Pasi Riissanen

# Remote Firmware Updating

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

30 November 2016

Metropolia

| | |
|---|---|
| Author<br>Title<br><br>Number of Pages<br>Date | Pasi Riissanen<br>Remote firmware updating<br><br>40 pages + 1 appendix<br>30 November 2016 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information Technology |
| Specialisation option | Embedded Systems |
| Instructor | Keijo Länsikunnas, Principal Lecturer |

The objective of this final year project was to develop a device to gather measurement data from an air quality sensor in a database over the Internet. The device was to be an add-on for the already existing sensor without having to change any components of it. The sensor's firmware was also required to be updated remotely. This thesis and the application that was developed is a part of a larger project including a database server for gathering data and a web application for displaying the gathered data.

The platform for the device was an Atmel microcontroller with a Debian Linux operating system. One part of the project was enabling a secure connection between the sensor client and the server. When sending measurement data over the Internet security aspects regarding data integrity and authenticity had to be taken into consideration. Another part of the requirements of this project was updating the sensor's firmware remotely. A program was developed for this purpose. Converting the firmware file from Intel Hex into binary form and utilizing the sensor's proprietary communication protocol were the challenges faced when writing the software.

The developed device met the set requirements and is able to communicate securely and to upload new firmware to the sensor. This project was a good learning experience regarding information security and autonomously functioning embedded devices.

| | |
|---|---|
| Keywords | IoT, SSH, Linux, socket, public-key authentication, Diffie-Hellman, Intel Hex |

| | |
|---|---|
| Tekijä<br>Otsikko<br><br>Sivumäärä<br>Aika | Pasi Riissanen<br>Laiteohjelmiston päivittäminen etäyhteydellä<br><br>40 sivua + 1 liite<br>30.11.2016 |
| Tutkinto | Insinööri (AMK) |
| Koulutusohjelma | Tietotekniikka |
| Suuntautumisvaihtoehto | Sulautetut järjestelmät |
| Ohjaaja | Lehtori Keijo Länsikunnas |


Insinööriyön tavoitteena oli kehittää laite, jolla kerätä ja lähettää ilmanlaatuanturin tuottamaa mittausdataa palvelimelle Internetin välityksellä. Laitteen tuli olla lisättävä osakokonaisuus jo olemassa olevan anturin yhteyteen, ilman että anturin kokoonpanoa tarvitsee muuttaa. Anturin ohjelmiston etäpäivittäminen oli myös yksi vaatimuksista. Tämä insinöörityö oli osa isompaa projektia, jossa osakokonaisuuksina on palvelin, jolle mittausdata kerätään, sekä verkkosivusto mittaustulosten esittämistä varten. Laitteen alustana toimii Atmelin valmistama mikrokontrolleri, johon asennettiin Debian Linux -käyttöjärjestelmä. Yksi osa projektia oli yhteyden luominen anturin ja palvelimen välille. Mittausdataa lähetettäessä Internetin välityksellä piti ottaa huomioon tietoturva-asioita datan eheyden ja luotettavuuden suhteen.

Toinen osakokonaisuus tässä projektissa oli anturin ohjelmiston päivittäminen, jota varten kirjoitettiin ohjelma ohjelmistotiedoston formaatin muuttamiseksi ja anturiin lataamiseksi. Ohjelman luomisessa oli haasteena Intel Hex -tiedostomuodon ymmärtäminen, ja anturin oman tiedonsiirtoprotokollan käyttäminen.

Projektin päättyessä laite vastasi sille asetettuja vaatimuksia ja kykenee tietoturvalliseen liikennöintiin ja lataamaan anturille uuden ohjelmiston. Projekti oli hyvä oppimistilaisuus tietoturvaan ja itsenäisesti toimivien sulautettujen laitteiden toimintaan liittyvistä asioista.



| | |
|---|---|
| Avainsanat | Esineiden Internet, SSH, Linux, socket, julkisen avaimen autentikointi, Diffie-Hellman, Intel Hex |

## Contents

## Abbreviations

## Abbreviations

| | |
|---|---|
| NB-IoT | Narrowband Internet of Things, a new NB radio technology to address the requirements of the IoT |
| LTE | Long-Term Evolution, mobile phone communication standard |
| GSM | Global System for Mobile Communications, mobile phone communication standard |
| Nelix | Unit that polls the air quality sensor for measurement data locally through ModBus. Running Linux OS |
| TCP | Transmission control protocol |
| USB | Universal serial bus |
| JSON | JavaScript Object Notation |
| SSH | Secure Shell |
| SCP | Secure copy, a tool provided with SSH |
| ASCII | American Standard Code for Information Interchange |

# 1 Introduction

The goal of this project was to add network connectivity to a sensor that measures quantity and mass of particles in air. The sensor can be interfaced with a PC, but only locally. This project started from the need to remotely and wirelessly gather data from the sensor to a remote database with an integrated device. Wireless connectivity provides ability to form a network of sensors to monitor air quality on multiple measure points over a wide area. By gathering the measurement data to a database, a comprehensive analysis can be made of the air quality in that area.

During the sensor's operational lifetime, its settings, calibration values and firmware need to be updated. The most convenient way to achieve this is remotely over an Internet connection. Wirelessly transferring data over an insecure network produces challenges with data security and integrity. To solve this, the security aspects needed to be evaluated and appropriate security measures implemented in the system.

This thesis focuses on two topics, creating a secure connection between the sensor and the database server and the program Armupdate that updates the sensor's firmware. Armupdate also handles a file format conversion from Intel ASCII Hex into binary. An Atmel SAMA5D4 series microcontroller was used to interface the already existing sensor through a ModBus/TCP interface. A 3G radio modem was installed in the microcontroller to gain reliable internet connectivity. A Linux Debian operating system was installed in the microcontroller for a practical way to handle files and to utilize the connection provided by the modem. This microcontroller unit is later referred to as Nelix. Reading the measurement data from the sensor is not one of the main topics in this thesis.

# 2 Internet of Things

The Internet of things, or IoT, is a fairly recently popularized term used to describe a network that does not require human interaction to gather data or send messages. In automotive industry sensor networks have been commercially available since the 1980's but only fairly lately data from these networks is sent over the internet for failure statistics and optimized maintenance plans [1]. The growth of IoT is thought of to be able to provide new services and new features on top of existing ones. Examples of IoT are being implemented in building automation and logistics services improving effectiveness and thus reducing costs.

## 2.1 New Network Technologies for IoT

Most IoT applications only need a small amount of bandwidth but require to have low power consumption in operation. As millions of IoT devices are expected to be in operation during the next ten years, new technologies are being developed solely for IoT machine-to-machine type operations [2].

Members of the Third-Generation Partnership Project decided in September 2015 to standardize a new radio technology narrowband Internet of Things, NB-IoT, also referred to as LTE-M2. It is targeted towards enabling the connectivity of machine-to-machine devices over a Wide Area Network. NB-IoT is designed to improve indoor coverage, support a massive amount of connected devices and to reduce end device power consumption. One major benefit of NB-IoT is the ability to utilize existing LTE and GSM frequency bands which should ease the deployment of the technology. In September 2016 Nokia together with Sonera tested the NB-IoT network technology in Finland transferring temperature, humidity and barometric pressure sensor data. [2;3.]

A French company called Sigfox utilizes its own proprietary low power wide area network technology for low power and low data throughput devices. The Sigfox network operates on unlicensed frequency ranges employing Ultra Narrow Band, UNB technology. Maximum bandwidth with Sigfox at 100 bits per second is even lower than with NB-IoT but is still enough for applications where the amount of data to be transferred is minimal. Low bandwidth allows for minimal power consumption with standby times which can be tens of years. [4;5.]

## 2.2 Security Concerns

With devices affecting and monitoring physical surroundings, potential attacks are no longer targeted solely on data or services but also privacy and physical security. [6] Baby monitors and digital video recorders with Internet access have often times poorly protected access control. One cause is the end user not changing the default password for their account but also the audio and video stream can be without any sort of encryption, resulting in a major privacy breach. [7.]

Some attacks benefit from the interconnectability of devices. If one of the devices in a Local Area Network or Controller Area Network has access to the internet and is breached through that connection, it should not provide access for the attacker to the rest of the devices on the network. A prime example of this is remotely controlling a Jeep Cherokee described in the paper *Remote Exploitation of an Unaltered Passenger Vehicle* by Charlie Miller and Chris Valasek. Through the car entertainment unit that is connected by a CAN bus to the rest of the on board computers, Miller and Valasek were able to control the cars accelerator, brakes and steering. [8.]

IoT devices having access to the Internet and therefore having an IP address have become a target for bot net creators. Bot net is a network of devices infected with software that is most often used for denial-of-service attacks. With thousands of infected devices, the bot net is able to attack a host by creating massive amount of service requests, overloading the target's resources and thus denying the service for even the legitimate users. For example, the Mirai malware infected hundreds of thousands IoT devices in 2015 by scanning for devices with factory default user name password combinations. [9;10.]

## 2.3 Secure Boot

One main topic in IoT security is the ability for the system to autonomously check for unauthorized software modifications. Secure boot is a procedure that enables a device to verify the authenticity of the application code before execution. Verifying the application before execution provides protection against a large amount of security breaches. Malware infected or otherwise modified software will not be launched. The application

code is verified by a root-of-trust program that is stored on a read-only-memory and executed on start up before starting the application. This operation takes place each time the device is powered on and thus does not protect from changes in the application that could happen during run time. The root-of-trust program must be stored in a non-modifiable memory such as ROM or internal flash memory that can be by other means assured to be modifiable only by an authorized party. [11.]

For the root-of-trust program to be able to verify the application code, the code must be digitally signed by the issuer of the application. The signature consists of a hash calculated from the application code which is encrypted using the private key of the signing party. The root-of-trust program verifies this signature by decrypting the hash from the signature with the public key of the signing party and calculating a hash from the application code. These two hashes must match for the application code to be considered valid. The public key of the signing party must be stored on the device in a non-modifiable memory so that it cannot be modified by any unauthorized party. The public key should also be protected by a checksum for integrity verification. Updating the software on a root-of-trust based system requires the new software to be also signed appropriately with the private key of the signing party. [11.]

# 3 Security and Cryptography

For a network to be considered secure it must guarantee confidentiality and that only known users are able to send and receive messages without anyone else accessing the content of the messages. The network must guarantee integrity of messages, making sure that they are free of alteration by unauthorized users or by technical malfunction. Users of the network must be able to prove their authenticity so that their messages can be proven to be from a certain origin and it must be impossible for unauthorized users to masquerade themselves as someone else. [12.]

## 3.1 Confidentiality with Encryption

Confidentiality in practice is achieved with encryption, which is a process where the content of the message is converted into a ciphertext. Even if a third party has access to the media and therefore to the message, they are unable to determine its content from the encrypted message. The algorithm used in the conversion process is known as a cipher. The secrecy lies in what algorithm is used and this information must be somehow shared between endpoints through another secure channel. Maintaining different ciphers unique for each user is practically impossible with any larger networks. To overcome this downside most ciphers use keys to encrypt and decrypt messages. With keys the secrecy is transferred from knowing the cipher to knowing the correct key. A single algorithm can be used but the key is the shared secret between endpoints. Symmetric key ciphers use the same key for encrypting and decrypting the message. Both endpoints must know this key before being able to communicate and thus again creating the need of sharing the key through an alternative secure channel beforehand. [12.]

Asymmetric key ciphers use two keys, one to encrypt and another to decrypt the message. Encrypting and decrypting with different keys is usually done by utilizing a trapdoor one way function where the outcome of the function for a chosen value is easy to calculate but reversing the operation is difficult. In cryptographic context difficult means that it takes an unfeasible amount of computing time to reverse the operation. In theory any encryption can be decrypted without knowing the key if given infinite resources and time. As asymmetric keys use two keys, the encryption key can be public. Only the decryption key must be kept secret. This again leads to the dilemma of sharing the decryption key but in most cases this only needs to be done once. When initializing a connection with

the aim of using a symmetric key cipher, within the first steps information of the cipher or key must be sent before the channel is secured. This enables any potential attacker to also know what cipher or key will be used. With asymmetric keys, the first message sent can be encrypted with the public encryption key. Only the targeted recipient of the message has the correct decryption key; therefore, the attacker is unable to determine any contents of the message. Asymmetric ciphers are also referred to as public key ciphers as the encryption key can be public. [12.]

## 3.2 Integrity

If a third party has access to the media and is able to send and receive encrypted messages, they cannot access the content but can still modify the messages. This way security is not compromised but the service is most likely unusable depending on the outcome of the modification after the message has been decrypted. One-way hash functions are used in cryptography to ensure message integrity. Typical for a one-way hash function is that it takes a variable length input and the result after conversion has a fixed length. The function's output is a hash that is used as a checksum when sent along with the message. By agreeing what hash function to use, both endpoints can calculate the hash from the message and come to the same end result.

Properties of a good hash function include that it is unfeasible to invert, meaning that it is next to impossible to recreate the message from the hash by trying all possible results. It is also desirable that a small change in the message leads to a big change in the hash to protect against transfer errors. In most scenarios a potential attacker would want to change only a small part of the message. Another desirable feature is that two different messages should never result in the same hash. This to ensure that an attacker cannot completely change the message and still being able to maintain a correct hash. If an attacker is able to modify the whole message, they are most probably also able to create a totally different hash. This situation then becomes an authentication issue which is described in chapter 2.3 Authentication and digital signatures. It is also possible to calculate the hash not only on the message content but also including the shared secret between the sender and receiver. As the attacker does not have the key, they cannot create a hash for the message. [12.]

### 3.3 Authentication and Certificate Authority

Authentication is a concept that ensures that a node in the network is who they claim to be. There are two main approaches in cryptography to ensure authentication, digital signatures and challenge response queries, both of which utilize asymmetric encryption with public keys. Challenge response queries consist of a random number that is sent to the node that is requesting authorization. The receiving node signs this random number with its private key and sends the signed response back to the other node. The other node can then decrypt this signature with the according public key and authenticate the node. This challenge can be repeated the other way round so that both nodes are certain of each other's identity. With digital signatures, a part of the message is encrypted with the private key. The receiver can decrypt this signature with the claimed sender's public key and confirm its authenticity. This method differs from public key encryption by the fact the encryption is done with the private key instead of the public key. Digital signatures are also often used with file transmissions to prove that the file is the same unaltered version as the provider claims it to be. [12.]

In the usual scenarios where public keys are used to authenticate nodes in the network, the public key is presented and proven to be authentic with challenge response queries. This does prove that the node has the correct private key and thus is the owner of the said public key, but there is no certain link between the identity of a user or node in the network and their key pairs. Public key authentication has a possibility for man-in-the-middle attacks if Node A does not know the identity of Node B and is unknowingly communicating with node X. Certificate authority is a third party whose only task is to keep a record which public key belongs to whom. The certificate authority issues a certificate which proves that the target of a certificate is known by the certificate authority. When Node A in the network wants to connect to Node B, it requests the target node's public key from certificate authority. Certificate authority returns the public key encrypted with its own private key. Node A can be certain that the public key for Node B is authentic as long as the certificate authority is trusted. All nodes in the network need only to know the public key of the certificate authority instead of knowing all keys for all other users in the network. [12;13.]

# 4 System Description

The key hardware components used in this thesis are the particle sensor and the Atmel SAMA5D4 series ARM based embedded microprocessor unit illustrated in figure 1, with a 3G modem. The Atmel microprocessor unit was used with an evaluation platform that had all the necessary connections and interfaces for application development. The server into which the measurement data was gathered and the web application for displaying the data is not described in this thesis.

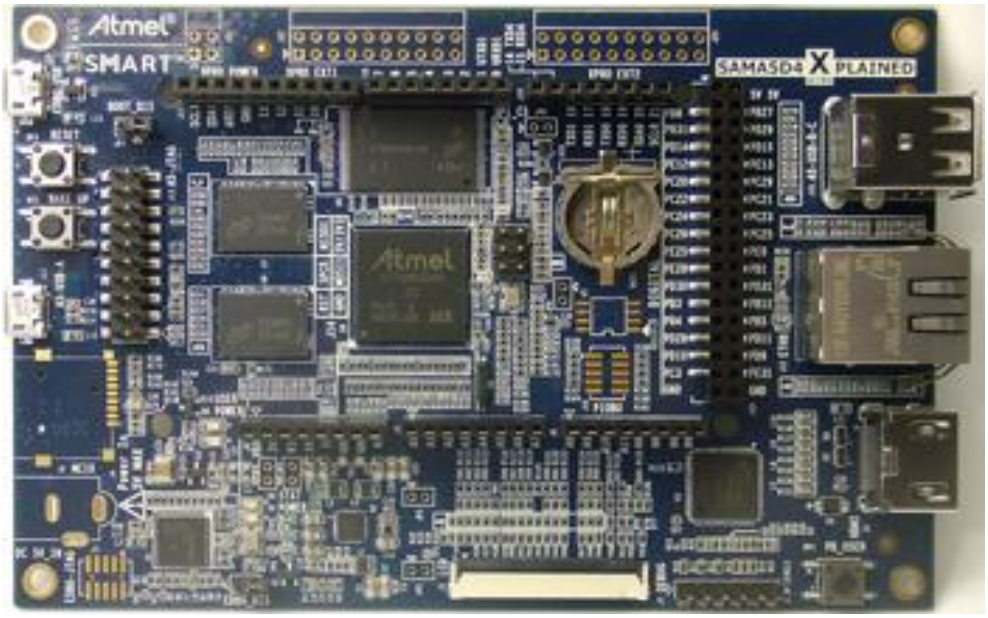


Figure 1. Atmel SAMA5D4-XULT evaluation kit. Copied from Atmel [14].

The air quality sensor unit has two interfaces for communication, a USB and an Ethernet port. The USB port utilizes the proprietary USB protocol for communicating with a proprietary software on a PC made for interfacing the sensor. With this software, all the sensor unit's functionalities are accessible, including reading measurements, setting new calibration values and uploading new firmware. In this project's application, the sensor was interfaced through the Ethernet port as illustrated in figure 2. The air quality sensor's Ethernet port supports both ModBus/TCP and the proprietary packet protocol. The ModBus/TCP bus is used for reading the sensor's holding registers which holds all the measurement data and possible error and status codes. The proprietary packet protocol over TCP is used for firmware updating. The Atmel microprocessor unit together with Linux operating system is later referred to as Nelix.

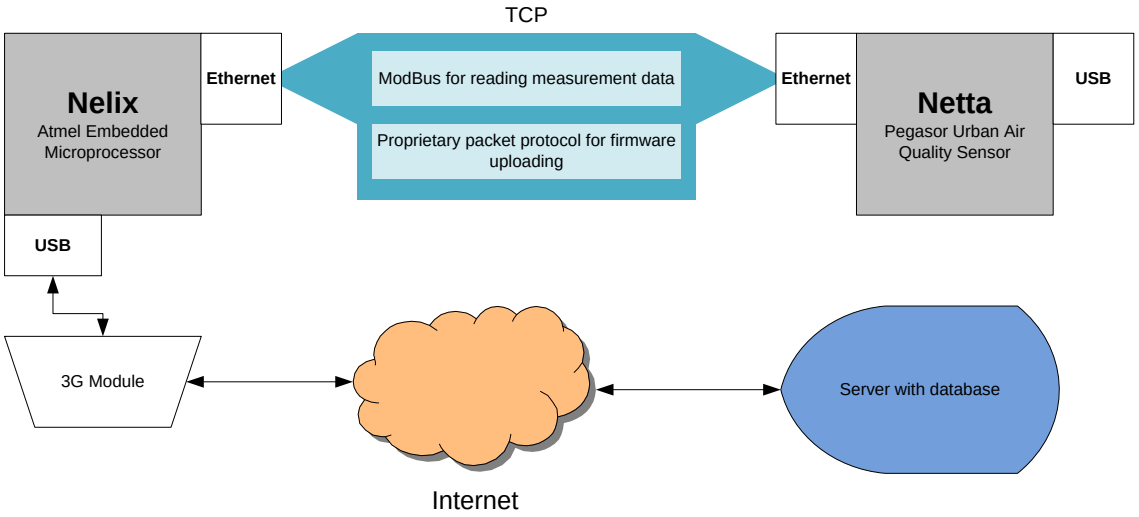


Figure 2. System layout diagram

## 4.1 Hardware and Operating System

The Atmel SAMA5D4-XULT is an evaluation kit for the Atmel SAMA5D4 series ARM-based embedded microprocessor units. The evaluation kit includes all required interfaces for this project. The Ethernet interface was used for opening an SSH connection locally to the board for development and debugging purposes and in a later stage to communicate between the Atmel board and the air quality sensor. The USB interface was used for connecting the 3G modem. The modem provides wireless internet access for data transfer from the microprocessor to the database server.

A Debian Linux operating system was installed to the Atmel Embedded Microprocessor to manage the hardware and software resources. Having an operating system enables the handling of the firmware image as a file and also running different programs simultaneously automated with scripts. All the functionalities of Nelix are driven by the main bash script Nelix_main.sh.

## 4.2 Air Quality Sensor

A part of the system is an air quality sensor which is designed for combustion engine emission monitoring and outdoor and indoor air quality monitoring. The sensor measures the amount of particles and mass concentration in gas. It can detect particles with size

ranging from a few nanometers up to 2.5 micrometers, depending on the sensor's configuration. The sensor itself requires clean and dry compressed air feed to function properly. At the beginning of the project, the sensor was installed in a laboratory at Metropolia University of Applied Sciences (UAS) with external air and moisture filters. Later, an all-in-one package, was installed at school premises as illustrated in figure 3. The unit has an air compressor and filters internally installed.

Figure 3. Air quality sensor unit in operation at Metropolia UAS premises

### 4.3 Development Tools

Development of Armupdate software was carried out in Windows environment with Visual Studio development environment. Visual Studio offers a graphical user interface for code editing and a debugger tool. Initial compilations and tests were run on Windows environment. Later the code was compiled on a target processor. A quicker way of compiling and testing would have been to have a cross-compiler that can compile for another kind of processor than the compiling host. Without a cross-compiler, the source needed to be transferred to the target processor and compiled there. Small modifications to the code were simple enough to be done on a/the target processor with a simple text editor such as the Unix native Vi. Bigger modifications were done on Windows for practical reasons and also to be able to upload the source code to Git version control.

Valgrind is a wrapper around a collection of tools and it functions similarly to a virtual machine. It utilizes just-in-time compilation techniques which means that the compilation is done during run time. The target program is translated to a form called Intermediate Representation and in that form it can be analyzed with Valgrind tools. After any analysis or modification is done to the Intermediate Presentation form, it is translated back into machine code and executed on the host processor. [15.] All software that was developed for this thesis was analyzed with MemCheck, which is a memory mismanagement detector that can detect memory leaks and misallocation errors. It was used to verify that there are no memory leaks in the programs. A memory leak could be fatal in a system that is supposed to be running continuously possibly in a remote location.

Most of the code debugging needed to be executed on the target processor. GNU Project Debugger (Gdb) was used for this purpose. Gdb requires both the source code and the compiled binary to enable the user to step through the source code while executing it. Program progression and variables and their values can be examined and the program can be executed line by line or until a set breakpoint is reached. There are graphical user interfaces for Gdb but they were not an option as the only way of interfacing the Nelix was with command line through a network interface.

Wireshark is a network traffic analyzing tool that can display the data being transferred over a network interface. It was used to investigate what kind of commands and responses the sensor's data interface transferred and to ensure that the software running on Nelix was sending the data in the right format. Wireshark can decode and identify most of the common communication protocols and display the headers and other fields of the protocols. The air quality sensor is running a proprietary protocol over the standard Modbus/TCP and Wireshark made analyzing and debugging this interface easy. Firstly, the sensor was interfaced with a desktop PC running Windows operating system and the proprietary configuration software. With Wireshark running on this computer it was possible to see what commands were sent from the PC and how the sensor responded.

Later when the sensor was interfaced with the Atmel SAMA5D4 with the Debian operating system, running Wireshark was not feasible. Tcpdump is a program that is included as a standard package in most Linux distributions and it can save the raw data passing through the Atmel's network interface to a file. The file can then be transferred to a PC and analyzed with Wireshark in the same manner as a locally captured log.

# 5 Secure File Transfer over an Insecure Network

The sensor clients connect to the measurement database server over internet. To ensure that data that the server receives is valid, the connection needs to be encrypted and the sender needs to be proven authentic. This is achieved with Secure Shell (SSHl). SSH is a suite of tools that enables an encrypted connection between hosts. It utilizes public-key cryptography for authentication and a symmetric shared secret for encryption. All traffic between the sensor and the measurement data server is initiated by the sensor client. This way the sensor clients do not need to respond to any potentially malicious inbound connection requests.

## 5.1 Cryptography and Authentication

Public-key cryptography is a cryptographic protocol system that the SSH protocol utilizes for client and server authentication, a proof that both parties are what they claim to be. Public-key encryption is also known as asymmetric cryptography as it uses a combination of two separate keys to encrypt and decrypt messages. One key is called public key as it is shared with the recipient of the message. The other is called private as it is kept secret and stored only locally on the client. The client signs a message with its private key and the recipient server decrypts it with the client's public key. These two keys are mathematically linked, in a way that it is practically impossible to deduce the private key from the public key. These key pairs are only used for authentication and cannot be used for encrypting the connection. [12;13;16.]

SSH utilizes symmetric encryption in order to secure the transmission of information. When a client initializes an encrypted connection to a server, the connection is encrypted with a shared secret created using Diffie-Hellman key exchange algorithm, after the client has verified the server's authenticity with public-key authentication. Typical for a Diffie-Hellman is that both parties participate in generating the key without allowing one end to control the outcome of the process. Also with Diffie-Hellman, the shared secret is generated without the need of sharing the secret over an insecure connection. The generated key is a symmetric key, as it used to both encrypt and decrypt messages. After the connection has been encrypted, the client can be authenticated to access the server. [12;13;16;17.] In the application described in this thesis, the authentication is accomplished with both public-key and a password.

## 5.2 Establishing a Secure Connection

Establishing an SSH connection starts with the client opening a TCP connection to port 22 on the target server. Port 22 is the default and an officially assigned port for SSH [16]. The connection process is illustrated in figure 4. During the first two steps, the client and the server exchange SSH version information and lists of supported encryption protocols. The encryption protocol is agreed on, based on what the client is able to support. If none of the protocols is supported by both parties, the connection is closed. The server also sends information about its identity in the form of a public key. If the host's public key is not found in the client's list of known hosts, a warning message is displayed, to inform that the host is previously unknown or its identity has been changed since the last connection. During the fourth step, a shared secret is negotiated between hosts. The algorithm used to calculate the shared secret is selected from the lists of supported ciphers sent at steps one and two. [12;13;17.] An encrypted communication channel is established after step four. It is safe to send a password and other login credentials through this encrypted channel.

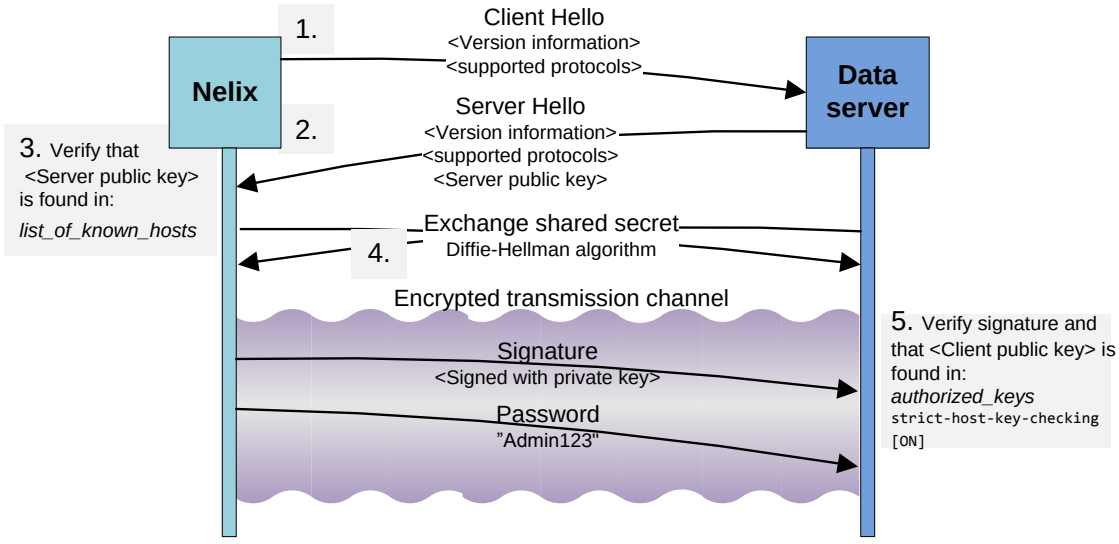


Figure 4. SSH connection initialization. Data gathered from Chandra and Bauer [12;13.].

On step five, the client creates a signature with its private key that is sent to the server. With the client's public key, the server can decrypt that signature, and verify that the client is the actual owner of the said public key. In addition to public key authentication,

a password is also required. The password can be sent as cleartext as the connection is already encrypted by the transport layer. Both the client and server must be running SSH version 6.2 or newer to be able to utilize multiple authentication methods. [13.]

## 5.3 Data Transfer during Normal Operation

During normal operation Nelix retrieves measurement data from the sensor and uploads it to the measurement data server. Nelix_main.sh bash script calls the measurement data reading program modbuspoller, which polls measurement data from the sensor through a TCP/ModBus interface. The script then uploads the gathered data as a JSON file to the measurement database server. After each upload, a new configuration file is retrieved from the server. This process is illustrated in figure 5. Handling the new configuration file is not implemented yet. The current implementation is an example of how the server is able to inform the Nelix unit if any new calibration parameters or a firmware update is available.

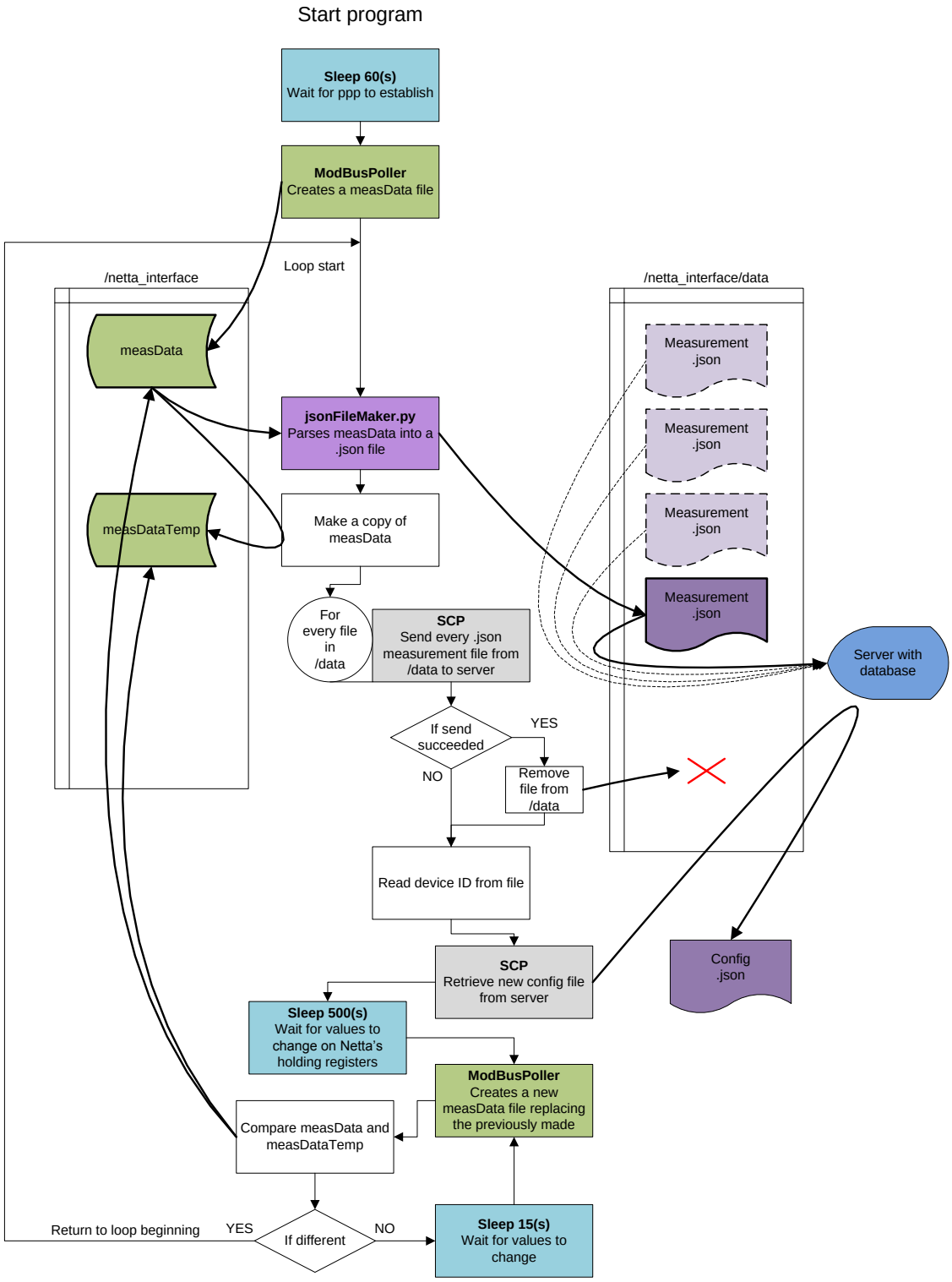


Figure 5. Flowchart diagram of bash script Nelix_main.sh

The JSON file is uploaded to the server with Secure Copy (Scp). Scp is a tool that comes with SSH and it is used to remotely copy one file from one host to another [18]. When running the scp command, an encrypted connection to the target host is created auto-

matically, as described in chapter 3.2 Establishing a secure connection. Multiple authentication methods are used. The target server will require public-key authentication and a password.

By design, SSH requires that the password is typed by an interactive keyboard user. This drawback that causes a challenge in automating the process is avoided by using sshpass to run the scp command. Sshpass makes the situation appear to SSH so that the password is typed from a keyboard instead of an automated script. [19.] Scp is used this way for all data transfer from Nelix, to both upload the measurement data to the server, and to download new configuration files or firmware.

## 5.4 File Integrity Considerations

Scp runs over the TCP transport layer protocol which employs a cyclic redundancy check for each packet's integrity verification. While it does catch most errors, it cannot be considered a guarantee for file integrity. The file is transmitted as separate packets and a missing packet could go unnoticed resulting in a corrupted file. A corrupted firmware image could render the sensor inoperable. The worst case scenario would be if the sensor's communication interface would stop responding. This would have the immediate effect of the sensor not being able to serve measurement data to Nelix as well as causing Nelix to have no chance of uploading another firmware onto the sensor.

Currently there is no extra file integrity verification implemented to the system. One way to overcome this risk of file corruption during transfer would be to calculate a checksum of the original firmware file and compare that against the downloaded file's checksum. Md5sum is a program installed by default with most Linux distributions. It can calculate and verify 128-bit MD5 checksums of files. A 128-bit MD5 checksum is considered a reliable way to verify that the file transferred correctly and intact. For verifying that the file was downloaded from an authentic source, the MD5 has become obsolete and should not be used for that purpose. [12.]

## 5.5 Key Sharing Protocol

With public key authentication there is always the dilemma of knowing who in the network has which key and which identity matches which key. Implementing a certificate authority as described in chapter 3.3 would be one solution but would require one extra instance the only function of which would be key management. In this project the network and the relation between sensor clients and the server is simple and straightforward. Sensor clients only need to communicate to one host as opposed to some other application where sensors would communicate with each other as well. In such a simple network as this it is easier to implement key management within the measurement data server. The measurement data server holds a list of all the sensor clients and their public keys. Only the keys that are present in that list can be used to authenticate with the server.

Each sensor node's public key is introduced to the measurement data server before the actual secure connection is established. The public key itself is not required to be kept secret. Introducing sensor clients to the measurement data server acts as a certificate registration function as in the certificate authority implementation. This eliminates the need for the data measurement server to accept inbound connections from previously unknown hosts over the Internet. Sensor nodes have a private key that corresponds to the public key and can prove their identity with it. Public keys of sensor nodes are added to the measurement data server's list of known hosts. During the development of the application described in this thesis, a local area network was used for key sharing, but any secure network would suffice.

Nelix has a cronjob that is set to run on the first boot and it runs the shell script key_upload.sh (see Appendix 1) that handles the key sharing procedure. After the key uploading is successful, the cronjob is deleted to ensure it will not be executed again as it reserves several resources by constantly trying to connect to the key server. With this design, after the Nelix unit has been programmed in production it can automatically generate and upload SSH keys through a local network connection as illustrated in figure 6.

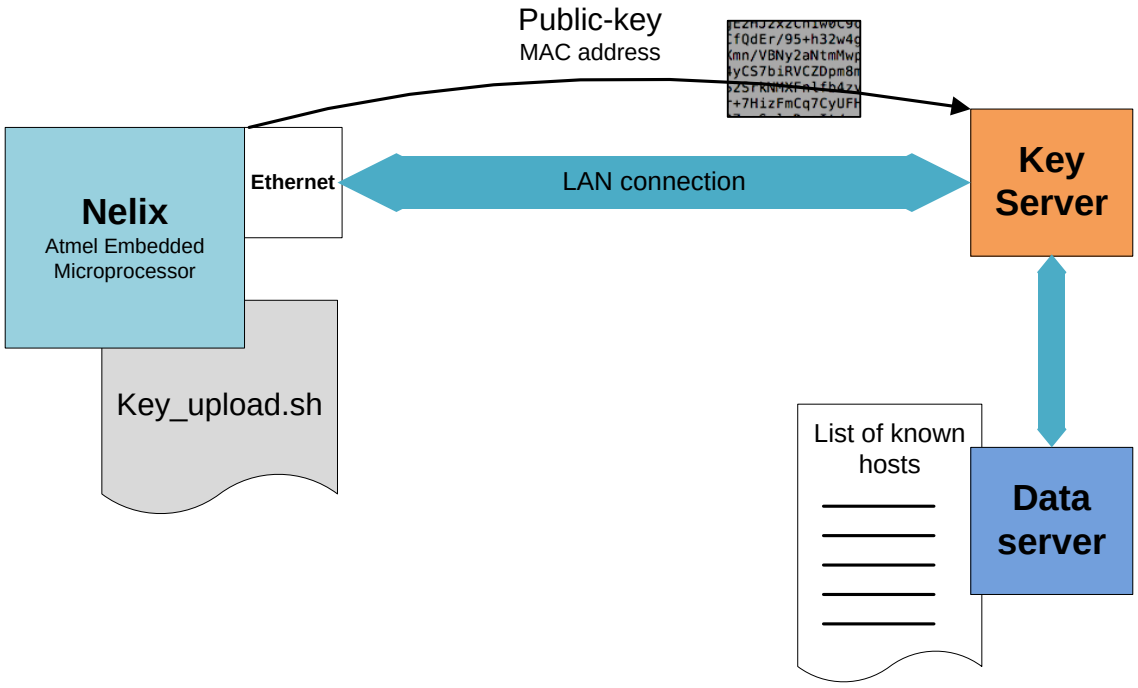


Figure 6. SSH key pair sharing over LAN

Key_upload.sh script starts with the generation of the public and private SSH key pairs.

```
ssh-keygen -f /home/user/.ssh/id_rsa -t rsa -N ''
```

This command generates the key pairs with default settings and with an empty pass-phrase. The private key is saved to /home/user/.ssh/id_rsa and the public key in /home/user/.ssh/id_rsa.pub.

The device's MAC address is assigned to a variable with the following command:

```
MAC_ADDRESS=`cat /sys/class/net/eth0/address`
```

The target IP address where the keys will be uploaded to can be configured by modifying the IP_ADDRESS variable in the script. The script sends a ping request to the target server. When the ping request is successful, the generated public key is uploaded to the key sharing server as request parameters of the wget command. The device's own MAC address is also sent to pair sensor units and keys accordingly.

```
wget –spider –S "http://$IP_ADDRESS/index.php?key=$PUB_KEY"
```

The key sharing server handles saving of the key and MAC address into a list as a pair. In this project, the MAC address was initially used to differentiate different sensor nodes

from each other but the sensor clients are also given a device ID which is used for client identification within the measurement data server. With this ID number, the measurement data server can maintain a list of clients, their configuration and firmware versions. Connection between the key sharing server and the measurement data server is not described in this document.

### 5.6 Possible Vulnerabilities

How the measurement data server's identity in the form of a public key is given to Nelix is not defined. With this approach the client does not have a way of making sure it is communicating with the authentic host. Any malicious third-party could pose as the host to the sensor client. The client would accept this previously unknown host without any verification. One simple way to avoid this situation would be to introduce the measurement data server to the clients during production, the same way as sensor nodes are introduced to the measurement data server. SSH configuration parameter StrictHostKeyChecking could be set to "yes" on the sensor client as well which would cause the sensor clients to automatically abort connections to servers that do not appear on the client-side list of known hosts.

In this project's application both public-key authentication and a password are required for server access. Public-key authentication alone results in a man-in-the-middle attack becoming somewhat impossible. The shared secret is calculated by the peers when initializing the connection, and as mentioned before, no peer can determine alone what this secret will be. This shared secret also includes a session identifier. In case of a man-in-the-middle attack, the attacker has two connections and two shared secrets, one to the client and one to the server. In case of public-key access authentication, the client will send a signature to the server using its private key. This signature includes among other things, the session identifier. As the attacker does not have the public key necessary to decrypt this signature, it cannot relay it to the server. The session identifier mismatch between the client and the server will lead to authentication failure. [20.]

# 6 Intel Hex File Format

Intel Hex is a file format used to present binary data as an ASCII text file. The Hex file consists of records, each of which contains six fields as illustrated in figure 7. The Intel Hex file format defines also six different record types:

- Data Record (8-, 16-, or 32-bit formats)
- End of File Record (8-, 16-, or 32-bit formats)
- Extended Segment Address Record (16- or 32-bit formats)
- Start Segment Address Record (16- or 32-bit formats)
- Extended Linear Address Record (32-bit format only)
- Start Linear Address Record (32-bit format only)

For the application described in this thesis, only necessary record types required are Start Segment Record, Extended Segment Address Record, Data Record and End of File Record. Start Segment Record defines the starting address of the object file's execution. Extended Segment Address Record contains a 16-bit segment base address that is multiplied by 16 and summed to the Data Record’s address to form the starting address for the data. This allows addressing up to one megabyte of address space. Data Record holds the data bytes that form a piece of the image to be loaded into the target device’s memory. End of File Record specifies the end of the object file. [21.]

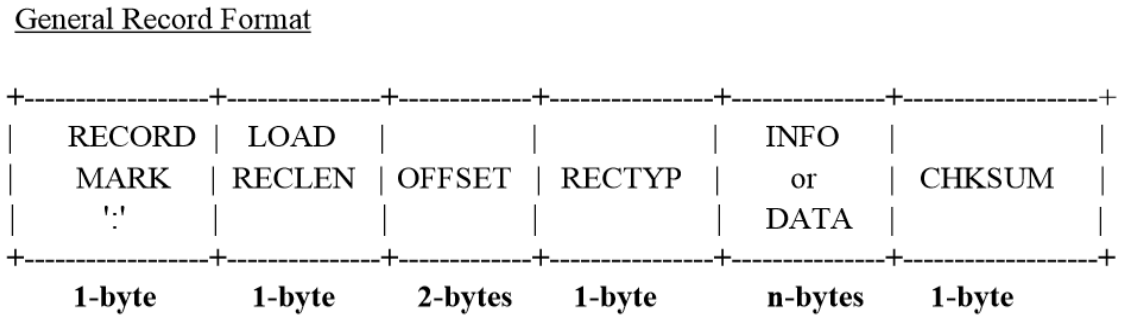


Figure 7. Intel Hex data record format. Copied from Intel [21].

Each of the above mentioned records consists of six fields as illustrated in figure 7. Each line starts with a 1 byte long RECORD MARK field, character ":". The second field LOAD RECLEN defines the length of the information or data field in bytes. This field is 1 byte;

thus, the maximum length for the data field is 255 bytes. The third field OFFSET is only used for data records and is 2 bytes long. It defines the address offset that is summed to the most previous Segment Address Record to form the complete address for the data. On other than data records, this field should be "0000". The fourth field RECTYP specifies the record type of the current line and is 1 byte long. The fifth field INFO or DATA is interpreted according to the RECTYP field. This field is the only one with variable length. LOAD RECLEN being only 1 byte long restricts the INFO or DATA field length to 255 bytes. The last field is CHKSUM and it contains the checksum for the current record. The checksum is calculated by summing together all the bytes in the line excluding the checksum and the RECORD MARK character in the beginning of the record. The final checksum is formed by taking a two's complement from that sum. [21.]

# 7 Description of Armupdate

Armupdate is a program written for Nelix to handle the air quality sensor firmware update process. It is written in the C++ language as it offers a higher level of abstraction than C. C++ enables easier error handling, memory management and container classes that automate many of the otherwise manual operations in C. The program consists of three classes: ByteLine, ImageHandler and TcpConnection. Each of these classes performs a part of the whole process. Distributing a complex program into smaller subprograms helps in the development process by enabling the developer to concentrate on one small task instead of a whole program.

The firmware to be updated on the air quality sensor is received to Nelix as an Intel Hex file. The sensor's uploading protocol requires the firmware image as binary. The hex file is converted to binary on Nelix before starting the uploading process. Converting Intel Hex file to binary is one of the operations included in the Armupdate program. Figures 8 and 9 give an example of the firmware image before and after conversion. Note that figure 9 is with hexadecimal encoding which represents the numeric binary values as hexadecimal numbers as the actual binary data would not be human readable.

```
1  :020000021000EC
2  :10000000E07F00103F01010009010100B915010066
3  :10001000D9150100FB15010019160100000000B0
4  :1000200000000000000000000000000013010100BB
5  :10003000150101000000000017010100B943010093
6  :1000400099150100296B01001B0101001B01010032
7  :10005001B01010031E401001B0101005D6701008B
8  :10006001B0101001B0101001B0101001B0101001C
```

Figure 8. Example of a part of firmware image in ASCII hex file format in text editor

```
1  e07f 0010 3f01 0100 0901 0100 b915 0100
2  d915 0100 fb15 0100 1916 0100 3226 f9ef
3  0000 0000 0000 0000 0000 0000 1301 0100
4  1501 0100 0000 0000 1701 0100 b943 0100
5  9915 0100 296b 0100 1b01 0100 1b01 0100
6  1b01 0100 31e4 0100 1b01 0100 5d67 0100
7  1b01 0100 1b01 0100 1b01 0100 1b01 0100
8  1b01 0100 1b01 0100 1b01 0100 1b01 0100
```

Figure 9. Example of a part of converted binary firmware image in text editor with hexadecimal encoding

Armupdate is started with three arguments that define the target address (hostname), target port (port) and the name of the firmware image file (file name). Three different instances of three classes are created within the program as illustrated in figures 10 and

11. The first object created is ImageHandler which handles the firmware image conversion from Intel Hex to binary. ImageHandler receives the name of the Intel Hex file as an instance call parameter and converts the addresses and data from that file into variables in dynamic memory. To convert a single data record into binary data, ImageHandler utilizes an instance of ByteLine class. File format conversion is described in more detail in chapter 5.2 Conversion from Intel Hex to binary. TcpConnection is the object that handles all the TCP connection operations necessary for communicating with the sensor's interface. It receives the target connection address and port as instance call parameters and opens a connection with the given parameters. TcpConnection also includes member functions which are used to send commands to the sensor.

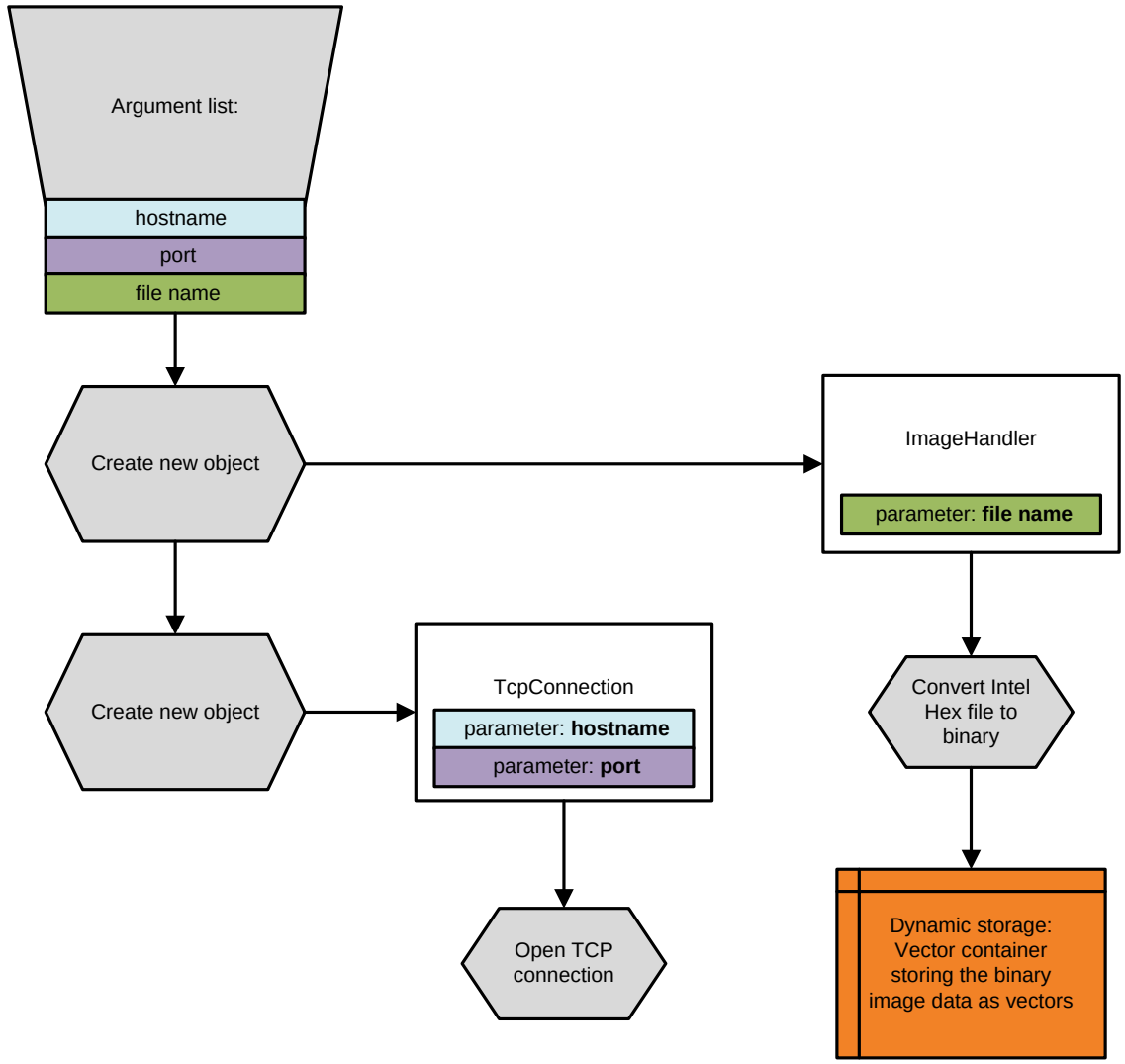


Figure 10. Flowchart diagram of the first steps in Armupdate main function

## 7.1 Using Socket API for Network I/O

Connection from Nelix to the air quality sensor unit is established with the Socket application program interface. Originally developed as a part of the BSD UNIX operating system, socket has become a de facto standard with adding network connectivity to UNIX and Linux based systems. Socket defines an interface for network input and output that loosely resembles the UNIX file I/O open-read-write-close paradigm. When a program requests a socket to be created, the operating system returns a handle to the socket that can be referenced by the program. [22.] A socket is created with the function *socket*, which takes three arguments as illustrated in listing 1.

```
// Create the unconnected socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

Listing 1. Creating a socket

The first argument defines how to interpret the target address as IPv4 and IPv6 addresses differ from each other. The second argument specifies the type of communication. In this case the type of communication is SOCK_STREAM which corresponds to a reliable stream delivery service. The third argument specifies the protocol but can be left empty as TCP is the only protocol that supplies reliable stream delivery service. At this point the socket is created without defining a destination address or port. Connecting the socket is not necessary for connectionless protocols such as UDP. With TCP, socket must be connected before it can be used. [22.] To connect the socket, function connect is used, as illustrated in listing 2.

```
if (connect(sockfd,(struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
    std::cout << "ERROR connecting";
usleep(10000);
```

Listing 2. Opening a socket

The first argument sockfd is the handle of the socket to be connected. The second argument is a pointer to sockaddr which defines an address to which the socket will be bound. The third argument specifies the length of the destination address in bytes. When creating a socket to utilize the TCP protocol, the connect function returns an error if opening a connection was unsuccessful. If using the UDP protocol, a connection is not opened at this point and the destination address is only saved to be used when reading or writing. Utilizing UDP also enables specifying the destination address on each input and output

operation separately. [22] Sending data to a destination address is done with the write function, illustrated in listing 3.

```
// write buffer to socket
n = write(sockfd, buffer, auxString.size());
usleep(10000);

    if (n < 0)
        std::cout << "ERROR writing to socket";

// clear the buffer
memset(buffer, 0, BUFFER_SIZE);
```

Listing 3. Writing to socket

The first argument for the write function defines the socket to be used. In this application the TCP protocol is used so the socket has to be connected before writing as previously described. The second argument buffer is a pointer that points to the data to be sent to destination. The third argument is the length of the buffer. The armupdate program has a method writeTcp that utilizes the write function. Listing 4 illustrates how different content is appended to auxString, which is an instance of a C++ String class container. The write function takes the buffer as a c-string, which is a pointer to an array of characters. The String class has a member function copy that is used to copy the contents of a C++ string into an array of characters. The buffer is then passed to the write function as illustrated in listing 3.

```
std::string auxString;
// clear the buffer
memset(buffer, 0, BUFFER_SIZE);
// append header
auxString = auxString + globalHeader;
// append payload size
auxString = auxString + integerSeparator16(payload.size());
// append payload
auxString = auxString + payload;
// copy String as a c-string to buffer array
auxString.copy(buffer, auxString.size(), 0);
```

Listing 4. Appending different fields to buffer array

After each write operation, a response is also read from the socket as illustrated in listing 5. The response is not used for anything but verifying that the sensor responded with a length more than zero bytes. No response would result in an error.

```
// read response from socket to buffer
n = read(sockfd, buffer, 255);
```

Listing 5. Reading from socket

After all necessary communication with the sensor is over, the socket is closed with the close function. The handle for the socket to be closed is the only argument for this function.

## 7.2 Firmware Conversion Classes

ImageHandler receives a file name as an instance call parameter and opens the ASCII hex file for reading and reads through the file line by line. As a line is read, ImageHandler creates an instance of ByteLine and passes the line read as an instance call parameter. ByteLine parses through the received string and checks the record's type, converts and saves the data to the appropriate public variable. Converted addresses are saved as numeric values in unsigned char variables and data records are saved to a vector container. How ImageHandler operates with ByteLine is illustrated in listing 7.

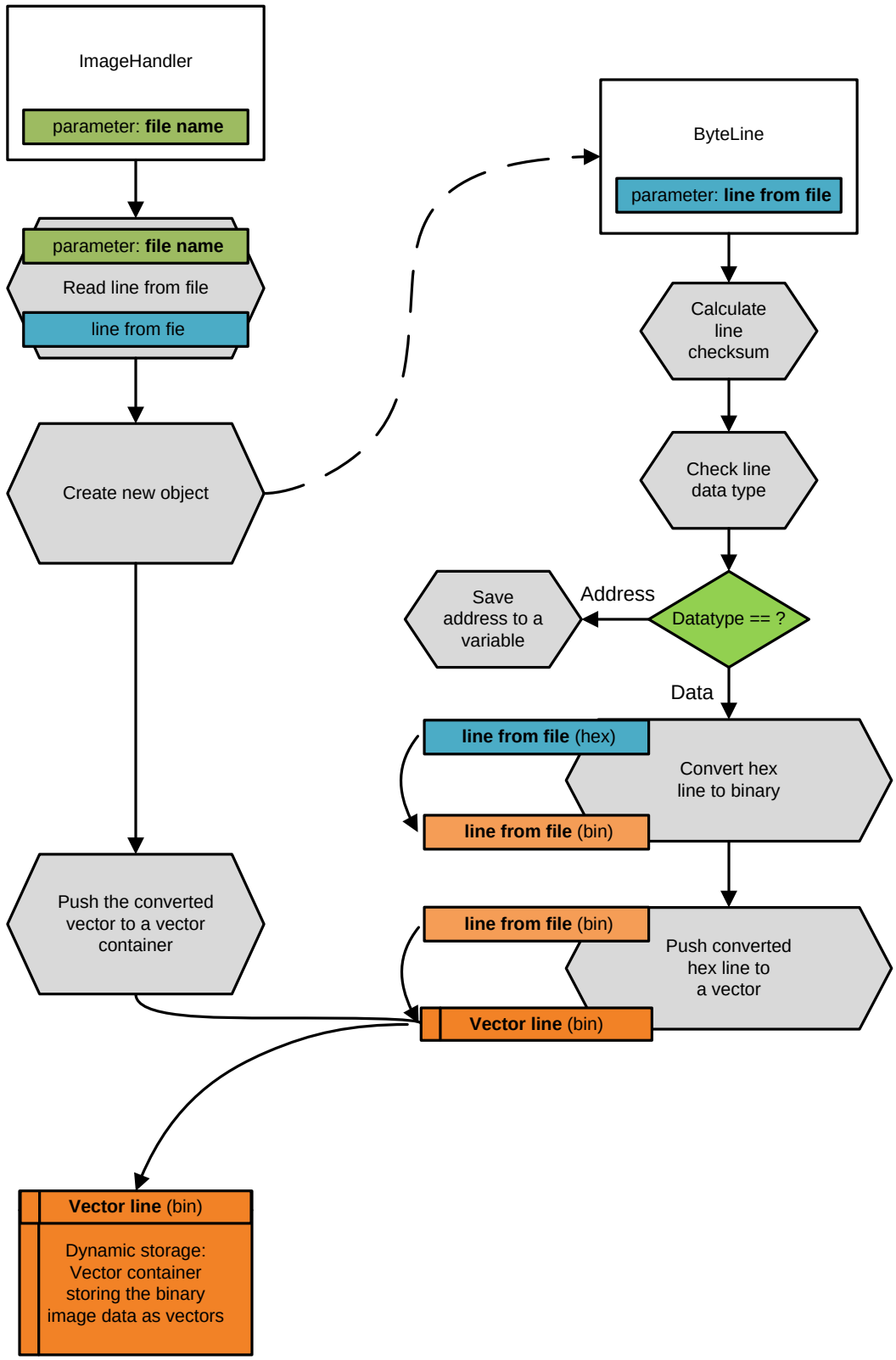


Figure 11. Flowchart diagram of ImageHandler and ByteLine objects

ImageHandler can access the ByteLine object's data members that are set to public, such as start and segment address and the data vector. Another way of accessing another object's data members would be to define the "set" and "get" methods inside the class that would allow modifying or retrieving the variables. After a data line is read and processed, ImageHandler saves the data vector formed by ByteLine as a new element in the vector container listVector. Each data line of the converted binary image is then held in a vector container as a single vector as illustrated in listing 7. When the whole ASCII hex file is processed, a checksum of the image is calculated and written to offset 1C in the firmware image. This image checksum is mandatory as it is required by the air quality sensor's firmware loader.

```
// Read string from the hex file and pass it to ByteLine
while (getline(hexFile, readLine)) {
    // remove first character ":"
    readLine.erase(0, 1);
    ByteLine *bl = new ByteLine(readLine, extendedSegAddress, startSegAddress);
    hexLineCount++;
    // store each data line to datavector
    if (bl->dataType == 0) {
        listVector.push_back(bl->dataVector);
    }
    delete bl;
}
```

Listing 7. Snippet from Image_Handler.cpp source code

## 7.3 Conversion from Intel Hex to Binary

The Intel Hex file is an ASCII text file. Each record's data field is converted into numeric values and their checksum is verified before conversion. Each pair of characters represents one 8-bit value (0 - 255). Both characters are first converted into 4-bit values (0 - 15). The most significant character's numeric value is bitwise shifted left four times before summing the most and least significant values together forming a single 8-bit value in binary format. The object ByteLine handles the single record format conversion. It receives a record read from the Intel Hex file as a string in its instance call parameters. A checksum of the record is calculated with the calculateChksum function and then compared to the record's last two characters as illustrated in listing 8.

```
ByteLine::ByteLine(std::string oneLine, unsigned char *extendedSegAddress, unsigned char *startSegAddress) {

    oneLineChksum = calculateChksum(oneLine);
    // check if the binary value of the last two characters of the string equals the checksum
    if (convertCharToHexDigit(&oneLine[(oneLine.length() - 2)]) != oneLineChksum) {
        std::cout << "Invalid single line checksum" << "\n";
        std::cin.get();
        std::exit(-1);
    }

    dataType = convertCharToHexDigit(&oneLine[6]);

unsigned char ByteLine::calculateChksum(std::string oneLine) {
    unsigned int i;
    unsigned char sum = 0;
    for (i = 0; i < oneLine.length() - 2; i = i + 2) {
        sum = sum + convertCharToHexDigit(&oneLine[i]);
    }
    sum = ~sum;
    sum++;
    return sum;
}
```

Listing 8. ByteLine object and single record checksum verification

All different record types are converted in the same way. Different record types are stored to different variables after conversion. This variable selection is done inside a switch case structure in which the record's data type functions as a selector. After conversion, data records are saved as a vector to a vector container and addresses are saved as numeric values in unsigned char variables. Data record's converted values are saved in a vector container using a vector class member function push_back which adds a new element to the end of the vector after its current last element. Push_back also increases the container's size within the limits of its capacity. [23.] Selecting into which variable to save the converted value and passing pointers to the characters is illustrated in listing 9.

```
int i;
int j = 0;
byteCount = convertCharToHexDigit(&oneLine[0]);
switch (dataType) {
    case 0:     // Data
        for (i = 0; i < byteCount * 2; i = i + 2) {
            dataVector.push_back(convertCharToHexDigit(&oneLine[i + 8]));
            j++;
        }
        break;
    case 1:     // End of file
        eofReached = 1;
        break;
    case 2:     // Extended segment address
        for (i = 0; i < byteCount * 2; i = i + 2) {
            extendedSegAddress[j] = convertCharToHexDigit(&oneLine[i + 8]);
            j++;
        }
        break;
    case 3:     // start segment
        for (i = 0; i < byteCount * 2; i = i + 2) {
            startSegAddress[j] = convertCharToHexDigit(&oneLine[i + 8]);
            j++;
        }
        break;
    default:
        std::cout << "Invalid single line datatype > 03 " << "\n";
        std::exit(-1);
}
```

Listing 9. Target variable selection and passing a pointer to conversion function convertCharToHexDigit

As the Intel Hex file uses two characters as a pair to present a single 8-bit value, a pointer to the first character is passed to the convertCharToHexDigit function. Inside this function, the received pointer is incremented by one to reach the following character. The first character's value is shifted left eight times because it presents the most significant bits in the final value. The second character presents the least significant bits. Function convertCharToHexDigit utilizes another function convertAlphabeticalHexDigitToNumeric to resolve numeric values represented by the hexadecimal characters. This process is illustrated in listing 10.

```
unsigned char ByteLine::convertCharToHexDigit(char *pointerToChar) {
    // convert two characters ([i]and [i]+1) to two bytes, sum them together
    return convertAlphabeticalHexDigitToNumeric(*pointerToChar) * 16
    + convertAlphabeticalHexDigitToNumeric(*(pointerToChar + 1));
}

unsigned char ByteLine::convertAlphabeticalHexDigitToNumeric(char hex) {
    // convert the alphabetical values (A - F) to numerical (10 - 15)
    if (hex >= 'A')
        return hex - 'A' + 10;
    else
        return hex - '0';
}
```

Listing 10. Converting two ASCII characters into their respective numeric values

Worth noticing is that as can be seen in listings 9 and 10, records are not converted to binary in full length. Each pointer passed to the conversion function has an offset of eight.

This is exactly the length of the RECLEN, OFFSET and RECTYP fields as previously described in listing 7, excluding the RECORD MARK byte that was removed before passing the record to ByteLine object illustrated in listing 7.

Initially, C-type arrays were to be used in this project to store the binary-converted data but were later replaced with vectors. A vector is a container class type in C++ that offers many useful member functions. The most useful property of a container class is that its size can be dynamic and automatically increased whereas C-type arrays require manual memory reallocation and resizing. Using vectors enables an easy way of accessing the container's size and adding new elements. [23.]

## 7.4 Update Protocol Packet Structure

The air quality sensor utilizes another proprietary packet structure on top of TCP for communication through the Ethernet interface. When armupdate opens a TCP connection and sends a message to the sensor, the TCP related headers and other content necessary for the protocol are created automatically into the packet. Inside a TCP packet there is another packet with three fields, a header, a payload size and a payload. The packet has a 12 byte long header which consists of a 9 byte long static identification field and two bytes for expressing the payload size. The content of the proprietary packet is created in a separate method for each command. All commands are sent with a common writeTcp method. For security reasons the content or structure of the packet is not described in this document. The payload can be one of the various commands that are used to communicate with the air quality sensor. When uploading new firmware to the sensor, the payload is a piece of the firmware image.

After the firmware image conversion, TcpConnection starts the firmware loader on the sensor by sending the appropriate command. The flash memory on the sensor is first erased and then overwritten. TcpConnection then assembles the binary image from the vector into 1024 byte sized packets, and uploads them one at a time to the sensor’s firmware loader with the write flash command. After the flash is written, a restart command is sent to the firmware loader with the first and last addresses of the new firmware image to let the firmware loader know at which memory area the program is. While the communication link to the sensor is open, a keepalive command is sent every 10 seconds to prevent automatic closing of the connection.

# 8 Future Development

In its current functioning condition, the system is able to gather measurement data from the sensor and upload it to the server. Nelix also has a proven ability to upload new firmware to the sensor. After each measurement data packet is uploaded to the measurement data server, a new configuration file is downloaded. This way the server has a way to transmit information to Nelix even though all traffic is initiated only by Nelix. The file downloaded is a JSON file, although with a small modification to the nelix_main.sh script it could be any other file format. A convenient way would be to replace the downloading of a JSON with a bash script file that when executed would download the firmware file and run the armupdate program to update the sensor's firmware.

Using a digital signature could be a way to ensure that the firmware file was received from the authentic host. This could be implemented using GnuPG, which is a program that can be used to encrypt and sign data. GnuPG, as well as SSH, uses public-key encryption for encryption and signatures. The server would generate a signature of the firmware file with its private key. The client receiving the file decrypts the signature with the server's public key. GnuPG uses its own set of private and public keys. [24.] These keys could also be used for SSH authentication but using the same keys for both authentications would invalidate the benefit of the file signature.

In a paper called Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice, the authors describe the weaknesses of Diffie-Hellman key exchange algorithm against a Logjam attack. They describe that with nation state resources it is possible to break a 1024bit cipher and recommend that keys with a bit length of less than 2048 bits not to be used. In the application described in this thesis, changing the key length would require modifying the server's SSH configuration to only support key exchange algorithm diffie-hellman-group14-sha1 which uses a 2048 bit long key. [25.] The key exchange method is selected from a list of supported methods provided by the client and server. Both ends must support the same method. The client must support this key exchange algorithm in order to be able to establish a connection to the server. Longer keys also require more calculation time and they are calculated in the beginning of each connection. This would most likely not result in any loss of performance on the system as new measurement data can be received from the sensor only once every ten minutes. Another recommendation is using the Elliptic Curve Diffie-Hellman algorithm for key exchange to avoid any feasible attacks on the key.

## 8.1 Key Revocation

In the case of one of the sensor clients being stolen or their private keys being otherwise compromised, a procedure for denying them access to the server must be defined. Applications where a certificate authority is used, a list of revoked keys is published at defined intervals. Any application using a certificate must check within this list whether or not to accept a certificate as trusted. In this project there is no certificate authority implemented in the traditional sense, as a similar service is implemented in the measurement data server as a list of known hosts as described in chapter 5.5. If the security of one of the sensor client becomes compromised, its access to the measurement data server can and must be denied by removing their entry from the list of known hosts. Along with the authentication keys the sensor clients also contain the address, username and password used to access the measurement data server. These alone do not suffice to access the server as long as the server denies access for the revoked public key.

If the security of the measurement data server would be compromised the sensor clients would need to generate new keys as their current ones provide access to the measurement data for whomever has access to the private key of the measurement data server. The only way to generate new keys on the sensor clients is to access them locally and reprogram them and use the same key sharing procedure as in production to gather the new public keys manually. If the sensor nodes are operating at volumes of hundreds or thousands, the time it takes to access all of them to manually generate new keys would be disastrous. An automated process for sensor client key revocation and regeneration would greatly reduce the down time in this sort of worst case scenario.

## 8.2 Implementing a Secure Boot

A root-of-trust implementation as described in chapter 2.3 is possible to achieve with the existing SAMA5D4-XULT hardware. A secure boot loader is embedded in the ROM that can be used to prevent loading or running unauthentic or unauthorized application software. Implementing secure boot would improve overall security and mitigate the chance of a third party being able to use the hardware and network connection of the device.

SAMA5D4 also includes TrustZone, which is an architecture extension that allows for the resources to be partitioned for two operating systems. Both secure and non-secure operating system share the same the hardware resources and peripherals. With TrustZone, the secure operating system can monitor and control the non-secure operating system's resources. The secure operating system can be small and simple and the non-secure a fully functioning operating system with network access and other peripheral functions. TrustZone also includes separate Advanced Interrupt Controllers and DMA controllers for both secure and non-secure access. [26.]

# 9 Conclusion

The project described in this thesis started as wireless connection functionality to be added to a standalone air quality sensor. Methods and tools changed along with the requirements and when the device was specified to be running an operating system, the ability to update the sensor's firmware became feasible. At first, writing a program to convert a file from a format to another seemed very difficult but ended up being quite a straightforward operation. Understanding the structure of the Intel Hex file format was challenging with all its complexity. Another goal of this project was to create a way to securely send data from a remote location to a server over the internet. This proved to be very challenging considering all the security related topics that needed to be investigated. A lot of the data sent over the internet needs to be somehow private and secure. This need has helped develop many advanced encryption techniques, SSH tunneling being one of the most popular methods today.

Along with the development of increased secrecy come the techniques built to break these encryptions. The Logjam paper that raised awareness about the probable security issues of Diffie-Hellman key exchange was released on May 2015 [25]. The development of Nelix was already finished then, and during development the 1024bit Diffie-Hellman key exchange algorithm was thought to be practically unbreakable. This example demonstrates how every developer has to stay with the newest trends and news in the industry. The current state of Nelix leaves a lot of room for improvement and new features. One key part of keeping the system secure was the requirements for Nelix not responding to inbound connections. This is proven possible with Nelix with its method of actively polling the server for new information and being able to retrieve it autonomously.

## References


1. Westerman George. The Internet-Connected Engine Will Change Trucking [online]. Harvard Business Review; November 2014.
   URL: https://hbr.org/2014/11/the-internet-connected-engine-will-change-trucking. Accessed 25 October 2016.

2. IDG News Service. NarrowBand IoT Atandard for Machines Moves Forward [online]. Computer World; September 2015.
   URL: http://www.computerworld.com/article/2984928/mobile-wireless/narrowband-iot-standard-for-machines-moves-forward.html. Accessed 17 September 2016.

3. Lehto Tero. Nokia ja Sonera testasivat uutta verkkoa: nopeus ei päätä huimaa, mutta laitteen akku kestää jopa 10 vuotta [online]. Tekniikka & Talous; September 15 2016.
   URL: http://www.tivi.fi/Kaikki_uutiset/nokia-ja-sonera-testasivat-uutta-verkkoa-nopeus-ei-paata-huimaa-mutta-laitteen-akku-kestaa-jopa-10-vuotta-6590915. Accessed 16 September 2016.

4. Poole Ian. SIGFOX for M2M & IoT [online]. Radio-Electronics.com.
   URL: http://www.radio-electronics.com/info/wireless/sigfox/basics-tutorial.php. Accessed 17 September 2016.

5. Morris Anne. Sigfox Rolls out IoT Network in Finland [online]. Fierce Wireless; June 3 2016.
   URL: http://www.fiercewireless.com/europe/sigfox-rolls-out-iot-network-finland. Accessed 17 September 2016.

6. SRI Consulting Business Intelligence. Disruptive Technologies Global Trends 2025 [online]. National Intelligence Council (NIC); April 2008.
   URL: https://fas.org/irp/nic/disruptive.pdf. Accessed 16 September 2016.

7. Goodin Dan. 9 Baby Monitors Wide Open to Hacks that Expose Users' Most Private Moments [online]. Ars Technica; September 2015.
   URL: http://arstechnica.com/security/2015/09/9-baby-monitors-wide-open-to-hacks-that-expose-users-most-private-moments/. Accessed 23 October 2016.

8. Miller Charlie, Valasek Chris. Remote Exploitation of an Unaltered Passenger Vehicle [online]. August 2015.
URL: http://illmatics.com/Remote%20Car%20Hacking.pdf. Accessed 24 September 2016.

9. Bonderud Douglas. Leaked Mirai Malware Boosts IoT Insecurity Threat Level [online]. Security Intelligence; October 4 2016.
URL: https://securityintelligence.com/news/leaked-mirai-malware-boosts-iot-insecurity-threat-level/. Accessed 24 September 2016.

10. Zeifman Igal, Bekerman Dima, Herzberg Ben. Breaking Down Mirai: An IoT DDoS Botnet Analysis [online]. Imperva Incapsula; October 10 2016.
URL: https://www.incapsula.com/blog/malware-analysis-mirai-ddos-botnet.html. Accessed 24 October 2016.

11. Loisel Yann, di Vito Stephane. Securing the IoT: Part 2 - Secure Boot as Root of Trust [online]. Maxim Integrated; January 11 2015.
URL: http://www.embedded.com/design/safety-and-security/4438300/Securing-the-IoT--Part-2---Secure-boot-as-root-of-trust-. Accessed 10 Sepember 2016.

12. Praphul Chandra. Bulletproof Wireless Security. Newnes; 2005.

13. Bauer Michael D. Linux Server Security, Second Edition. O'Reilly Media Inc; 2005.

14. Atmel. Picture of SAMA5D4 Xplained Ultra Evaluation Kit.
URL: http://www.atmel.com/tools/ATSAMA5D4-XPLD-ULTRA.aspx. Accessed 27 November 2016.

15. Seward J, Nethercote N, Weidendorfer J. Valgrind 3.3 - Advanced Debugging and Profiling for Gnu/Linux Applications. Network Theory Ltd; 2008.

16. MSDN. Public Key Infrastructure [online]. Microsoft;
URL: https://msdn.microsoft.com/en-us/library/windows/desktop/bb427432. Accessed 16 September 2016.

17. Ylönen T, Lonvick C. The Secure Shell (SSH) Transport Layer Protocol [online]. The Internet Engineering Task Force; January 2006.
URL: https://www.ietf.org/rfc/rfc4253.txt. Accessed 9 May 2016.

18. Rinne Timo, Ylönen Tatu. Secure Copy, Linux Man Page [online]. April 2013.
URL: https://linux.die.net/man/1/scp. Accessed 17 November 2016.

19. Sshpass Linux man page [online].
URL: http://linux.die.net/man/1/sshpass. Accessed 9 May 2016.

20. Bezroutchko Alexandre. SSH Man-in-the-Middle Attack and Public-Key Authentication Method [online]. Gremwell.com; 25 December 2010.
URL: http://www.gremwell.com/ssh-mitm-public-key-authentication. Accessed 9 May 2016.

21. Hexadecimal Object File Format Specification Revision A [online]. Intel; January 6 1988.
URL: http://microsym.com/editor/assets/intelhex.pdf. Accessed 17 November 2016.

22. Comer Douglas E. Internetworking with TCP/IP Volume One, Fifth Edition. Pearson; 2005.

23. Vector C++ Reference [online]. Cplusplus.com;
URL: http://www.cplusplus.com/reference/vector/vector/. Accessed 17 November 2016.

24. Copeland Matthew, Grahn Joergen Grahn, A David. The GNU Privacy Handbook [online]. The Free Software Foundation; 1999.
URL: https://www.gnupg.org/gph/en/manual.html. Accessed 9 May 2016.

25. Adrian David, Bhargavan Karthikeyan Bhargavan, Durumeric Zakir, Gaudry Pierrick, Green Matthew, Halderman J. Alex, Heninger Nadia, Springall Drew, Thomé Emmanuel, Valenta Luke, VanderSloot Benjamin, Wustrow Eric, Zanella-Béguelin Santiago, Zimmermann Paul. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice [online]. CCS '15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security; 16 October 2015.

URL: https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf. Accessed 9 May 2016.

26. Introduction to ARM TrustZone [online]. Atmel; October 2014.
URL: http://atmel.force.com/support/servlet/fileField?id=0BEG000000002Ur. Accessed 20 November 2016.

## Bash Script key_upload.sh

```
#!/bin/bash

# Pasi Riissanen 2015
# For Nelix, particle sensor project.
# rsa key uploader to local server
# Designed to be ran as a daemon.
# Generates a new key pair and tries to upload it
# until succesful.
# All stdout outputs saved in /var/log/key_upload.log

IP_ADDRESS=10.10.10.1 #Address of the local server
LOG_ENTRY=/var/log/key_upload.log

rm /var/log/key_upload.log
echo "Log started at: " `date` >> $LOG_ENTRY

#Remove older key pairs
rm /home/user/.ssh/id*
echo "Removed old key pairs" >> $LOG_ENTRY

#Create ssh keypair
ssh-keygen -f /home/user/.ssh/id_rsa -t rsa -N ''
echo "Identification saved in /home/user/.ssh/id_rsa" >> $LOG_ENTRY
echo "Public key saved in /home/user/.ssh/id_rsa.pub" >> $LOG_ENTRY

#Find out eth0 HWaddr
MAC_ADDRESS=`cat /sys/class/net/eth0/address`
echo "This device has a mac address: $MAC_ADDRESS" >> $LOG_ENTRY

#Read public key from file id_rsa.pub
PUB_KEY=$(</home/user/.ssh/id_rsa.pub)
echo "Generated public key: $PUB_KEY" >> $LOG_ENTRY

#Test connection to local server
#Loop until connection successful
echo "Trying to connect to $IP_ADDRESS" >> $LOG_ENTRY
while ! ping -c 1 $IP_ADDRESS; do echo "- timeout :(" >> $LOG_ENTRY;
done; {
 echo "$IP_ADDRESS responds to ping" >> $LOG_ENTRY
 HTTP_RESPONSE_OK=200
 CHECK_VALUE=0
 while [ "$CHECK_VALUE" -ne "$HTTP_RESPONSE_OK" ]
  do
   echo "Sending public rsa key.." >> $LOG_ENTRY
   CHECK_VALUE=`wget --spider -S "http://$IP_ADDRESS/in-
dex.php?key=$PUB_KEY" 2>&1 | grep "HTTP/" | awk '{print $2}'`
   echo "HTTP returned: $CHECK_VALUE" >> $LOG_ENTRY
  done

 CHECK_VALUE=0
 while [ "$CHECK_VALUE" -ne "$HTTP_RESPONSE_OK" ]
  do
   echo "Sending HWaddr.." >> $LOG_ENTRY
```

```
    CHECK_VALUE=`wget --spider -S "http://$IP_ADDRESS/in-
dex.php?mac=$MAC_ADDRESS" 2>&1 | grep "HTTP/" | awk '{print $2}'`
    echo "HTTP returned: $CHECK_VALUE" >> $LOG_ENTRY
  done
}

#Remove the cronjob that runs this script on boot
#Must be ran as root :|
crontab -l -u root | grep -v key_upload | crontab -u root
```