University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

ECCforContiki: An ECC-based Security Solution for Contiki-based Sensor Networks

Madeleine von Heyl
Zurich, Switzerland
Student ID: 08-737-801

Supervisor: Corinna Schmitt, Eder John Scheid
Date of Submission: October 9, 2018

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

CSG
Department of Informatics
Communication Systems Group

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Zusammenfassung

Wireless Sensor Networks (WSNs) sind Netzwerke, welche sich aus Constrained Devices zusammensetzen. Diese verfügen definitionsgemäss über eingeschränkte Rechen- und Speicherkapazität. Den Datenverkehr zwischen den einzelnen Netzwerkknoten durch Verschlüsselung abzusichern, ist bei WSNs nicht trivial, da kryptographische Algorithmen normalerweise sehr ressourcenintensiv sind.

In dieser Arbeit wird ECCforContiki präsentiert, ein Sicherheitsprotokoll für WSNs. ECCforContiki basiert auf Elliptic Curve Cryptography bei der Schlüsselerzeugung und beim Schlüsselaustausch. Bei der eigentlichen Verschlüsselung kommen verschiedene Techniken aus symmetrischen Verschlüsselungsverfahren zur Anwendung. Die Sicherheitslösung benötigt keine Zertifizierungsstelle, sondern die öffentlichen Schlüssel werden vor der Inbetriebnahme auf die Netzwerkknoten geladen, wodurch die Authentizität gewährleistet wird. Zudem können je nach Netzwerkgrösse verschiedene Optimierungen erfolgen: Bei kleineren Netzwerken ist es aus Performance-Gesichtspunkten sinnvoll, aufwändige Schlüsselaustauschalgorithmen vor der Inbetriebnahme auszuführen, während dies bei grossen Netzwerken (<1000 Sensoren) zu Speicher-Engpässen führt.

Das Protokoll wurde auf einem WSN-Prototypen mit insgesamt 4 Sensoren implementiert, auf welchem das Aggregationsprotokoll TinyIPFIX auf dem Betriebssystem Contiki läuft. Die Evaluation hat gezeigt, dass die präsentierte Lösung den Datenverkehr zwischen den Netzwerkknoten erfolgreich absichert.

Mögliche Weiterführungen dieser Arbeit könnten das entwickelte Protokoll auf den erwähnten grossen Netzwerken mit Hilfe von Simulator-Software implementieren. Eine andere mögliche Weiterentwicklung betrifft das Speichern der Schlüssel auf den einzelnen Sensoren: Beim entwickelten Prototypen geschieht dies manuell, was bei grossen Netzwerken schlecht möglich ist.

Abstract

Wireless Sensor Networks (WSN) are networks composed of constrained devices. One problem related to WSN is how to secure the communication between the nodes of such a network, as cryptographic algorithms are usually resource intensive.

Therefore, in this thesis, ECCforContiki, a security solution for WSNs is presented. ECCforContiki uses elliptic curve cryptography (ECC) for key generation and key exchange, and various symmetric key techniques for encryption between the network nodes. It does not rely on a certificate authority to provide authenticity, but uses asymmetric pre-shared keys. Depending oh the network size, different optimizations can be made: If the network size is rather small (<1000 network nodes), it is advisable to perform the key exchange before deployment, whereas with large networks (>=1000 nodes), this will lead to memory shortage.

The solution was implemented on a small WSN prototype of 4 sensor nodes running the aggregation protocol TinyIPFIX on the operating system Contiki. As shown by the performed evaluation, the presented solution successfully secures the communication between sensor nodes without a central certificate authority.

Possible future work entails implementation on the aforementioned large networks, possibly with the aid of simulation software. Another further development would be the implementation of a key distribution service, which would make the deployment of a larger network easier.

Acknowledgments

I would like to thank my supervisors Eder John Scheid and Dr. Corinna Schmidt for their detailed feedback and patient explanations. Further, I want to thank Prof. Burkhard Stiller for his support of this thesis. I would also like to thank Michael Petersen for his patient moral support and supply of fresh groceries. Finally, I am very grateful for my lab mate Severin Siffert's continuous tips and help with mysteriously blinking LEDs and misbehaving Makefiles.

Contents

Chapter 1

Introduction

Due to the growth of the Internet and increasing diversity of devices connected to the Internet, the Internet of Things (IoT) has become a relevant theme in research and industry. IoT is not limited to Peer-to-Peer (P2P) networks and devices such as servers, computers, and routers anymore, but also includes wireless sensor devices that form an individual network, called a Wireless Sensor Network (WSN). Those devices present a challenge for developers, because they are limited in memory, energy, and computational capacities. In order to connect them with the Internet, they must support IP communication by using an IPv6 implementation called 6LoWPAN [30]. The possible topology of WSNs can range from star topology to P2P topology, but usually a combination of both topologies is common in WSN deployments. This means that the network consists of Full-Function Devices (FFD) and Reduced-Function Devices (RFD) that both support different functionalities, depending on their location in the WSN. This functionality can range from simple data collection and forwarding to preprocessing. Usually the communication between nodes is performed wirelessly and over UDP. Furthermore, the packet size is limited and due to existing IPv6 implementations, like 6LoWPAN, it is possible to support packet fragmentation and compression in order to connect such limited devices to the IoT. [10]

Many use cases for IoT involve the collection and transmission of sensitive data. Yet, many deployments currently do not protect this data through suitable security schemes [27]. Different end-to-end security schemes were built upon existing Internet standards, specifically the Datagram Transport Layer Security protocol (DTLS), but might not be applicable to WSNs due to the use of constrained devices with limited memory resourcse. By relying on an established standard, existing implementations, engineering techniques, and security infrastructure can be reused that enable easy security uptake from application developers. [14] A promising approach to bring high security in constrained networks is to use Elliptic Curve Cryptography (ECC) [13]. As is evident from figure 1.1, ECC requires shorter keys to achieve the same level of security of solutions with longer key sizes.

In this thesis, ECCforContiki is presented, a standard compliant security solution for resource constrained sensor nodes, that is implemented on the application layer. The solution was developed in conformity with an end-to-end security architecture. Moreover,

Table 1.1: Recommended bit lengths for security levels 80, 128, 192, 265. [23]

Algorithm Family	Cryptosystems	Security Level (bit)			
		80	128	192	256
Integer factorization	RSA	1024 bit	3072 bit	7680 bit	15360 bit
Discrete logarithm	DH, DSA, Elgamal	1024 bit	3072 bit	7680 bit	15360 bit
Elliptic curves	ECDH, ECDSA	160 bit	256 bit	384 bit	512 bit
Symmetric-key	AES, 3DES	80 bit	128 bit	192 bit	256 bit

the solution satisfies the paradigm of end-to-end security and supports two-way authentication.

ECCforContiki's main contribution will be a security solution with pre-shared keys for Contiki, which emphasizes efficiency, compared to a key server solution, as the extra traffic for key establishment via the key server will be unnecessary. Additionally, several efficiency levels are introduced, which can be implemented, depending on the network size and simplicity requirements. The detailed design decisions are discussed in Chapter 3.

The thesis is structured as follows: In Chapter 2, the theoretical bases for the solution architecture are laid out, including an explanation of the used ECC components. In Chapter 3, the solution design is presented, including the security goals and the cryptographic functions provided. In Chapter 4, the implementation is explained. Chapter 5 presents the evaluation of the implementation. Lastly, in Chapter 6, conclusions about the design and the implementation are drawn, together with a pointer to future work.

Chapter 2

Background

In this chapter, the different basics and building blocks for ECCforContiki will be outlined and explained. First, definitions for the types of devices and networks to which ECCforContiki applies are given. Next, Contiki, the operating system and its peculiarities on which ECCforContiki is implemented are mentioned. Subsequently, Elliptic Curve Cryptography and its cryptosystem components that are used for ECCforContiki are explained. Lastly, several approaches that are related to ECCforContiki are analyzed.

2.1 Definitions

Internet of Things (IoT): Garcia-Morchon et al. define the Internet of Things as "the interconnection of highly heterogeneous networked entities and networks that follow a number of different communication patterns such as: Human-to-Human (H2H), Human-to-Thing (H2T), Thing-to-Thing (T2T), or Thing-to-Things (T2Ts)" [7].

Constrained device/constrained node: A broad definition is the following by Borman et al. [1]: "A node where some of the characteristics that are otherwise pretty much taken for granted for Internet nodes at the time of writing are not attainable, often due to cost constraints and/or physical constraints on characteristics such as size, weight, and available power and energy."

A conceptual subset of IoT are **Wireless Sensor Networks (WSN)**, for which ECCforContiki is designed. Often, network nodes of WSNs are constrained devices, because their task in WSNs (e.g. recording sensor data and sending it to a sink) are usually resource-intensive. Providing more computational power or memory than needed would be unnecessarily expensive.

2.2 Contiki

Contiki is an open-source, lightweight operating system written in C. It is especially suitable for constrained devices, as it needs very little memory and energy resources (e.g.

2KB of RAM and 60KB of ROM [4]). Two of its features are loadable programs even for very constrained networks, and protothreads - lightweight, stackless threads, which are implemented on top of the event-driven kernel. Protothreads combine the advantages of event-driven systems (which do not require stacks, but cannot be preempted) with preemptive threads. This is achieved by dynamically loading protothreads as a library only for programs which explicitly require it. This makes an event-driven operating system architecture possible even with constrained resources, which can be greatly exploited for WSNs [5, 4].

2.3 Basic Cryptographic Notions

There are two basic possibilities to secure network traffic: (i) symmetric cryptography (i.e, shared key), and (ii) asymmetric cryptography, also know as Public Key Cryptography (PKC). With symmetric cryptography, two parties A and B have a common key established, which is used for both encryption and decryption [23]. With PKC, each party has two keys, a public key and a private key. To encrypt a message M, party A uses B's public key and its own private key. To decrypt the ciphertext $C = encrypt(M)$, B uses his private key and A's public key. As the public key can be transmitted over an insecure channel, the encryption algorithm should be easy in one direction and hard in the other, so that attackers cannot decrypt the ciphertext, even if they have both the ciphertext and the public key. These kinds of algorithms are called one-way function. Apart from encryption, there are three other important PKC components [23]:

- Key establishment: A function that allows to establish cryptographic keys over an insecure channel.
- Identification/Authentication: A function through which entities can be identified irrefutably via cryptographic signatures.
- Nonrepudiation: With the help a nonrepudiation function, it can be made sure that a message has not been changed after the sender has sent it.

Another important building block of ECCforContiki is the **Cyclic Redundancy Check** (CRC), where a bit sequence is divided by a pre-defined binary polynomial. The remainder is the resulting CRC check [32].

2.4 Number Theory

In the following section,various number theory concepts needed for ECC will be defined. All definitions, unless otherwise noted, were taken from *Understanding Cryptography* by C. Paar and J. Petzl [23].

Group: A set of elements plus an operation ($\circ$). The operation must satisfy the four group axioms:

- Closure: For all elements of the group G, the result of the group operation on any two elements has to be in G as well.
- Associativity: For any a, b and c in G: $(a \circ b) \circ c = a \circ (b \circ c)$
- Identity element: There is one element e in G such that, for every element a in G, $e \circ a = a$ holds.
- Inverse element: For each a in G there is an inverse element a^{-1} in G such that $a \circ a^{-1} = e$.

An **abelian group** is a group where a fifth axiom, commutativity, always holds $(a \circ b = b \circ a)$.

Congruence (class): The modulo operation can be considered a congruence relation. Two integers a and b are congruent modulo n if a and b have the same remainder when divided by n. This relation is written as follows:

$$a \equiv b \pmod{n}$$

The respective congruence class consists of integers that are congruent modulo n. [34]

Field: A group on which the addition and multiplication operations are defined (cf. the group definition).

- For each element, there exists an additive and a multiplicative inverse.
- For both addition and multiplication, there exists an identity element ("0" resp. "1").
- The associativity axiom must be satisfied, similar to above.
- Both addition and multiplication are commutative.
- Finally, the distributivity axiom must be satisfied, i.e. $a \circ (b + c) = (a \circ b) + (a \circ c)$, as known from basic algebraic rules.

A **finite field or Galois field** is a field with a finite number of elements, where the number of elements is called the order of the field. A field of order m exists, if $m = cn$, where n is a positive integer and c is a prime number called the characteristic of the field. A **prime field** is a finite field of order p where p is prime and the operations are done modulo p (0, 1, ... p-1).

The additive and multiplicative group of a field F: As the addition and multiplication are defined, the field forms an abelian group under addition, resp. under multiplication. These groups are called additive group resp. multiplicative group of F. An example of a multiplicative group is the multiplicative group of integers modulo n $(\mathbb{Z}/n\mathbb{Z})^*$, which means, roughly speaking "take two integers from the group and check that they are coprime (i.e. their greatest common divisor is 1). If they are, calculate modulo n on the product of the two and you will see that the result is in the group." $(\mathbb{Z}/n\mathbb{Z})^*$ is the basis of a number of cryptosystems, such as RSA and Diffie-Hellman key

exchange. $(\mathbb{Z}/n\mathbb{Z})^*$ is defined as the integers $0, ..., n-1$ that are coprime to n. Now, if it is really a group, multiplying two members and calculating $\mod n$ on the result will in turn result in a group member (closure). Associativity is given by the multiplication; 1 is obviously the identity element. Now, what is the inverse? - The inverse of each element can be computed using the extended Euclidean algorithm, which to explain would go beyond the scope of this thesis.

The multiplicative group of a finite field $GF(q), q = p^2$ is another important example of a multiplicative subgroup of a field.

The **order n of an element a** of a group G with group operation $\circ$ is the smallest possible integer k such that k times a applied to itself $(a * k)$ equals the identity element.

Cyclic group: A group G which contains an element a with maximum order $= |G|$ is said to be cyclic. a is then called generator or primitive element. Less formally: If there is an element that generates all elements, it is a cyclic group. Example: $a = 2$ for $Z11$. If $|G|$ is prime, then all elements except the identity element are generators.

Discrete logarithm problem (DLP): Given a cyclic group $(\mathbb{Z}/n\mathbb{Z})^*$, its generator a and some element b from $(\mathbb{Z}/n\mathbb{Z})^*$, solve

$$a^x \equiv b \mod n$$

or, more informal: which power of a is congruent to $b \mod n$? When a and n resp. $(\mathbb{Z}/n\mathbb{Z})^*$ are chosen correctly, the DLP is considered intractable, i.e. a computation takes very long, even with huge resources. Figure 1.1 in the previous chapter shows what "correctly" means in this context: To achieve a security level of 80 bit, i.e. for the best known attack to need 280 steps, the recommended key length, i.e. the order (i.e. "size" = n) of $(\mathbb{Z}/n\mathbb{Z})^*$, for the discrete logarithm problem is 1024 bit.

2.5 Elliptic Curves

An elliptic curve EK defined over a field K of characteristic $c \neq 2$ or 3 is the set of solutions $(x, y) \in K^2$ to the equation 2.1 plus a "point at infinity" [13].

$$y^2 = x^3 + ax + b, a, b \in K \tag{2.1}$$

Additionally, the discriminant $\Delta = 4a^3 + 27b^2$ cannot be 0 for the elliptic curve to be non-singular (roughly speaking, this means that the curve cannot have "special" points like intersections, cusps or isolated points). [34]

In cryptography, the most common choice of the field K are prime fields (see section 2.4). An example of an elliptic curve by Paar et al. [23] would then be

$$y^2 = x^3 + 2x + 2 \mod 17$$

How can points on the curve be added, such that the group axioms are satisfied? Recall, the result of the operation must be an element of the group again, so performing a scalar addition is not possible, as the result may lie anywhere and not necessarily in the group resp. on the curve. The group operation is defined as follows: Two points on the curve are added by drawing a line through them. The point where the line intersects the curve again is mirrored on the x-axis, and that point is defined as the sum of the first two points. For an illustration see Figure 2.1. If a point P is added to itself, i.e. $2P$ is to be computed, the tangent is used instead to intersect the curve. The inverse of a point P is defined as the point mirrored on the x-axis ($-P$). Now, to make use of elliptic curves for the DLP, a cyclic group is needed. Under certain conditions, all points on an elliptic curve defined over a finite field form a cyclic group, which means that all points are generators [23]. A point P on the elliptic curve added to itself results in the next element in the group (at $2P$). If P is again added to the second element, $3P$ will be the next result and so forth. If all the points on the curve form a cyclic group, eventually P will be the result.

2.5.1 The Standard NIST curves

The National Institute of Standards and Technology (NIST) published a list of elliptic curves with pre-defined parameters, which are especially suitable to quickly perform computations on. All NIST curves are so-called *Koblitz curves*, on which the *Frobenius expansion* has been performed, such that computations them are faster because the field that the curve is defined on are now smaller. Recall that the order m of a field is equal to c* n, where c is the characteristic of the field. If the c= 3, then n can be represented in 64 bits, which is an advantageous size for cryptographic computations and makes them faster. [16]

2.5.2 The Elliptic Curve Discrete Logarithm Problem (ECDLP)

The DLP for elliptic curves will be “Given two points P and Q, how many hops does P have to make to get to Q?”, or more formal: given an elliptic curve E, defined over a finite field F with order m, and two points P and Q, find an integer x such that Q = xP. This does not seem like a particularly difficult problem. Considering that the key size (i.e. the secret factor x) should be at least 160 bits (see Figure 1.1), it is easy to see that computing all possibilities of x can take a long time.

2.6 Elliptic Curve Cryptosystems

In the following section, the three cryptographic functions with ECC that will be used for implemented solution will be described, namely the Elliptic-curve Diffie-Hellman key exchange (ECDH), the Elliptic Curve Integrated Encryption Scheme (ECIES) and the Elliptic Curve Digital Signature Algorithm (ECDSA).

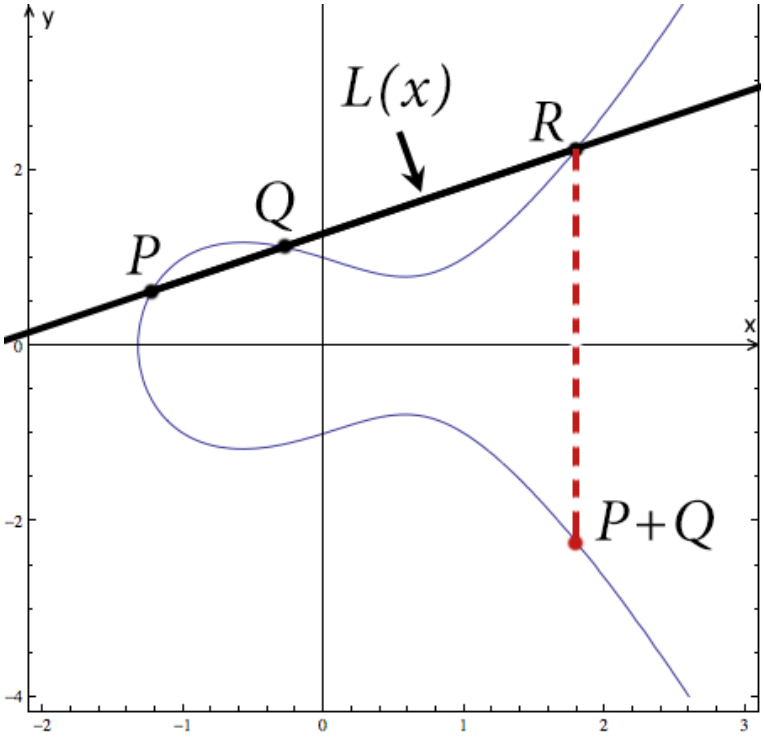

Figure 2.1: Illustration of the point addition on elliptic curves [15]

2.6.1 Key exchange

The ECDH key exchange is similar to the regular Diffie-Hellman key exchange, in that each party chooses private and a public key, fetches the public key from the other party and computes the shared key with the other party's public key and their own private key. The main difference to the regular DH protocol is that the basis of ECDH are points of an elliptic curve. The input of the protocol, also called the domain parameters, are the following items:

- An elliptic curve E, defined by its parameters a and b over a field F
- The field F. If F is a prime field, it is defined by the prime number p
- The generator Q of the additive group G of F

The protocol has the following steps [23]:

1. The three domain parameters E, p, Q a openly agreed to by both parties. They do not pose a vulnerability.
2. Each party chooses a private key x, such that $1 < x < n$, where n is the number of points on the curve E. Naturally, x cannot be made public under any circumstances.
3. Both parties compute their public key, $R = xQ$, i.e. they multiply the published generator Q with their private key x, considering the multiplication rules shown above. Each public key consists of E, Q and R.

4. Finally, each party computes the shared key by calculating x * R, where x is their own private key and R is the other party's public key.

2.6.2 Encoding and Decoding Points on a Curve

To perform a key exchange, an option to store and transmit a representation of a point on an elliptic curve is required. To represent a point in a straightforward and unambiguous manner, it has to be mapped to an integer or other simple data structure. The question arises how to obtain this mapping function.

There are several ways to represent a (two-dimensional) point on an elliptic curve (e.g. as an integer or byte array), of which two will be mentioned, both of which are described in Jivsov, 2014 [9].

A rather intuitive mapping function works as follows: If the curve and its parameters, as well as the x coordinate are known, the y coordinate can almost be inferred by solving the elliptic curve equation, as there are two points per x value on a curve. Hence it suffices to represent the y coordinate in a very compressed way, e.g. a sign bit. As Jivsov explains, this approach has several drawbacks, such as wasting an entire byte for the one bit that is needed, and the fact that the resulting representation is not an actual integer, but a pre-defined sequence of sequence of bytes. Luckily, for some protocols, such as ECDH, the y value is not even needed, the key exchange works with only the x values, hence the x value of a point is an unambiguous representation of it.

An alternative way to uniformly map points on an elliptic curve to integers would look as follows: The key generation algorithm is modified in the following manner:

1. Generate a key pair $(k, Q = k * G)$, where k is the private key and Q the public key.
2. if $y! = min(y, p - y)$, where p is the order of the finite field of the elliptic curve) go to step 1
3. return the key pair (k, Q)

As half the points on a curve have a negative y value, the expected number of iterations is 2.

To encode (= map) a point as an integer, the x value is returned. To decode a point represenentation, compute the following values are computed:

1. $y' = sqrt(C(x))$, where $C(x)$ is the Weierstrass equation of the curve, e.g. $C(x) = y^2 + a * x + b$
2. $y = min(y', p - y')$, where p is the order of the finite field of the elliptic curve
3. $Q = (x, y)$

Evidently, this method is "backwards compatible" with simply using the x value of a point, which suffices for ECDH.

2.7 Related Work

In the following section, several related approaches to ECCforContiki are reviewed and contrasted with this thesis.

Many of the better-known security approaches for constrained networks use symmetric-key cryptography, of which a few important ones will be cited.

Karlof, Sastry and Wagner designed TinySec [11], "the first fully-implemented link layer security architecture". It secures communication between nodes with symmetric block ciphers and Message Authentication Codes (MAC) to ensure confidentiality and integrity. It allows an authentication-only mode, where encryption is not deemed necessary, in case speed and energy efficiency is prioritized over confidentiality.

Eschenauer and Gligor [6] also opt for symmetric key cryptography, as they consider public key cryptography to be impractical for constrained networks. Their contribution is a key pre-distribution scheme, which randomly distributes a set of keys per sensor node. After the key distribution, the nodes broadcast their set of nodes which are reachable during the path discovery phase. At a certain size of the set (e.g. 250 for a sensor network of 10'000 nodes), paths between any two sensor nodes can be established during it. Key ring (= key set) revocation mechanisms ensure resilience against node capture attacks.

Zhu, Setia and Jajodia [35] designed LEAP+ (Localized encryption and Authentication Protocol), which implements security mechanisms for various ranges of communication. On the smallest level, between two neighboring nodes, pairwise symmetric keys are used. Within a cluster, which is smaller than the entire network, the nodes share a cluster key, which is used for local broadcasts. The global key is used for global broadcasts. As the entire network can be compromised if an attacker can find out the global key, there is a protocol in place for replacing the global key securely. Finally, for communication between a node and the base station, each node shares an individual key with the base station. The individual key, the pairwise shared key with the neighbors and the cluster key can be computed incrementally by only loading each node with an id, the same initial key and various key generation algorithms: First a master key for each node is generated with the initial key (which will be deleted after a while as a defense against node cloning attacks) and the id, then two neighbors compute a pairwise shared key, then a cluster key is agreed by a number of neighbors. The global key is preloaded and there is a rekeying algorithm in place, namely the one of μTesla, which will be discussed next.

SPINS (Security Protocols for Sensor Networks) [24] by Perrig et al, is the last of the symmetric cryptography protocols to be examined. It is designed to secure base station-to-sensors-broadcasts, base station-to-sensors individual messages and node-to-base stations communication. It consists of two building blocks, SNEP and μTesla, the latter of whose purpose is securing broadcasts. SNEP uses a block cipher in counter mode and message authentication codes (MAC) to ensure confidentiality, integrity, replay protection and semantic security. μTesla also uses symmetric cryptography together with a hash chain to protect the broadcasts against replay attacks: A block cipher in cipher block chaining is repeatedly calculated on an initial number. Each result is used as a verification key, such that the verification key for the MAC of each broadcast message is sent with the

next message. The nodes store the message and with the next one, they can verify the previous MAC.

Each of these papers deem asymmetric cryptography unusable for constrained networks. In contrast, Ma et al. propose a "Certificateless Searchable Public Key Encryption Scheme for Industrial Internet of Things". For their approach, a user's identifying information serves as the public key, and the private key is in part generated by a Key Generation Center (KGC) and in part chosen by the user, which mitigates problems with a malicious KGC.[18]

This approach is promising, but unusable for our scenario, as, even though it is not explicitly stated, the concerned parties seem to not have to deal with constraints concerning computational power or restricted memory; additionally, the parties in the ECCforContiki protocol are sensor nodes, which likely do not have enough distinctive features that might serve as a public key space.

Du and Ning [3] propose an asymmetric design with pre-shared public keys. The avoid the problem of scarce memory, which aggravates with the number of concurrently communicating nodes by using hash trees to authenticate the pre-loaded public keys: Of each node's public key, the hash is computed. That hash value constitutes the label of a leaf in a hash tree. Now the hash tree is built in the following fashion: each internal node is labelled the hash value of its two children, up to the root, whose label is stored on every node. If some node wants to authenticate another node's public key, it requests the public key of the node and its sibling, then it computes the hash value of the two, and so forth, up to the root, which should be the same value as the computed one. The intermediate hash values can be cached on the authenticating node. If there are n nodes in the network and the size of a hash is L, the cache will be at most L * log n. Now, if not all nodes need to communicate with each other, there can be separate, smaller hash trees, which reduces the computation and communication cost.

Finally, a few more basic solutions for sensor networks with less extreme requirements regarding the number of nodes are explained, or where not all nodes are required to communicate with each other. This makes special solutions for key management unnecessary.

P. Lowack [17] implemented TinySAM, an asymmetric security solution, for TinyOS, an operating system for constrained networks, which has a similar purpose as Contiki. The solution uses a certificate authority for authenticity. S. Siffert [31] developed a similar approach which uses public keys with a certificate authority, but for Contiki instead of TinyOS. M. Noack [22]finally implemented an ECC-based two-way-security solution on TinyOS, using pre-shared keys for authenticity. This last approach will serve as a basis for ECCforContiki.

ECCforContiki is roughly based on Noack's approach, which was implemented on TinyOS, whereas ECCforContiki will be implemented on Contiki. Furthermore, Noack's solution will be simplified, respectively enhanced:

1. Noack's key exchange will be simplified: Noack pre-loaded identifying data on each node and implemented a multi-step handshake to ensure authenticity, which can be

greatly simplyfied while keeping the same security level, by directly preloading the public key to the nodes. Authentic data packets cannot be forged, as an attacker does not have the shared key corresponding to the pre-shared public key.

2. Noack used ECC encryption, which can be rather resource-intensive. For this reason, the encryption step in ECCforContiki will be symmetric for better performance.

3. Finally, instead of a signature, a hash value will be computed over each message, which further saves resources compared to an asymmetric signature computation and verification.

Chapter 3

Solution Design

A widely used scheme for securing wireless sensor network communications is symmetric key cryptography with preloaded keys, which are distributed to the sensor nodes before deployment [11, 35]. One drawback, apart from the fact that with this approach is that the nodes have to store one shared key per sensor node that they communicate with, which can fill up the memory quickly. With PKC, storing the public keys of the communication partners is not necessary, as the public keys needed for encryption can be fetched from the nodes for each session and authenticated against a hash of one or several public keys, thus saving space, as shown by Du and Ning [3]. Hence, public key cryptography is in some cases the better, more viable solution, especially with very large networks, which is why public key cryptography will be chosen for the design of ECCforContiki.

3.1 Network topology

The prototype network consists of four different groups of devices with regard to their role in the network. For a network schema see Figure 3.1. On the network runs an implementation of the TinyIPFIX protocol, which is a push-protocol for WSN, suited for collecting sensor data [26]. The network is comprised of four type of devices:

- Collectors: They record sensor data (e.g. temperatures) and send it to the aggregators (see below). They perform the IPFIX Exporting Process. In the case of ECCforContiki, the data will be encrypted before sending it to the aggregator via UDP. In order to encrypt the data, each collector has its aggregator's public key stored [2] [26].
- Aggregators: They register the data sent by the collectors (IPFIX Collecting Process), perform an aggregation function on the received data and send the aggregated data as a single data point to the border router [2] [26]. ECCforContiki will decrypt the data, deliver it to the aggregation function and encrypt the aggregated data again, before it the data is sent to the border router via UDP. Each aggregator stores the public keys of the collectors that are allocated to this aggregator. The

aggregator uses these keys for decrypting the received packets and the public key of the server for encrypting the aggregated data.

- Border router: Generally, there is one border router per WSN. It is plugged into the server, where the data is processed further. The border router receives all aggregated data from the aggregators and forwards it without alterations to the server via the Serial Line Internet Protocol. [29]

- Server: The server is a configuration, management and data handling framework called CoMaDa and is written in Java. It collects the data from the aggregators, saves it into an SQL database and renders it via a visualization module, which can be accessed via a web browser. [25] It also contains an ECCforContiki Java module, which decrypts the received data before processing it further. It has the public key of each aggregator stored.

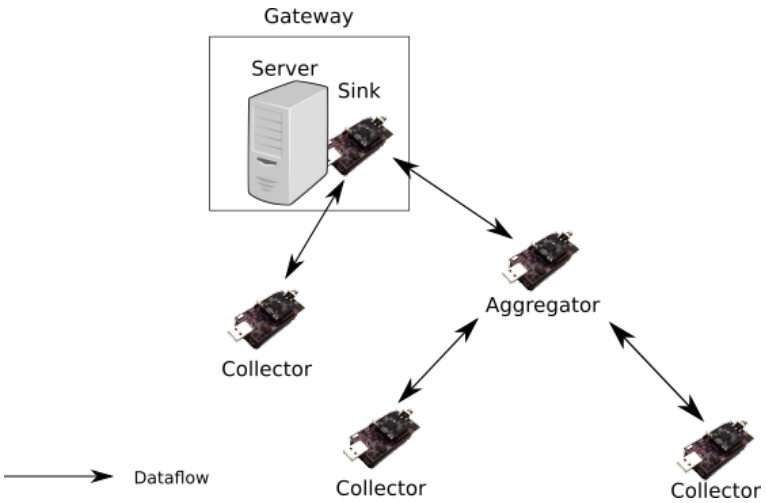

Figure 3.1: Network topology [31, p. 4]

3.2 Security goals

For security sensitive network communication, there are three main security goals, which are guaranteed by the chosen design:

- Confidentiality: The two communication partners should be able to communicate confidentially, i.e. a third party should not get access to the exchanged information, and it should be ensured that the communication partners are the intended ones. Confidentiality is achieved by encrypting the packets that are exchanged between collectors and aggregators, resp. between aggegators and the server. This ensures that only the intended party can access the data being sent.

- Integrity: A third party should not be able to alter the message without the intended receiver noticing. Integrity is ensured by attaching a hash - a CRC16 check, precisely

- of the message, which serves as a MAC, after the message itself, and encrypting the message together with its hash. By choosing hashes instead of signatures, the relatively costly signature generation and verification can be avoided, which would put a heavy load on the rather small processors of the collectors and aggregators.

- Authenticity: A third party should not be able to participate in the communication. In our case, this means that it should be impossible for an attacker to send fake sensor data to the aggregator or impersonating an aggregator to send fake data to the border router (with the goal of skewing the aggregated data). [11] This includes re-sending legitimate messages by either party. Authenticity is ensured by encrypting the data with the shared key of the two parties, which is only known or can only be computed by holders of the private-public key pair. Re-sending a legitimate packet is prevented by attaching a counter to the message. If the received counter is not new with each message, the message is rejected.

3.3 Basic Cryptographic Functions

In this section, the coverage of the basic functions of public key cryptography by ECC-forContiki - identification (authentication), encryption, key establishment and nonrepudiation - will be discussed.

3.3.1 Identification/Authentication

In general, authenticity is provided by signatures or message authentication codes, but either of these alone do not suffice. Consider the following scenario: A wants to communicate securely and in an authenticated manner with B. A sends B a certificate with his public key and a digital signature and requests in turn B's public key. Upon reception, A can encrypt a message with his private key and B's public key. But how does B know it was in fact A who sent the certificate with the public and the encrypted message? This can be achieved by using a designated certificate authority, whose task it is to issue certificates. [23]

In the case of the present work, this would require a designated node and would induce an overhead caused by the certificate transmissions. Moreover, a previously authenticated channel would be required, which is not a trivial task.

But a peculiarity of WSNs can be exploited here: Other than with, for example, the Internet, where the communication partners are not known from the beginning, with WSNs, the sensor nodes that will be in the network are known, and before the deployment, the communication channel is supervised and therefore authenticated. During this pre-deployment phase, certain information (such as keys) can be distributed.

For these reasons, the chosen design will entail a pre-shared keys architecture, as for example Du et al [3] suggest, instead of one with a certificate authority.

With a network this small, hard-coding the shared keys is the most efficient solution and there are no security-wise drawbacks. For networks with significantly more nodes (500-1000 collectors per aggregator), there are advantages in storing the public keys in a condensed form like a hash tree, but if there are only a handful of nodes, the overhead would be unnecessarily large. As the key exchange mechanisms are already in place, expanding the security mechanism to the 'authenticate against hash tree' approach would be simple.

As for the server: it does not have significant memory restrictions, which favors directly saving the shared keys. An important precondition for this decision is that access to the server is restricted to few persons, which can safely be assumed as servers can usually not be accessed unrestrictedly.

3.3.2 Encryption

For efficiency considerations, the actual encryption is done by XORing the message together with the counter and its hash with the shared key, instead of encrypting it with the elliptic curve encryption algorithm. This sort of hybrid approach can be found in the Elliptic Curve Integrated Encryption Scheme (ECIES), where only signatures and key exchange are ECC-based, and the actual encryption is done in a symmetric manner [8].

M. Noack [22] measured the performance of ECC algorithms on the OpenMote-CC2518 (see Figure 3.1), on which the ECCforContiki prototype is implemented (see Chapter 4). The speed of computation lead to the decision not to implement ECC encryption for ECCforContiki, but to choose a hybrid design, where for the key exchange protocol ECDH is chosen, but for the actual encryption a symmetric approach will be implemented.

The encryption algorithm is an simplification of AES, which has been preferred over the full-fledged AES approach for performance considerations.

Table 3.1: Measured performance of ECC operations on TelosB (collectors, third column) and OpenMote-CC2538 (aggregators, second column) sensor nodes, on the latter of which the ECCforContiki prototype is implemented [22]

	Aggregator [s]	**Collector [s]**
EC Key Generation	4.77 ± 0.14	8.77 ± 0.17
SHA-1	< 0.1	< 0.1
ECDSA Sign	5.14 ± 0.19	9.28 ± 0.18
ECDSA Verifiy	10.20 ± 0.19	18.51 ± 0.19
ECIES Encrypt	5.98 ± 0.15	9.41 ± 0.18
ECIES Decrypt	4.96 ± 0.19	-

This setup offers protection against any kind of eavesdropping attack or man-in-the-middle attack, ensuring authenticity of the received data. There is no possibility to decrypt the traffic, and the messages cannot be tampered with, as they are signed.

3.3.3 Nonrepudiation

An assumption made with the design is that the sensors are operating in a secure environment, i.e. there is no possiblity for an attacker to get ahold of a sensor and copy its contents or change the behavior of a sensor. This assumption is important when discussing a possible nonrepudiation function of the architecture.

Nonrepudiation becomes relevant when dealing with legal ramifications of an application, for example when it has to be ensured that a message has not been changed after sending. A possible scenario is the exchange of contracts or legal statements. As the sensor nodes of the network simply record sensor data and send it off to the server, there are no legal consequences of any sort. Furthermore, a node "retracting" their message would require a mechanism to somehow modify the already recorded data, which is nonexistent. This fact renders the nonrepudiation discussion superfluous.

Chapter 4

Implementation

In this chapter, the implementation of the chosen solution is explained, and the hardware on which the solution is deployed is presented.

4.1 Hardware

The prototype of ECCforContiki was deployed on OpenMotes, more precisely a combination of the OpenMote-CC2538 Rev.E and the OpenUSB Rev.B. The OpenMote-CC2538 consists of the CC2538 Texas Instruments processor with 512KB storage and 32KB memory. It has three LEDs (red, yellow and green). The OpenUSB is equipped with a USB port, sensors for light, temperature, humidity, and acceleration. The sensor node can be powered on via the USB port or with two AA batteries. [33] [31]

This particular hardware combination does not have a I/O interface, which means that it is not directly feasible to debug via the command line. Debugging is hence done via the LEDs.

4.2 TinyIPFIX

TinyIPFIX is a push protocol for transmitting sensor data in a power saving fashion. It is based on IPFIX and reduces redundancy by separating data and header information and sending the header information, which stays the same for a long time, only periodically. There are two processes, the Exporting Process and the Collecting Process, the former of which records, for example, sensor data and pushes it to the Collecting Process. Similar to IPFIX, TinyIPFIX allows the reduction of messages by aggregating the data received by the Collecting Process [26] [2].

4.3 ECC Library

The ECC library that was used for ECCforContiki is micro-ecc by Kenneth MacKay [19]. It was chosen over OpenSSL [12] with regard to the very small resources of the nodes and because it solves exactly the needs of the problem at hand, while OpenSSL is a rather comprehensive solution for various problems. The micro-ecc library has implementations for key generation, ECDH (key exchange) and ECDSA (digitial signature generation and verification), and different optimization levels, 0 being "unusably slow for most applications" [19]. It supports 5 standard NIST curves (see Chapter 2) from which the *secp256k1* is used for computing the keys.

4.4 Protocol Implementation

In the ECCforContiki prototype, there are four physical sensor nodes available in total, one of which is the border router, which means there are two collectors and one aggregator. The two collectors record their data and after a certain interval, they encrypt it and send it to their aggregator (in bigger networks, there may be more than one aggregator and the collectors have one designated aggretagor to which they send their data). According to the IPFIX protocol, the data is split into template and data packets. The aggregator checks which collector each packet comes from and decrypts it with the correct shared key. It aggregates the data according to the aggregation function chosen at deployment, encrypts it again and sends it to the border router, which forwards all packets to the server without modification.

During each encryption and decryption step, several algorithms are performed: Before encrypting a packet, the sender computes a CRC checksum on the message, which takes two bytes. The message counter is updated and serves as replay attack protection. The checksum, the message counter and the message are concatenated and the padding length is computed as the number of missing bytes to the next full block, i.e. block size - (message length + checksum length + counter length), where the block size is a multiple of the shared key length, depending on the length of the original message. The message together with the CRC checksum is then padded with random characters. In the last byte of the padded block the padding length is stored. The entire padded message is encrypted (message length/block size) times. Finally, the first four bytes of the sender's public key is stored in the first four bytes of the packet.

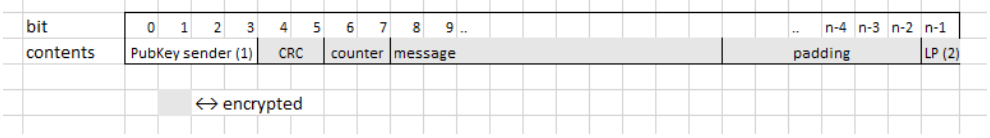

Figure 4.1: ECCforContiki packet schema.

After receiving a message, the aggregator or server first checks the first four bytes for a matching public key. If it matches with one that is stored, it decrypts the rest of the packet with the appropriate shared key. Next, it checks the message counter received. If

it is higher than the stored counter, it proceeds with removing the padding by reading the padding length on the end of the message. Finally, it computes the CRC check on the message and compares it with the one found in the packet. If they match, the message is returned to the next protocol layer, TinyIPFIX. (See Listing 5.1 for debug output of this process.)

If an attacker tries to forge a message by copying the first four bytes and sending some random bytes with the rest of the message, it would be rejected at the checksum stage latest, as the attacker cannot forge a message where the - decrypted - third and fourth byte randomly matches the CRC check of the message, if the whole packet is encrypted.

One peculiar detail should be noted: The first four bytes of the message correspond to the sender's public key. They serve as an ID to the receiver of the message. This design choice is especially convenient as it is not dependent on the network addresses assigned by TinyIPFIX, and thus the two protocol layers are completely independent. Another advantage is that if the network grows very large, the public keys can simply be appended in their entire length and the key authentication can be made more efficient by implementing hash trees, as suggested by Du and Ning [3].

4.4.1 Protocol function calls

The protocol implementation resides in the `ecc_crypto.c` file in the `ECC_Protocol` folder. For easier re-keying, there are two different encryption and decryption methods for the collector resp. the aggregation process, even though the encryption algorithm is exactly the same for both types of sensor nodes. These process-specific methods are located in `collector-openmote/ECC_encrypt_collector.[c|h]`, respectively `aggregator/ECC_encrypt_aggregator.[c|h]`. These files contain the specific keys for each sensor and have to be updated with new keys before deployment. The `ECC_encrypt_[type]()` method (and ECC_decrypt_collector() method for aggregators) in the sensor type specific files call the protocol methods `ecc_udp_sendto()` and, in the case of the aggregator, `decrypt()`. This sensor specific function call can be seen in Listing 4.1.

The `ecc_udp_sendto()` method, called in line 16 of Listing 4.1, in the `ecc_crypto.c` file encrypts and packages the cleartext as described in Section 4.4: It computes and attaches the CRC check on the input data together with the sender's public key, updates and attaches the current message counter, adds the padding, so that the packet has the correct length, encrypts the entire message, adds the first four bytes of the public key of the sender and finally sends it to the address given by the `to` parameter via UDP..

The `decrypt()` method, called in line 28 of Listing 4.1, mirrors the `ecc_udp_sendto()`, so to speak: It decrypts the cleartext with the given shared key, checks if the message counter is higher than the one that is currently stored on the sensor node (corresponding to the `*counter` parameter), re-computes the CRC check and compares it to the one received. Finally, it computes the original data length, which can be computed by subtracting the length of the padding from the length of the received ciphertext (not counting the first four bytes, that consist of the public key of the sender), which can be found in the very

```
1
2   //for decryption of collector packets
3   uint8_t pubKCollector1[64] = { ...}; //actual keys omitted here
4   uint8_t sharedSecretColl1Aggr[32] = { ...};
5
6   uint8_t pubKCollector2[64] = {... };
7   uint8_t sharedSecretColl2Aggr[32] = { ...};
8
9   //for encryption
10  uint8_t pubKAggregator[64] = { ... };
11  uint8_t sharedSecretComada[32] = { ... };
12
13  int ecc_encrypt_aggregator(struct simple_udp_connection *c, uint16_t
        msg_ctr,
14     const void *data, uint16_t datalen, const uip_ipaddr_t *to){
15     //function call to ECC_Protocol
16     return ecc_udp_sendto(c, msg_ctr, data, datalen, to, pubKAggregator,
17        sharedSecretComada);
18  }
19
20  uint16_t decrypt_collector(uint8_t *in, uint16_t datalen, uint16_t
        *original_datalen,
21     uint8_t *out, uint16_t *counter){
22
23     //check if public key received with the message corresponds to one that
           is stored,
24     // and if yes, which one; call the decryption method with the correct
           shared key
25     if(memcmp(in, pubKCollector1, 4) == 0){
26        return decrypt(in, datalen, original_datalen, out, counter,
              sharedSecretColl1Aggr);
27     } else if(memcmp(in, pubKCollector2, 4) == 0) {
28        return decrypt(in, datalen, original_datalen, out, counter,
              sharedSecretColl2Aggr);
29     } else {
30        return 0;
31     }
32  }
```

Listing 4.1: ECC_encypt_aggregator.c methods, each with shared key and public key parameters. ECC_encrypt_collector.c is analog, except there is no decryption call

last byte and returns an array of length `original_datalen` with the original message. It returns the new message counter, to which the sensor's stored message counter is updated.

As the border router simply forwards the traffic to the server (written in Java), the incoming packets from the aggregators have to be decrypted at the server, so that the next layer - TinyIPFIX - can process the received data. Hence, a decryption method in Java is needed. It is one of the implemented methods, among others such as checksum and message counter checking methods, that are called in the process() method that returns the processed data to the TinyIPFIX process. The ECCProtocol process can be inspected in Listing 4.2. First, the first four bytes are compared with the public key of the aggregator, which is stored on the server. If they match, the rest of the message is decrypted with the shared key (around line 20). Then, the received message counter is checked against the currently stored message counter (line 24). If it is higher, the stored one is updated to the received. If it is not, the packet is discarded (line 26). Next, the received CRC is checked by re-computing it with the received payload an the public key of the aggregator. If it does not match, the packet is discarded. Otherwise, the packet is returned to the next lower layer (line 39).

As the encryption algorithm, that runs on the server, works exactly the same with collector and aggregator payloads, respectively, the server implementation of `decrypt()` corresponds exactly to the C implementation that is run on the aggregators, where they decrypt collector packets.

Before deployment, the sensors have to be provided with their keys. The `make_keys.c` file contains the a key generation method, which is run locally and in turn calls the ECC key generation algorithm from the micro-ecc library [19], which contains the ECC implementation that was used for the prototype.

```
1
2   private final String sharedSecret = " ... "; //actual keys omitted here
3   private final String pubKAggregatorString = " ...";
4   private final byte[] pubKAggregator =
        hexStringToByteArray(pubKAggregatorString);
5   private final byte[] sharedSecretBytes = hexStringToByteArray(sharedSecret);
6
7   private int messageCounter= 0;
8
9   @Override
10  public WSNProtocolPacket process(WSNProtocolPacket packet) throws
        WSNProtocolException {
11
12  byte[] pubKeySender = Arrays.copyOfRange(packet.getPayload(), 0,4);
13
14  if(!(Arrays.equals(pubKeySender, Arrays.copyOfRange(pubKAggregator, 0,
        4)))) {
15     System.out.println("[ECCProtocol.process] Pub key of sender doesn't
           match; likely not a
16       ECCProtocol packet");
17
18     return null;
19  }
20  byte[] decrypted =
        decrypt(bytesToHex(Arrays.copyOfRange(packet.getPayload(), 4,
        packet.getPayload().length)), sharedSecret);
21
22  byte[] counterReceived = Arrays.copyOfRange(decrypted, 0, 2);
23
24  if(!(checkMsgCounter(messageCounter, counterReceived))){
25     //counter mismatch, possibly replay attack
26     return null;
27  }
28
29  byte[] crcCheckReceived = Arrays.copyOfRange(decrypted,2,4);
30  byte[] payl = Arrays.copyOfRange(decrypted, 4, decrypted.length);
31
32  //CRC check incorrect; possibly bogus data
33  if(!(checkCrc(crcCheckReceived, payl))){
34     return null;
35  }
36
37  System.out.println("[ECCProtocol.process] payload: "+bytesToHex(payl));
38  //return cleartext to lower layer
39  return new WSNProtocolPacket(packet.getID(), payl,
        packet.getSourceAddress());
40  }
```

Listing 4.2: ECCProtocol.process() function

Chapter 5

Evaluation

In this chapter, three different evaluations are performed: Memory size measurement, energy consumption measurement, and test methods. Moreover, the different results and their usefulness will be discussed.

It is important to highlight that as the sensor nodes do not present an I/O interface, the measurement of energy, memory consumption, and code complexity analysis was performed indirectly (i.e., using external tools).

5.1 Energy consumption

Energy consumption can be estimated by connecting a multimeter to one side of a battery and the power socket of a sensor, as shown in Figure 5.1.

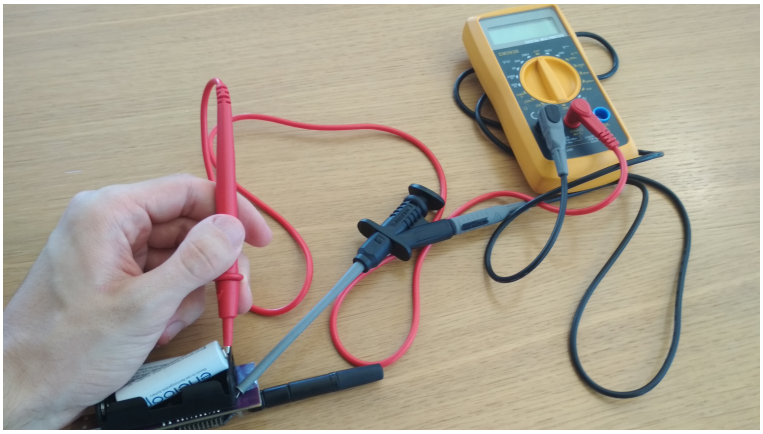

Figure 5.1: Energy consumption measurement setup [31]

The measurements does not seem to vary with or without encryption. For the aggregator, between two messages being sent, the power consumption varies between 31.9mA and 32mA. It increases to approx. 33.9mA while a message is being sent. As the reception of the collectors' packets happens directly before the transmission of the aggregated

Table 5.1: TinyIPFIX (unencrypted) and ECCforContiki (encrypted) size according to `size` utility (all numbers in bytes).

	Collector		Aggregator	
	ROM	RAM	ROM	RAM
Unencrypted	48250	15999	48359	17073
Encryptyed	48909	16095	49602	17297
Difference	659	96	1243	224
Change	+1.37%	+0.6%	+2.6%	+1.3%

messages, it is infeasible to determine if the reception or the transmission requires more power, or to determine the point where the receiving phase stops and the transmission phase starts.

The collectors consume approx. 31.2mA to 31.3mA while idling and approx. 32.0mA while sending.

When comparing these measurements with Siffert's *sTiki* [31], who measured around 30mA during idling and approx. 31.5mA during transmission, a slight difference is conspicuous. As the unencrypted version is the same with both approaches, and both the unencrypted and the encrypted version measured around the same energy consumption, this difference seemingly cannot be explained by more or less efficient approaches. The measuring method seems to be too inconsistent to draw reliable conclusions about the energy consumption.

One dependable statement about the energy consumption of ECCforContiki is that it does not seem to consume significantly more energy that the unencrypted protocol.

5.2 Memory consumption

Memory and storage consumption can be measured with the `size` tool. It receives as input the compiled .elf file with which is loaded onto the sensors for deployment. The storage and memory requirements of the binary of the encrypted version, of the unencryted version (both include the memory requirements of the operating system) and of the difference between the two can be gathered from the table below. ROM corresponds to the `text` output parameter, whereas RAM corresponds to the `data` and `bss` parameters.

It is apparent that the security solution is memory-efficient: The increase in RAM or ROM is within the single-digit percentage change.

Compared to S. Siffert's solution *sTiki* [31], which is implemented on the same hardware, which encrypts the TinyIPFIX protocol with a certificate authority, and which uses 52796 bytes of ROM and 16367 bytes of RAM on the collector and 52913 bytes of ROM respectively 17441 bytes of RAM on the aggregator, it can be seen that ECCforContiki is significantly more efficient with regards to ROM and marginally more efficient with regards to RAM consumption.

M. Noack, who also implemented an ECC solution, but on TinyOS, used OpenMotes for his aggregators, too. He reported a total ROM size of 46216 bytes, of which 13042 bytes accounted for the cryptography part, and a RAM consumption of 8470 in total, where 1018 bytes were due to additional RAM consumption by his implementation. [22]

Comparing the ECCforContiki numbers, it is obvious that the unencrypted version on Contiki already uses significantly more RAM and ROM, but the difference caused by the encryption is much smaller with ECCforContiki (1243 bytes of ROM, 224 bytes of RAM for the aggregator). It is apparent that the hybrid ECCforContiki solution does indeed save a lot of resources. This is obvious when comparing the memory consumption, and it is to be suspected that it is also much faster, even though there is no possiblity to measure the time consumption of ECCforContiki (see Table 3.1 for Noack's time measurements).

Considering the ROM size of 512KB in total, this means that the directly storing the shared keys of the communication partner is feasible up to a significant network size.[1]

5.3 Proof of Operability

As the sensor nodes do not have an I/O interface, proving that the protocol works as intended cannot happen directly during operation, i.e. via command line outputs by the nodes during operation. But as the server runs locally, it running correctly can be shown. As the aggregators send their packets directly to the server, their content can be examined as well. Debug messages or print statements are not available.

In Listing 5.1, the debug output that appears when running CoMaDa is shown: Lines 1-4 indicate that the startup and authentication at the database has been completed successfully. Line 5 indicates that a packet with a lenght 132bytes has arrived. Lines 5-9 are the hexadecimal representation of the packet bytes. The first four bytes (256C26AE) correspond to the public key of the sender. As described in Chapter 4, the rest of the message is encrypted and looks like a random string. Next, in line 10, the length of the actual encrypted message is printed (132/4 = 128). It should be divisable by 32, as that is the block size, and the remainder is padded with random bytes. The next lines (11-14) are a printout of the decrypted packet. Next, the padding is removed, as indicated by line 20.

The first two bytes of the decrypted message correspond to the sender's message counter, which is 3 in this case, which is an indication that three other messages were already sent before the server completed its startup. As the received message counter simply has to be bigger than the one stored on the server, this is not a problem. This means that the message counter is not vulnerable to lost packets, as it simply updates its counter to the one received, if it is bigger. The third and fourth byte of the decrypted message are the CRC check that was sent with the message. As can be seen in line 31 and 32, it is equal to the CRC check that was computed by the server, the input of which being the decrypted payload, without CRC check and counter, but with the entire public key of the sender.

[1] It is to be suspected that the lower protocol layers which handle the actual sending and receiving of messages will fail before the sensor runs out of storage, if the network size was increased.

The final payload is printed out on lines 33 to 36 and returned to the next lower layer. The debug output of it can be seen in the last 5 lines.

Analysing the packets from the collectors is impossible; it can merely be shown that they arrive correctly in an aggregated state at the server. It is impossible that the data somehow arrives directly at the server, because packets that do not originate from the aggregator are rejected.

In Figure 5.2, the data packet visualization of CoMaDa is pictured. The aggregated data packets can be discerned: Several measurements of the same parameter can be seen in the same message, which is characteristic for the *message aggregation* mode of operation of TinyIPFIX. The lower part of the listing is a representation of a parsed template packet.

The last option available to test and verify the correct operation of the collectors is to run parts of their code locally. Encrypting (including the CRC check, counter and padding) and decrypting (including CRC check, counter check and padding removal) mock data is the functionality that can and should be tested.

The same can be done in unit tests for the server side of the ECCforContiki protocol. The `testDecrypt()` function takes the output from the encryption test function from the sensors. It is evident that the two functions (encryption and decryption) mirror each other perfectly, as the original input results again after encryption and subsequent decryption, as can be seen in Listings 5.3 and 5.4:

In summary, the available evaluation methods are not exceedingly insightful, but do reveal that 1) the ECCforContiki implementation is quite efficient especially regarding memory requirements and 2) that it barely uses any more energy than the unencrypted method. It compares well with sTiki, which is implemented on the same hardware, and uses significantly to marginally less memory and roughly the same amount of energy. The difference can likely be explained by the unreliableness of the available means of measuring.

The proof of operability is not trivial, because at least the operation of the collector nodes cannot be observed directly, but only indirectly through correct sensor data arriving at the server. Packets from the aggregators can be examined directly. Finally, correct operation of the protocol can be shown by testing the protocol programatically.

```
Establish WebSocket connection to pull interface: wss://pull.webmada.csg.uzh.ch
WebSocket connection established to: pull.webmada.csg.uzh.ch/130.60.156.11:443
Authenticate at pull.webmada.csg.uzh.ch/130.60.156.11:443...
Authentication successful.
length: 132 message:
256C26AEF1EF3BEB9FF55E5076EDA2CFD13B5D58CD12B1109296C5AB9539B5DD18FC54FFC
5BA7F072892A54342BBD6CFD53B5D58CD12B1109696C6AB9539B5DD1AFC54FFC5BA7F072
A92A34342BBD6CFD33B5D58CD12B1109096C3AB9539B5DD1EFC54FFC5BA7F072E92A0434
2BBD68E72F74A86AC78583F5E343C3B42E76B6538933BF8
[ECCProtocol.decrypt] length hexstring = 128
[ECCProtocol.decrypt] decrypted:
    00033C6C0467FF0100000C80B000021234567880B100021234567880B200021234567880B30
0041234567880B400021234567880B500011234567880B000021234567880B1000212345678
80B200021234567880B300041234567880B400021234567880B5000112345678C113CC15CC
553C91AF7DA2FB82E388A638926F6D15
[ECCProtocol.decrypt] len padding: 21
[ECCProtocol.decrypt] data stripped of padding:
00033C6C0467FF0100000C80B000021234567880B100021234567880B200021234567880B30
0041234567880B400021234567880B500011234567880B000021234567880B1000212345678
80B200021234567880B300041234567880B400021234567880B5000112345678
[ECCProtocol.process] decrypted message, stripped of padding length:
    107message:
    00033C6C0467FF0100000C80B000021234567880B100021234567880B200021234567880B30
0041234567880B400021234567880B500011234567880B000021234567880B1000212345678
80B200021234567880B300041234567880B400021234567880B5000112345678
[ECCProtocol.checkMsgCounter] counter = 00
[ECCProtocol.checkMsgCounter]: counter received = 3, new counter stored = 3
[ECCProtocol.process] message counter correct
[ECCProtocol.process] counter received: 0003, new counter stored: 3
[ECCProtocol.checkCrc] data:
0467FF0100000C80B000021234567880B100021234567880B200021234567880B30004123456
7880B400021234567880B500011234567880B000021234567880B100021234567880B200021
234567880B300041234567880B400021234567880B5000112345678
[ECCProtocol.checkCrc] computedCrc: 3C6C, receivedCrc: 3C6C
[ECCProtocol.checkCrc] crc checks equal: true
[ECCProtocol.process] payload:
    0467FF0100000C80B000021234567880B100021234567880B200021234567880B
300041234567880B400021234567880B500011234567880B000021234567880B1
00021234567880B200021234567880B300041234567880B400021234567880B50
00112345678
[IPFIXParser._parseTemplate] sequence number: 4294967295
[IPFIXParser._parseTemplate] nodeID: 64768
[IPFIXParser._parseTemplate] set id: 2
[IPFIXParser._parseTemplate] fieldcount: 12
[IPFIXParser._parseTemplate] data packet length: 0
```

Listing 5.1: Debugging output in CoMaDa

```
|+--[64768] Data received Sun Oct 07 23:44:57 CEST 2018
|
|----- Temperature (SHT12)[2] (305419896 - 32944): 28.44 Â°C
|----- Humidity (SHT12)[2] (305419896 - 32945): 49.65 %
|----- MAX44009 Light[2] (305419896 - 32946): 258.75 Lux
|----- Node Time[4] (305419896 - 32947): 80 sec
|----- NodeID[2] (305419896 - 32948): 38891
|----- Pull Flag[1] (305419896 - 32949): 0
|----- Temperature (SHT12)[2] (305419896 - 32944): 26.9 Â°C
|----- Humidity (SHT12)[2] (305419896 - 32945): 53.33 %
|----- MAX44009 Light[2] (305419896 - 32946): 545.48 Lux
|----- Node Time[4] (305419896 - 32947): 100 sec
|----- NodeID[2] (305419896 - 32948): 39887
|----- Pull Flag[1] (305419896 - 32949): 0


+--[64768] Template: 256, received Sun Oct 07 23:45:02 CEST 2018
|

|----- Field 32944, enterpriseNumber: 305419896, length: 2
|----- Field 32945, enterpriseNumber: 305419896, length: 2
|----- Field 32946, enterpriseNumber: 305419896, length: 2
|----- Field 32947, enterpriseNumber: 305419896, length: 4
|----- Field 32948, enterpriseNumber: 305419896, length: 2
|----- Field 32949, enterpriseNumber: 305419896, length: 1
|----- Field 32944, enterpriseNumber: 305419896, length: 2
|----- Field 32945, enterpriseNumber: 305419896, length: 2
|----- Field 32946, enterpriseNumber: 305419896, length: 2
|----- Field 32947, enterpriseNumber: 305419896, length: 4
|----- Field 32948, enterpriseNumber: 305419896, length: 2
|----- Field 32949, enterpriseNumber: 305419896, length: 1
```

Listing 5.2: CoMaDa Visualization

```
1
2   *******
3
4   Input of length 31, current message counter = 2:
5   0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C00
6   encrypting with shared key of collector 1 - aggregator ...
7
8   ECC UDP packet of length 68:
9   02EAF3E90D991B875C034700F452C2C0BA296A904514EC1125148CB6F9E942B8
10  48C6DC150196C5C59061385B001424E27B99CA6EB4FCA667EC4C7EA61D04C4C218E3B304
11  new messagecounter: 3
12  decrypted and unpacked packet of length 31 (1f):
13  0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C00
14
15
16  *******
17
18  Input of length 62, current message counter = 279:
19  0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C0
20  A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C0000
21  encrypting with Comada shared key...
22
23  ECC UDP packet of length 100:
24  256C26AEF0F4816A9199AD5B7DE1A4446D315446F34FC59A289ACDB2AD65C65
25  1A0F75AE1FDE00D8C9798AA5D7CE6A2456A375541F54EC29C299DCBB3AA63C7
26  56A6F05AE1F1EC34185208C7637B5A9F17C2617A17FC5391797D426C0B6CA956
27  E9FEED58F3
28
29  new messagecounter: 280
30  decrypted and unpacked packet of length 62 (3e):
31  0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C0A0
32  B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C0000
```

Listing 5.3: testEncryptDecrypt() output from sensor nodes, run locally

```
1  [decryptTest] cleartext:
2  0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C0
3  A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C0000
4  [decryptTest] Length ciphertext: 96
5  [ECCProtocol.decrypt] length hexstring = 96
6  [ECCProtocol.decrypt] decrypted:
       011886ED0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C
7  0C0C0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0
8  C0C0C0000339FC99A66320DB73158A35A255D051758E95ED4ABB2CDC69BB45
9  4110E1E
10 [ECCProtocol.decrypt] len padding: 30
11 [ECCProtocol.decrypt] data stripped of padding:
12 011886ED0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0
13 C0C0C0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0
14 C0C0C0C0000
15 [decryptTest] decrypted and stripped of checks:
16 0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C
17 0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0A0B0C0C0C0C
18 0000
```

Listing 5.4: CoMaDa testDecrypt() output

Chapter 6

Summary and Conclusions

The thesis herein presented a solution that addresses the issue of securing the communication between nodes in a WSN. The hybrid design of the solution is based on ECC for key exchange and symmetric encryption and was implemented on a small prototype sensor network. The solution covers all required security goals (authentication, integrity, and confidentiality). After evaluating the memory requirements and energy consumption, it could be shown that the solution is exceedingly efficient and hardly adds any overhead to the unencrypted protocol.

There are two variations to the solution, one where the shared keys, shared by two communication partners, are directly stored on the nodes, and the other where the public/private key pair is stored on the nodes. With the latter variation, the public key of the sender is sent with each message and authenticated against a hash tree, which uses a fraction of the memory compared to storing each public key of every communication partner, but induces a communication overhead

It was shown that depending on the network size, it is reasonable to deploy one or the other variation: If there are only few sensor nodes in the network (> 1000), it is reasonable to store the shared keys directly on the sensors, because the overhead from authenticating the public keys through the hash tree and computing the shared keys from the public keys is larger than the efficiency profit. When the network is large ($>=1000$ nodes), the second variation seems advantageous, as storing the public keys of alle the communication partners will become a problem considering the memory size of 512KB the OpenMotes. It has to be seen how the lower protocol layers can cope with the actual message handling with a network this large. For this task a simulator would be necessary, which is not available for the current TinyIPFIX implementation.

Finally, another possible improvement concerns the deployment process: Currently, all sensor nodes have to be manually equipped with their shared keys, respectively with the public and private keys. For TinyOS there exists a Web service for keying the sensor nodes, which would have to be ported to Contiki. The implementation of such a web service would only ease the work of manually deploying the keys, but would not affect on the general security of the protocol.

Bibliography

[1] C. Bormann, M. Ersue, and A. Keranen: *Terminology for Constrained-Node Networks* Internet Engineering Task Force (IETF), RFC 7228, Fermont, CA, USA, May 2014, DOI 10.17487/RFC7228, `https://www.rfc-editor.org/info/rfc7228`.

[2] B. Claise, B. Trammell, and P. Aitken, *Specification of the IP Flow Information Export (IPFIX) Protocol for the exchange of IP Traffic Flow Information*, STD 77, RFC 7011, September 2013, DOI 10.17487/RFC7011, `http://www.rfc-editor.org/rfc/rfc7011.txt`

[3] W. Du, R. Wang, and P. Ning: *An Efficient Scheme for Authenticating Public Keys in Sensor Networks*, Electrical Engineering and Computer Science, Vol. 17, pp 1-10, 2005, `https://surface.syr.edu/eecs/17`

[4] A. Dunkels, B. Gronvall, and T. Voigt: *Contiki - a lightweight and flexible operating system for tiny networked sensors*, 29th Annual IEEE International Conference on Local Computer Networks, pp 455-462, November 2004, ISBN 0-7695-2260-2, DOI 10.1109/LCN.2004.38, `https://ieeexplore.ieee.org/document/1367266/`

[5] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali: *Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems*, Proceedings of the 4th international conference on Embedded networked sensor systems, pp 29-42, November 2006, ISBN 1-59593-343-3, DOI 10.1145/1182807.1182811, `https://dl.acm.org/citation.cfm?id=1182811`

[6] L. Eschenauer and V. D. Glgor: *A Key-Management Scheme for Distributed Sensor Networks*, Proceedings of the 9th ACM conference on Computer and communications security, pp 41-47, ACM, New York, 2002, ISBN 1-58113-612-9, DOI 10.1145/586110.586117, `https://dl.acm.org/citation.cfm?id=586117`

[7] O. Garcia-Morchon, S. Kumar, and M. Sethi: *State-of-the-Art and Challenges for the Internet of Things Security*, Internet Engineering Task Force, IETF, draft-irtf-t2trg-iot-seccons-15 (work in progress), version 15, pp 1-47, Fermont, CA, 2017, `https://tools.ietf.org/html/draft-irtf-t2trg-iot-seccons-02`

[8] V. Gayoso Martínez, L. Hernandez Encinas, C. Sánchez Ávila: *A Survey of the Elliptic Curve Integrated Encryption Scheme*, Journal of Computer Science and Engineering. Vol. 2, Issue 2, pp 7-13, 2010, `https://www.researchgate.net/publication/255970113_A_Survey_of_the_Elliptic_Curve_Integrated_Encryption_Scheme`

[9] A. Jisov: *Compact representation of an elliptic curve point*, Internet Engineering Task Force (IETF), Fermont, CA, USA, March 2014, https://tools.ietf.org/id/draft-jivsov-ecc-compact-05.html (draft).

[10] H. Karl and A. Willig: *Protocols and Architectures for Wireless Sensor Networks*, John Wiley & Sons, Vol. 1, ISBN: 0470519231, GB, 2007

[11] C. Karlof, N. Sastry, and D. Wagner: *TinySec: A Link Layer Security Architecture for Wireless Sensor Networks*, Proceedings of the 2nd international Conference on Embedded networked Sensor Systems (SenSys '04), pp 162-175, Baltimore, MD, 2004, ISBN 1-58113-879-2, DOI 10.1145/1031495.1031515, http://dl.acm.org/citation.cfm?id=1031515

[12] E. Käsper: *Fast elliptic curve cryptography in OpenSSL*, International Conference on Financial Cryptography and Data Security, pp 27-39, Springer, Berlin, Heidelberg, 2011, ISBN 978-3-642-29889-9, DOI 10.1007/978-3-642-29889-9_4, https://link.springer.com/chapter/10.1007/978-3-642-29889-9_4

[13] N. Koblitz: *Elliptic Curve Cryptosystems*, Mathematics of Computation, Vol. 48, No. 177, ISSN 1088-6842, pp 203-209, 1987, DOI 10.1090/S0025-5718-1987-0866109-5, https://pdfs.semanticscholar.org/c7c5/47ede2da32aba645edb11e33f1d32af735e2.pdf

[14] T. Kothmayr, C. Schmitt, W. Hu, M. BrÃ¼nig, and G. Carle: *DTLS based security and two-way authentication for the Internet of Things*, Journal Ad Hoc Networks, ELSEVIER, Vol. 11, Issue 8, ISSN 1570-8705, pp 2710-2723, November 2013, DOI 10.1016/j.adhoc.2013.05.003, https://www.sciencedirect.com/science/article/pii/S1570870513001029

[15] J. Kuhn: Elliptic Curves as Algebraic Structures. https://jeremykun.com/2014/02/16/elliptic-curves-as-algebraic-structures/, queried on July 17, 2018.

[16] T. Lange: *Koblitz curve cryptosystems*, Finite Fields and Their Applications, Vol. 11, No. 2, ISSN 1071-5797, pp 200-229, April 2005, DOI 10.1016/j.ffa.2004.07.001, https://www.sciencedirect.com/science/article/pii/S1071579704000395

[17] P. Lowack: *Key Management in Wireless Sensor Networks with Support for Aggregation Nodes*, Master Thesis, Faculty of Informatics, Technische Universität Munich, Munich, 2013

[18] M. Ma, D, He, N. Kumar, and K. R. Choo: Certificateless Searchable Public Key Encryption Scheme for Industrial Internet of Things, IEEE Transactions on Industrial Informatics, Vol. 14, No. 2, ISSN 1941-0050, pp 759-767, 2018, DOI 10.1109/TII.2017.2703922, https://ieeexplore.ieee.org/document/7927473/

[19] K. MacKay: *micro-ecc* (Readme). https://github.com/kmackay/micro-ecc, last visited October 1, 2018.

[20] V. Gayoso Martínez, F. Hernández Álvarez, L. Hernández Encinas, and C. Sánchez Ávila: *A Comparison of the Standardized Versions of ECIES*, 2010 Sixth International Conference on Information Assurance and Security, Atlanta, GA, August 2010, pp 1-4, DOI 10.1109/ISIAS.2010.5604194, `https://ieeexplore.ieee.org/document/5604194/`

[21] A. Menezes and S. Vanstone: *Elliptic curve cryptosystems and their implementation*, Journal of Cryptology, Vol. 6, No. 4, ISSN 1432-1378, pp 209-224, 1993, DOI, 10.1007/BF00203817, `https://link.springer.com/article/10.1007/BF00203817`

[22] M. Noack: *Optimization of Two-way Authentication Protocol in Internet of Things*, Master Thesis, Communication Systems Group CSG, Department of Informatics IfI, University of Zurich UZH, Switzerland, 2014

[23] C. Paar, J. Pelzl: *Understanding Cryptography. A Textbook for Students and Practitioners*, Springer, 2010, ISBN 978-3-642-04101-3.

[24] A. Perrig R. Szewczyk, J. D. Tygar, V. Wen and D. E. Culler: *SPINS: Security Protocols for Sensor Networks*, Kluwer Academic Publishers, Netherlands, Wireless Networks, Vol. 8, No. 5, pp 521-34, 2002, DOI 10.1023/A:1016598314198, `https://dl.acm.org/citation.cfm?id=582464`

[25] C. Schmitt et al.:*CoMaDa: An adaptive framework with graphical support for Configuration, Management, and Data handling tasks for wireless sensor networks*, Proceedings of the 9th Conference on Network and Service Management, pp 211-218, Zurich, Switzerland, October 2013, ISBN 978-3-901882-53-1, DOI 10.1109/CNSM.2013.6727839, `https://ieeexplore.ieee.org/document/6727839`

[26] C. Schmitt, B. Stiller, and B. Trammell: *TinyIPFIX for Smart Meters in Constrained Networks*, RFC 8272, November 2017, DOI 10.17487/RFC8272, `https://www.rfc-editor.org/rfc/rfc8272.txt`

[27] J. Sen: *A Survey on Wireless Sensor Network Security*, International Journal of Communication Networks and Information Security (IJCNIS), Vol. 1, No.2, pp 55-78, 2009, `https://arxiv.org/abs/1011.1529`

[28] L. Sgier: *Optimization of TinyIPFIX. Implementation in Contiki and Realtime Visualization of Data*, Internship, Communication Systems Group CSG, Department of Informatics IfI, University of Zurich UZH, Switzerland, 2016

[29] L. Sgier: *TinyIPFIX Aggregation in Contiki*, Internship, Communication Systems Group CSG, Department of Informatics IfI, University of Zurich UZH, Switzerland, 2017

[30] Z. Shelby and C. Bormann: *6LoWPAN: The Wireless Embedded Internet,* Wiley Series on Communications Networking & Distributed Systems, John Wiley & Sons, Vol. 1, ISBN: 0470747994, GB, 2009

[31] S. Siffert: *Secure Data Transmission in Contiki-based Constrained Networks Offering Mututal Authentication*, Bachelor Thesis, Communication Systems Group CSG, Department of Informatics IfI, University of Zurich UZH, Switzerland, 2018

[32] A. Tanenbaum, D. Wetherall: *Computer Networks*, Pearson, 2013, ISBN 978-1-292-02422-6, `https://dl.acm.org/citation.cfm?id=2559493`

[33] *Texas Instruments: CC2538 A Powerful System-On-Chip for 2.4-GHz IEEE 802.15.4-2006 and ZigBee Applications*, December 2012, `www.ti.com/product/CC2538`, last visited October 1, 2018.

[34] D. Wätjen: Kryptographie. Grundlagen, Algorithmen, Protokolle. Spektrum Akademischer Verlage, Heidelberg, 2004, ISBN 10 3827414318

[35] S. Zhu, S. Setia, and S. Jajodia: *LEAP+: Efficient Security Mechanisms for Large-Scale Distributed Sensor Networks*, Transactions on Sensor Networks, Vol. 2, No. 4, pp 500-528, 2006. ISBN 1-58113-738-9, DOI 10.1145/948109.948120, `https://dl.acm.org/citation.cfm?id=948120`

List of Figures

List of Tables

Listings