification: Theory and Practice*, pages 225–242. Springer, 2003.

[CM12] C. J. F. Cremers and S. Mauw. *Operational semantics and verification of security protocols*. Springer, 2012.

[CMdV06] C. J. F. Cremers, S. Mauw, and E. P. de Vink. Injective synchronisation: an extension of the authentication hierarchy. *Theoretical Computer Science*, 367(1):139–161, 2006.

[Coo09] Atmel Cooperation. Avr dragon. http://www.atmel.com/tools/avrdragon.aspx, 2009. [Online - accessed 28-December-2013].

[Cre08a] C. J. F. Cremers. The Scyther Tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, USA, Proc.*, volume 5123/2008 of *Lecture Notes in Computer Science*, pages 414–418. Springer, 2008.

[Cre08b] C. J. F. Cremers. Unbounded verification, falsification, and characterization of security protocols by pattern refinement. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 119–128. ACM, 2008.

[Cre13a] C. J. F. Cremers. The scyther tool. http://www.cs.ox.ac.uk/people/cas.cremers/scyther/index.html, 2013. [Online - accessed 9-June-2014].

[Cre13b] C. J.F. Cremers. The scyther tool - exercise set. http://www.cs.ox.ac.uk/people/cas.cremers/scyther/scyther-exercises.html, 2013. [Online - accessed 10-June-2014].

[Cre14] C. J. F. Cremers. *Scyther User Manual - Draft February 18, 2014*. Oxford, 2014.

[CV02] Y. Chevalier and L. Vigneron. Automated unbounded verification of security protocols. In *Computer Aided Verification*, pages 324–337. Springer, 2002.

[DY83] D. Dolev and A. C. Yao. On the security of public key protocols. *Information Theory, IEEE Transactions on*, 29(2):198–208, 1983.

[EB06] R. Birkeland E. Blom, E. Technical satellite specification. Technical report, 2006.

[EG83] S. Even and O. Goldreich. On the security of multi-party ping-pong protocols. In *Foundations of Computer Science, 1983., 24th Annual Symposium on*, pages 34–39, Nov 1983.

[Gom12] GomSpace. Cubesat space protocol - a small network-layer delivery protocol designed for cubesats. http://www.libcsp.org/, 2012. [Online - accessed 20-May-2014].

[HKC97] M. Bellare H. Krawczyk and R. Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, Internet Engineering Task Force, 1997.

[IB03] H. Irazabal and S. Blozis. *I^2C Manual.* Philips Semiconductors, 2003.

[Ins10] National Instruments. Call perl and python scripts from labview. http://www.ni.com/white-paper/8493/en/pdf, 2010. [Online - accessed 19-June-2014].

[ITU12] ITU. Radio regulations articles. Technical report, 2012.

[JMT+13] D. Jonassen, S. Mikkelsen, M. Teilgård, V. Edvardsen, M. H. Arnesen, and H. L. Andersen. Systemovervåkning i studentsatellitt. Project work, Department of Electronics and Telecommunications, Trondheim, 2013.

[KFN00] C. Kaner, J. Falk, and H. Q. Nguyen. *Testing Computer Software Second Edition*. Dreamtech Press, 2000.

[LB92] K. Y. Lam and T. Beth. *Timely authentication in distributed systems*. Springer, 1992.

[Lin12] Linear Technology, Milpitas. *LTC4415 - Dual 4A Ideal Diodes with Adjustable Current Limit*, 2012.

[Low97] G. Lowe. A hierarchy of authentication specifications. In *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pages 31–43. IEEE, 1997.

[Max13] Maxim Integrated, San Jose. *DS3231 - Extremely Accurate I2C-integrated RTC/TCXO/Crystal*, 2013.

[Mea03] C. Meadows. Formal methods for cryptographic protocol analysis: Emerging issues and trends. *Selected Areas in Communications, IEEE Journal on*, 21(1):44–54, 2003.

[Men13] B. L. E. Mendez. Link Budget for NTNU Test Satellite. Master's thesis, Norwegian University of Science and Technology, Trondheim, 2013.

[Mjø11] S. F. Mjølsnes. Cryptographic protocols. In *A Multidisciplinary Introduction to Information Security*, pages 93–112. Chapman & Hall/CRC, 2011.

[MSDL99] J. Mitchell, A. Scedrov, N. Durgin, and P. Lincoln. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols*. Citeseer, 1999.

[Mü14] M. Münch. Development of a security module for the uplink of the nuts student satellite. Technical report, Department of Telematics, 2014.

[NAS96] NASA. Space radiation effects on electronic components in low earth orbit. Practice No. PD-ED-1258, 1996.

[NS78] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[NS93] B. C. Neumann and S. G. Stubblebine. A Note on the Use of Timestamps aus Nonces. Technical report, Information Sciences Department, Los Angeles, 1993.

[Pra12] S. Prasai. Access control of NUTS uplink. Master's thesis, Norwegian University of Science and Technology, Trondheim, 2012.

[PvO95] B. Preneel and P. C. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *Advances in Cryptology — CRYPT0' 95*. Springer, Berlin, 1995.

[SA09] Y. Sasaki and K. Aoki. Finding preimages in full md5 faster than exhaustive search. In *Advances in Cryptology-EUROCRYPT 2009*, pages 134–152. Springer, 2009.

[Sin02] B. Sintay. Unix time stamp . com. http://www.unixtimestamp.com, 2002. [Online - accessed 21-June-2014].

[Sop12] D. S. Soper. Satellite encryption. In J. R. Vacca, editor, *Computer and information security handbook*. Newnes, 2012.

[Sta11] W. Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, 2011.

[TK06] J. Travis and J. Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun (National Instruments Virtual Instrumentation Series)*. Prentice Hall PTR, 2006.

[UC09] US-CERT/NIST. Vulnerability summary for cve-2009-2415. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2009-2415, 2009. [Online - accessed 10-July-2014].

[Vis11] V. Visockas. Access control and securing of the NUTS uplink. Project work, Department of Telematics, Trondheim, 2011.

[WY05] X. Wang and H. Yu. How to break md5 and other hash functions. In *Advances in Cryptology–EUROCRYPT 2005*, pages 19–35. Springer, 2005.

Appendix A
SDPL Files

A.1 coarse_timestamps.sdpl

```
hashfunction hash;
usertype Msg;
usertype T;
protocol NAP(B,S)
{
  const m: Msg;
  const r: Msg;

  role B
  {
    var t: T;

    recv_1(S,B,{t}sk(S));
    send_2(B,S,m,t,hash(m,t,k(B,S)));
    recv_3(S,B,{r}sk(S));

    claim_b5(B,Secret,k(B,S));
  }

  role S
  {
    var t: T;

    recv_!T1(S, S, t);
    send_1(S,B,{t}sk(S));
    recv_2(B,S,m,t,hash(m,t,k(B,S)));
    send_3(S,B,{r}sk(S));

    claim_s1(S,Alive);
    claim_s2(S,Weakagree);
    claim_s3(S,Niagree);
    claim_s4(S,Nisynch);
    claim_s5(S,Secret,k(B,S));
  }
}
```

A.2 coarse_timestamps_extended_auth.sdpl

```
hashfunction hash;
usertype Msg;
usertype T;

protocol NAP(B,S)
{
  const m: Msg;
  const r: Msg;

  role B
  {
    var t: T;

    recv_1(S,B,{t}sk(S));
    send_2(B,S,m,t,hash(m,t,k(B,S)));
    recv_3(S,B,{r,hash(m,t,k(B,S))}sk(S));

    claim_b1(B,Alive);
    claim_b2(B,Weakagree);
    claim_b3(B,Niagree);
    claim_b4(B,Nisynch);
    claim_b5(B,Secret,k(B,S));
  }

  role S
  {
    var t: T;

    recv_!T1(S, S, t);
    send_1(S,B,{t}sk(S));
    recv_2(B,S,m,t,hash(m,t,k(B,S)));
    send_3(S,B,{r,hash(m,t,k(B,S))}sk(S));


    claim_s1(S,Alive);
    claim_s2(S,Weakagree);
    claim_s3(S,Niagree);
    claim_s4(S,Nisynch);
    claim_s5(S,Secret,k(B,S));
  }
}
```

A.3 precise_timestamps.sdpl

```
1  hashfunction hash;
2  usertype Msg;
3  usertype T;
4
5  protocol NAP(B,S)
6  {
7    const m: Msg;
8    const r: Msg;
9
10   role B
11   {
12     var t: T;
13
14     recv_1(S,B,{t}sk(S));
15     send_2(B,S,m,t,hash(m,t,k(B,S)));
16     recv_3(S,B,{r,hash(m,t,k(B,S))}sk(S));
17
18     claim_b1(B,Alive);
19     claim_b2(B,Weakagree);
20     claim_b3(B,Niagree);
21     claim_b4(B,Nisynch);
22     claim_b5(B,Secret,k(B,S));
23   }
24
25   role S
26   {
27     fresh t: T;
28
29     send_!T1(S, S, t);
30     send_1(S,B,{t}sk(S));
31     recv_2(B,S,m,t,hash(m,t,k(B,S)));
32     send_3(S,B,{r,hash(m,t,k(B,S))}sk(S));
33
34     claim_s1(S,Alive);
35     claim_s2(S,Weakagree);
36     claim_s3(S,Niagree);
37     claim_s4(S,Nisynch);
38     claim_s5(S,Secret,k(B,S));
39   }
40 }
```

A.4 precise_timestamps_extended_auth.sdpl

```
hashfunction hash;
usertype Msg;
usertype Timestamp;

protocol NAP(B,S)
{
  const m: Msg;
  const r: Msg;

  role B
  {
    var t: Timestamp;

    recv_1(S,B,{t}sk(S));
    send_2(B,S,m,t,hash(m,t,k(B,S)));
    recv_3(S,B,{r,hash(m,t,k(B,S))}sk(S));

    claim_b1(B,Alive);
    claim_b2(B,Weakagree);
    claim_b3(B,Niagree);
    claim_b4(B,Nisynch);
    claim_b5(B,Secret,k(B,S));
  }

  role S
  {
    fresh t: Timestamp;

    send_!T1(S, S, t);
    send_1(S,B,{t}sk(S));
    recv_2(B,S,m,t,hash(m,t,k(B,S)));
    send_3(S,B,{r,hash(m,t,k(B,S))}sk(S));

    claim_s1(S,Alive);
    claim_s2(S,Weakagree);
    claim_s3(S,Niagree);
    claim_s4(S,Nisynch);
    claim_s5(S,Secret,k(B,S));
  }
}
```

Appendix B

Satellite Code

B.1 security.h

```
#ifndef SECURTY_H_
#define SECURTY_H_

#define VALIDATE_INVALID_TIME -1
#define VALIDATE_INVALID_HMAC -2
#define VALIDATE_INVALID_DATALEN -3
#define VALIDATE_SECURITY_DISABLED -4
#define VALIDATE_SUCCESS 1
#define VALIDATE_AND_RESPOND_OUT_OF_MEMORY -5

#define HMAC_KEY "ABCD"
#define TOLERANCE 8
#define MAX_PACKET_SIZE 4096

#define SECURITY_MODE_OFF 0
#define SECURITY_MODE_NO_TIME 1
#define SECURITY_MODE_BASIC_AUTH 2
#define SECURITY_MODE_EXTENDED_AUTH 3

#define PROTO_POS_TIMESTAMP 0
#define PROTO_POS_HMAC 4
#define PROTO_POS_DATALEN 20
#define PROTO_HEADERLEN 22
#define PROTO_HMACLEN 16

void security_init(int init_mode);
int security_set_mode(int new_mode);
int security_validate_and_respond(char *inbuf, char **outbuf,
  int (*cb_success)(char *, char **, unsigned short), int (*cb_fail)(
    int, char **));
static void security_response(char **outbuf, unsigned short outlen,
    char *hmac);
static int security_validate(char *inbuf);
static unsigned short security_extract_data_length(char *inbuf);
#endif /* SECURTY_H_ */
```

B.2 security.c

```
1 #include <stdio.h>
2 #include "FreeRTOS.h"
3 #include "task.h"
4 #include "i2c.h"
5 #include "rtc.h"
6 #include "polarssl/md5.h"
7 #include "security.h"
8 #include <string.h>
9
10 int mode;
11
12 void security_init(int init_mode)
13 {
14   i2c_init();
15   rtc_init();
16   mode= init_mode;
17 }
18
19 int security_set_mode(int new_mode){
20   if (new_mode+1 == mode || new_mode-1 == mode || new_mode >=
      SECURITY_MODE_OFF || new_mode <= SECURITY_MODE_EXTENDED_AUTH){
21     mode = new_mode;
22     return 1;
23   }
24   return -1;
25 }
26
27 /* The core interface to the security module
28  * Params:
29  * inbuf: Pointer to the incoming buffer containing the packet being
      validated
30  * outbuf: Pointer where the output shall be stored
31  * *cb_succes: Callback function which shall be called to process the
      data - format: cb_success(char *data, char *outbuf, int datalen)
32  * *cb_fail: Callback function which shall be called for error handling
      - format: cb_fail(int errorcode, char *outbuf)
33 */
34 int security_validate_and_respond(char *inbuf, char **outbuf, int (*
      cb_success)(char *, char **, unsigned short), int (*cb_fail) (int,
      char **)){
35   unsigned short datalen;
36   char *data, response;
37   int validate;
38   unsigned short response_length;
39
40   validate = security_validate(inbuf);
41
42   if(validate <0){
43     cb_fail(validate, outbuf);
44     return validate;
45   }
```

```
46
47   datalen = security_extract_data_length(inbuf);
48   response_length = cb_success(&inbuf[PROTO_HEADERLEN],outbuf,datalen);
49
50   if (mode> SECURITY_MODE_BASIC_AUTH){
51     char hmac[PROTO_HMACLEN];
52     memcpy(&hmac,&inbuf[PROTO_POS_HMAC],PROTO_HMACLEN);
53     security_response(outbuf,response_length,hmac);
54   }
55   return 1;
56 }
57
58 static void security_response(char **outbuf, unsigned short outlen,
     char * hmac){
59   *outbuf = realloc(*outbuf,outlen+PROTO_HEADERLEN);
60   if (*outbuf){
61     memmove(*outbuf+PROTO_HEADERLEN,*outbuf,outlen);
62     memset(*outbuf+PROTO_POS_TIMESTAMP,'-',sizeof(int32_t));
63     memcpy(*outbuf+PROTO_POS_HMAC,hmac,PROTO_HMACLEN);
64     memcpy(*outbuf+PROTO_POS_DATALEN,outlen,sizeof(unsigned short));
65   }
66   //realloc failed, dont change anything
67 }
68
69
70 static unsigned short security_extract_data_length(char *inbuf){
71   unsigned short datalen;
72   memcpy(&datalen,&inbuf[PROTO_POS_DATALEN], sizeof(short));
73   return datalen;
74 }
75
76 static int security_validate(char *inbuf){
77   unsigned char hmac_recv[PROTO_HMACLEN];
78   unsigned char hmac_calc[PROTO_HMACLEN];
79   unsigned short datalen;
80   int32_t timestamp_recv, time;
81   if(mode > SECURITY_MODE_OFF){
82
83     datalen = security_extract_data_length(inbuf);
84     if(datalen < 1 || datalen > MAX_PACKET_SIZE-PROTO_HEADERLEN){
85       return VALIDATE_INVALID_DATALEN;
86     }
87
88     if (mode>SECURITY_MODE_NO_TIME){
89       memcpy(&timestamp_recv,&inbuf[PROTO_POS_TIMESTAMP], sizeof(
     int32_t));
90       time = rtc_get_timestamp();
91       if( time > timestamp_recv || time+TOLERANCE < timestamp_recv)
92       return VALIDATE_INVALID_TIME;
93     }
94
95     memcpy(&hmac_recv,&inbuf[PROTO_POS_HMAC],PROTO_HMACLEN);
```

```
96        memset(&inbuf[PROTO_POS_HMAC],0,PROTO_HMACLEN);
97        md5_hmac((const unsigned char *) &HMAC_KEY, sizeof(HMAC_KEY)-1,
          inbuf, datalen+PROTO_HEADERLEN, hmac_calc);
98        memcpy(&inbuf[PROTO_POS_HMAC],&hmac_recv,PROTO_HMACLEN);
99        if (memcmp(hmac_calc,hmac_recv,PROTO_HMACLEN)){
100         return VALIDATE_INVALID_HMAC;
101       }
102       return VALIDATE_SUCCESS;
103     }
104     return VALIDATE_SECURITY_DISABLED;
105   }
```

Appendix C

Base Station

C.1 create_nap_header.pl

```
1 #!/usr/bin/perl
2 use warnings;
3 use strict;
4 use Digest::HMAC_MD5 qw(hmac_md5 hmac_md5_hex);
5
6 my $ifilename = shift;
7 my $timestamp_int = shift;
8 my $key = shift;
9 my $data;
10
11 open FILE, "<", $ifilename or die $!;
12 {
13   local $/ ;
14   $data = <FILE> ;
15 }
16 close FILE;
17
18 my $timestamp = pack "N",int($timestamp_int);
19
20 my $digest = "\0"x16;
21 my $length = pack "n", length($data);
22 my $uplink_data = $timestamp.$digest.$length.$data;
23 my $hmac_hex = hmac_md5_hex($uplink_data, $key);
24 my $filename = "secured_out/$timestamp_int\_$hmac_hex";
25 $digest = hmac_md5($uplink_data, $key);
26
27 $uplink_data = $timestamp.$digest.$length.$data;
28
29 open FILE, "+>", $filename or die $!;
30 print FILE $uplink_data;
31 close FILE;
32
33 print $filename;
```

C.2 get_nap_hmac.pl

```
1 #!/usr/bin/perl
2 use warnings;
3 use strict;
4
5
6 my $filename = shift;
7
8 my $data;
9 open FILE, "<", $filename or die $!;
10 {
11     local $/ ;
12       $data = <FILE> ;
13 }
14 close FILE;
15
16 my $hmac = substr($data,4,16);
17 my $hmac_hex = unpack('H*', $hmac);
18 print $hmac_hex;
```

Appendix D

Testing Code

D.1 link_simulation.h

```
1  #ifndef LINK_SIMULATION_H
2  #define LINK_SIMULATION_H
3
4  int beacon_out(char *msg);
5  void simulation_link_task();
6
7  void simulation_beacon_task();
8  int simulation_cb_fail(int errorcode, char **outbuf);
9  int simulation_cb_success(char *data, char **outbuf, unsigned short
       datalen);
10
11 #endif /* LINK_SIMULATION_H_ */
```

D.2 link_simulation.c

```c
#include "link_simulation.h"
#include "security/security.h"
#include "FreeRTOS.h"
#include "task.h"
#include "security/rtc.h"
#include "string.h"
#include "stdio_usb.h"

int beacon_out(char *msg){
  printf("[BEACON] NTNU Test Satellite; %s\n", msg);
  return 1;
}

void simulation_beacon_task(){
  const portTickType delay = 5000 / portTICK_RATE_MS;
  char timebuf[10];
  for(;;){
    int32_t time = rtc_get_timestamp();
    sprintf(timebuf,"%d",time);
    beacon_out(&timebuf);
    vTaskDelay(delay);
  }
}

int simulation_cb_success(char *data, char **outbuf, unsigned short
    datalen){
  //process data
  *outbuf = malloc(30 * sizeof(char));
  snprintf(*outbuf,30,"Packet successful validated!\n");
  return 30;
}

int simulation_cb_fail(int errorcode, char **outbuf){
  *outbuf = malloc(30 * sizeof(char));
  snprintf(*outbuf,30,"Validation failed: %d\n", errorcode);
  return 30;
}

void simulation_link_task(){
  TickType_t timeStart, timeEnd;
  int status;
  unsigned short length;
  char recbuf[MAX_PACKET_SIZE];
  char **outbuf = malloc(sizeof(char *));
  for(;;){

    fread(&length, sizeof(short), 1, stdin);
    if(length == 0 || length > MAX_PACKET_SIZE){
      continue;
    }

```

```
51      printf("[DEBUG] Incoming packet with length: %d\n", length);
52
53      memset(recbuf,0,MAX_PACKET_SIZE);
54      fread(recbuf,sizeof(char),length, stdin);
55
56      timeStart = xTaskGetTickCount();
57      status = security_validate_and_respond(recbuf, outbuf,
        simulation_cb_success, simulation_cb_fail);
58      timeEnd = xTaskGetTickCount();
59
60      printf("[DEBUG] Called validate and response function: %d\n",status
        );
61      printf("[DEBUG] Elapsed Time: %d ms\n",timeEnd-timeStart);
62      printf("[RESPONSE] %s\n", *outbuf);
63
64      free(*outbuf);
65    }
66  }
```

D.3 main.c

```
#include <asf.h>
#include "FreeRTOS.h"
#include "task.h"
#include "security/security.h"
#include "simulation/link_simulation.h
void simpleLEDtask(void) {
  while(1) {
    gpio_toggle_pin(AVR32_PIN_PB03);
    vTaskDelay(1000);
  }
}

void uc3a3init(){
  struct pll_config pllcfg;
  pm_switch_to_osc0(&AVR32_PM, 12000000,
    AVR32_PM_OSCCTRL0_STARTUP_2048_RCOSC);
  pll_config_init(&pllcfg, PLL_SRC_OSC0, 1, 96000000 / FOSC0);
  pll_enable(&pllcfg, 0);
  pll_wait_for_lock(0);
  sysclk_set_prescalers(1, 3, 1); // CPU clock is 48MHz, PBA is 12MHz,
    PBB is 48MHz
  sysclk_set_source(SYSCLK_SRC_PLL0);
}

int main(void) {
  uc3a3init();

  stdio_usb_init();
  security_init(2);

  xTaskCreate(&simulation_link_task, (const signed portCHAR *)"Link
    Simulation Task", configMINIMAL_STACK_SIZE+MAX_PACKET_SIZE, NULL,
    tskIDLE_PRIORITY+1, NULL);
  xTaskCreate(&simulation_beacon_task, (const signed portCHAR *)"Beacon
    Simulation Task", configMINIMAL_STACK_SIZE+512, NULL,
    tskIDLE_PRIORITY+1, NULL);


  xTaskCreate((TaskFunction_t)simpleLEDtask, "SimpleLEDtask",
    configMINIMAL_STACK_SIZE+512,NULL, 1, NULL);
  vTaskStartScheduler();
}
```

D.4 nap_middleware.pl

```
1 #!/usr/bin/perl
2 
3 use warnings;
4 use strict;
5 use Win32::SerialPort;
6 use IO::Socket::INET;
7 use threads;
8 use IO::Select;
9 
10 $|++;
11 
12 if($#ARGV < 3){
13   die "Usage: $0 <serial port> <uplink port> <downlink port> <beacon
      link port>\n";
14 }
15 my $serialPort = shift;
16 my $uplinkPort = shift;
17 my $downlinkPort = shift;
18 my $beaconPort = shift;
19 my $PortObj = Win32::SerialPort->new($serialPort) or die "Failed to
      open $serialPort\n";
20 print "[*] Established serial connection\n";
21 
22 my $uplinkServer = new IO::Socket::INET(
23   LocalHost => '0.0.0.0',
24   Proto => "tcp",
25   LocalPort => $uplinkPort,
26   Listen => 1,
27   Timout => 10,
28   reuse => 1,
29 ) or die "Failed to open Socket on port $uplinkPort: $!\n";
30 
31 my $downlinkServer = new IO::Socket::INET(
32   LocalHost => '0.0.0.0',
33   Proto => "tcp",
34   LocalPort => $downlinkPort,
35   Listen => 1,
36   Timout => 10,
37   reuse => 1,
38 ) or die "Failed to open Socket on port $downlinkPort: $!\n";
39 
40 my $beaconServer = new IO::Socket::INET(
41   LocalHost => '0.0.0.0',
42   Proto => "tcp",
43   LocalPort => $beaconPort,
44   Listen => 1,
45   Timout => 10,
46   reuse => 1,
47 
48 ) or die "Failed to open Socket on port $beaconPort: $!\n";
49 
```

```
50 print "[*] Waiting for client to connect\n";
51 my $uplinkClient = $uplinkServer->accept;
52 my $downlinkClient = $downlinkServer->accept;
53 my $beaconClient = $beaconServer->accept;
54 my $peerAddr = $uplinkClient->peerhost();
55
56 print "[+] Client connected from $peerAddr\n";
57 my $thread_1 = threads->new(\&serialThread)->detach();
58
59 while (1){
60   my ($in, $pre);
61   $uplinkClient->recv($in,4096);
62   last if (length($in)==0);
63   $pre = addPseudoNetworkLayer($in);
64   $PortObj->write($pre);
65   $PortObj->write($in);
66 }
67
68
69 print "[-] Connection closed, Exiting...\n";
70
71 $uplinkClient->close();
72 $uplinkServer->close();
73 $downlinkClient->close();
74 $downlinkServer->close();
75 $beaconClient->close();
76 $beaconServer->close();
77
78
79 sub addPseudoNetworkLayer{
80   my $in = shift;
81   my $out = pack "n", length($in);
82   return $out;
83 }
84
85 sub serialThread{
86   $PortObj->user_msg(1);
87   $PortObj->error_msg(1);
88   $PortObj->databits(8);
89   $PortObj->baudrate(115200);
90   $PortObj->parity("none");
91   $PortObj->stopbits(1);
92   $PortObj->buffers(4096, 4096);
93   $PortObj->read_interval(10);
94   $PortObj->read_char_time(5);
95   $PortObj->read_const_time(10);
96   $PortObj->write_settings;
97   my $response = "";
98
99   while(1){
100     $response .= $PortObj->input;
101     if ($response =~ /\n/){
```

```
102         if ($response =~ /\[RESPONSE\](.*)/ ){
103           $downlinkClient->send($response);
104           $response = "";
105         }
106
107         if ($response =~ /\[BEACON\](.*)/ ){
108           $beaconClient->send($1);
109           $response =~ s/\[BEACON\](.*)\n//;
110         }
111
112       }
113     }
114
115 }
```

D.5 nap_packetgenerator.pl

```
1 #!/usr/bin/perl
2 use warnings;
3 use strict;
4
5 my $size = shift;
6 my $num = shift;
7 my $prefix = shift;
8 my $dir = "./out";
9
10 open RAND, "<", "/dev/urandom" or die $!;
11
12 for(my $i = 0; $i < $num ; $i++){
13   my $bytes = "";
14   read(RAND,$bytes,$size);
15   my $fi = sprintf '%05d', $i;
16   open FILE, "+>", "$dir/$prefix\_$fi" or die $!;
17   print FILE $bytes;
18   close FILE;
19 }
20 close RAND;
```

D.6 nap_base_station_simulation.pl

```
#!/usr/bin/perl
use warnings;
use strict;
use threads;
use threads::shared;
use Digest::HMAC_MD5 qw(hmac_md5 hmac_md5_hex);
use IO::All;
use IO::Socket::INET;
use Time::HiRes qw(usleep);

$|++;
my $ip = shift;
my $uplinkPort = shift;
my $downlinkPort = shift;
my $beaconPort = shift;
my $proto = 'tcp';
my $response = "";

my $dir = './out';
my $timestamp : shared = undef;
my $key = "ABCD";

my $sent_packages = 0;
my $fingerprint_counter = 0;


my $uplinkSocket = IO::Socket::INET->new(
  PeerHost => $ip,
  PeerPort => $uplinkPort,
  Proto => $proto,
  Timeout => 100,
) or die "Unable to connect to uplink at $ip:$uplinkPort - $!\n";

my $downlinkSocket = IO::Socket::INET->new(
  PeerHost => $ip,
  PeerPort => $downlinkPort,
  Proto => $proto,
  Timeout => 10,
) or die "Unable to connect to downlink at $ip:$downlinkPort - $!\n";

my $beaconSocket = IO::Socket::INET->new(
  PeerHost => $ip,
  PeerPort => $beaconPort,
  Proto => $proto,
  Timeout => 10,
) or die "Unable to connect to beacon link at $ip:$beaconPort - $!\n";

my $recv_t = threads->new(\&recvThread)->detach();
  while(!defined($timestamp)){}

my $incTimestampt_t = threads->new(\&incTimestampThread)->detach();
```

```
52
53 opendir (DIR, $dir) or die $!;
54 my @files = sort readdir(DIR);
55 while (my $file = shift @files) {
56   my ($data, $recv, $response);
57   my $ts = $timestamp;
58   my $replay_counter = -1;
59
60   next unless (-f "$dir/$file");
61
62   my $ofile = `./create_nap_header.pl $dir/$file $ts $key`;
63   my $saved_hmac = `./get_nap_hmac.pl $ofile`;
64   open FILE, "<", $ofile or die $!;
65   {
66     local $/ ;
67     $data = <FILE> ;
68   }
69   close FILE;
70
71   print "Sending: $ofile:  \n";
72   $sent_packages++;
73
74   print "Packet Length: ", length($data), "\n";
75
76 #Goto Label for replay testing
77 REPLAY:
78   #Simulate uplink transmission delay
79   my $transdelay = int(length($data)* 8 / 9600*1000*1000);
80   usleep($transdelay);
81
82   $uplinkSocket->send($data);
83   $downlinkSocket->recv($recv,1024);
84
85   if($recv =~ /\[RESPONSE\]\s(.*)/){
86     $response = $1;
87     my $respfile = "./in/tmp";
88     open FILE, "+>", $respfile or die $!;
89     print FILE $response;
90     close FILE;
91
92     print $recv;
93
94 ###Only required for testing extended auth#################
95 #   my $recv_hmac = `./get_nap_hmac.pl $respfile`;
96 #   if ($recv_hmac eq $saved_hmac){
97 #     print "[ENAP] HMAC matches!\n";
98 #     $fingerprint_counter++;
99 #   }else{
100 #     print unpack 'H*' ,$response;
101 #     $recv =~ s/[^[:print:]\n]+/./g;
102 #     print "$recv\n\n";
103 #   }
```

```
104 # }
105 #}
106 #print "[ENAP RESULT] $fingerprint_counter / $sent_packages";
107 ##################################################################
108
109 ###Normal excution flow###
110   }
111 # Simulate donmlink transmission delay
112 # $transdelay = int(length($response)* 8 / 9600*1000*1000);
113 # usleep($transdelay);
114
115 ###Only required for testing immediate replay attacks######
116 # if($recv !~ /\[DEBUG\].*-/){
117 #       $replay_counter ++;
118 #       goto REPLAY;
119 # }
120 # if($replay_counter == -1 ){ print $recv};
121 # print "Number of succesful replays: $replay_counter\n";
122 #################################################################
123 }
124
125 $uplinkSocket->close();
126 $downlinkSocket->close();
127 $beaconSocket->close();
128 closedir(DIR);
129
130
131 sub recvThread{
132   my $response;
133   while($beaconSocket->connected){
134     $beaconSocket->recv($response,4096);
135     $response =~ /.*?(\d{9,})/;
136     $timestamp = $1+4;
137
138   }
139 }
140
141 sub incTimestampThread{
142   while(1){
143     $timestamp += 1;
144     sleep(1);
145   }
146 }
```