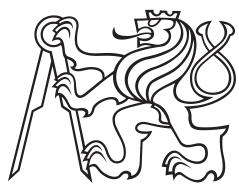


Master Thesis



Czech
Technical
University
in Prague

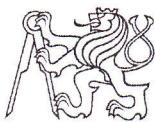
F3

Faculty of Electrical Engineering
Department of Measurement

IoT devices for smart buildings with ECC-based data signatures

Bc. Kateřina Juranová

Supervisor: doc. Ing. Radislav Šmíd, Ph.D.
May 2017



Akademický rok 2016-17

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Kateřina Juranová**

Studijní program: **Inteligentní budovy**

Název tématu česky: **IoT zařízení pro inteligentní budovy s digitálními podpisy dat pomocí kryptografie eliptických křivek**

Název tématu anglicky: **IoT Devices for Smart Buildings with ECC-based Data Signatures**

Pokyny pro vypracování:

Cílem diplomové práce je navrhnout a implementovat protokol umožňující ověření integrity dat posílaných z IoT senzorů do agregátorů. Tako by mělo být možné ochránit tyto systémy proti vložení nedůvěryhodných dat z potenciálně škodlivých IoT zařízení na IoT síti inteligentních budov spojené s narušením provozu.

Vzhledem k tomu, že IoT zařízení mají obvykle nízký výkon i operační paměť, zaměřte se na implementaci podepisování zasílaných zpráv pomocí kryptografie eliptických křivek a dosažení nízkých požadavků na paměť a procesor při zachování silné kryptografické bezpečnosti podepisovaných dat. Referenční implementace by měla obsahovat grafické rozhraní pro správu a zobrazování dat ze skupiny teplotních čidel a bude použita v testovací lokalitě pro získání reálných dat.

Seznam odborné literatury:

- [1] Liu Z., Großschädl J., Li L., Xu Q. , Energy-Efficient Elliptic Curve Cryptography for MSP430-Based Wireless Sensor Nodes. In: Information Security and Privacy. ACISP 2016. Lecture Notes in Computer Science, vol 9722. Springer 2016
- [2] Bormann, C., Ersue, M., Keranen, A.: Terminology for Constrained-Node Networks. Internet Engineering Task Force, Light-Weight Implementation Guidance Working Group, RFC 7228, May 2014
- [3] Hankerson, D.R., Menezes, A.J., Vanstone, S.A.: Guide to Elliptic Curve Cryptography. Springer, New York 2004

Vedoucí diplomové práce: doc. Ing. Radislav Šmíd, Ph.D.

Datum zadání diplomové práce: 1. února 2017

Platnost zadání do¹: 30. září 2018



V Praze dne 1. 2. 2017

¹ Platnost zadání je omezena na dobu tří následujících semestrů.

Acknowledgements

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 25, 2017

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 25. května 2017

Abstract

In this thesis we develop an optimized version of digital signature algorithm based on Schnorr's algorithm using a Twisted Edwards curve over a finite field that can run on low-power and low-memory devices. We use this implementation as a basis for adding message authenticity feature to communication between sensor devices and computer program in a wireless sensor network. The sensor devices sign the message and the computer program verifies the signature and recognizes known devices by their public keys. For demonstration purposes a simple graphical user interface is presented to allow visualizing the temperatures and signature verification results.

Keywords: IoT, ECC, msp430, message authentication, low-power MCU, IoT security, ECC-based data signatures

Supervisor: doc. Ing. Radislav Šmíd, Ph.D.

Abstrakt

Tato práce se zabývá implementací optimalizované verze digitálního podpisu na základě Schnorrova algoritmu za použití eliptické křivky typu "Twisted Edwards curves" nad prvočíselným konečným tělesem, kterou je možné využít v mikrokontrolérech s malým výkonem a malou velikostí paměti. Tato implementace je následně použita pro přidání možnosti autentizace zpráv při komunikaci senzorů a počítačového programu v bezdrátové síti senzorů. Zařízení se senzorem podepíší zprávu s daty a počítačový program následně tento podpis ověří a rozpozná registrovaná zařízení na základě jejich venkovních klíčů. Jako ukázka použití této funkcionality bylo realizováno jednoduché grafické rozhraní, které zobrazuje teploty a stavy ověření podpisů jednotlivých senzorů.

Klíčová slova: IoT, ECC, msp430, autentizace zpráv, nízko-energetické MCU, IoT bezpečnost, podpisy dat pomocí Kryptografie eliptických křivek

Překlad názvu: IoT zařízení pro inteligentní budovy s digitálními podpisami dat pomocí kryptografie eliptických křivek.

Contents

1 Introduction	1
1.1 Common hardware	3
1.2 Elliptic curve cryptography	4
2 MSP430 family	7
2.1 eZ430-RF2500 Sensor Monitor Demo	7
2.2 Memory configuration	8
3 SimpliciTl	11
3.1 Architecture overview	11
3.2 Topology	12
3.3 SimpliciTl configuration	14
3.4 Frame	15
3.5 eXtended Tiny Encryption Algorithm (XTEA)	16
4 Protocol design	19
4.1 Replay attack	20
4.2 Ed25519	21
5 Digital signatures	23
5.1 Porting Daniel Beer's implementation c25519	24
5.2 Optimization	25
5.2.1 Verification of optimizations .	27
5.3 Key Generation	28
5.4 Verification	31
6 Protocol implementation	33
6.1 Communication between Access Point and End Device	33
6.2 Protocol testing	34
6.3 Verification	34
6.4 Measurement	35
6.4.1 Key generation measurement	35
6.4.2 Signature generation measurement	36
7 Thermo GUI	39
7.1 Attaching devices	40
7.2 Toolkit	40
7.3 Verification	42
8 Conclusion	45
Bibliography	47

Figures

1.1 Smart building.[7]	2
1.2 Beaglebone with BB view.[20]	3
2.1 Wireless development eZ430-RF2500 Tool [22].	7
2.2 Sensor Monitor demo [21] opened in CCS studio [2].	8
3.1 SimpliciTI Architecture[17].	12
3.2 Legend[17].	12
3.3 Direct Peer-to-peer[17].	12
3.4 Store-and-forward and peer-to-peer through Access Point[17].	13
3.5 Direct peer-to-peer through Range Extender[17].	13
3.6 Store-and-forward peer-to-peer through Range Extender and Access Point[17].	13
3.7 SimpliciTI Encryption scheme[16].	17
3.8 SimpliciTI frame with encryption enabled.[17]	17
4.1 Replay attack scheme.	21
5.1 Our small Wireless Sensor Network.	23
5.2 The graph of function calls.	24
5.3 MSP430 Random number generation.[15].	28
6.1 Functions called by edsige_sign.	37
7.1 Thermo GUI before devices connect.	41
7.2 Thermo GUI with average temperature.	42
7.3 Our small Wireless Sensor Network.	43

Tables

1.1 EC scalar multiplication times	5
2.1 Memory configuration of MSP430.	9
3.1 SimpliciTI frame structure(disabled security) RD*: Radio-dependent populated by MRFI or handled by the radio itself.	15
3.2 Device info bit values[17].	16
4.1 3-byte message payload	19
4.2 New protocol packed size.	19
4.3 Protocol design.	20
4.4 Splitting the ed25519 key and signature.	21
5.1 Daniel Beer's stack usage figures for key functions [12].	24
5.2 Not optimized stack usage of functions in c25519 [12].	25
5.3 Optimized stack usage of functions in c25519 [12].	27
6.1 Protocol implementation example.	34
6.2 Key generation measurement.	36
6.3 Signature generation measurement.	36
7.1 Device status table.	41

Chapter 1

Introduction

The connection between Internet of Things (IoT) and Elliptic curve cryptography (ECC), described in this thesis, should solve the lack of security in smart devices. By security, we mean confidentiality and authentication. We are tackling the authentication aspect of security by utilizing ECC signatures.

There are many definitions of Internet of Things (IoT), there is no formal specification at this time. Historically, the concept of a IoT utilizing was firstly discussed in 1982, with a modified vending machine known as "The "Only" Coke Machine on the Internet", at Carnegie Mellon University, becoming the first Internet-connected application[9]. The Coke machine was able to report the whole inventory and if the newly added drinks were cold. For our formal definition of IoT, we rely on Network of 'Things' NIST specification [14].

IoT connects multiple technologies such as telecommunication, data mining, machine learning, control systems, sensor networks and embedded systems. When these technologies are together we are able to do incredible things such as making our home smart. IoT applications are, in general, composed of:

- **Sensors**, devices mainly used for measuring properties of environment
- **Aggregators**, devices for data collection
- **Communication channel**, ensures data exchange between all the elements of network
- **eUtilities**, external utility for data processing
- **Decision Trigger**, makes decision based on collected data

The Smart homes concept is becoming very popular as a sense of sustainability and eco-friendliness become more pervasive. People are more interested in "saving" the planet by decreasing our energy consumption as much as possible. In the future, we can expect to be living in smart cities, full of smart buildings, cars, and more. In this thesis, we are building a small Wireless Sensor Network (WSN) where sensors, referred to as End devices, are transmitting data such as temperature, signal strength, and voltage to a receiver known as an Access point. The Access point receives data from the End devices and sends the data to the more powerful device for the further processing.

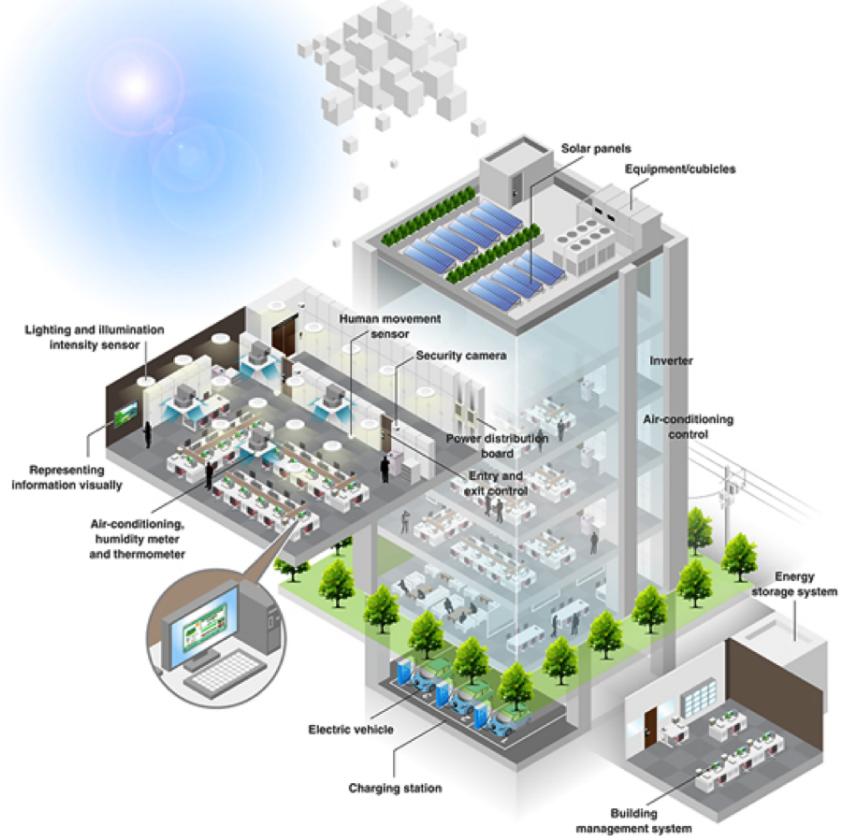


Figure 1.1: Smart building.[7]

We deploy our End devices to the chosen positions in our smart house for measuring temperature. Problems can occur when an intruder acquires the End device and starts sending forged data. Then we cannot rely on sensors anymore and the building becomes vulnerable especially fire or flooding detection. We must ensure that the data is reliable, especially when making critical decisions such as those based on fire detection.

Current wireless IoT protocols only use symmetric block ciphers. For example, Zigbee uses Advanced encryption standard (AES) and SimpliciTI uses Extended Tiny Encryption Algorithm (XTEA) [16]. We chose to use SimpliciTI for its low size only around 4k bytes comparing to Zigbee around 40k bytes.

Because our WSN is constrained by memory and computational power in all of its components and we want our End nodes to last as long as possible if they are running on batteries, we decided to use ECC based digital signatures for their energy-efficiency, [18] high-speed, high security, small size [8]. We will discuss examples of the most common hardware used in the Internet of things later in this chapter.

1.1 Common hardware

Raspberry Pi is very popular and fairly cheap board used mostly for experiments such as creating media centers or for educational purposes. It is possible to extend this mini computer by adding kits and capes such as a wireless card or touch display. It is possible to use precompiled images of various Linux distributions like Raspbian which can be easily downloaded and copied to SD card. There are lot of public open source implementations of many IoT applications such as “Internet of things toilet” which can be downloaded even by non-technical people, like hobbyists, without not any further deep technical knowledge. The mentioned example attempts to lower water consumption of the toilet and detects if we have run out of toilet paper [19].

This application is a very simple implementation with only two sensors. The first sensor is an aquarium liquid level floating switch which is used to detect the toilet tank level and the second sensor is a photo cell located in paper holder socket which checks when the spindle is removed. The data is sent through the wireless protocol to the Raspberry Pi which contains the Python API gspread, transmitting the data to a Google Drive spreadsheet.

Beaglebone is very similar to the Raspberry Pi, but is less popular amongst non-technical people and is more often used for academic proofs of concept. In our proof of concept, we also used the Beaglebone as a device for visualization and aggregation of our data. We connected Access point to the beaglebone for data processing and visualization using the BB-view cape which contains a touchscreen display.



Figure 1.2: Beaglebone with BB view.[20]

Arduino is very popular and is widely used in home automation, as well as in robotics. Early ATMega boards were not able to contain a normal operating system, but these days we have Arduino boards with ARM competing with Beaglebone and are able of containing full Linux installation. There are many popular applications openly shared by users and can be used immediately. It is also possible to buy full kits for various purposes.

Wireless chips such as **flutter** or **MSP430** are generally micro controllers not capable of having a full operating system due to a significant lowering power consumption to extend the life time of the batteries. Normally we connect these chips to more powerful devices to be able to aggregate and process data.

■ 1.2 Elliptic curve cryptography

The Elliptic curve cryptography (ECC) is asymmetric cryptography scheme of public-key cryptography based on elliptic curves over the finite fields. In general, asymmetric cryptography can be used for key exchange as well as for digital signatures. Since purpose of this thesis is focusing on the lack of authenticity in the IoT message exchange. This scheme is perfect, not only because its high speed properties, but also the small size of the keys.

We are using implementation Curve25519 and Ed25519 for low-memory systems [12]. “Curve25519 is a state-of-the-art Diffie-Hellman used for wide variety applications” as stated in [10] such as encrypting messages between two users. For our purposes we are using Ed25519 which is public-key signature system using elliptic-curve signatures [8] and with various useful features such as:

- Fast single-signature verification.
- Even faster batch verification.
- Very fast signing.
- Fast key generation.
- High security level.
- Foolproof session keys.
- Collision resilience.
- No secret array indices.
- No secret branch conditions.
- Small signatures.
- Small keys.

For 128-bit security the size of the key is 32 bytes (256 bits) which compared to the RSA algorithm, the equivalent would be RSA 3072-bit with a key size 384 bytes. As we can see, for IoT as well as general purpose low-memory systems, RSA is not usable at all. Edwards-curve Digital Signature Algorithm (EdDSA) digital signature system based on on Twisted-Edwards curves using Schnorr Algorithm for digital signatures. Schnorr’s signature algorithm is based on an oracle model and it is efficient in generating short signatures.

Curve type	159 bit	191 bit	223 bit	255 bit
Montgomery (variable base)	$3.86 \cdot 10^6$	$6.00 \cdot 10^6$	$8.79 \cdot 10^6$	$12.34 \cdot 10^6$
Twisted Edwards (fixed base)	$1.92 \cdot 10^6$	$3.01 \cdot 10^6$	$4.45 \cdot 10^6$	$6.29 \cdot 10^6$

Table 1.1: Execution time (in clock cycles on an MSP430F1611) of variable-base scalar multiplication on a Montgomery curve and fixed-base scalar multiplication on a twisted Edwards curve over 159, 191, 223, and 255-bit fields.[18]

We use a special implementation for Twisted Edwards curve Ed25519 algorithm. This pseudo-mersenne prime number $2^{255} - 19$ allows us to do fast modular arithmetic and as Table 1.1 shows, is significantly faster than Montgomery curve on MSP430.

Chapter 2

MSP430 family

We are using MSP430 Wireless development Tool which has MSP430F2274 ultra-low-power MCU. The MSP430 is a low power micro controller with 16-bit CPU designed for low cost embedded applications. The MSP430 can use six different low-power modes which include the ability to disable needed CPU clocks. MSP430 can be woken up from the sleep mode very quickly, around 1 microsecond, which minimize current consumption.

Our Wireless MSP430 also contains CC2500 chip has low power transceiver and allows wireless communication on 2.4 GHz with ISM band multi channel. Additionally, our MSP430 supports MSP430 Application UART, allowing communication with the receiver and our PC through a USB serial connection. Using this, we can connect our Receiver to the Beaglebone for data processing and aggregation.

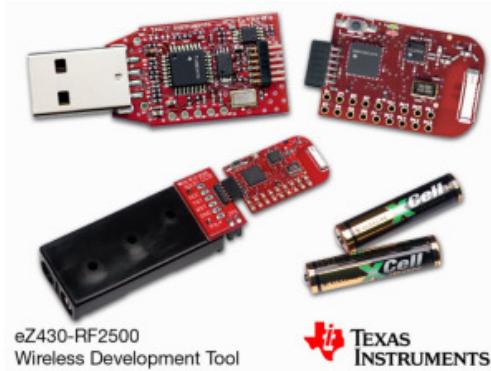


Figure 2.1: Wireless development eZ430-RF2500 Tool [22].

The specifics of this particular model are discussed in relevant sections below. This micro controller unit is a typical example of what is used in a WSN with battery-operated devices, as battery lifetime is an important issue.

2.1 eZ430-RF2500 Sensor Monitor Demo

Texas Instruments provides demonstrations on their pages for the eZ430 Kit Sensor monitor demo of sample applications for these devices. We can choose

2. MSP430 family

an integrated development environment (IDE) of our choosing to work in, however, Texas Instruments provides whole projects for IAR [4] and CCS [2]. Since the IAR studio is not free, we decided to use CCS.

We download the zip file from Texas Instruments web page and open it in CCS studio as the project. We can choose from two implementations for MSP430 one called “Access Point” and other called “End Device”. Every time we make change to the code we just can compile it and see the compiled binary in eZ430-RF2500_WSM/End Device/eZ430-RF2500_WSM.out depending if it is the Access point code or End device one. For our purposes we had to edit both source codes. We describe these changes in the protocol implementation, in chapter 6.

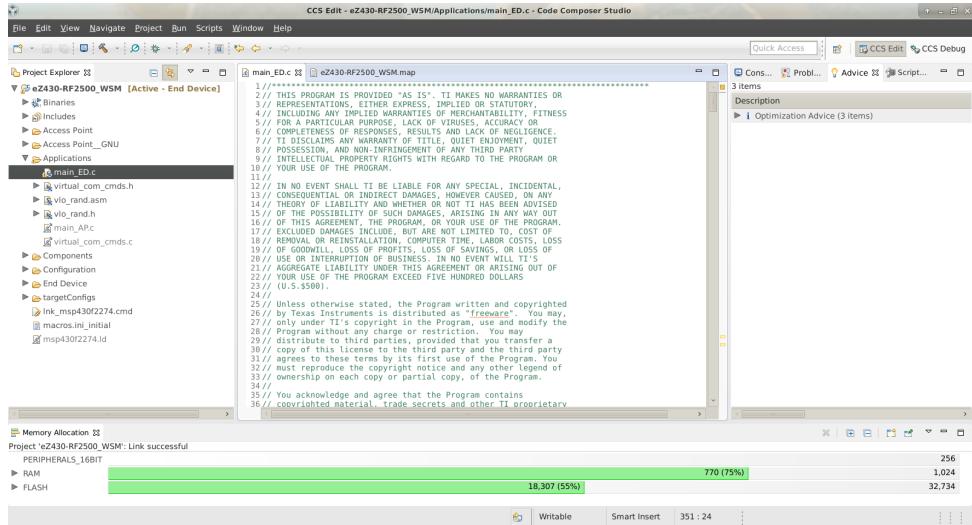


Figure 2.2: Sensor Monitor demo [21] opened in CCS studio [2].

2.2 Memory configuration

We can see the memory configuration of our device in the map file eZ430-RF2500_WSM.map during compilation the CCS studio which we are using also show us the location map.

As we can see from the memory configuration there are parts of the FLASH which are not used for example INFOC and INFOD. We can optimize our key signing algorithms by storing the private and public keys in INFOD and the expanded secret key in INFOC.

Further explanations of these optimizations follow in later chapters. To be able to actually write to the flash, we have to acquire some of the knowledge how to open the flash memory for writing. After studying the Texas Instruments provided demo mentioned earlier in this chapter, we found when the End device code is creating the random address to save it into FLASH

name	origin	length	used	unused	attr
SFR	00000000	00000010	00000000	00000010	RWIX
PERIPHERALS 8BIT	00000010	000000f0	00000000	000000f0	RWIX
PERIPHERALS 16BIT	00000100	00000100	00000000	00000100	RWIX
RAM	00000200	00000400	00000302	000000fe	RWIX
INFOD	00001000	00000040	00000000	00000040	RWIX
INFOC	00001040	00000040	00000000	00000040	RWIX
INFOB	00001080	00000040	00000000	00000040	RWIX
INFOA	000010c0	00000040	00000000	00000040	RWIX
FLASH	00008000	00007fde	00004783	0000385b	RWIX
INT00	0000ffe0	00000002	00000002	00000000	RWIX
INT01	0000ffe2	00000002	00000002	00000000	RWIX
INT02	0000ffe4	00000002	00000002	00000000	RWIX
INT03	0000ffe6	00000002	00000002	00000000	RWIX
INT04	0000ffe8	00000002	00000002	00000000	RWIX
INT05	0000ffea	00000002	00000002	00000000	RWIX
INT06	0000ffec	00000002	00000002	00000000	RWIX
INT07	0000ffee	00000002	00000002	00000000	RWIX
INT08	0000fff0	00000002	00000002	00000000	RWIX
INT09	0000fff2	00000002	00000002	00000000	RWIX
INT10	0000fff4	00000002	00000002	00000000	RWIX
INT11	0000fff6	00000002	00000002	00000000	RWIX
INT12	0000fff8	00000002	00000002	00000000	RWIX
INT13	0000ffa	00000002	00000002	00000000	RWIX
INT14	0000ffc	00000002	00000002	00000000	RWIX
RESET	0000ffe	00000002	00000002	00000000	RWIX

Table 2.1: Memory configuration of MSP430.

memory they are using specific commands to open the FLASH memory for writing.

The empty flash memory is always full of 1s after unlock we are able to flip the individual bits down to the 0s but only once. Erasing the flash memory can be done only by segments of size 64 bytes. Later in this thesis, we describe erasing the segments by use of a small shell script while flashing code to the devices.

```
void createRandomAddress()
{
    unsigned int rand, rand2;
    do
    {
        // first byte can not be 0x00 or 0xFF
        rand = TI_getRandomIntegerFromVLO();
    }
    while( (rand & 0xFF00)==0xFF00 || (rand & 0xFF00)==0x0000 );
    rand2 = TI_getRandomIntegerFromVLO();

    BCSCTL1 = CALBC1_1MHZ;           // Set DCO to 1MHz
```

2. MSP430 family

```
DCOCTL = CALDCO_1MHZ;
// MCLK/3 for Flash Timing Generator
FCTL2 = FWKEY + FSSEL0 + FN1;
FCTL3 = FWKEY + LOCKA;           // Clear LOCK & LOCKA bits
FCTL1 = FWKEY + WRT;           // Set WRT bit for write operation

Flash_Addr[0]=(rand>>8) & 0xFF;
Flash_Addr[1]=rand & 0xFF;
Flash_Addr[2]=(rand2>>8) & 0xFF;
Flash_Addr[3]=rand2 & 0xFF;

FCTL1 = FWKEY;                  // Clear WRT bit
FCTL3 = FWKEY + LOCKA + LOCK;   // Set LOCK & LOCKA bit
}
```

From this code, we are able to figure out how we should operate with the FLASH memory and how to open it. We decided to store private public and extended private key to the FLASH memory. As we learned from the Texas instrument's code before we write to the flash we have to set WRT bit for write operation as following:

```
/* Unlock flash memory for write */
FCTL3 = FWKEY; // Clear LOCK bit
FCTL1 = FWKEY + WRT; // Set WRT bit for write operation
```

After we are done writing to the memory we have to lock it:

```
/* Lock flash memory after write */
FCTL1 = FWKEY; // Clear WRT bit
FCTL3 = FWKEY + LOCK; // Set LOCK
```

This is very important finding for our optimizations later in this thesis which will allow us to reach our goal. From this same code for creating the random address for the End devices we later utilize this in creating a random number generator for generating the private keys 5.3.

Chapter 3

SimpliciTI

SimpliciTI is a low-power, radio frequency (RF) protocol aiming for simple and small networks such as Wireless Sensor Networks in smart buildings. It was mainly developed for building networks with Texas Instruments chips such as MSP430 where the end devices are powered by battery.

This protocol was designed to minimize power consumption as much as possible to extend the life time of batteries. In our application, the chip supports 2.4GHz. The network device types are limited and one physical device can be more than one logical device. Types of the logical devices:

- **Access point**, functions are network address management and storing or forwarding messages on behalf of the sleeping devices. Only 1 AP for network is permitted and have to be always on.
- **Range extender**, re-transmits every frame thus extending the range of the other node. Range extender has to be always on for not missing any frames.
- **End device**, transmits, receives the data or both. Depending on configuration can be always on or asleep.

This protocol is very widely used in Home automation for various applications such as garage door openers, light sensors, glass breakage detectors and smoke detectors.

3.1 Architecture overview

Communication through the simpliciTI protocol is provided by a set of API symbols for initialization, configuration, and reading or writing messages through air. The SimpliciTI protocol supports 3 layers of the OSI model:

- **Application layer**, needs to be developed by the user. The simpliciTI API is used for such an application to send or receive messages from device to device.
- **Network layer**, handles message exchange between nodes based on their 4 byte addresses.

- **Data link (PHY) layer**, no formal PHY layer as in OSI Reference model. The framed Data are received from the radio which performs functions of formal PHY layer.

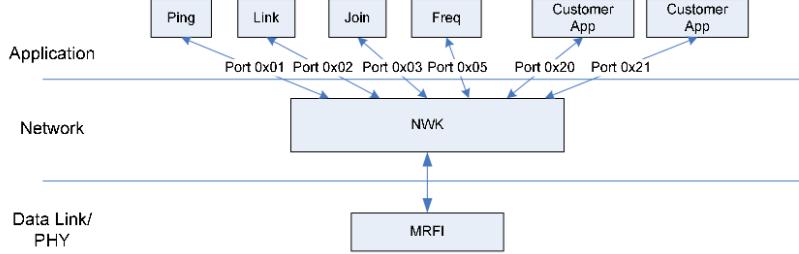


Figure 3.1: SimpliciTI Architecture[17].

The SimpliciTI protocol supports different types of radios, but the basic interface for the network layer is same for all of them. The Minimal Radio frequency Interface (MRFI) shields the programmer from implementation specific details which might be slightly different in various protocol configurations on different devices. Explicit routing is not supported.

The receiver devices receives data directly from the source or through Range extender. Sleeping receivers can poll data from the AP or through Range extender.

3.2 Topology

The example of topologies which SimpliciTI provides can be seen below in the figures.

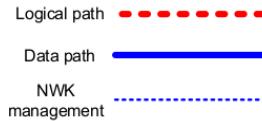


Figure 3.2: Legend[17].



Figure 3.3: Direct Peer-to-peer[17].

On the figure 3.3 we see Direct Peer to peer connection where the End Device 1 (ED1) is sending data to the End Device 2 (ED2). This topology can be useful in more complex networks 3.4.

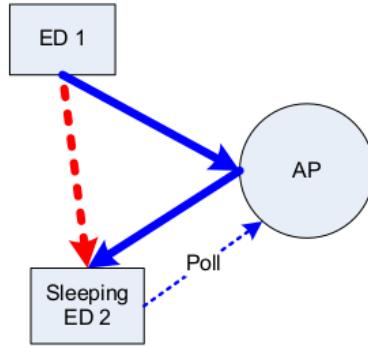


Figure 3.4: Store-and-forward and peer-to-peer through Access Point[17].

The next figure is an example of a more complex network where we use the Peer to peer topology together with the Access Point. The data is sent from ED1 to AP and then AP stores and manages the data on behalf of ED2 when it's asleep^{3.4}.

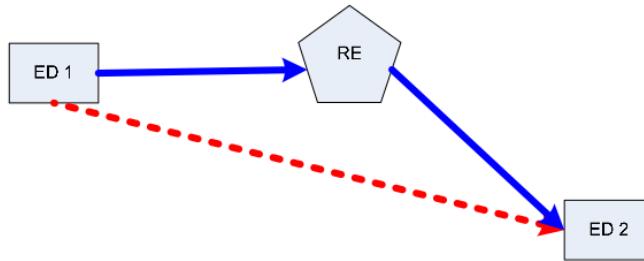


Figure 3.5: Direct peer-to-peer through Range Extender[17].

We can also combine the Peer to peer with Range extender meaning if the devices are not in range of each other we use extender which provides some kind of middle hop between them and they can communicate through it 3.5.

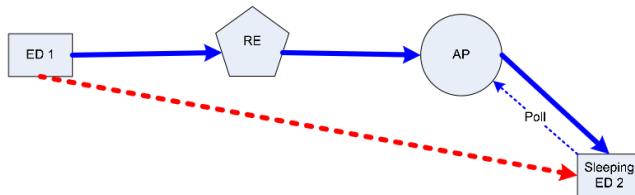


Figure 3.6: Store-and-forward peer-to-peer through Range Extender and Access Point[17].

Eventually we combine all discussed topologies to one as it can be seen on the last figure 3.6. We are using there range extender to ensure communication with the ED1 storing the data from the ED2 and polling to AP. The AP manages the data from the sleeping devices.

3.3 SimpliciTI configuration

The configuration of SimpliciTI can be customized by the end user in the **smpl_nwk_config.dat**. The build time options for custom configuration can be seen below:

- Hops are jumps between the network devices.

```
# Max hop count to AP
--define=MAX_HOPS=3

# Max hops away from and AP. Keeps hop count and therefore
# replay
# storms down for sending to and from polling End Devices.
# Also used
# when joining since the EDs can not be more than 1 hop away
#
--define=MAX_HOPS_FROM_AP=1
```

- The payload are transmitted data. We have to change APP payload to from 10 to 17 more in 4

```
# Maximum size of Network application payload. Do not change
# unless
# protocol changes are reflected in different maximum
# network
# application payload size.
--define=MAX_NWK_PAYLOAD=9

# Maximum size of application payload
--define=MAX_APP_PAYLOAD=17
```

- Set the default link and join token.

```
# Default Link token
--define=DEFAULT_LINK_TOKEN=0x01020304
# Default Join token
--define=DEFAULT_JOIN_TOKEN=0x05060708
```

- The ACK acknowledge of received message from the receiver sent to the origin device.

```
# Remove '#' to enable Frequency Agility (frequency hopping
# when acks not received)
# Requires APP_AUTO_ACK
#--define=FREQUENCY_AGILITY
```

```
# Remove '#' to enable application auto-acknowledge support.
    Requires extended API as well
# Note: while enabling this option will increase robust
    operation, it also
# increase current consumption.
#define=APP_AUTO_ACK

# Enable Extended API
#define=EXTENDED_API
```

- Security discussed later in this chapter in 3.5.

```
# Remove '#' to enable security.
#define=SMPL_SECURE
```

- Enables GET and SET support for NV objects.

```
# Remove '#' to enable NV object support
#define=NVOBJECT_SUPPORT
```

- Enabling the SW timer.

```
# Insert '#' to disable software timer
#define=SW_TIMER
```

3.4 Frame

The SimpliciTI frame consists of 3 logical parts from which one of each one serves to its layer.

PREAMBLE	SYNC	LENGTH	MISC	DSTADDR	SRCADDR	PORT	DEVICE INFO	TRACITD	App payload	FSC
RD*	RD*	1	RD*	4	4	1	1	1	n	RD*

Table 3.1: SimpliciTI frame structure(disabled security)

RD*: Radio-dependent populated by MRFI or handled by the radio itself.

PREAMBLE and *SYNC* are for radio synchronization inserted by the hardware radio. *LENGTH* is the length of the remaining packet in bytes inserted by firmware on the transmitter and it can be partially filtered on receiver. *MISC* depends on the type of radio and can be missing. Destination address (*DSTADDR*) and Source address (*SRCADDR*) are byte arrays handled by MRFI for the address mapping based on configuration of network and radio. *PORT* is holding the port number in bits from 0-5. The bit 6 is

Bit	7	6	5-4	3	2-0
Description	Acknowledgement request	00: Controlled listen	00: End Device	Acknowledgement reply	Hop count
	01: Sleeps/polls	01: Range Extender			
		10: Access Point			
		11: Reserved			

Table 3.2: Device info bit values[17].

enabled when the encryption is requested. The bit 7 is telling us if the frame was forwarded by the AP or not.

DEVICE INFO bit values can be seen in table 3.2. Transaction ID (*TRACTID*) is used by NWK layer for example for matching the the replies of messages or to help detect a duplicate frames. *APP PAYLOAD* reserved for the application data we want to send. Finally Frame check sequence (*FSC*) depending on radio is some kind of CRC appended for the checking the sequence.

3.5 eXtended Tiny Encryption Algorithm (XTEA)

The SimpliciTI protocol is using the modified counter mode (CTR mode). The eXtended Tiny Encryption Algorithm (XTEA) is used for encryption with a fixed number of rounds 32. XTEA is eXtended version of older Tiny Encryption Algorithm which is trying to correct the weaknesses such as vulnerability to “related-key attack” requiring 2^{32} plain texts and time complexity 2^{32} . It is a block cipher designed by David Wheeler and Roger Needham from Cambridge Computer Laboratory is 64-bit Feistel cipher using 128-bit key.

The 64-bit block containing 32-bit Initialization vector and 32-bit counter which is changed for each encryption. The change guarantees unique block for encryption. On the figure 3.7 see the schematic of the XTEA encryption in SimpliciTI. Between the 64-bits cipher block and the next 64-bits of plain text it uses exclusive bitwise logical XOR. When the remaining plain text is smaller than 64-bits it discards the extra cipher block. Otherwise, for the remaining plain text it creates next cipher block by counter incrementation. Each time application sends the frame the payload is encrypted before sending and on the receiver the payload is decrypted before it is available to the application.

The security of encryption is maintained by the 3 components:

- 128-bit key
- 32-bit Initialization Vector
- 32-bit counter

The key and the Initialization vector are fixed during the build time. The starting value of the counter is exchanged during the link session between application peers. These values are independent for the sending and the receiving transmissions. The decryption is done by the same scheme as in figure 3.7 but in the reverse order of the encryption one.

3.5. eXtended Tiny Encryption Algorithm (XTEA)

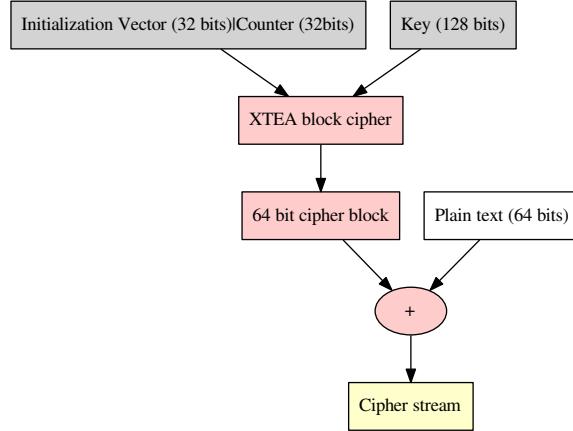


Figure 3.7: SimpliciTI Encryption scheme[16].

The code of the XTEA security can be found in SimpliciTI components part of the Texas Instruments provided demo[21]. The file containing whole security logic is **nwk_security.c** and this encryption can be enabled by build time option **SMPL_SECURE** in the **smpl_nwk_config.dat** as described previously in this chapter. The SimpliciTI security method contains

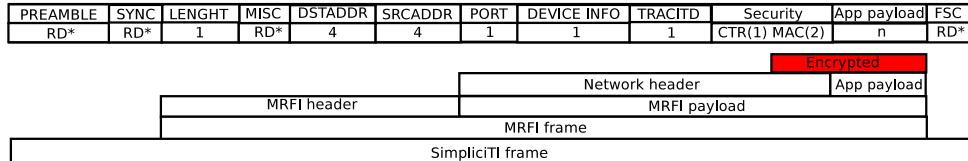


Figure 3.8: SimpliciTI frame with encryption enabled.[17]

minimal Authentication object (MAC). The MAC is a 2 byte object consisting first byte containing fixed value and second byte representing a frame check sequence (FSC). This helps defending against rogue and replayed frames.

Frame format has to be changed for the encryption support as it can be seen on the figure 3.8. Firstly the encryption valid bit is set after frame encryption and secondly another 3 bytes are added to the frame consist of one byte counter hint and 2 byte MAC.

XTEA is for ensuring data confidentiality and should by no means be considered a message authentication scheme. In this project, we opt not to use it as this work is related on message authentication. It should be noted that if the digital signatures developed as a part of this thesis are used in XTEA-encrypted SimpliciTI network, many attacks on protocol messages would be mitigated.

Chapter 4

Protocol design

We need to design a protocol for sending not only the sensor data but also the public key and cryptographic signature from End devices to Access Point. The sensor data from ez430 demo is only 3 bytes large and for demonstration purposes this is all we need to sign.

The original 3-byte message payload can be seen in table 4.1. This is the message we need to sign in order to provide message authentication.

Field:	degC LB	degC UB	volt
Byte:	0	1	2

Table 4.1: Original 3-byte message payload containing the temperature in little-endian two-byte format and single-byte voltage value.

Writing the payload byte-by-byte can be done as follows:

```
// The actual payload starts at msg+1
msg[1] = degC&0xFF;
msg[2] = (degC>>8)&0xFF;
msg[3] = volt;
```

The public key is 32 bytes long and signatures – regardless of the signed data length – are 64 bytes long. That is total 99 bytes of data to be transferred.

However SimpliciTI does not allow us to send so much data in one packet. The default packet size is 10 bytes and although it is possible to recompile the wireless stack with larger packet size, the maximum recommended packet size tops at 50 bytes.

Therefore we opt to use multiple SimpliciTI packets for sending all the data we need to transfer.

Field:	Counter	Data
Size [B]:	1	16

Table 4.2: New protocol packed size.

As the nature of the data is easily aligned on 16 byte size, we configure the

wireless stack to use 17 byte long packets with first byte representing packet number within the message (a counter) and the rest 16 bytes are the actual payload as seen in table 4.2.

Although we need to sign only 3 bytes of data, we reserve the space for actual sensor data in our protocol to be also 16 bytes is that allows for simple extension in the future and falls in line with the rest of the protocol.

The resulting packet descriptions comprising the whole message can be seen in table 4.3. On packet of sensor data, two packets with the device public key and four packets containing the signature of the sensor data by a private key corresponding to the public key transferred earlier.

Number of message	16 bytes message
0	data + nothing
1	1-16 bytes of public key
2	16-32 bytes of public key
3	1-16 bytes of signature
4	16-32 bytes of signature
5	32-48 bytes of signature
6	48-64 bytes of signature

Table 4.3: Protocol design.

In order to transfer the whole message, the device first sends a packet number 0 with the sensor data. Then it sends packets number 1 and 2 with the public key. We define in advance, that this should happen immediately after the measurement. After sending these three packets, the cryptographic signature should be created. Creating this signature may take a long time and only after it has been calculated, packets 3 to 6 should be sent.

One of the measurements in section 6.4.1 uses this delay to measure how long does the creation of Ed25519 signatures take on the msp430 MCU.

Only after the access point receives all the messages, it should act upon the data received. First it should verify the signature against provided public key and if it is correct, it should look it up in the end devices database and mark it as known or unknown. Should the signature verification fail, the software should display it to the user accordingly.

4.1 Replay attack

A replay attack is a form of network attack where the original valid data from the sender is spoofed and replayed to the receiver. In that case, the receiver gets same messages with valid data signature and the key and does not notice there is something wrong.

In way the attacker can send any time this data to the receiver and make receiver think that it was sent from our original End device. In our case, it can easily happen when for example there is fire in the room and temperature rises, the attacker can easily send this old data to our receiver and pretend

there is nothing wrong. For these cases, we left space in the first message

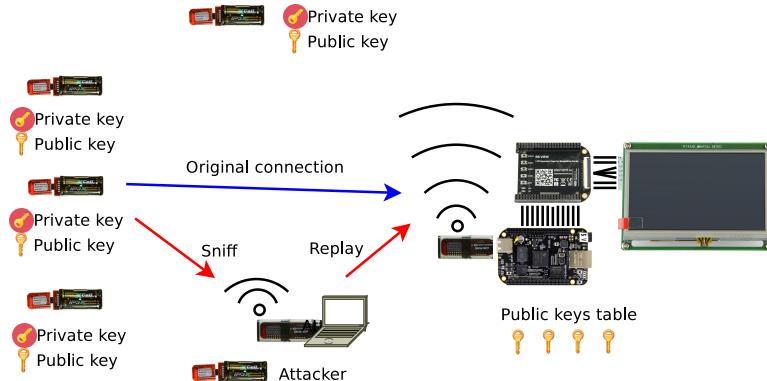


Figure 4.1: Replay attack scheme.

number 0 as it can be seen in 4.3. In the future we can, instead of “nothing”, put some kind of counter changing every transmission and prevent replay attack. Then, if the attacker tries to resend the messages, the receiver will know it is an old message with obsolete counter number and reject the data.

Because it is out of the scope of this thesis, space was left in the initial message for replay attack countermeasures.

4.2 Ed25519

We are using Ed25519 scheme which is using elliptic curve signatures providing us very high speed without compromising security. Public keys in Ed25519 are 256-bit keys of size 32 bytes. For 32-byte key transmission, we just need to send 16 bytes two times for the device identification in the network showed on table 4.4.

Field	Size
Sensor data	3 (+13)
Public key	32
Signature	64

Table 4.4: Splitting the ed25519 key and signature.

The signatures in this implementation are 512-bit which gives us the 64 bytes size of the whole signature. That is twice more larger than the public key but we only need 4 transmissions for the sending the whole signature to the Access point. That is convenient and not very problematic and does not limit us at all.

The signed payload we are transmitting is 3 bytes consisted of the voltage and temperature. There is still 13 bytes left in the first transmission

4. Protocol design

so the payload can be increased by the simple constant in the SimpliciTI configuration file discussed in 3.3.

Chapter 5

Digital signatures

Digital signatures are used for authentication of messages or documents. In our case, we wish to know if message was really send from End Device that we set up, if is whole and there was any problem during the transmission. If the digital signature is valid, the receiver can easily validate whenever the sender is allowed to perform such a transmission and that the message was not corrupted or tampered with[3].

To accomplish this, we use Public key cryptography, where we have pair of keys private and public key. On each End device, we generate the private key (ECC 32-byte key in our case) and send a corresponding public key to the receiver so that it can check future message validity. After we have our pair of keys generated on every device we make table public keys in our receiver of “Trusted devices” where we put all the public keys as we can see in figure 5.1.

Now we can finally send our message with the public key and signature and the receiver can validate if the message was not corrupted and if the device’s public key is in the table of “Trusted devices”. The ECC is not needed on the Access point, data is passed to the application over USB serial for the further processing.

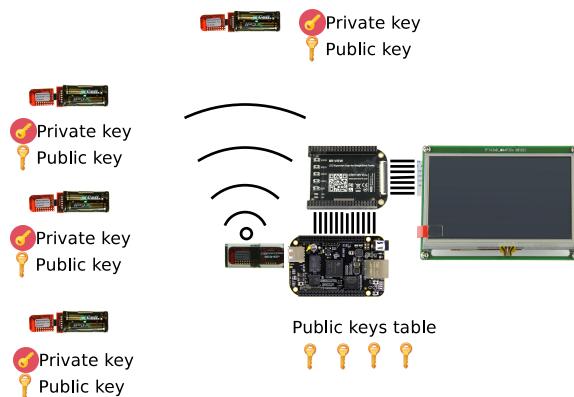


Figure 5.1: Our small Wireless Sensor Network.

5.1 Porting Daniel Beer's implementation c25519

We are using the only known embedded implementation of Ed25519 Daniel Beer's "Curve25519 and Ed25519 for low-memory systems" [12]. As Daniel Beer describes on his page the cost of the key functions on the stack usage with our 1K byte of memory we supposed that it should work. Unfortunately, during the implementation, we discovered that the values from the Table 5.1 are not correct and we have to optimize many key functions to even be able to sign messages.

We discuss these optimizations later in this chapter. We rendered the graph of function calls in figure 5.2 where we can see that path from `ed25519_smult` to `ed25519_double` or `ed25519_smult` to `ed25519_add` which is the longest path taking the most of time.

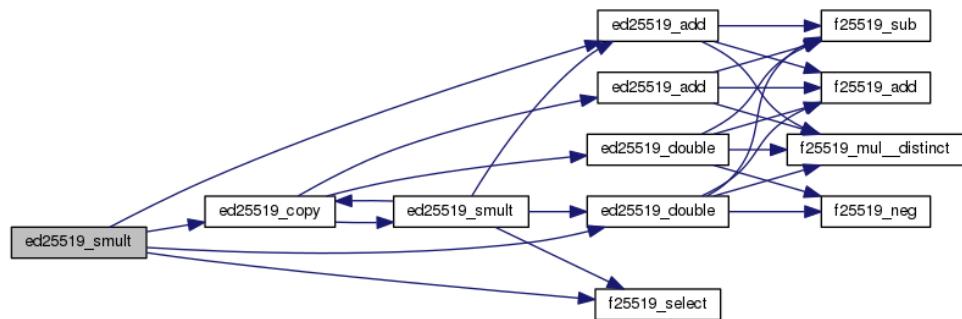


Figure 5.2: The graph of function calls.

However, during the porting of the c25519, we ran into trouble. The generation of public key went fine, however when we tried to generate signatures for our data, we were not able to do so. After investigating the `main_ED.asm` file, we created the table 5.2, where we can see that we actually do not have enough memory to use function `edsign_sign` and create the desired signature. This means that if we want to sign our data, we have to reduce

Func	Cost	Frame	Height
edsign_verify	1078	364	6
edsign_sign	1050	336	6
edsign_sec_to_pub	786	72	6
c25519_smult	462	280	3

Table 5.1: Daniel Beer's stack usage figures for key functions [12].

the stack usage as much as possible, otherwise, the stack gets corrupted and our device is not usable anymore. We describe these optimizations later in this chapter.

5.2 Optimization

We can find an approximate stack usage from main_ED.asm and see which function have memory allocations and how many bytes on the stack. Functions allocating the most memory can be seen in the table 5.2. Our first small optimization was to save expanded secret key the FLASH memory saving 64 bytes in the some of the key functions. Instead of expanding it every time we expanded the secret key just once and then saved it to readable FLASH memory INFOC, as described in 2.2.

This simple optimization saves the 64 bytes in edsige_sec_to_pub which is function where we generate public key from our private key and edsige_sign. We also saved some time during generation of the keys because we do not have to use expand_key function that many times.

Function	Stack usage
edsige_sign	192
ed25519_smult	256
ed25519_double	224
ed25519_add	256
sha12_block	238

Table 5.2: Not optimized stack usage of functions in c25519 [12].

The key optimizations as we can read from table 5.2 all other three functions since they have the biggest usage on the stack. If we look at the **ed25519_smult** function closely below.

```
#define F25519_SIZE 32

/* Projective coordinates */
struct ed25519_pt {
    uint8_t x[F25519_SIZE];
    uint8_t y[F25519_SIZE];
    uint8_t t[F25519_SIZE];
    uint8_t z[F25519_SIZE];
};

void ed25519_smult(struct ed25519_pt *r_out, const struct
                    ed25519_pt *p,
                    const uint8_t *e)
{
    struct ed25519_pt r;
    int i;

    ed25519_copy(&r, &ed25519_neutral);

    for (i = 255; i >= 0; i--) {
        const uint8_t bit = (e[i >> 3] >> (i & 7)) & 1;
        struct ed25519_pt s;

        ed25519_double(&r, &r);
    }
}
```

5. Digital signatures

```

        ed25519_add(&s, &r, p);

        f25519_select(r.x, r.x, s.x, bit);
        f25519_select(r.y, r.y, s.y, bit);
        f25519_select(r.z, r.z, s.z, bit);
        f25519_select(r.t, r.t, s.t, bit);

    }

    ed25519_copy(r_out, &r);
}

```

We can see it allocates enormous number of bytes, $32 \cdot 4$, just to eventually copy the data to the same struct `ed25519_pt`. We can easily just not use the variable `r` at all and use just `r_out`, saving desired bytes on the stack, as seen below.

```
void ed25519_smult(struct ed25519_pt *r_out, const struct
ed25519_pt *p,
                     const uint8_t *e)
{
    int i;

    ed25519_copy(r_out, &ed25519_neutral);

    for (i = 255; i >= 0; i--) {
        const uint8_t bit = (e[i >> 3] >> (i & 7)) & 1;
        struct ed25519_pt s;

        ed25519_double(&s, r_out);
        ed25519_copy(r_out, &s);
        ed25519_add(&s, r_out, p);

        f25519_select(r_out->x, r_out->x, s.x, bit);
        f25519_select(r_out->y, r_out->y, s.y, bit);
        f25519_select(r_out->z, r_out->z, s.z, bit);
        f25519_select(r_out->t, r_out->t, s.t, bit);
    }
}
```

So far, we saved $64 + 32 \cdot 4$ bytes. Unfortunately, it is not enough, and we have to find additional optimizations. The functions **ed25519_double** and **ed25519_add** derive from the optimizations described in “Twisted Edwards Curves Revisited” [13].

The functions `ed25519_double` and `ed25519_add` use, by default, very inefficient allocation of local variables which make no use of the output variables until the calculation is done.

If we slightly reorder the formulas used there, we can exploit the fact that some variables used in the first half of the calculation are not used in the second half. Another useful storage for temporary variables are the output variables which are filled-in at the end of the calculation. We can use them in the first half of the calculation.

Function	Stack usage
edsign_sign	64
ed25519_smult	134
ed25519_double	96
ed25519_add	96
sha12_block2	112

Table 5.3: Optimized stack usage of functions in c25519 [12].

From first glance, we see that it is again possible to use the output value “r” instead of allocating all the points and dynamically #define and #undefine these points during the calculations. From the table 5.3, we can see that by this utilizing optimization, we saved in both functions more than half of the remaining stack usage.

Using this technique, it was possible to reduce the stack usage of the doubling and adding functions by 128 and 160 bytes respectively, down to 96 bytes each. Both functions have 4 32-byte output variables (contained in the “struct edpoint” – x, y, z, t) and both need to allocate only 4 more 32-byte variables to hold the intermediate values.

5.2.1 Verification of optimizations

We have to verify if we did not make a mistake during our optimizations. Meaning the key generation and signature generation works properly otherwise we would not be able to verify signed messages. For this purpose we use part of the Daniel Beer’s tests which are part of his c25519 implementation [12].

We need to adapt the code for our purpose so it generates the public key from the given private key and if the private key gets properly expanded. We choose the **test_c25519.c** and simply just add functions **expand_key** and **edsign_sec_to_pub** and print the results out. Such as that we generate the private key and public key on our End Device and with our testing code. We compare these two results and see if they are equal as they should be. Our device generates following:

```
Privkey:
d05189d1670c9fb4e0b73a3dfd173a0270cd8d3e47527d281db4c011020e3e40

Pubkey:
f21801453e50e5b447cc73b5cebae9f92159cd7e580c0191e888b7caeade84736

Expandedkey: 103433d12171db0d0941478a7f7eb6c27336ec98be366dd6 ..
```

We take this private key and put it to our adapted script from Daniel Beer’s implementation and get following:

```
./tests/c25519.test
Privkey:
d05189d1670c9fb4e0b73a3dfd173a0270cd8d3e47527d281db4c011020e3e40
```

Pubkey :

f21801453e50e5b447cc73b5cebae9f92159cd7e580c0191e888b7cae ad84736

Expandedkey : 103433d12171db0d0941478a7f7eb6c27336ec98be366dd6 ..

We clearly see that they are equal. We can now be sure that our optimizations of the key functions are correctly implemented.

5.3 Key Generation

As we described previous in this chapter, we are utilizing public key cryptography to sign our messages and data. To accomplish this goal, we need to generate private key which should be as random as possible. To generate the private key, we are using “Random Number Generation Using the MSP430”[15], which is already used in our demo from Texas Instruments for Address random generation.

The very low frequency oscillator (VLO) together with digitally controlled oscillator (DCO) are used as the two clock independent systems. To generate stream of random bits, we use the difference between these two clocks as we can see on 5.3.

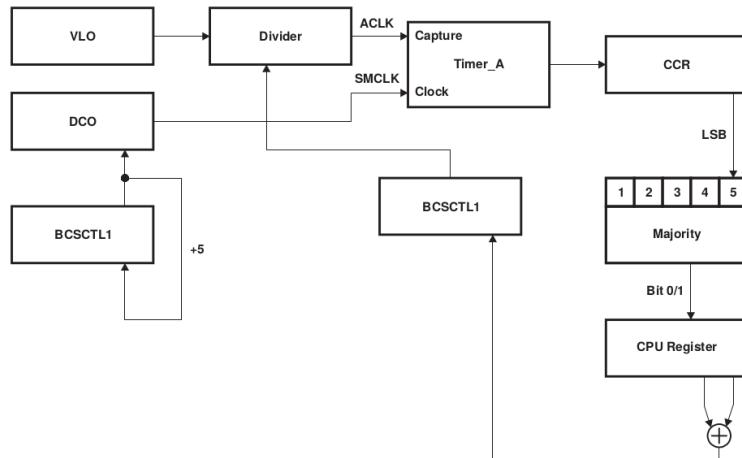


Figure 5.3: MSP430 Random number generation.[15]

The randomness of this method was tested with the tests described in Federal Information Processing Standards (FIPS) with the `fips_tests.c` source code which implements FIPS 140-2 tests for randomness. The results can be viewed with a debugger. These tests are only included for the statistical information this application was not officially tested or certified by FIPS. For the private key creation we add to the Texas Instruments demo [21] function `createPrivateKey()`:

```

unsigned int rnm=TI_getRandomIntegerFromVLO();
srand(rnm);

```

```

int gen=1,i;
for (i=0;i<32;++i) {
    if (private_key[i]!=255)
        gen=0;
}
if (gen) {
    /* Unlock flash memory for write */
    FCTL3 = FWKEY; FCTL1 = FWKEY + WRT;

    /* Generate private key */
    private_key[0]=(rand()%256)&0xF8;
    for (i=1;i<31;++i) {
        private_key[i]=(rand()%256)&0xFF;
    }
    private_key[31]=((rand()%256)&0x7F)|0x40;
}

```

As soon as we have our private key generated, we create a corresponding public key using function “edsign_sec_to_pub” from c25519[12].

```

void edsign_sec_to_pub(uint8_t *pub, const uint8_t *secret){
    /* Expand secret key*/
    expand_key(expanded, secret);
    sm_pack(pub, expanded);}

```

These two keys are always generated during flashing if the address where we store them is empty. If we want to force generation, even if we already have to keys we just remove the corresponding segments of the FLASH memory.

For these purposes, we made simple bash scripts which are using Daniel Beer’s MSPdebug [11]. By running following command:

```
sh flash.sh EDevice.out
```

As we can see in the code listing, we need to erase whole memory and then also flash segments we are using to ensure there are no previous key left. Then we program and load our code and run it.

```

(echo "erase"
echo "erase segment 4160"
echo "erase segment 4096"
echo "prog $1"
echo "load $1"
echo "run"
)| mspdebug rf2500 --fet-force-id MSP430F2274 -q
BIN=$1-'date +%Y%m%d%H%M%S'.bin
(
echo "save_raw 4096 64 $BIN"
)| mspdebug rf2500 --fet-force-id MSP430F2274 -q
privkey='dd if=$BIN bs=1c count=32 skip=32 | \
hexdump -v -e "1/1 \"%02x\""'
publickey='dd if=$BIN bs=1c count=32 skip=0 | \
hexdump -v -e "1/1 \"%02x\""'

```

5. Digital signatures

```
echo "Private key: $privkey"
echo "Public key: $publickey"
```

We are using function **save_raw** to save the keys which we generated in the file so we can print them afterwards. For our purposes, we had to also save the expanded secret key to the FLASH because of not enough stack space which we describe in the following chapter. During the flashing, we wait until both led diodes are shining to know that we are done.

```
(mspdebug) save_raw 4096 64 Etest34.out-20170507143550.bin
Done, 64 bytes total
(mspdebug)
Private key:
8888a8e6b3fddecd851fc87d1ea884863271dcfcc978811dbdb2d608f196906b

Public key:
dee82e9354cf01b6761bcfc4ddf4b014745d6c56e67a2ee849b01d0fd877d2f7
```

Here, we can see the example what our script prints when we are done. When we have the keys we can finally create signature and send our messages. Messages in our case are comprised of temperature and voltage data sent in 3 byte message.

Now, we have our pair of keys which are needed for Public key infrastructure and also message we want to sign. We will just use the **edsign_sign** function, which will create 64 bytes signature for us.

```
edsign_sign(sig,msg+1,3);
```

The “sig” is the created signature the “msg” is our message and 3 is the length of the message.

5.4 Verification

We also wrote a simple script in Ruby[5] using red25519 [6] to verify if the public keys and signatures that we are generating are correct. That means if there was a problem – such as not enough memory or any other coding problem we could make – it would be spotted.

This small script has one parameter “-p” where we put private key and our script generates public key using red25519.

```
ruby edtest.rb -p
d05189d1670c9fb4e0b73a3dfd173a0270cd8d3e47527d281db4c011020e3e40

SigningKey:
8888a8e6b3fddedc851fc7d1ea884863271dcfcc978811dbdb2d608f196906b
VerifyKey:
dee82e9354cf01b6761bcfc4ddf4b014745d6c56e67a2ee849b01d0fd877d2f7
```

We see that the public key we generated in previous section is identical with the one which was generated by our small ruby script. We can now know for sure that the public keys that we have generated on our devices are correct.

Chapter 6

Protocol implementation

As we described in Protocol design, chapter 4 we have to split messages into more packets as SimpliciTI protocol requires this of us.

6.1 Communication between Access Point and End Device

To be able to print received messages to serial port, we have to slightly modify the Access point source code from eZ430 Sensor monitor demo [21]. The desired function we want to edit is located in Code/Applications/virtual_com_cmds.c and called *transmitData()*. This function, as we can see below, takes address, rssi and msg delivered from End devices.

```
void transmitData(int addr, signed char rssi, char msg[MESSAGE_LENGTH])
```

Then, it simply calls the function *trasmitDataString()*, which prints the formatted data to the serial port. However, for our purpose, it is not enough and we have to modify the *trasmitData()* to be able to print keys and signatures to the serial port.

```
char txtmsg[] = {"\r\n00 000 #\n00000000000000000000000000000000"};\ntxtmsg[2] = '0'+((addr+1)/10)%10;\ntxtmsg[3] = '0'+((addr+1)%10);\ntxtmsg[5] = '0'+(rssi_int%10);\ntxtmsg[6] = '0'+((rssi_int/10)%10);\ntxtmsg[7] = '0'+((rssi_int/100)%10);\ntxtmsg[9] = '0'+msg[0];\nint i;\nfor (i=0;i<16;i++) {\n    txtmsg[11+i*2]=hexnum((msg[i+1]>>4)&0xf);\n    txtmsg[11+i*2+1]=hexnum(msg[i+1]&0xf);\n}\nTXString(txtmsg, sizeof(txtmsg));
```

We defined new message output which will be printed to the serial port. First two numbers which can be seen are always address of the End device responsible for the data. The next 3 packets are the Received Signal Strength Indicator which we get from the Access point. Our ninth packet is the message counter which increases until we send all the data we want. And the last 16 packets can be the public key, message or signature, depending on the message counter.

The reason for this approach, modifying the string in place and transmitting it as whole – is to avoid dependency on including the `printf` function with the code as it would add unnecessary data and code to the resulting program image. The `printf` function is too big to fit into our memory.

6.2 Protocol testing

Changing the payload to 17 in the `smpl_nwk_config.dat`.

```
# Maximum size of application payload
--define=MAX_APP_PAYLOAD=17
```

Message number	16 bytes message
0	0b011c + 9b93c4dbccb6c59297d1809e01
1	1ddfb0493e0468ccdaf65fc58d5ad058
2	38f1cd1ca5c2b81239d1e30165a85b85
3	3f096b2ef28445bda16a588545c17e37
4	5b73e6e122861a59b58211b2766a7e56
5	fd5974770df40f9694d9e1e0eede4ccf
6	9d85a144cc47f050db6cb8bfec7020db

Table 6.1: Protocol implementation example.

We can read from the table 6.1 that message we are signing, in our case temperature and voltage, is “0b011c”. In the messages 1-2 we can find our public key and messages 3-6 contains 64 bytes signature of our “Temperature message”.

6.3 Verification

To find out if our code and signatures works properly, we modified small ruby script we described in previous chapter 5.4. We have to add signature and message verification.

We have to add two more parameters “-m” for the message and “-s” for the signature. For private key and signature:

```
Private key:
1ddfb0493e0468ccdaf65fc58d5ad05838f1cd1ca5c2b81239d1e30165a85b85
Signature: 3f096b2ef28445bd...
```

We use command:

```
ruby verify.rb -p $private_key -m 0b011c -s $signature
```

We can see from our printed output below the public key used for the purposes of verification in the previous chapter, signatures generated and given one and validation. In this case, the generated signature identical with the given one and thus, validation is successful.

```
Pubkey:
1ddfb0493e0468ccdaf65fc58d5ad05838f1cd1ca5c2b81239d1e30165a85b85
Message:"0b011c"
Generated signature: "3f096b2ef28445bd..."
Given signature: "3f096b2ef28445bd..."
Valid:true
```

We can now confirm that our protocol implementation was successful. We are able to sign messages and send them to the receiver.

6.4 Measurement

We implemented and verified that our protocol and key functions work. Now we have to evaluate if it is usable for our application. We know that our devices are low-power MCUs with low memory specially designed for low power consumption – which means they are also much slower than usual PC-class CPUs.

In this section we measure the duration of most computational-intensive operations and perform a simple statistical analysis of the results to ensure the measurements are consistent.

6.4.1 Key generation measurement

As we do not possess the required hardware for better debugging and therefore interaction with the devices in question we devised a simple way of measuring the most complex operations.

In the code for key generation we added a statement to enable the green LED just before the key generation starts and another statement to turn it off when the process successfully finishes. Then, with a stopwatch, we measured the period between these two events manually and repeated the measurement 10×.

It should be noted that most of the time is spent in **ed25519_smult** function.

Number	1	2	3	4	5	6	7	8	9	10
Time[s]	148	148	155	146	149	147	149	148	148	148

Table 6.2: Key generation measurement.

Average running time is:

$$\bar{t} = \frac{1}{10} \sum_{i=1}^{10} t_i \approx 148.6s$$

For this average time the standard deviation is:

$$\sigma = \sqrt{\sum_{i=1}^{10} (t_i - \bar{t})^2} \approx 2.2891046284519194$$

Therefore we can state the running time is:

$$t = 148.6 \pm 2.29 s$$

■ 6.4.2 Signature generation measurement

To generate the signature, we also have to use **ed25519_smult** which is the slowest operation as we discussed earlier in section 5.1 and visualized in figure5.2.

Number	1	2	3	4	5	6	7	8	9	10
Time[s]	157	156	158	156	156	158	158	157	155	155

Table 6.3: Signature generation measurement.

Average running time is:

$$\bar{t} = \frac{1}{10} \sum_{i=1}^{10} t_i \approx 156.6s$$

For this average time the standard deviation is:

$$\sigma = \sqrt{\sum_{i=1}^{10} (t_i - \bar{t})^2} \approx 8.077128202523474$$

Therefore we can state the running time is:

$$t = 156.6 \pm 8.077 s$$

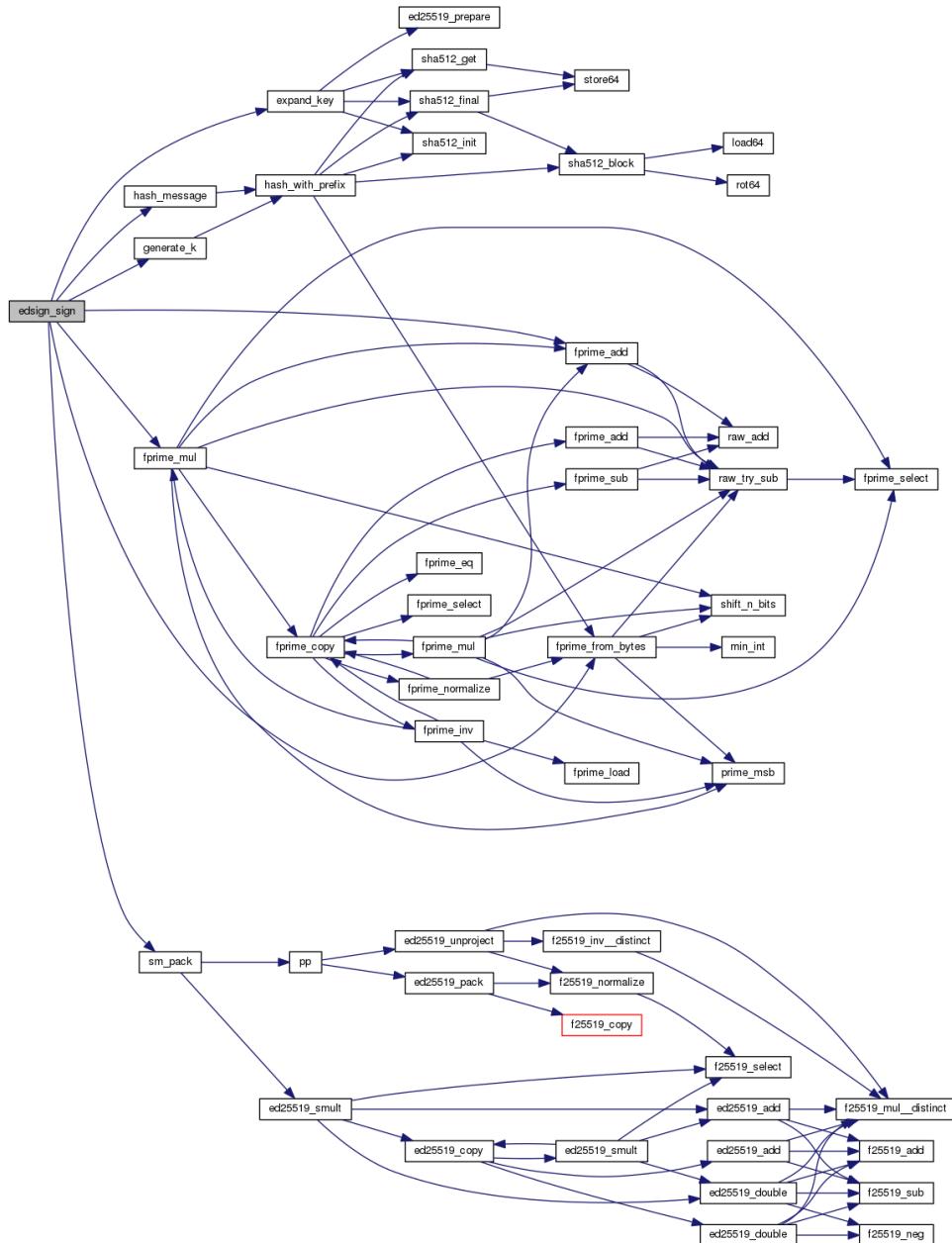


Figure 6.1: Functions called by `edsign_sign`.

In the figure 6.1, we can see what functions are called by `edsign_sign` this scheme is a bit more complicated then the key generation so we can expect it will take a bit longer.

In conclusion, we can say that the most time consuming function is `ed25519_smult`, taking around $t = 148.6 \pm 2.29$ s. This means that the implementation is not usable for the real time applications but more than sufficient for this implementation.

Chapter 7

Thermo GUI

We made a simple application called Thermo GUI for visualizing the data and key verification. This application run on a Beaglebone with the bb-view cape. This Beaglebone is running Debian Linux and the LXDE graphical desktop environment. This Debian image was downloaded from the Beaglebone official web pages [1]. We chose to write the Thermo GUI in Ruby [5], leveraging the availability of a multitude of “gems” – pre-packaged library-bindings for any functionality we might need.

There are readily available gems for GTK+ graphic user interface toolkit, gem for Ed25519 signatures and also a gem for communication over a serial port.

The Thermo GUI application consists of the following modules:

ez430.rb handles communicating via serial port over USB with the ez430 AP device,

octave.rb allows for writing temperature processing functions in MATLAB-like language,

plotview.rb is a simple graph plotting widget for GTK+,

thermo.rb is the main application gluing all the modules together

thermokeys.txt is not a module per-se, but it is important as it contains public keys of all the registered devices.

The application, as a whole, provides a GUI both on a standalone computer (or laptop) as well as in an embedded environment like on Beaglebone platform.

7.1 Attaching devices

When the application starts, it tries to find a USB-connected ez430 device and open its USB serial port. If there is no such device connected, the program waits and polls until the device appears.

In order to setup the serial port and communicate over the device a ruby gem “serialport” must be installed:

```
gem install serialport
```

This allows the application to read data sent from Access Point connected to the Beaglebone platform.

Finding the device is straightforward, as Linux kernel gives us all the required information. The information about all the console-like devices (TTYs), to which the serial ports and USB-emulated serial ports belong, is located in sysfs file system under the /sys directory. We also know the device and vendor id for the ez430 device we are looking for:

```
# Where is information about all TTYs located
SYS_CLASS_TTY="/sys/class/tty"
# Vendor and product USB id
ID_VENDOR=0x0451
ID_PRODUCT=0xf432
```

Based on this information, we can find the ez430 device and its ttyACM device number.

```
Found following devices: ttyACM3 .
```

After finding which serial port the ez430 is connected to, we can start reading messages and process them. For this application, we do not need to process every message immediately as it arrives, but we want to handle them after they've formed complete message from the End device. In our case, we have 7 messages so we create buffer which will collect the messages, until we have all of them. Eventually we can handle it as a whole and update the GUI components based on the results.

7.2 Toolkit

The GUI was created using the GIMP toolkit (GTK+), which allows us create labels, show temperatures, and more. To use it, we need to install the following gem:

```
gem install gtk3
```

After the installation, we use the GTK+ gem by writing `require 'gtk3'` at the beginning of the application. At this point, we can then use the functions we need.

On the left-hand side of the GUI, we can see temperatures for each device. Next to these temperatures, we added the labels which show us the status of the device in the network. To demonstrate the status of the devices, we use the pictures depicted in table 7.1.

?	unknown
!	signature not valid
🔑	public key is not in the “Trusted devices” table
✓	signature is valid and public key is in the “Trusted devices” table

Table 7.1: Device status table.

In figure 7.1 we can see the GUI before it receives any data from the End devices.

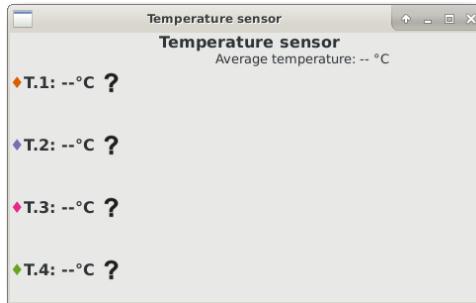


Figure 7.1: Thermo GUI before devices connect.

Now, we are able to recognize the message authentication status of our devices and we can calculate the average temperature in Octave.

```
function [ta] = average(t1,t2,t3,t4)
t=[t1,t2,t3,t4];
ta=mean(t);
end
```

The reason why we are using Octave to calculate average temperature is that, in the future, we plan to use this sensor with more sophisticated Octave script, which will have the capacity to calculate other values. Using octave as the platform for processing the data gives us the opportunity to write the functions in MATLAB-like notation. Then the future, it will be possible to just change the name of the script in the code and calculate and display different values.

For temperature visualization, we implemented a custom graph plotting widget. We are plotting the temperatures of the End devices and the average temperature, as calculated by the Octave script.

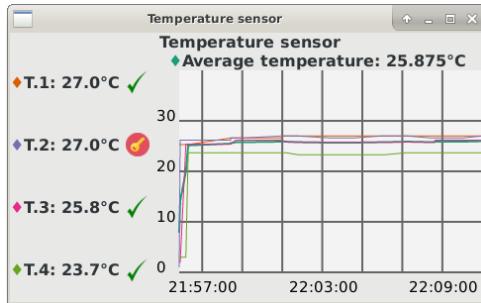


Figure 7.2: Thermo GUI with average temperature.

Each device has its own color showed next to the temperature and is plotted on the graph. In figure 7.2, we can see the calculated average temperature and temperatures of all the devices plotted in the graph.

7.3 Verification

Verification is done using the red25519 library [6] written in ruby. This library can also be installed as ruby gem and just required at the beginning of the code:

```
gem install red25519
```

The signature is verified by the **verify** method of the **pubkey** object which is an instance of Ed25519::VerifyKey class. We initialize the public key object using the public key data we received and then pass the received signature and message data to the **verify** method.

```
# Prints received signature
puts "sign: #{@cryptodevices[unit][:sign]}"
p pubkey # Prints received public key
sig = hex_to_bin(@cryptodevices[unit][:sign])
msg = hex_to_bin(@cryptodevices[unit][:msghex])
# Verify the signature
vr = pubkey.verify(sig, msg)
# Prints true or false depending on verification
puts "signature ok: #{vr}"
```

As for the verification, we print the output to console to see if everything went as we designed. Getting a "true" value when looking at the "Trusted devices" table indicates that we found the correct public key, and the device has not been compromised.

We show the picture in our GUI depicting the device status according to the table 7.1. The whole system with the Beaglebone running the Thermo GUI can be seen in figure 7.3.

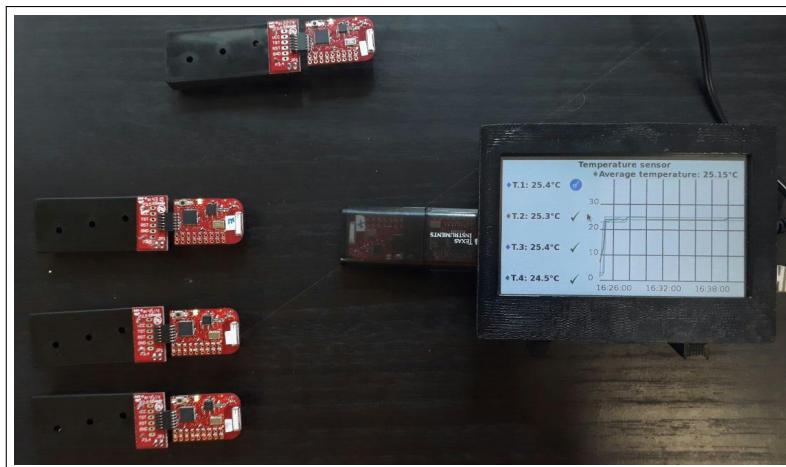


Figure 7.3: Our small Wireless Sensor Network.

Chapter 8

Conclusion

This thesis offered a solution for adding message – and therefore device – authentication to devices with low-power and low-memory MCUs, typically used for sensors in the Internet of Things domain.

Commonly the sensor security in typical IoT applications nowadays is completely missing or there is symmetric block cipher used for message encryption optionally adding shared-key hash-based message authentication code (HMAC). This approach is far from perfect, because this cryptographic scheme can be easily broken by compromising single device with the shared key and it is then easy to impersonate any other device using the same key.

Although some of the constraints imposed by these MCUs on programmers are very limiting, we have shown that it is possible to devise and implement a message signing scheme that is usable for practical applications for these devices. The demonstration application we show typically needs to transfer the sensory data every 10 minutes and therefore – even though the message signing time is more than a two minutes and a half – it can be put to sleep for 75% of its running time, saving batteries lifetime.

The message asymmetric cryptographic scheme we created is built upon elliptic-curve cryptography using the class of elliptic curves called Twisted Edwards curves. This allowed us to keep the cryptographic material relatively small – each number is only 32 bytes, compared to 128 bytes for 1024-bit RSA, this is a huge improvement memory-wise.

Fitting the algorithm into 1024 bytes of RAM was a challenging task which required re-using regions of memory for multiple operations, pre-calculating intermediate forms of the cryptographic keys and re-ordering certain operations in order to avoid stack overflow. It turned out it was possible to shrink the memory requirements more than enough to keep spare RAM for actual sensor applications.

Saving the memory beyond the original goal also allows more “intelligence” of smart buildings – such as preliminary data processing and filtering – to be implemented in the sensor MCUs. This makes it possible to provide not only device authentication (and therefore trust) but also a foundation for innovative applications distributed over the sensor networks.

Our implementation and its optimizations are not limited only to the MSP430 MCU, but it can also be used for any other 16-bit CPU.

8. Conclusion

We have solved the lack of device and message authentication in sensor networks built with common 16-bit MCUs. We have shown the solution works in real-world scenarios using the ez430-RF2500 wireless thermometers sensor network designed for usage in Smart buildings.

Bibliography

- [1] beagleboard.org. <http://beagleboard.org/>.
- [2] Code Composer Studio (CCS) Integrated Development Environment (IDE). <http://www.ti.com/tool/ccstudio>.
- [3] Digital Signature. https://en.wikipedia.org/wiki/Digital_signature.
- [4] IAR IDE. <https://www.iar.com/iar-embedded-workbench/>.
- [5] Ruby. <https://www.ruby-lang.org/en/>.
- [6] Ruby wrappers for the Ed25519 public key signature system. <https://rubygems.org/gems/red25519/versions/1.1.0>.
- [7] Smart building picture. <http://www.datastax.com/wp-content/uploads/2014/06/SmartBuilding.png>.
- [8] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 12 2012.
- [9] Carnegie Mellon University. The "Only" Coke Machine on the Internet. https://www.cs.cmu.edu/~coke/history_long.txt, Retrieved 10 November 2014.
- [10] D. J. Bernstein. A state-of-the-art Diffie-Hellman function. <https://cr.yp.to/ecdh.html>.
- [11] Daniel Beer. MSPDebug. <http://dlbeer.co.nz/mspdebug/>.
- [12] Daniel Beer. Curve25519 and Ed25519 for low-memory systems. <http://www.dlbeer.co.nz/oss/c25519.html>, 25 April 2014.
- [13] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. *Twisted Edwards Curves Revisited*, pages 326–343. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
- [14] Jeffrey Voas. Networks of ‘Things’. <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-183.pdf>.

Bibliography

- [15] Lane Westlund. Random Number Generation Using the MSP430. <http://www.ti.com/lit/an/slaa338/slaa338.pdf>.
- [16] Larry Friedman. Application Note: SimpliciTI Security. <http://www.dianyuan.com/upload/community/2014/06/27/1403852642-91399.pdf>.
- [17] Larry Friedman, Texas Instruments. SimpliciTI: Simple Modular RF Network Specification.
- [18] Zhe Liu, Johann Großschädl, Lin Li, and Quliang Xu. *Energy-Efficient Elliptic Curve Cryptography for MSP430-Based Wireless Sensor Nodes*, pages 94–112. Springer International Publishing, Cham, 2016.
- [19] Liz Upton. Internet of things toilet. <https://www.raspberrypi.org/blog/internet-of-things-toilet/>.
- [20] Texas Instruments. Beagbone with BB view. <https://www.element14.com/community/dtss-images/uploads/devtool/image/large/Expansion+Board+for+BeagleBone+family+with+7+inch+LCD+5511ec1858262.png>.
- [21] Texas Instruments. eZ430-RF2500 Sensor Monitor Demo (Rev. G). <http://www.ti.com/general/docs/lit/getliterature.tsp?baseLiteratureNumber=slac139&fileType=zip>.
- [22] Texas Instruments. MSP430 Wireless Development Tool. <http://www.ti.com/tool/ez430-rf2500>.