CHALMERS
UNIVERSITY OF TECHNOLOGY

Securing Chaos

An Ultra Low-Latency Wireless Network Protocol for Mission Critical Applications

Master's Thesis in Computer Systems and Networks

ROBIN KARLSSON

Department of Computer Science & Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's Thesis 2016

Securing Chaos

An Ultra Low-Latency Wireless Network Protocol for Mission Critical Applications

ROBIN KARLSSON

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

Securing Chaos
An Ultra Low-Latency Wireless Network Protocol for Mission Critical Applications
ROBIN KARLSSON

Supervisor: Olaf Landsiedel, Department of Computer Science & Engineering
Examiner: Elad Schiller, Department of Computer Science & Engineering

Master's Thesis 2016
Department of Computer Science & Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2016

Securing Chaos
An Ultra Low-Latency Wireless Network Protocol for Mission Critical Applications
ROBIN KARLSSON
Department of Computer Science & Technology
Chalmers University of Technology

Abstract

Wireless Sensor Network (WSN) is a field on the rise with the advent of Internet of Things (IoT), where small and cheap computers are distributed to collect information about their local environment. To ensure resistance against malevolent adversaries, and ultimately correct functionality, security is commonly used. Chaos [23] is a novel network primitive with built-in high performance all-to-all data sharing that is designed for use in WSNs. This thesis investigates how to secure Chaos from the ground up. As part of this thesis, we add a link layer security layer to the existing network, and design and implement a protocol based on Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) for securely enabling new motes to join. This Elliptic Curve Cryptography (ECC) based approach enables flexible extensions in future iterations. We show that our dynamic join protocol performs sufficiently well for Chaos, such as being able to build a complete network of 32 motes from a single mote in less than 18 minutes on the TelosB platform, and that the approach is viable despite the high limitations of the Chaos protocol and the TelosB motes.

Keywords: Chaos, IoT, Wireless Sensor Network Security, TelosB, OpenMote, Cryptography

Acknowledgements

I would like to thank my supervisor, Olaf Landsiedel, for giving me the opportunity to work on this thesis, and for providing good guidance. I also thank Beshr Al Nahas for his generous advice and help in dire times, and Elad Schiller for being a good examiner.

Robin Karlsson, Gothenburg, October 2016

Contents

1
Introduction

This thesis investigates ways to secure the communication in the Chaos [23] all-to-all network primitive. We explain how this can be done and show our approach to it.

In this chapter, we give a short introduction to the field of WSN, their use cases, and properties. Later on, we explain our contributions and then an outline of how the thesis is structured.

1.1 Motivation and Context

In recent years, there has been great progress in computer technology. Transistor sizes have been decreasing steadily for decades, and with it, chip size, price and power consumption. Small, cheap, disposable, and low powered computers (henceforth called nodes) can be distributed over an area, and then use attached sensors to collect information about their surroundings, which they communicate with the network wirelessly. Such a network is called a WSN. Noteworthy applications of WSNs are as monitors of geological activities, such as volcano eruption detection [43] and landslide detection [34], and in military environments, such as sniper detection and localization systems, and submarine detection [11].

A WSN is also one of the fundamentals of the IoT. IoT is the concept that, ultimately, all of our everyday items and everything we want to interact with will be connected to the Internet. To accomplish such a task, one must rely on small and cheap computers. There already exist protocols for connecting cheap nodes to the Internet, such as the IPv6 port 6LowPAN, which aims to apply the Internet Protocol to even the smallest devices [28].

The sensor data collected by all the sensor nodes need to be collected and aggregated somehow. One common network principle is to let all sensor nodes send their collected data to a central collector node. An operator can then connect to this central node and view the aggregated data of all the nodes on the network. It does have weaknesses, however, such as a lack of redundancy and scalability issues.

It is beneficial to let all nodes share their sensor data over the whole network. This requires a form of all-to-all network communication. The well established way to do this has been to let all nodes send their data to a central node, which aggregates the data, and then sends it back to all sensor nodes. Chaos improves upon this

by utilizing in-network processing of data, and thereby lowering the latency for full network consensus [23]. Chaos also relies on synchronous transmission combined with capture effect to allow multiple nodes to successfully transceive simultaneously [23].

1.2 Problem statement

Up until now, Chaos has not offered any protection of network data against malicious adversaries. Data protection is a core requirement for applications, such as for monitoring patients' sensitive medical data [39]. Chaos needs to support it if it is going to be used as a building block for a varied range of applications. Unfortunately, most nodes in a sensor network are low powered and have severe constraints on CPU, power, memory, and network packet size. A highly streamlined solution which provides for the security must be adapted in order for Chaos to be moved outside of lab environments, while continuing to perform well.

This is related to the thesis's first research question:

- RQ1: Find an efficient solution that secures Chaos's link layer traffic, without severely affecting Chaos's high performance by adding too much overhead.

One of the largest problems for security in WSNs is how to handle cryptographic keys correctly, and particularly, how to secure communication between new nodes that do not yet share a common secret. General purpose computers commonly use Public Key Cryptography (PKC) for mutual authentication and key establishment. Historically, WSN researchers have considered PKC based schemes too computationally demanding for sensor nodes [8][13][7], and have therefore focused their attention on key establishment schemes solely based on symmetric key cryptography. However, recent research shows that PKC schemes in fact are viable even on low powered sensor nodes [15][26].

This leads into the second research question:

- RQ2: Combine a PKC based solution with the Chaos protocol to enable complex, dynamic security functionalities, such as a secure join.

1.3 Contributions

As a way of answering the research questions, the thesis accomplishes the following:

- We add a layer of security features on top of all regular network traffic in Chaos. We then confirm that the current implementation of Chaos on the TelosB platform still performs well with the added overhead.
- We develop an application through which a Chaos network operator can establish new symmetric keys with a low cost and high reliability.
- We adapt a PKC framework that allows for new, dynamic, functionalities to be added. We then design and implement a functionality based on this that

enables new nodes to join an existing network without any pre-shared secret.

1.4 Thesis outline

The thesis is structured as follows: after this Introduction chapter, Chapter 2 is background where we name and explain the technologies that are used as a foundation for our design and implementation. In Chapter 3, we discuss work that is related to this thesis, but does not align well enough with our goals to be used as standalone solutions. In the following Chapter 4, we discuss the requirements of a solution to the problem statement, and how we chose our design to fulfil these requirements. In Chapter 5, we describe how these design choices manifest concretely, and we also show our algorithms in greater detail. Chapter 6 describes how we test the performance of our implementation on different topologies and set ups. Lastly, Chapter 7 uses the results of the evaluation as a foundation for revisiting the contributions and problem statements in this Introduction Chapter, and decide whether they are actually fulfilled, and what needs to be done in future work.

2
Background

In this chapter, we give the required background for the thesis. First off, we explain the context of Chaos, how it works and its advantages. Then we present the underlying technologies we work with, and finally, we give a theoretical explanation for the different security mechanisms that are relevant for this thesis. This information is the foundation for the rest of the thesis, and in particular, the design choices we make in Section 4.

2.1 Wireless Sensor Network Communication

A WSN can consist of a large number of nodes spread out in a varied manner that is not always known before. The nodes are sometimes deployed by airplanes, and therefore, the topology can be hard to predict. The topology is also prone to change as a consequence of nodes eventually losing power, varying radio reception conditions, and mobile nodes moving around. One of the most basic forms of communication in WSNs lets all nodes flood the network with messages without routing. This has the advantages of being simple and redundant, and downsides of being power hungry and having large message overheads.

The standard approach for achieving all-to-all communication has historically been to take a centralized solution and then modify it through a few extra steps, like the following:

1. All nodes send their respective "data to be processed" to a central node.
2. The central node processes and aggregates all data.
3. The central node disseminates the processed data back to the network.

Chaos, in contrast, incorporates all these steps into a distributed and concurrent in-network processing of the data. This leads to short, intense bursts of radio traffic that Chaos thrives upon and makes use of through simple, yet clever, principles. This seemingly chaotic nature of Chaos is what allows it to perform with low latency and power consumption compared to its competitors [23].

2.1.1 Chaos Overview

Chaos is a tightly synchronized, highly distributed, flooding based network primitive. Chaos lets all nodes start communication periodically in rounds. In each Chaos round, all nodes should reach consensus for the payload. Every Chaos round is composed of a limited amount of time slots, each lasting a few milliseconds. Through

a set of rules, all nodes individually decide actions for each time slot, such as whether to transmit or receive on the radio the following time slot.

2.1.1.1 Chaos Scheduling

For this to work, all Chaos nodes need to be well synchronized. This is done by a dedicated node called the initiator, which starts the first round by sending a few packets on the radio. All other nodes associate to these round messages, after which they all share time slot length and the Chaos round interval, and so they can all calculate the time offset of when to start the following round, as well as turn the radio on and off for energy conservation reasons. The nodes can also adjust for clock drift by comparing the expected time of incoming packets to the actual time.

The well timed transmissions allow the capture effect [24] to take place [14]. Capture effect is the physical phenomenon where if two or more signals reach a receiver, that receiver receives only the strongest signal despite there being multiple colliding signals. Capture effect is central to Chaos's design, as it allows multiple nodes to transmit simultaneously, and work despite the increased interference. Another physical effect that strengthens Chaos is constructive baseband interference [23]. Constructive baseband interference takes place when two or more nodes transmit identical packets with small delay in between (within 0.5 µs on IEEE 802.15.4 [14]), which makes receivers interpret it as one single stronger signal.

2.1.1.2 Chaos Data Aggregation

Chaos has built-in mechanisms for utilizing in-network processing of messages. Chaos manages this by letting all nodes in the network have a flag in each packet (usually 1 bit) in a fixed location. These flags represent nodes' acknowledgements that the data in the packet is the aggregated value of all nodes with their respective flags set. When a node receives a packet, it has the option to merge the incoming data with its own data and then merge the flags and transmit the merged packet the following slot. Nodes receiving this packet know that the incoming packet is the aggregated data of all the nodes whose flags are set.

The merge operator is programmable by the Chaos network operator and is applied each time an incoming packet has flags set that are not set in the already received packets. A simple example of such a merge operator is to calculate the maximum value on the network, as seen in figure 2.1.

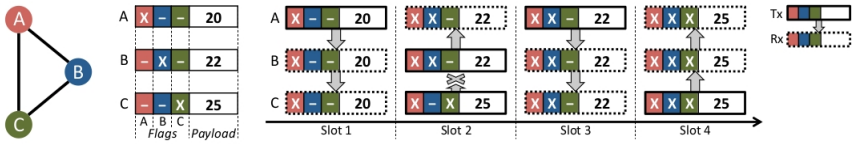

Figure 2.1: Chaos maximum aggregate scenario. In slot 1, A sends its value to B and C, who merge A's flag with their own and set the new max value. In slot 2, both B and C transmit their aggregated maximum value, but only B succeeds. Then in slot 3, node A has the maximum value of A and B and the corresponding flags set, which it transmits. In slot 4, node C transmits the aggregate with all flags set, after which all nodes have received the aggregated value of the whole network [23].

Chaos network operators can implement their own merge operators, but since the time slots should be short for low latency communication reasons, there are restrictions on a merge operator's complexity.

When a node receives a packet with all flags set, it aggregates the data from all nodes and goes into the completion phase and aggressively shares it, so that its neighbours likely receive the fully aggregated data as well. After a few time slots when this state is completed, the node locally ends the round. A Chaos round is successful if all nodes reach the identical conclusion by receiving the final aggregated value. According to measurements, this occurs with a high reliability in varying topologies [23].

There are cases when nodes reach the final state, whereas others do not, and the nodes do in fact not reach consensus. Although this occurs rarely, there are cases when the reliability needs to be increased even further. The probability of reaching consensus is increased by running Chaos multiple times and adding yet another flag for each node for each extra round, and then letting these extra flags signify an acknowledgement of finalizing the last round. This is how Two-Phase commit (2PC)[37] and Three-Phase commit (3PC)[37] are implemented in Chaos.

2.2 Hardware nodes

In the previous sections, we gave an overview of the Chaos protocol. Chaos can be implemented on any wireless system. However, Chaos was designed for use on low powered cheap computers. In the current version that this thesis is based on, Chaos is implemented on the lightweight operating system Contiki [9] to work with the TelosB [40] platform. Work is also in progress to port Chaos to the next generation OpenMote [30] platform. In this section, we describe these platforms' capabilities.

2.2.1 TelosB

The TelosB platform is a small, low powered sensor node with a basic radio and CPU. It has historically been a popular choice in academia for building WSNs, and its capabilities are well researched. This made it a good choice for an implementation of the Chaos protocol.

The TelosB [40] platform specification:

- 16-bit MSP430 from Texas Instruments, running at ~4 MHz CPU.
- 48 kB ROM, 10 kB RAM.
- Sensors for temperature, relative humidity, and light.
- A IEEE 802.15.4 compliant cc2420 radio unit [17]. This radio unit also has hardware accelerated AES (128 bit key length) computations in a number of modes, such as AES-CTR, AES-CBC and AES-CCM*.
- Power supply through 2xAA batteries or USB.

2.2.2 OpenMote 2538

The OpenMote 2538 [30] platform is the next generation of nodes to run Chaos. Chaos is currently not fully supported on this node, but it is relevant since it will be ported in the future. Compared to the TelosB platform, the OpenMote 2538 has much stronger computational resources, and in particular, has ECC hardware acceleration:

- 32-bit ARM M3 CPU, running at up to 32 MHz.
- 32 kB RAM, up to 512 kB flash.
- A IEEE 802.15.4 compliant CC2520-like radio unit.
- Hardware support for Advanced Encryption Standard (AES) (128/256 bit key lengths), ECC (128/256 bit key lengths), and Secure Hash Algorithm 2 (SHA2).

2.3 IEEE 802.15.4

The IEEE 802.15.4 radio standard [16] is designed to work well with low powered nodes and can be seen as the de-facto standard for WSNs. It includes rules for both the physical layer, such as packet size and modulation scheme, as well as the MAC layer. The standard describes security features that operate on the MAC layer, such as how to encrypt data and how to communicate security settings in the MAC header. Both the OpenMote and the TelosB have hardware support for these security mechanisms. The current version of Chaos on Contiki implements this standard to a large extent.

2.4 Contiki

Contiki [9] is a lightweight, open source operating system that is developed for use on hardware with restricted resources. This makes it suitable for an implementation of the Chaos protocol and for the TelosB and OpenMote platforms. Contiki

has support for simple, low memory overhead threading of applications through protothreads [10].

Contiki also has support for basic real time scheduling, as well as regular sequential non-preemptive process scheduling. In Contiki, threads run in two different modes: cooperative mode and preemptive mode.

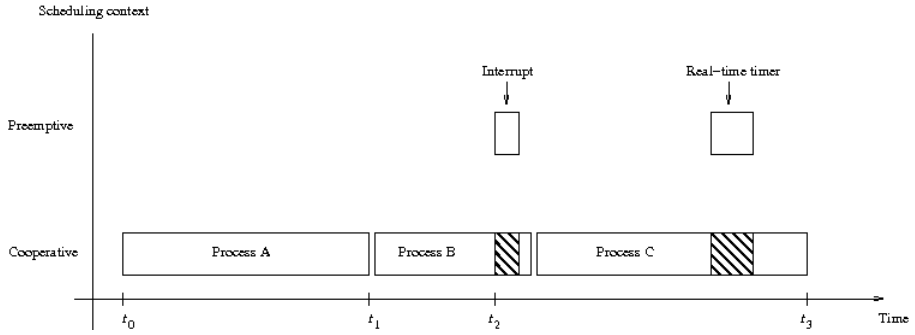

Figure 2.2: Process A, B, and C run in cooperative mode, and run in a sequence with respect to one another. A calls B which calls C. Preemptive code interrupts process B, which runs in cooperative mode, at time t_2. Control returns to process B when the interrupt routine is complete [3].

The main Chaos process runs in a preemptive mode and is called in a regular time interval for all nodes. Any computations that were done in cooperative mode are temporarily put on hold, until this timing critical Chaos thread completes. This is what allows nodes to wake up synchronously and turn on the radio and transceive simultaneously, while still doing useful computations in between rounds.

2.5 Cooja

Cooja is a network simulator well adapted for WSNs. It has support for the TelosB platform, which makes it relevant for this thesis. We have used Cooja to promptly test applications on top of the Chaos protocol, such as the key exchanges. The strength of Cooja is that it can simulate a large number of different topologies and node setups. Often, it is easier to set up a simulated environment than to set up actual nodes on a physical test bed. Cooja has support for different network models, and can simulate the diminishing signal strength as distance grows. However, Cooja lacks support for the hardware accelerated cryptographic operations that are vital for the functioning of the OpenMote and TelosB platforms.

2.6 Security

This section explains general computer security mechanisms, what they accomplish, and how they work. We then present specific security technologies that are central to this thesis.

A WSN is often an open and vulnerable system. Inherently, the radio is an open medium on which all parties in the vicinity can both transmit and receive, and the nodes themselves are potentially physically accessible, and consequently, vulnerable to manipulation. This opens up attack vectors for a malicious adversary that needs to be remedied for the WSNs functionality to be secure, and ultimately, correct.

One common way to achieve security in computer networks is to rely on cryptography. Through cryptography, developers can establish a multitude of important features, such as data confidentiality, data authenticity, and replay protection. In this section, we describe two different approaches to cryptography that facilitate this: asymmetric key cryptography and symmetric key cryptography, and their respective strengths and weaknesses, and how they generally complement each other.

We also describe in each category the basic building blocks that are available and suitable for the current implementation of Chaos on TelosB and OpenMote running Contiki.

2.6.1 Symmetric key cryptography

The core idea behind cryptography can be understood from studying its etymology. Cryptography comes from the Greek words *kryptos* - "secret", and *-graphy* - "writing". In other words, cryptography is the art of making a text unreadable to an unwanted third party. Symmetric key cryptography uses the same secret for both encryption and decryption.

$$Encrypt(Key, Message) \rightarrow Ciphertext \tag{2.1}$$

$$Decrypt(Key, Ciphertext) \rightarrow Message \tag{2.2}$$

2.1 and 2.2 showcase an abstract view of symmetric key cryptography. The encryption algorithm needs two parameters: the Key, and the "Message to encrypt". The encryption algorithm manipulates the Message together with the Key in a way that is hard to predict without prior knowledge of the correct Key. Consequently, the inverse/corresponding decryption algorithm is not possible without either knowing the Key or guessing it.

By sharing this secret between a number of parties, who consequently can correctly decrypt the message, one can establish a secure communication channel. These nodes can then confidentially send ciphertexts to each other, even over an open medium, without revealing the secret Message.

Relying solely on this encryption mechanism only enforces confidentiality of data. Like stated in the beginning of the section, there are other important security properties that must be enforced for a truly secure communication channel, such as data integrity. This can, however, be solved in a similar fashion of encryption, as exemplified in the next section.

The main weakness of symmetric key cryptography is that, ultimately, the shared secret has to be established over a secure communication channel. WIFI in PSK mode solves this by simply letting users input an administrated password into the supplicant device. Alternatively, one of strengths of asymmetric key cryptography is that it complements this weakness of symmetric key cryptography: an asymmetric key cryptography based scheme can establish a shared secret and secure channel, even over an open medium.

2.6.1.1 Advanced Encryption Standard

AES[4] is a popular and strong symmetric key algorithm. Both the TelosB and OpenMote have hardware support for AES, which makes it a good alternative for these nodes, especially for securing the link layer traffic.

AES encryption works by dividing the message into blocks of 128 bits. It then performs a set of permutations and substitutions together with the key on each block. AES does these operations in multiple rounds, and finally, each block is individually transformed into a ciphertext.

Using solely this approach has one severe weakness. Since the algorithm is deterministic, encrypting identical blocks found in different parts of the message results in identical encrypted ciphertext blocks. This causes an information leak which an adversary might be able to abuse. To combat this, AES is often run in a mode of operation that causes identical blocks to transform into differing cipher blocks. The relevant modes for this thesis are AES in Counter mode (AES-CTR), AES in Cipher Block Chaining (AES-CBC), and AES Counter with CBC-MAC (AES-CCM), since they are supported and configurable on the TelosB and OpenMote radio chips.

AES-CTR
AES-CTR solves the problem of recurring cleartext blocks by encrypting each block differently, depending on its position in the message.

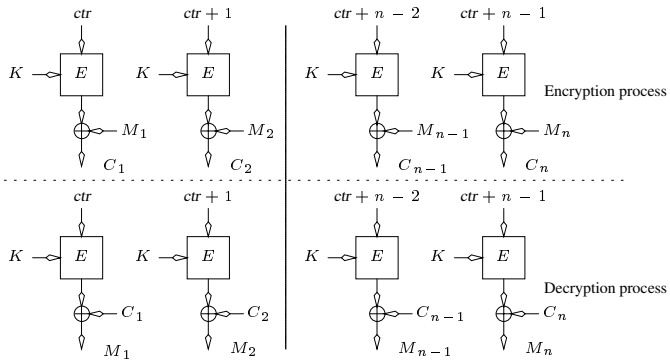

Figure 2.3: Showcases AES encryption and decryption in CTR mode. *Ctr* represents a unique counter [25].

AES-CBC

Similarly to AES-CTR, AES-CBC also solves the problem of recurring cleartext blocks, but AES-CBC does this by chaining all blocks together.

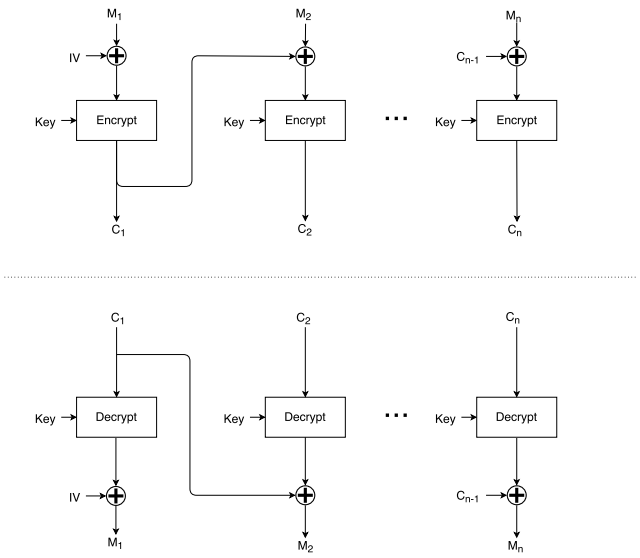

Figure 2.4: Showcases AES encryption and decryption in CBC mode [12]. M represents message to be encrypted, C represents the encrypted ciphertext, and IV represents the initialization vector.

AES-CBC-MAC
AES in Cipher Block Chaining Message Authentication Code (AES-CBC-MAC) is a special case of AES-CBC. AES-CBC-MAC does the same computations that AES-CBC does, but the difference is that it only saves the last block. This block is then a form of a cryptographic message digest that is appended to the message to ensure its integrity. Since a modification of any block ripples down to the last block, any change to the ciphertext is discovered by the decrypting node when doing the AES-CBC-MAC computation on the whole message and comparing it with the appended message digest in the message. If there is a difference, the receiver notices this and concludes that the message has been tampered with.

AES-CCM
AES-CCM is a combination of AES-CBC-MAC and AES-CTR. When encrypting with AES-CCM, it first runs AES-CBC-MAC and appends the digest to the end of the message. It then runs AES-CTR using the same key. By using AES-CCM, one enforces both data confidentiality and data integrity. This type of mode is called authenticated encryption.

2.6.2 Public Key Cryptography

PKC, also known as asymmetric key cryptography, differs from symmetric key cryptography in the sense that it utilizes two distinct, but mathematically related, cryptographic keys. One of the keys is made public and the other must be held private.

The revolutionary use cases of PKC when Whitfield Diffie and Martin Hellman introduced it in 1976 [5], was that it allowed for confidentially establishing secrets over an insecure channel, and that a message can be signed by one party's private key, and the message's validity and sender identity can be verified by any party that has the signer's public key.

These simple facts are used as building blocks for advanced and scalable infrastructures, such as the Public Key Infrastructure (PKI) found on the Internet. For one, a holder of both keys does not need a secure channel to establish a secure connection with other parties, but can rely on a scheme such as Elliptic Curve Diffie-Hellman (ECDH) to establish new keys.

The general disadvantages of PKC compared to symmetric key cryptography, is that it is computationally expensive to encrypt and decrypt messages, and that the cryptographic keys need to be longer for a comparable level of security. To get around this, developers of security functionalities often rely on PKC solely to verify identities and establish symmetric keys so that the rest of the communication is secured by a cheaper symmetric key cryptography algorithm, such as AES.

2.6.2.1 Elliptic Curve Cryptography

ECC[27][19] is a PKC approach that has gained popularity in recent years. ECC requires shorter key lengths than alternatives, such as RSA, for a comparable security level [26]. This property makes it suitable for the memory restricted nodes commonly found in WSNs.

The core idea behind ECC is to use elliptical curves and rely on the trapdoor function principle. An elliptic curve is a mathematical function of the form:

$$y^2 = x^3 + ax + b \tag{2.3}$$

The curve parameters in equation 2.3, a and b, are publicly known for all nodes that need to do ECC operations, as well as the starting point G on the curve.

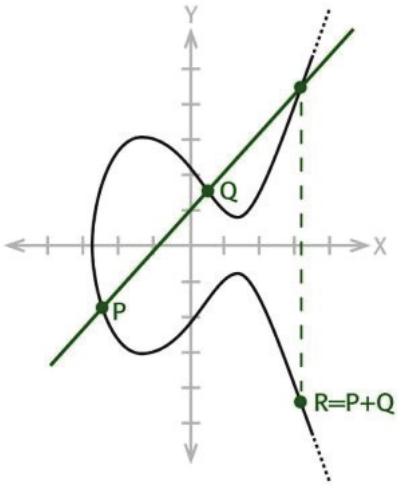

Figure 2.5: Addition of two points, P and Q, on an elliptic curve to get a third point, R [18].

When a point is added to itself, like in figure 2.5, one uses its derivative to find the next intersection. Multiplying a point with a number n is done by adding a point to itself and then adding that point with itself, and repeating like this $log_2(n)$ times.

In the context of ECC, a private key d is a random number, whereas the public key Q is a point on the elliptic curve. The public key is, more specifically, a shared point G on the curve multiplied by the private key, as can be seen in equation 2.4.

$$Q = d * G \tag{2.4}$$

Equation 2.4 showcases the elliptic curve operation required to build a public key from the private key. From this equation, one can also understand the trapdoor principle central to ECCs success. Given knowledge of the private key d and the point G, one can calculate the public key Q through "multiply and add", which has worst case complexity $O(k)$, where k are the number of bits in the key. An attacker wanting to compute d based on Q and G faces the elliptic curve discrete logarithm problem, for which there is no known efficient algorithm. Given that the curve is secure, reversing a public key to find the private key is impractical.

Users can establish private and public keys through these ECC operations, but need dedicated algorithms for more advanced purposes than that, such as ECDH for establishing shared secrets, and Elliptic Curve Digital Signature Algorithm (ECDSA) for authentication and building trust.

2.6.2.2 Elliptic Curve Diffie Hellman Key Exchange

ECDH is a way to establish a shared secret between two parties through their respective public and private keys. In principle, it is similar to the first published version of Diffie Hellman [5], but ECDH works in the context of elliptic curves. The

core principles behind ECDH are exemplified by the following scenario where two parties, A and B, establish a shared secret:

- A: generate d_a, derive $Q_a = d_a * G$
- A $\rightarrow$ B: Q_a
- B: generate d_b, derive $Q_b = d_b * G$, and calculate shared secret $= d_b * Q_a$
- B $\rightarrow$ A: Q_b
- A: calculate shared secret $= d_a * Q_b$

$$A's\ Secret = d_a * Q_b = d_a * d_b * G = d_b * d_a * G = d_b * Q_a = B's\ Secret \tag{2.5}$$

Equation 2.5 clarifies how the two parties agree on the same secret, while not allowing a third party to decipher the secret.

3
Related Work

In this section, we discuss research that works on solving similar problems as ours. We show the most relevant solutions and explain why we have not opted to use them.

3.1 Background

A good security solution depends on the application, as well as the underlying communication protocol, and security schemes often have tradeoffs between complexity, overhead, and adaptability. Consequently, there is much research in the area.

Padmavathi et al.[31], Wang et al.[42], and Pathan et al.[32] give a broad security overview for WSNs. Padmavathi et al.[31] also present an extensive model for possible attack types.

Similar to security, fault tolerance aspects, such as *byzantine fault tolerance*[22], are related to a system's correctness. A *byzantine fault* is when a node transmit differing data depending on the receiving node, causing uncertainty of what it said. This kind of problem is, however, not solved through cryptography, but instead through overhead in communication.

Because of the limited resources in wireless sensor networks, traditional approaches typically rely on pre-shared symmetric keys at the expense of flexibility. PKC, however, is flexible, dynamic, and scalable, but also computationally heavy, and time consuming [26].

It is likely that as the number of nodes become prevalent in the future, combined with the increase in memory size and computational power of each node, scalable solutions, such as those provided by PKC schemes, will be increasingly relevant, juxtaposed to pure symmetric key based schemes.

3.2 Symmetric Key Based Establishment Schemes

A simple solution for establishing shared trust in between nodes is to rely on a single network-wide shared key for all cryptographic operations. Such a solution has low overhead and is simple to implement, but has no resiliency against node capture, which could compromise the security of the whole network. Therefore,

researchers have proposed schemes with increased resiliency, such as a pair wise pre-distributed key scheme where each node shares a key with all other nodes in the network (n-1 keys are stored on each node, where n is the number of nodes in the network). However, this naive scheme requires much memory for key storage and is not scalable as node count increases. Not all nodes are necessarily neighbours, and links between them may not be directly possible, which makes the storage space for these keys wasted. Blom[2] discusses an improvement to the naive scheme, by storing keying material on each node, so that nodes themselves can establish secret keys with their respective neighbours.

Du et al.[8] also improve upon the naive pair wise pre-distributed scheme by explicitly defining the resiliency as a security parameter, λ, which denotes the number of nodes that can be compromised while the rest of the network is still secure. Increasing λ increases the overhead, while improving the security.

However, a scheme that relies on pair wise keys has disadvantages when applied to Chaos. Chaos is meant to function with low latency coupled with high reliability, but pair wise schemes lower this since they work on a point-to-point basis, whereas Chaos thrives upon broadcasts. Establishing pair wise keys would also be troublesome, since Chaos inherently relies on all-to-all communication.

3.2.1 Probabilistic Key Establishment

Eschenauer et al.[13] describe a novel approach for establishing keys among a set of nodes. They build a large pool of keys from which all nodes draw a subset of keys. When deployed, these nodes try to form a network by connecting to neighbouring nodes with shared keys, and then use this to establish secure communication links between their neighbours. The size of the key pool can be adapted according to the network so that neighbouring nodes likely share a key, and a fully connected graph is likely to occur.

Du et al.[7] show improvements to the previous scheme, based on deployment topology knowledge, and by selecting keys with a higher probability to match between neighbours. This provides resiliency against node capture, since each node only has a subset of all keys, and this subset of keys can be deleted from all other nodes. The secure nodes can then establish new session keys with the non-compromised keys, which means the compromised node will be excluded from the network.

These schemes can work with group keys. New nodes can join the network if there is an overlap in sets of keys between the network and the joining node. If there is, they can establish a secure connection and transmit the group key. This scheme might be good for Chaos as it is fast and adaptable, as well as potentially resilient to node capture, but if there is no overlap of key subsets, a joining node can not join the network.

Adaptable Pairwise Key Establishment Scheme (APKES)[21] is a symmetric key scheme that is used for establishing session keys between neighbouring nodes, and

relies on a pre-shared master key. When the master key is drawn from a random subset of pre-distributed keys, the scheme can provide good resiliency against node capture. The scheme may be used for establishing unique session keys between each node-to-node link, and it can also be used to establish group keys.

Adaptable Key Establishment Scheme (AKES)[20] improves upon APKES and solves the problems of handling anti-replay data on rebooted nodes, as well as node mobility. Additionally, AKES handles reboots without storing data in non-volatile memory. This also solves the problem that arises when nodes have been communicating with the same key for a long time, and the replay mechanism (the frame counter) starts repeating.

These schemes can work with group keys, which is preferable for Chaos, by letting a joining node establish a secure connection with an already existing Chaos network if it shares the joining node's subset of keys overlaps with the subset of keys found on the network. This approach could be good for Chaos as it is fast and adaptable, as well as potentially resilient to node capture, but if there is no overlap of key subsets, a joining node can not join the network.

3.3 Public Key Based Establishment Schemes

Historically, researchers have considered asymmetric cryptographic schemes as too computationally heavy for practical use in WSNs. Gura et al.[15] and Liu et al.[26] show that PKC schemes are in fact viable, even on nodes with moderate computational resources, such as the TelosB platform. It is also reasonable to assume that sensor nodes will only become more powerful in the future, and therefore, an adaptable PKC scheme Chaos is both usable today, as well as prove scalable in the future.

There exists PKC schemes, such as RSA, that easily handles encryption and decryption through the public and private keys themselves. ECC works in a different manner, and encryption with the public key itself is not directly possible in the same manner. In ECC, a shared secret should be established through a key establishment protocol, such as ECDH.

Yow et al.[44] introduce a lightweight mutual authentication and key exchange protocol, based on ECC, but with lower run time than comparable schemes. However, it relies on node power inequalities and puts the majority of heavy computations on a powerful server, which is not necessarily found on a Chaos network.

There have been research on porting security protocols commonly found on the internet to work with IoT, such as Lithe [36], which integrates Datagram Transport Layer Security (DTLS) and Constrained Application Protocol (CoAP). Research like these are important for providing end-to-end security between nodes in the network, as well as for remote hosts. To accomplish this, Lithe compresses the headers, and has support for common cipher suits including ECDHE. Similarly, Raza et al.[35]

have published a lightweight Internet Key Exchange version 2 (IKEv2) port for use on sensor nodes, through which it is possible to establish end-to-end communication between nodes.

Porambage et al.[33] introduce a PKC scheme that relies on ECC to authenticate joining nodes and establish session keys between these and neighbours. This approach is similar to ours, but they utilize a Cluster Head (CH), which is a more powerful node that functions as a CA, and the underlying communication protocol is not necessarily an all-to-all primitive.

3.3.1 Discussion

One big disadvantage of PKC schemes is their large cost in terms of time and energy consumption. For instance, ECDSA signature verification takes a long time [26]. However, in a WSN, joining nodes might be rare, and when mainly used for joining nodes, the overhead is hopefully small when averaged over time. The heavy join computation also opens up for Denial of Service (DoS) attacks against the verifier, in the case of a fake node sending fake messages that the verifier has to compute. Research has shown that there are mechanisms available to mitigate this type of attack. Arazi et al.[1] exemplify this by instead of using ECDSA, which has a long verification computational requirement, it relies on RSA whose verification algorithm is much faster. Dong et al.[6] also mitigate the DoS attack, but instead rely on pre-shared group keys to act as a first hand filter against the non-valid signatures.

4
Design

In this chapter, we specify the security requirements and functionalities that Chaos should provide. We also describe and discuss our design choices, and our methodology for accomplishing these.

Our design approach is split into two parts:

- The first part deals solely with symmetric key cryptography and our approach to securing regular link layer network traffic, as well as a simple way to establish and handle symmetric keys.
- The second part deals with designing and implementing a PKC scheme that solves the disadvantages and weaknesses of the symmetric key cryptographic scheme. Our main focus here is to enable new nodes to join an existing Chaos network without an "a priori" shared secret, as well as solving the problem of a leaked network key.

4.1 Chaos Security Requirements

Chaos needs the option to secure all radio packets. First and foremost, all data sent on the network must fulfil data confidentiality, data authenticity, and replay protection, which the IEEE 802.15.4 standard ensures [16]. However, the IEEE 802.15.4 standard states no built-in way of sharing cryptographic keys, as this must be taken care of on an upper abstraction layer.

One important principle in Chaos is that all nodes should be able to read every incoming packet in order to successfully apply a relevant merge operation. Otherwise, Chaos's performance deteriorates heavily, which we must avoid. Therefore all Chaos nodes need to at least share keys with their respective neighbors.

4.2 Delimitations

Chaos nodes are potentially physically accessible, which opens up for a multitude of side-channel attacks, such as tampering of sensors, power consumption inspection, computational delay measurement, radio jamming, or simply reading the key when it is set on the radio. If a node is captured, more attack vectors are available for an attacker, and the current key may leak. Dealing with side-channel attacks and

physical attacks are outside the scope of this thesis, however, we try to make our solution adaptable enough for future research to solve issues such as these.

4.3 Link Layer Security

The link layer security corresponds to all security related mechanisms that are applied on a frame to frame basis. It encompasses and ensures the correct transmissions on each individual Chaos time slot.

4.3.1 Network-Wide Link Layer Key

In this section, we describe our approach to sharing link layer keys among Chaos nodes. We first discuss how we bootstrap the security, and then shortly describe our application for updating the link layer key on the network. This corresponds to the first part mentioned in this chapter's introduction.

Due to the all-to-all communication properties of Chaos, it is suitable to use a network-wide shared cryptographic key. This means that all nodes in the network use the same symmetrical key for encryption, decryption, and message authentication. The key is distributed on the network by hardcoding it into each node during compilation.

Using a shared key like this means lower overhead than other symmetric key schemes, such as the pairwise key scheme mentioned in Chapter 3, and it is also easy to implement. A key known to all nodes in the network also aligns well with Chaos's all-to-all communication, and works with a dynamic network topology, such as new nodes joining, radio signal strength changes, and mobile nodes.

However, it provides no resiliency if the key leaks, after which all network communication based on it is compromised. Despite this, using a network-wide security key is an acceptable approach, due to its low memory overhead and high performance, and since we take further precautions to mitigate the consequences of node compromise.

4.3.1.1 Establishing New Symmetric Keys

This network key is used by all Chaos nodes for securing regular radio traffic. The IEEE 802.15.4 standard specifies that a 16 octet nonce should be used for the AES operations.

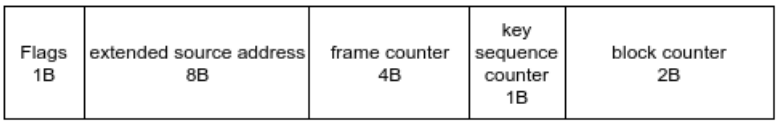

Figure 4.1: The structure of the nonce used in the IEEE 802.15.4 AES operations.

A unique key and nonce combination for each AES operation is a requirement for replay protection. As we show in Chapter 5, these nonces may start repeating within a network's operation time, and we must therefore confront this to ensure the full link layer that we mentioned in our requirements. Our solution is to build a key exchange application, through which all Chaos nodes agree on a new key to use before all nonce values for the current key are consumed and start repeating.

When establishing a new network key, all nodes should agree on using the identical key simultaneously. Otherwise, they will not be able to encrypt, decrypt and authenticate packets correctly, and will practically be excluded from the network. For establishing new keys, we require that the nodes reach consensus with a high reliability. We let the Chaos initiator create a new network key, and then utilize a 2PC consensus protocol through which it disseminates the new key to all the nodes in the network. The existing link layer security ensures that the new key is exchanged securely. The solution provides a good tradeoff between messages sent and a high reliability that all nodes reach the same decision of either changing keys, or not changing keys.

Establishing new keys has other advantages. For instance, if the key is leaked at one point in time, only traffic based on this key is compromised, whereas old traffic encrypted with an older key will not necessarily be decipherable by an attacker.

4.3.2 Problems With Joining Nodes

The strengths of our approach for establishing new symmetric keys are that it is simple to implement, and it has fast and cost efficient performance. Its main weakness, though, is that it relies on the link layer security secured through a pre-shared network key, and if this network key is not known, there is no way to decipher the new key. By itself, it can not expand the secured network, but at best preserve the existing one. Nodes may fail in a round while others succeed, after which a subset of nodes update the key whereas others do not, and they are thereafter unable to communicate.

It also breaks down if the current network key leaks, since then the link layer security is broken, and all future exchanges are compromised.

We want to solve the aforementioned symmetric key scheme's problems and add the following features:

- Enabling a new node to join an existing Chaos network without prior knowledge of the current network key.
- Securely establishing new network keys after network key compromise.
- Excluding compromised network nodes from the network and all subsequent network communication.

4.4 Elliptic Curve Join - EC-Join

In this section, we describe our application that enables new nodes to join the network without prior knowledge of the network key. PKC has ways of establishing new keys, revoking nodes' rights to the network, and enabling new nodes to join the network. Our goal here is to create a system that serves as a foundation for how PKC can successfully be used in conjunction with the Chaos protocol.

Our focus in this section is to present a way for new nodes to join an existing Chaos network. Currently, there is already a join application that manages network nodes and their flags. However, the join application assumes that the communication is secured on the link layer, which joining nodes lack the ability to do due to not having the network key. Our approach is therefore to build a PKC scheme that allows network nodes to securely share the network key with a new node. When the new node has received the Chaos network key, it can finally join the network through the existing join application.

We expect that the PKC scheme will not be used often, which loosens the time constraints. We also consider that optimizations and more powerful nodes solve the current computational time problems in the future. However, we do require that it manages to solve the problems within a reasonable amount of time, even on the TelosB platform.

4.4.1 Application Requirements

A new node should be able to securely join a Chaos network without knowing the current network key. To facilitate this, we break down the problem into the following required parts:

- Mutual authentication:
  - The joining node should prove its identity to a holder of the network key.
  - The network key holder should prove that the network is the joining node's desired network.
- Secure exchange of network key:
  - A node in the network should send the network key to the new node without a third party being able to decipher it.
  - The exchanged network key should be authenticated, and a third party can not manipulate it without the joining node detecting it.

The ECC based secure join application is designed to run on all Chaos node, including the TelosB platform, which Liu et al.[26] demonstrated requires multiple seconds for common ECC operations. Chaos time slots are in the order of a few milliseconds [23], which means the ECC operations are too computationally heavy to be used on individual Chaos time slots for the TelosB. In fact, operations can potentially span over multiple Chaos round periods, as we show in Chapter 6. Consequently, we build our secure join on the application layer. This means nodes exchange information

required for the Secure Join using Chaos simply as a network primitive for Elliptic Curve Join (EC-Join).

4.4.2 Mutual Authentication and Key Exchange Application

We base our key exchange on ECDH, and add features compared to the basic version in Chapter 2. One core difference is that we rely on lightweight certificates to build a trust chain and achieve mutual authentication. All nodes that can establish a secure connection must share the public key of a trusted third party, which we hardcode into all Chaos nodes.

4.4.2.1 Application Overview and Protocol Message Flow

Figure 4.2 shows an overview of the messages sent in the key exchange application. This represents the core of the protocol that occurs when a joining supplicant and a network authenticator has coupled. Currently, we assume that the supplicant already has the authenticator's anti-replay counter, and Chapter 5 shows how this is done.

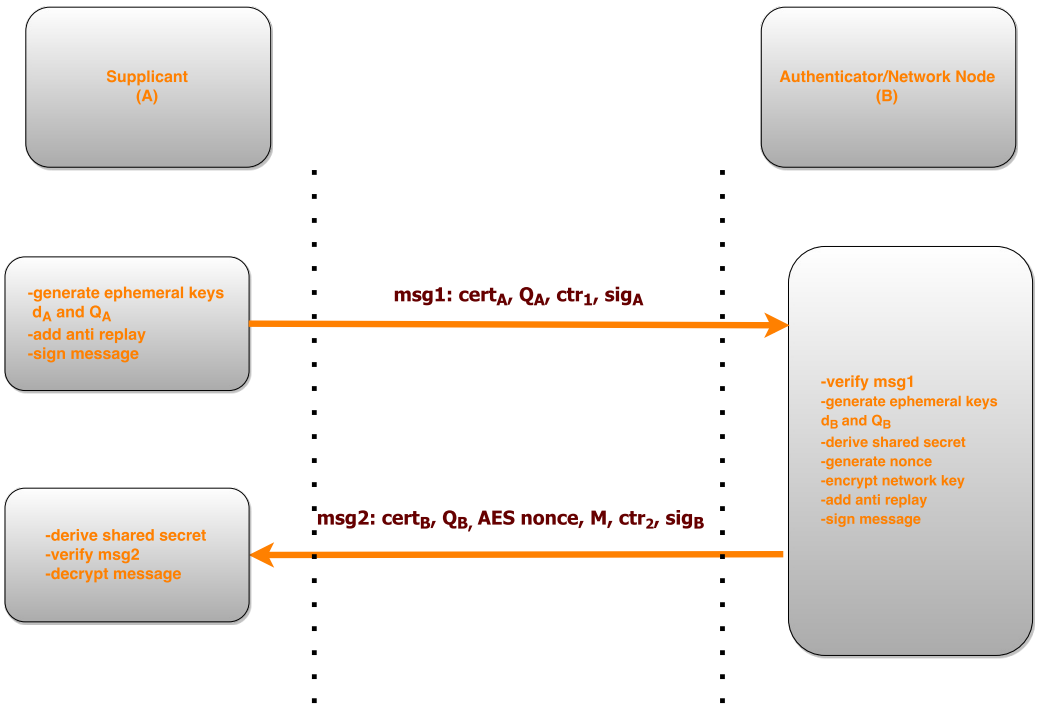

Figure 4.2: Visualizes the application's message flow. $Cert_A$ and $cert_B$ refer to A and B's public certificate. Q_A and Q_B represent A and B's public ephemeral keys. *Ctr* is a counter that is used to protect against replayed messages. Sig_A and Sig_B represent the nodes' respective ECDSA signature over the whole message. The authenticator sends the AES nonce together with M, the encrypted network key.

Mutual Authentication

At the end of the exchange, both nodes share each other's public certificate. These public certificates are signed by the trusted third party's ECDSA signature. The trusted third party's public key is hard coded into both nodes, and they both have the means to verify the identity of each respective certificate.

Both nodes must also verify that their session partner is the holder of the verified certificate, in order for the session to be authenticated. They accomplish this by ECDSA signing their respective message's hash digest with their certificate's corresponding private key. The receiver then verifies the message's appended signature. This means the signature in this case both serves for message integrity, and for authenticating that the trusted party has verified the other party's identity, and should therefore be trusted.

Secret Sharing
Both nodes share a common secret through the exchanged messages, just like ECDH as explained in Chapter 2. The authenticator uses this shared secret to derive an AES key, which it uses to encrypt the payload and then signing the message and sending it to the supplicant. In this case, the payload is the network key. When the supplicant receives message 2, it derives the same shared secret, and can decrypt the network key. An outsider can not decipher this payload, due to the ECDH protocols nature, as explained in Chapter 2.

Ephemeral Elliptic Curve Keys
We want to allow the network to handle compromised nodes. If a node is captured, that node's private key is possibly leaked. A leaked private key means that all corresponding ECDH sessions based on this private key are decipherable by the new holder, since the attacker who has saved the network traffic can perform that node's computations, including shared secret derivation used for encryption and decryption of payload. Consequently, if a captured node has ever used ECDH for EC-Join, then that agreed shared secret is compromised, as well as the network key that was encrypted based on the shared secret. This means all future link layer security is compromised starting from the first moment that node used ECDH to encrypt and send the network key.

A solution to this problem is to use ephemeral public and private keys for one key exchange and then discard them. This makes each individual key exchange session protected and indecipherable in the future. This property is called forward secrecy. There is also a performance tradeoff involved, since using ephemeral keys is heavier than each node's long term keys due to new keys having to be generated each session.

Anti-Replay Mechanism
One attack against the protocol is that an attacker resends valid messages from a valid ECDHE session. This type of attack would not allow the attacker to join the network, however, it would consume time and energy from the authenticator. We add an anti-replay mechanism to the ECDHE session that mitigates the consequences of fake messages.

counter = (ecdhe_session_counter || authenticator_id || supplicant_id)

The ecdhe_session_counter is a 32 bit value that is incremented after each use and stored on each node in the network. Each potential authenticator transmits it to pending supplicants in the pre-join rounds. Supplicant candidates save the anti-replay counter corresponding to the authenticator it couples with. The authenticator then only accepts incoming messages with the expected counter. The ecdhe_session_counter prevents messages from old sessions from being replayed, whereas the authenticator_id prevents valid messages from parallel sessions from being replayed to another authenticator.

4.4.3 Pre-Join Network Synchronization and Friendly Neighbour Finder

In Chapter 2, we describe how Chaos nodes synchronize to the network, both in terms of time and correct application scheduling the following rounds. New Chaos nodes must synchronize accordingly before joining the network. However, the network traffic is link layer encrypted, which means new nodes can not access the data, and therefore synchronize with the network. For this reason, we utilize a set of unencrypted, periodic, pre-join synchronization rounds, which allow new nodes to synchronize with the network before attempting the EC-Join. The pre-join is implemented as a Chaos application that is scheduled periodically.

Pre-join also serves for the new nodes to notify the Chaos network that they wish to join, which enables the network to schedule the secure key exchange and authentication application the following rounds. If no joining node is able to notify the network of its presence, the network schedules another pending application, such as the max-app example in Chapter 2, instead of the key exchange application.

The application's scheduling is configurable through the Chaos scheduler. A long delay between the application's rounds leads to a low overhead, but also means that joining nodes can not join as often, whereas a short delay means more nodes can join sooner, but at a higher cost for the network.

4.4.3.1 Coupling

Since ECDH secret sharing is point-to-point based, new nodes need to couple with one of the network nodes. There are different approaches to this, one of them is to let the network denote an authenticator node, with which all supplicants couple with. This works well due to Chaos's all-to-all communication, but it limits the number of simultaneous joining nodes to 1.

Another approach is to consider that all network nodes share the same network key, and have the same privilege and ability to share it. Supplicants can then couple with their closest neighbour, which minimizes latency and network load. It also means that all network nodes can, in theory, simultaneously authenticate one supplicant each. The tradeoff is that if all network nodes authenticate, there is likely

much interference between neighbouring links, which lowers the link reliability. We discuss solutions to this problem in Chapter 7.

4.5 Discussion

Although node compromise detection and its removal from the network are outside the scope of this thesis, we design our applications to be adaptable enough to solve these issues in future work.

4.5.1 Trust Infrastructure

One of our design choices is to remove the core trust element from being stored on the network, to avoid its compromise. In this case, the trusted third party's private key should not be stored on the network, but instead on a secured remote location.

There should, however, to be a link between this secure location and the Chaos network in order to facilitate dissemination of key revocation messages. The link ought to be either direct, through a permanent radio link to a few dedicated network nodes, or indirect through a mobile node which carries the trusted third party's signed message and connects to the network for a dissemination of the message.

4.5.2 Key Revocation

Revocation of nodes' rights can be accomplished by sending a revocation message containing the revoked nodes' public keys, signed by the trusted third party. The nodes on the network then verify the message, and add the keys in the message to their list of revoked public keys. If a node tries to connect to a network with a blacklisted key, the corresponding network node will detect this and abort the session.

This revocation message must be disseminated to the whole network, since all nodes in the network can establish a key exchange session with an incoming node. The revocation message must also be shared with all future joining nodes, since otherwise they run the risk of being fooled into joining a Chaos network run by one of the compromised nodes.

4.5.3 Leaked Network Key

A leaked network key means that all future network communication based on that key is compromised. For this reason, a new network key must be established. This can not be solved by our symmetric key exchange scheme presented earlier. A solution is to discard the current symmetric network key, and then let all network nodes perform an ECDHE session with an uncompromised node.

5
Implementation

This chapter describes our working solution for security functionalities. We start off by describing our methodology for implementing the common security operations used on each Chaos slot. We then show our symmetric key exchange implementation. Later on, we show the implementation of everything PKC related, and explain the ECC library's available building blocks.

5.1 Link Layer Security

The link layer security has large timing constraints, and to maximize Chaos's performance, we utilize hardware acceleration when available. The cc2420 radio unit found on the TelosB platform has hardware support for AES, as mentioned in Section 2. We implement the security features according to the IEEE 802.15.4 standard by extending the cc2420 Contiki driver.

5.1.1 Chaos Frame Counters

Since Chaos relies on all-to-all sharing, keeping track of individual frame counters from all neighbour nodes is not practical. Additionally, Chaos relies on a single shared network key, and consequently, the frame counter must be shared, because it is the combination of key - frame counter that must not be repeated.

Link layer frame counters in Chaos are set in the Chaos header. We make sure that the frame counter does not repeat randomly, through it being encoded as a combination of Chaos time slot and Chaos round number, as shown by listing 5.1.

Listing 5.1: Chaos Frame Counter

```
typedef union {
  uint32_t security_frame_counter;
  struct {
    uint8_t slot_number_frame_counter;
    uint8_t seq_number;
    uint16_t round_number;
  };
} security_frame_counter_t;
```

The seq_number variable is currently unused, and is available to be used to increase the maximum number of Chaos slots, or the number of Chaos rounds. If the extra byte is used to extend chaos_round when it overflows, then the Chaos frame counter practically supports 2^{24} Chaos rounds. If Chaos runs once every second, the frame counter will start repeating after approximately 6 months ($\frac{2^{24}}{60*60*24} \approx 194\ days$). This may not be enough for all applications, which is why we need to update the symmetric key before this occurs through the key update application.

5.1.2 AES Functionalities

The cc2420 radio unit has 3 types of AES interfaces: stand alone encryption, on-the-fly encryption on the TX buffer, and decryption on the RX buffer. We utilize the two latter types for our link layer security, as these natively handle the AES modes of operations discussed in Section 2.

To encrypt transmitted messages, we issue an STX_ON strobe on the radio, in conjunction with flags that represent which encryption configuration to use. To signal that the frame is security enabled, we utilize a flag in the Chaos header that is set when any link layer security setting is active.

Receiving nodes detect that the security flag is set in the Chaos header, and act according to this. They update the current AES nonce with the frame counter used for encrypting the packet, and issue an SRXDEC decryption strobe. This turns the decryption engine on to operate on the RX buffer, asynchronously. To avoid reading parts of the RX buffer before decryption is done, we only read if CC2420_FIFO_IS_1 pin is 1, which signals that there is at least one readily readable byte in the buffer.

If the radio security configuration includes MIC, the encryption engine automatically outputs the appropriate number of AES-CBC-MAC message digest bytes to the end of the packet before transmission. The in-line decryption also automatically computes the AES-CBC-MAC, and compares the locally computed digest with the appended MIC from the purported sender, and outputs either 0x00 if they match, or 0xFF if they differ. This is all done below the surface, and we only need to verify the MIC by comparing each frame's last byte, and discard it if when non-zero.

To prevent replayed packets, all Chaos nodes have a shared frame counter. Receiving nodes must decrypt with the frame's frame counter that was used in conjunction with encryption. After decryption, the receiving nodes then verify that the frame counter that was used to encrypt and decrypt the packet is valid and has not been used before with this key. If its invalid, they discard the packet.

5.1.2.1 Configurable Security Settings

In our implementation of the common security operations, we rely heavily on the TelosB platform's radio unit. As described in the Chapter 2, there are three available

AES modes of operation for the cc2420 radio, each with different properties.

Currently, security configurations are implicitly shared when nodes are compiled. Through the function cc2420_set_security() found in the cc2420 driver, Chaos network operators can decide appropriate security levels, based on their applications' needs and tradeoffs.

Security Level	Mode of operation	Meaning
0x00	No security	Data encryption OFF. Data authentication OFF.
0x01	AES-CBC-MAC-32	Data encryption OFF. Data authentication ON.
0x02	AES-CBC-MAC-64	Data encryption OFF. Data authentication ON.
0x03	AES-CBC-MAC-128	Data encryption OFF. Data authentication ON.
0x04	AES-CTR	Data encryption ON. Data authentication OFF.
0x05	AES-CCM-32	Data encryption ON. Data authentication ON.
0x06	AES-CCM-64	Data encryption ON. Data authentication ON.
0x07	AES-CCM-128	Data encryption ON. Data authentication ON.

Table 5.1: Our different security levels and their corresponding meaning, according to the 802.15.4 standard [16]. The number following the mode of operation in the second column describes the number of bits in the CBC-MAC digest. One simple way of interpreting the table, is that the higher security level, the higher the security is, which means AES-CCM-128 is the most secure mode due to it both utilizing encryption and having the largest possible cryptographic message digest.

There is a tradeoff associated with the security level. As we show in Chapter 6, each of these modes require differing amounts of time, both for encryption and decryption of packets, as well as packet size. A large message digest also correlates to a smaller maximum payload size on the radio.

When deciding which mode to use, an operator should know what he wants to accomplish with his application and what security level is absolutely required. Confidentiality is not always absolutely necessary, in which case AES-CBC-MAC might suffice.

It is also possible to solely use AES-CTR, that is, encryption with no authenticity. There is already a CRC check done automatically on the radio, so that these AES authenticity checks are not necessary if one only needs to check for corrupt messages due to radio interference. However, CRC is not sufficient protection against a malicious adversary, since the checksum operation can be calculated without a key. It is also common practice not to use an encryption scheme without cryptographic authenticity, since it does not actually protect against an adversary manipulating the packet. In such a case, the receivers can be certain that no unallowed third party has read the packet, but they can not know if the packet received is in fact the packet sent.

It is also possible to utilize dynamic security levels that varies depending on the application, but then further steps may be needed in order to avoid security downgrade attacks, and we implement no such feature.

5.2 Chaos Application and Scheduler Background

Chaos supports multiple running Chaos applications through the Chaos scheduler. We give a short background to understanding the Chaos scheduler here. Chaos applications are implemented as structs according to the C code:

```
typedef struct chaos_app{
  char* name;
  uint16_t slot_length;
  uint8_t max_slots;
  uint8_t requires_node_index;
  int (*is_pending)(const uint16_t round_count);
  void (*round_begin)(const uint16_t round_count, const
     uint8_t id);
  void (*round_begin_sniffer)(chaos_header_t* header);
  void (*round_end_sniffer)(const chaos_header_t* header);
} chaos_app_t;
```

A Contiki protothread preempts all nodes' current execution every Chaos period. The protothread runs the Chaos scheduler in which all nodes determine the current application to schedule, and the Chaos initiator reads all applications' corresponding round_begin() function to determine the application to schedule the following round, before calling the current application's round_begin() which starts Chaos radio communication.

Each round, the Chaos initiator sends out the current application ID, as well as the ID of the application to be scheduled the following round. The rest of the network adapts according to this, and all synchronized nodes therefore always run the same application.

5.3 Symmetric Key Update Application

We implement the symmetric key update mentioned in Chapter 4 using a consensus protocol for increased reliability. We utilize a 2PC library to increase the application's reliability. If an even higher reliability is required in the future, the library can be exchanged for another protocol, such as a 3PC.

The communication is already secured on the link layer, we can simply send the new key on the network. We do this by letting the Chaos leader generate a new network key through a random function, and then disseminate it to the network through two_pc_round_begin().

When all nodes agree to commit the new network key, they update the current Chaos key through the chaos_update_key(*key) function and set the new key on the radio through cc2420_set_key(*key). This erases the old network key. Since all nodes discard the network key after a key update, the old network traffic is not

readable anymore.

5.3.1 Chaos Frame Counter Reset

One important goal of our symmetric key exchange is that it should solve the problem of repeated frame counters per key. Since it is the unique combination of a frame counter and key that must not be repeated, changing to a new key re-validates the old frame counters. For this reason, we have two anti replay frame counters stored on each node, one representing the first used frame counter for this key, and the other the last frame counter used with this key. This handles frame counter wrapping around the maximum value. Incoming frames are invalidated if they fall between these. After a key update, we re-validate used frame counters by setting the first used frame counter equal to the last used frame counter.

5.3.2 Symmetric Key Update Application Scheduling

The Chaos network operator can configure key update application's periodicity to fit his requirements. It is possible that he may want to change the network key often for an increased network overhead, but also mitigate the risk of leaked key since old traffic is no longer decipherable.

We also add a mechanism that schedules the key update application automatically when required. We do this through the is_key_pending() function, which returns 1 if the difference between the last used frame counter and the first used frame counter is below a configurable threshold. The threshold provides a margin to successfully do the key update and works as a safeguard against key - frame counter reuse.

5.4 Elliptic Curve Cryptography Library

For ECC operations on the TelosB, we utilize a library developed by NIST. The library is a slightly modified version of ContikiECC, which is a port of TinyECC [26] [29]. The library is designed for use on embedded devices, and implements common ECC operations, such as generating private and public keys, ECDSA signing and verification, and ECDH operations for establishing a shared key between two parties. It also has lightweight certificates, which we use for authentication and establishing trust among nodes.

There are multiple curve configurations in the library, such as 128-bit, 160-bit, and 192 bit length ones. According to Stebila[38] it is advisable to use curves of at least 160-180 bits. This is configurable in the NIST library, and we use 160-bit curves.

5.4.1 Certificate Structure

The following code represents the public and private versions of a certificate as found in the NIST library [29]. The public certificate is sent on the network and contains the public key, issuer ID, and the issuer's signature, whereas the private certificate also holds the private key corresponding to the public key found in the public certificate. These certificates are condensed and lack fields commonly found in other types of certificates like x.509, such as time stamp. That means they are valid until an explicit revocation message is sent from the trusted third party, as discussed in Chapter 4.

```
/** A public certificate
 * for in-memory representation
 * note: base point is not necessary here,
 as it is shared by all other nodes */
typedef struct
{
        /** ECC pubkey */
        point_t pub;
        /** hash of the ECC pubkey of signing party */
        uint8_t issuer[SHA256_DIGEST_LENGTH];
        /** signature of the signing party (ECDSA signature)
            (1/2) */
        NN_DIGIT signature_r[NUMWORDS];
        /** signature of the signing party (ECDSA signature)
            (2/2) */
        NN_DIGIT signature_s[NUMWORDS];
        /** @} */
} s_pub_certificate;

/** A private certificate
 * for in-memory representation */
typedef struct
{
        /** public part of the certificate */
        s_pub_certificate pub_cert;
        /** private key */
        NN_DIGIT secret[NUMWORDS];
} s_certificate;
```

5.5 Chaos Pre-Join Synchronization Application

The pre-join synchronization application serves multiple purposes. It is crucial for letting non-network nodes synchronize to the network, which all non-initiator nodes already do automatically when they do not have an initiator. It also lets the non-network nodes notify the network that they wish to join, which they do by setting

the node_wants_to_join flag in the communicated payload, as seen in the struct below.

Since the non-network nodes' packets are invalid according to the network nodes, link layer encryption must be turned off. Nodes disable the link layer security before these rounds by setting a different security configuration through the function cc2420_set_security(), and by setting the security flag in the Chaos header to 0 in all transmitted packets.

```
typedef struct __attribute__((packed)) prejoin_t_struct {
  uint8_t dummy[APP_DUMMY_LEN];
  uint8_t node_wants_to_join:1;
  uint8_t has_network:1;
  uint8_t wants_to_couple:1;
  uint8_t id;
  uint32_t ecdhe_anti_replay_counter;
  uint8_t flags[];
} prejoin_t;
```

Through the pre-join application, all nodes can also find a set of their respective neighbours by saving the transmitter's ID of all received packets. Although this is not required for our pairwise key exchange application, it is a logical choice since it allows nodes to establish a connection with a single-hop neighbour. This approach taxes the network less than routing all traffic through the network, and enables us take a distributed, scalable approach with our ECDHE application.

Our neighbour discovery mechanism is simple and has room for improvements. For instance, it currently only reports the last visible neighbour, which might not be the neighbour with the best connection. Moreover, multiple other nodes might select the same neighbour as their couple, while other neighbours open for connection are left alone. However, due to time constraints, we do not improve this and instead focus on other areas.

5.6 Chaos ECDHE Application

Like other Chaos applications, the key exchange is divided into a Contiki thread, and a round_begin() function that is called by the Chaos scheduler. The logical distinction between them that we have made, is that round_begin() does the network communication such as calling the Chaos round and buffering arbitrarily large payloads, whereas the main process acts on the complete key exchange messages. Similarly to the pre-join synchronization application, we disable the link layer security before these rounds through cc2420_set_security(), and by setting the security flag in the Chaos header to 0 in all transmitted packets. After the application is completed, these are set to the previous configuration.

5.6.1 Contiki Process

The heavy ECC computations in the key exchange are done in a Contiki PROCESS_THREAD(). This thread is polled regularly by the ecdhe_round_begin() function when new network data arrives. The process is largely composed of three parts:

- The supplicant's first message creation.
- The authenticator's verification and processing of its supplicant's message, as well as the creation of the succeeding response message.
- The supplicant's verification and processing of the authenticator's message.

Both nodes in a session know their corresponding role, as well as the current sequence. We utilize two state variables for this purpose. One keeps track of the local computations it has done, whereas the other variable keeps track of the last agreed sequence. It is therefore simple for all nodes to decide the appropriate processing step.

We also utilize a time out mechanism after which a node quits the protocol, in order to prevent faulty states from remaining for too long.

5.6.2 Networking Library

The ECC public certificates' sizes in this library range from 112 bytes to 144 bytes, depending on key length. The maximum Chaos payload depends on the security level, however, the maximum physical packet size in IEEE 802.15.4 is 127 bytes, excluding the length field. That means not all certificates fit into a single frame. Therefore, we run multiple Chaos rounds for a single application message.

For simplicity, we do this on the application layer, i.e., we run multiple Chaos rounds to disseminate the current payload. The following C struct exemplifies the messages that are sent. The current round's disseminator copies the message into a temporary buffer of maximum length of ECJOIN_VALUE_LENGTH, which it sends on the network. After the Chaos round, the agreed network data is appropriately handled in the ecdhe_round_begin() function and copied into a temporary buffer, if both the authenticator and supplicant nodes have set their corresponding flags. This continues until the whole message is successfully disseminated.

```
typedef struct __attribute__((packed)) ecjoin_t_struct {
  uint8_t priority; //this round's sender
  uint8_t ecdhe_pre_join:1;
  uint8_t sequence_number:5;
  uint16_t offset;
  uint16_t total_size;
  uint8_t diss_size;
  uint8_t authenticator_id;
  uint8_t supplicant_id;
  uint8_t diss[ECJOIN_VALUE_LENGTH]; //disseminated value
```

```
  uint8_t flags[];
} ecjoin_t;

typedef struct __attribute__((packed)) ecjoin_t_local_struct
    {
  ecjoin_t ecjoin;
  uint8_t flags[FLAGS_ESTIMATE];
} ecjoin_t_local;
```

Earlier, we stated that the supplicant does not yet have a Chaos flag. We have circumvented this fact by hardcoding the supplicant's flag into one position and the authenticator's flag into another. Since both nodes know each other's ID, and we have this ID in the message, they know this message is exclusively for them to set the flags. Other, neighbouring nodes ignore these flags, since they see that they are not part of the session. This solution is appropriate for pre-join ECDHE sessions, and when we apply the ECDHE application on the existing network in case of a compromised network key, the Chaos nodes can use their regular flags.

5.6.2.1 Enabling Multiple Parallel Sessions

We also implement support for multiple parallel ECDHE sessions in this library. All supplicants instigate an ECDHE session with one of their neighbouring network authenticators by setting these two nodes' IDs in the ecjoin_t struct, and then sending the first ECDHE message as described earlier. All non-partaking neighbouring nodes abstain from retransmitting incoming frames belonging to sessions which they are not part of, which makes the traffic local and reduces interference.

Multiple supplicants may try to couple with the same authenticator, even though each node should only partake in one ECDHE session at a time. We solve this problem by letting the authenticator couple with the supplicant of the first received frame. The authenticator then combines the flags with the supplicant's flags, sets the authenticator_id and supplicant_id, and sends out this modified frame. When the other supplicants receive the frame from their proposed authenticator, they infer that the authenticator has already coupled with another supplicant, and therefore quit the session and stop transmitting data, in order to mitigate radio interference. The supplicants who backed off can try to couple with another neighbouring authenticator, however, we do not implement this optimization. Instead they back off and wait until after the next pre-join.

This is also how we prevent multiple ECDHE sessions per node: each authenticator and supplicant have their session state and coupling, and ignore frames from other sessions.

5.7 Scheduling Overview

We schedule the pre-join application periodically through its corresponding is_pending() function. If a new node wants to join, it notifies a neighbour, which spreads this on the network. The Chaos leader then schedules the ECDHE application where new nodes couple with neighbouring Chaos network nodes for the session, and send their corresponding first ECDHE message.

The key exchange application is scheduled until the next pre-join synchronization round, where current supplicants and authenticators, as well as new nodes can notify the network of their need to schedule more key exchange rounds, in order to complete an exchange. This continues until no more node notifies the network to schedule the key exchange application. When this occurs, another application is scheduled, such as the max application example mentioned in Chapter 2.

This scenario functions as a proof of concept and needs improvements before use in real life scenarios, such as limiting the number of scheduled key exchange rounds between pre-join rounds, and using a more robust way for nodes that has the network key, but have not yet joined, to notify the network to schedule a join round immediately.

6
Evaluation

In this chapter we measure the cost of our implementations. First, we evaluate the link layer related security in order to answer our RQ1 in Chapter 1: Can we use link layer security without detrimentally affecting Chaos's low latency?

Afterwards, we measure the cost of the key update and the ECDHE based key exchange and mutual authentication application. We answer our second RQ of whether this approach is good for Chaos on the TelosB and OpenMote platforms. Here we consider the major relevant constraints that operators face in the field of WSNs. This will show how usable our implementations are, as well as pinpointing the most urgent room for improvements.

Finally, we analyse the security aspects of our solutions and discuss their potential vulnerabilities, both with regards to our approach and design, as well as the current implementation.

6.1 Cost of Link Layer Security Operations

In this section, we evaluate and present the overhead cost of link layer related security configurations, such as running Chaos with encryption or not, and varying authentication message lengths, as described in Chapter 5. Although the cc2420 radio sheet presents example costs for encrypting and decrypting packets, this is incomplete and does not show the total cost for transmitting and receiving packets. Since Chaos network operators may have differing requirements with regards to security and performance, we chose to present costs for all security levels.

6.1.1 Cost of AES Operations

To encrypt packets, we issue a STXON strobe on the cc2420 radio, after which the radio automatically encrypts and authenticates all packets to-be-transmitted on the fly. For this reason, we opt to measure the total time it takes for transmission function to complete. We measure this by first measuring the time it takes for the NETSTACK_RADIO_fast_send() function to complete with different security settings, through the DCO_NOW() C preprocessor macro, which calls the high precision timer on the TelosB platform.

We also measure the required time for decryption on different security settings, however, decryption is issued manually before reading each packet through a SRXDEC

strobe. When the SRXDEC decrypt command strobe is sent, the radio sets FIFO and FIFOP pins to 0 to signal that no byte should be read. When a byte is ready to be read, the radio sets FIFO to 1 to signal this, which means that we poll the pin before reading each byte.

Security Level	Mode of operation	Packet Size (static payload size)	Time to Decrypt and Read	Time to Transmit
0x00	No security	108 Bytes	$3001\mu s$	$4051.5\mu s$
0x01	AES-CBC-MAC-32	115 Bytes	$2988.75\mu s$	$4277.5\mu s$
0x02	AES-CBC-MAC-64	119 Bytes	$3103.75\mu s$	$4414.75\mu s$
0x03	AES-CBC-MAC-128	127 Bytes	$3333.75\mu s$	$4675.5\mu s$
0x04	AES-CTR	111 Bytes	$2873.75\mu s$	$4143.5\mu s$
0x05	AES-CCM-32	115 Bytes	$3340\mu s$	$4274.25\mu s$
0x06	AES-CCM-64	119 Bytes	$3450.5\mu s$	$4423.75\mu s$
0x07	AES-CCM-128	127 Bytes	$3682.75\mu s$	$4686.5\mu s$

Table 6.1: Timings for transmitting and receiving packets with different security configurations

The results show that the security level does affect the required time on the radio, but when packet size is normalized, it is clear that the time to transmit is greater compared to the security operations' required time. The overhead in terms of packet size is 16 bytes Message Integrity Code (MIC) and 3 bytes in the frame counter. With a radio transmission of 250 kb/s, this corresponds to a transmission delay of:

$$\frac{1}{250000} * 19 * 8 = 608\mu s \tag{6.1}$$

Due to the cc2420 radio's on fly encryption, there is little actual time delay involved as a consequence of turning on different security levels. A little over $600\mu s$ between no security and maximum security, whereas the average slot length is usually less than 10 ms. This means a reasonable overhead, and is doable unless heavy merge operations already push the limits.

We see that generally, it takes longer time to transmit than to decrypt/read a packet. Decrypting also shows unexpected results in that reading a packet without decryption takes longer time than to decrypt the packet and read. It may seem strange that it is faster to read a packet when first decrypting, however, these minor differences are likely due to differences in polling the radio for when the frame is ready to be read. We do this either by checking the FIFO pin is 1, which signals that first byte is ready to be read, or FIFOP pin is 1, signals that whole frame is done.

These measurements show that the link layer security has time overhead, although small. This leads to operators having to adapt the Chaos slot length per application, and increase it by at most 1 ms.

6.2 Symmetric Key Update Application

To measure the cost of the symmetric key update application, we use real hardware, since Cooja does not support TelosB's hardware accelerated security features. Our evaluation scenario consists of 4 TelosB motes connected to a USB hub for power supply. We then measure the time the application spends in the CPU and radio through the energest energy estimation module. The energest module is a pure software construction which measures the time between two measurements. It is therefore to be considered as an approximation. Figure 6.1 shows the average energy consumption for one round.

An energy estimate is computed from the measured time, and an estimated voltage and current according to the TelosB datasheet:
$Energy\ [mJ] = time[s] * I[mA] * Voltage[V]$

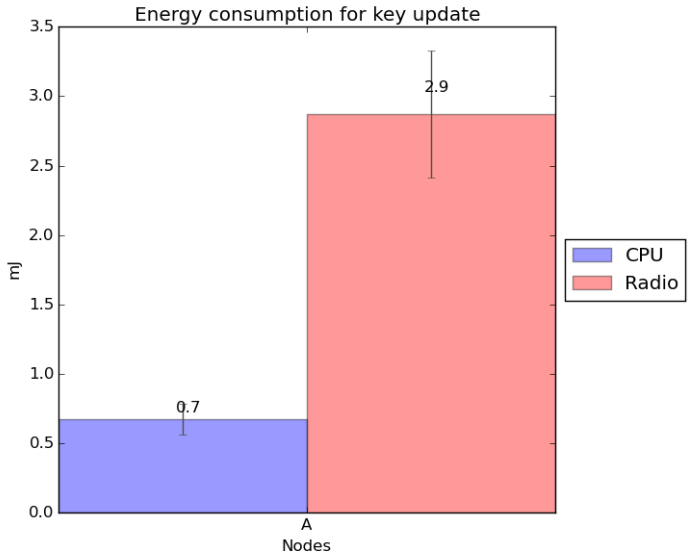

Figure 6.1: Energy cost average and standard deviation for a single key update

Compared to the ECDHE based key exchange in later sections, the symmetric key update application is light in terms of Chaos rounds and power consumption. Especially when considering that the key update application needs to run at most a few times every 2^{24} Chaos rounds. We also ran the scenario for a long while, and in our measurements, the motes succeeded in most rounds, and take thousands of rounds before a node gets excluded (round 5677 in one run and 38205 in another). When this occurs, that node is excluded from the network, and as long as it occurs rarely, it is fine, since it can go through the ECDHE based key exchange to rejoin the network.

We also measure the application's approximate memory overhead at compile time. We do this by measuring the compiled image of multiple Chaos applications, and then compare this with another compiled image without the key update application.

Metric	Total
ROM	1336B
RAM	106B

Table 6.2: ROM and RAM requirements of the symmetric key update application.

Given that the TelosB has 48kB of flash memory and 10kB of RAM, the symmetric key update application is cheap, even by WSN standards.

6.3 ECDHE Applications Related Operations Costs

Here we measure the cost of the operations required for our ECDHE application, determined on the TelosB platform. These values are later aggregated to determine the total cost of the ECDHE application for the TelosB. We then evaluate the corresponding operations on the OpenMote hardware accelerated engine to approximate the application's cost when Chaos is finally ported to it. We then have an approximation for its performance, and how well it likely will perform in the future.

We also include the approximate memory overhead of the dynamic join. It was measured similarly to how we measured the memory overhead of the key update application. This measurement also includes the pre-join synchronization application, since it is part of the same suite.

Metric	Total
ROM	18018B
RAM	2924B

Table 6.3: ROM and RAM requirements of the secure join applications.

These applications take up a large portion of the TelosB's memory. A Chaos network operator may need to prioritize away other important applications to fit into the TelosB. OpenMote's larger memory of 512kB makes these applications more viable.

6.3.1 TelosB

The NIST library we utilize supports the following curves: secp128r1, secp160k1, and secp192k1. We evaluate our the application according to the 160 bit secp160k1 curve, since it strikes a good balance of being secure [41] and having good performance. This is a compilation configuration and is changeable if a higher level of security is required in the future. Low level cryptographic operations optimizations are outside the scope of this thesis.

Metric	ECDSA Init	Cert Sign	Cert Verify	Host Ephemeral Key Gen	Network Ephemeral Key Gen	SHA256 (KDF)
Time	4.13 s	9.65 s	15.66 s	3.1 s	3.83 s	0.068 s

Table 6.4: Timings of different ECC operations for the TelosB platform.

Table 6.4 shows the time the most relevant ECC operations in the NIST library take.

6.3.2 OpenMote Hardware

Here we measure the operations found in our ECC based protocol on the OpenMote platform. The OpenMote supports multiple curves and curve sizes up to 256 bit. Due to implementation details, we evaluate the 256 bit curves. ECC operations with curves of lower order ought to be at least as fast as the 256 bit variant, and therefore, these values can be seen as the worst case.
Evaluation of ECC operations.

Metric	ECDSA Sign	ECDSA Verify	Host Ephemeral Keys Gen	Network Ephemeral Key Gen	SHA256 (KDF)
Time	612 ms	967 ms	343 ms	359 ms	<1 ms

Table 6.5: Timings of different ECC operations on the OpenMote platform.

Table 6.5 and 6.4 show the time the relevant ECC related operations take. We will later on use these to evaluate how well our protocol performs today, and estimate how well the OpenMote will perform in the future.

6.3.3 ECDHE Total Cost Comparison

In this section, we ascertain supplicant's and authenticator's total computational cost for the secure join applications. For the TelosB, for which our implementation already works, we measure this through the aforementioned energest module (we both use the individually measured operations and the total time measured here). Since the ECC operations are heavy, this measured total is close to adding up the individual costs in previous section as the measurement shows. For this reason, we can add up the evaluated ECC primitives for our ECDHE application to get an approximated total cost for the authenticator's and supplicant's computations even for the OpenMote.

The key exchange and mutual authentication protocol is largely split into the following steps:

- Stage 0: Both supplicant and authenticator generate ephemeral ECC keys.
- Stage 1: The supplicant builds its message and ECDSA signs it.
- Stage 2: The authenticator ECDSA verifies the supplicant's certificate and message ECDSA signature, derives a shared secret and symmetric key, AES encrypts payload, ECDSA signs the message and sends it.
- Stage 3: The supplicant verifies the authenticator's certificate and message signature, derives shared secret and symmetric key, and AES decrypts payload.

We utilize this knowledge for comparing the evaluated numbers for the TelosB platform with the approximated values for the OpenMote.

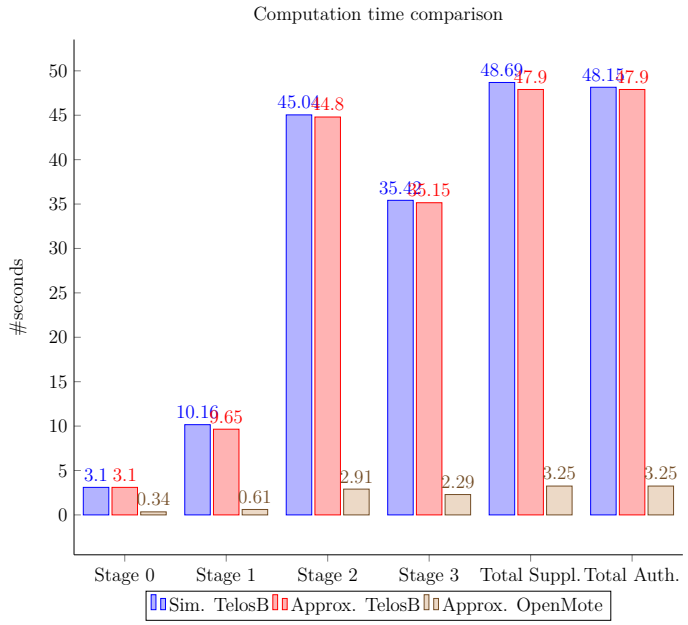

Figure 6.2: Computational time of OpenMote and TelosB

Figure 6.2 shows the time that the application spends in the different process stages for a successful key exchange. We approximate this by utilizing the measurements in Table 6.4 and 6.5. We also evaluate this by running the application and measuring the actual time it spends in the application in total, to see whether it differs much from the approximated values. For the OpenMote, we only relied on our earlier measurements for each individual ECDHE related heavy operation, and summed these according to the different stages, as mentioned in the stages bullet list. The OpenMote timings are approximations since Chaos does not currently run on it.

The platforms' energy consumption varies over time. The TelosB datasheet expects the CPU to draw 3V and 1.8 mA during computation. The OpenMote CPU draws an estimated 3.3V and 13 mA. For the TelosB radio, we assume a 18.8 mA and 3V according to the datasheet. Figure 6.3 then shows the corresponding energy cost for the time measurements we did in Figure 6.2, which allows us to compare OpenMote and TelosB in yet one aspect: energy efficiency.

As Figure 6.2 shows, both authenticators and supplicants have the approximate same total costs, although distributed differently in different stages. The OpenMote also proves to be much faster than the TelosB at performing these kinds of operations, and if the TelosB now has shown to be useful for this use case, the OpenMote is indeed a good improvement. The OpenMote is much faster than the TelosB. It also draws much more power. Figure 6.3 shows that this tradeoff favors the OpenMote well, since it is approximately twice as power efficient as the TelosB.

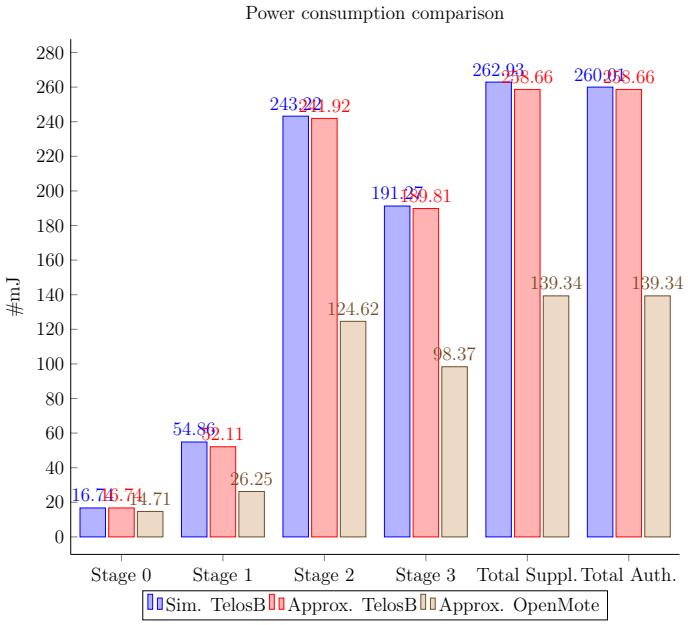

Figure 6.3: Describes the power consumption for OpenMote and TelosB.

6.4 Evaluating ECDHE Application Scenarios

In this section, we analyse and discuss the approach's cost on the network, both when no joining node is nearby, and also as the number of joining nodes increases. During these evaluations, we utilize a Chaos period of 1.5 seconds, since this strikes a good balance between fast completion of message transmission, and long enough time for computations to take place between the rounds. We also use a pre-join period of 26 Chaos rounds. These numbers affect the number of Chaos rounds that are run, and the corresponding radio traffic. These scheduling configurations are easily changed at compile time.

We make no claim that the configuration is optimal, and to get a good idea of the real future cost, we divide the cost into three different categories: radio, CPU energy spent in the ECDHE process, and CPU energy spent on network communication for the applications. The ECDHE process CPU ought to represent future iterations well, but the application CPU and radio traffic are likely to change based on the scheduling of the applications.

6.4.1 Testing Configurations

For evaluating energy, we utilize the energest module for measuring the time the nodes spend in different parts of the code. There is a bug in Chaos that makes the interrupt for scheduling applications preempt a long computation only once, but not again. This means nodes that perform the ECDHE application computations, which can take over half a minute, will not communicate on the network until the heavy ECC computations are done. When the Chaos initiator authenticates, the rest of the network therefore goes into association mode, and does busy wait and

listens on the radio for the initiator's synchronization messages.

To bypass this, we have made a workaround solution where the Chaos initiator only authenticates once and after that, not again. This makes the Chaos initiator node able to communicate regularly on the radio so that the rest of the network does not become desynchronized and go into association mode.

In our evaluations henceforth, we solely rely on Cooja simulations. This allows us to easily set up our testing environment and different network configurations. Although we relied on 2 network topologies. One containing 6 motes in total, and the other 32 motes.

Due to implementation problems with the scheduling, when measuring the cost of a joining session, we do not present the cost of an actual join, but instead we refer to the time and cost that it takes for a new node to successfully finish the ECDHE based mutual authentication and key exchange, and thereafter receive the correct network key. Our plan to solve this scheduling issue in the future is to let a node inform the network that it has joined by modifying the Chaos header and signing this with the agreed network key.

During the evaluation, we hardcode the trusted third party's private certificate into all nodes, and let each node generate their private and public certificates locally. This is a shortcut that will not be done on real deployment, due to severe security issues, and for this reason we do not include this part in our measurements. In the future, all nodes will receive their signed certificates from a trusted third party directly.

Additionally, we remove the anti-replay checks in our evaluation, as these fail due to synchronization problems that we have not fixed yet.

6.4.2 Cost of a Single Join Session

To start, we measure the energy required for the whole network when a single join session takes place. We measure this by making a single node join the network through a single authenticator, while the rest of the network idles in wait for connections. To bypass the association problems that occurs when the initiator authenticates, we do not let the Chaos initiator couple, but instead, the rest of the network motes are available for the ECDHE new node.

Figure 6.4 represents both the energy in terms of radio traffic for the three different types of motes in the exchange, as well as CPU cost in the process and applications. We choose to present the CPU cost of the process and the applications separately, since the application and radio costs will change based on scheduling optimizations (which will be needed in the future), whereas the process cost is unlikely to change. Non-partaking nodes' CPUs consume little energy compared to the session nodes. However, their radio is expensive, compared to the session motes. The radio cost is misleading at the moment, due to a temporary flaw in the implementation, which

causes motes to not to be preempted by the Chaos thread responsible for radio communication. This means session motes trade off expensive radio communication for a less expensive CPU calculations.

We note that in this measurement, we see almost the expected energy drawn in the process, although within a small margin of error.

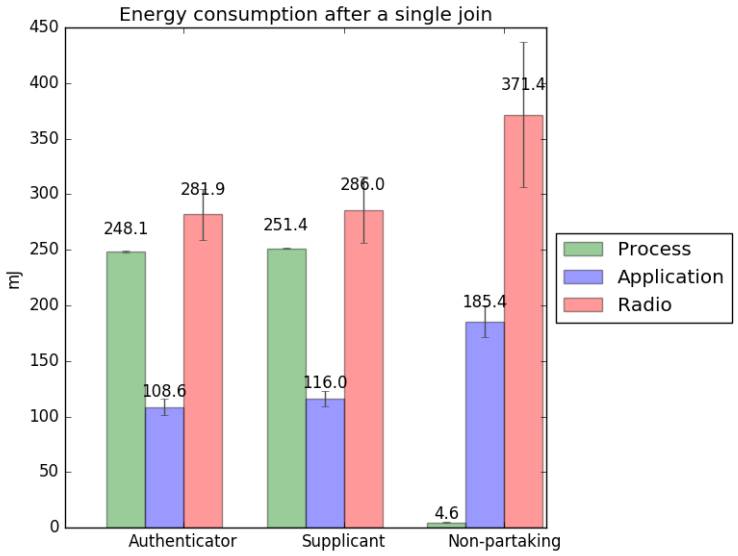

Figure 6.4: Energy cost average and standard deviation for a single join for different motes.

6.4.3 Cost of Building a New Network

One aspect of our secure join is the cost required to build a network from only the Chaos initiator node. Here, we run different scenarios to evaluate how much our application costs when used for building a new network from only a single Chaos initiator node. We do this by running a simulation for a single join, and another scenario is made up of 6 motes, and the last of 32 motes. This also shows how well our application scales with Chaos.

Figure 6.5 represents the total energy required to build a network from only a single initiator node. In both cases, the Chaos initiator only authenticates one node and not more, in order to not desynchronize the network. In future iterations, when our preemption works correctly, this circumvention should not be needed.

As Figure 6.5 shows, the largest energy consumption comes from the radio, and the applications' parts responsible for communication. The process where ECDHE operations are done is not as heavy. The average cost for building a network of 32 motes is greater than for building a network of 6 motes. For the radio and application, this seems to scale with the increased time of approximately 2.5 times that it takes to create the full network, but the process only increases by 1.5 times.

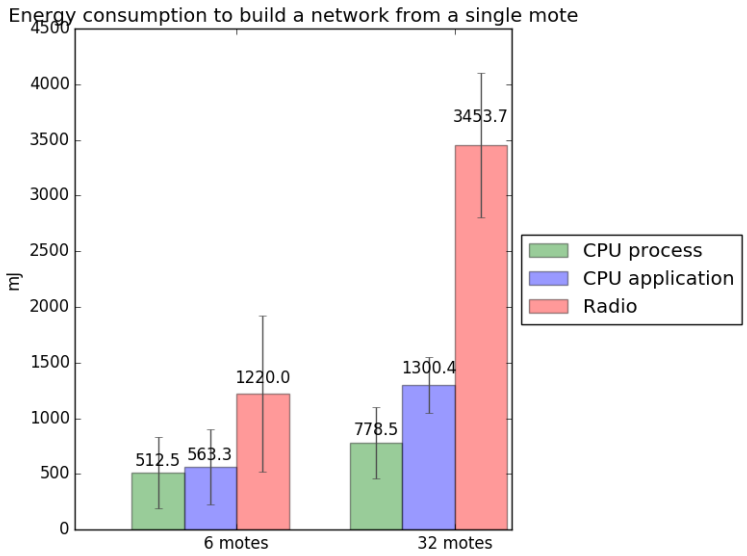

Figure 6.5: Total energy cost for building a network for each node on average.

Figure 6.6 represents a standard run for building a network of 32 motes. The simulation starts from only the Chaos initiator, and like our earlier evaluations, we only let the initiator authenticate once and then only communicate over radio to keep motes synchronized. This causes a slower start than we would otherwise expect.

This curve resembles an s-curve in that it takes off slowly due to the lack of authenticators, accelerates in the middle when most supplicants can quickly couple with a free authenticator fast, and at the end it slows down again due to few remaining supplicants. As we can observe, building a network of 6 motes takes an average of 7 minutes, whereas in double that time, a network of 25 motes can form, and in less that 18 minutes a full 32 mote network can form. This means our dynamic join seems to scale according to our expectations, and should work fine when building a network from scratch in a WSN environment.

6.5 Security Analysis

In this section, we analyse our solution's security strengths and weaknesses. We start by evaluating its design and inherent limitations, and later move on to our current unresolved implementation issues.

6.5.1 Design Limitations

- One inherent limitation of our design, is the choice of using a single shared network key, which provides no resiliency. This choice was done in order to preserve Chaos's low consensus latency, as described in Chapter 4. We have mitigated the consequences of a compromised network key through PKC, but

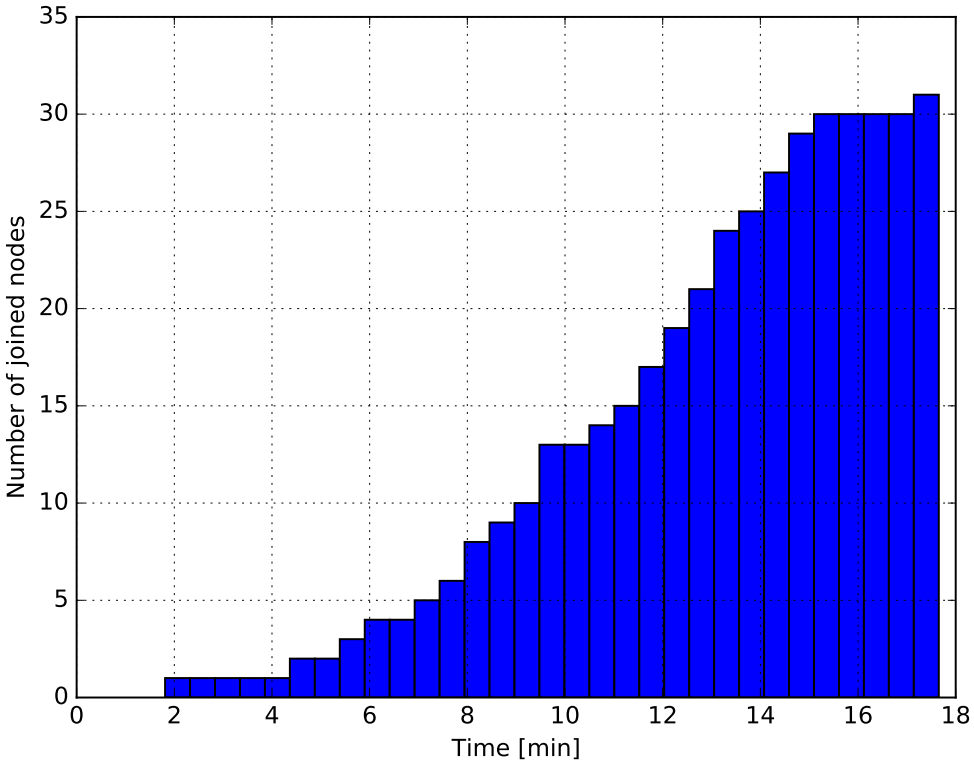

Figure 6.6: Time to create a network of 32 motes from 1 starting mote.

these countermeasures are only reasonable as long as a compromise is detected early on.

- The certificates are lightweight and lack commonly sought after fields, such as date fields, which means that they are valid until the trusted third party sends a revoke message. The choice to omit fields like this improves the performance and is acceptable due to the tight restrictions.
- The pre-join key exchange has no link layer security. This means an attacker can modify packets and send new ones at his discretion. The attacker can set the Chaos slot in the header to the maximum value so that all present nodes finish the round prematurely, or abuse our key exchange abortion after a faulty supplicant-authenticator coupling and send fake key exchange messages that cause an existing supplicant to decouple with its authenticator. The same is also true for the pre-join synchronization rounds, where an attacker can cause suppliants to couple with wrong neighbours.
- An adversary can cause the network to schedule unnecessary pre-join key exchange rounds. This means other rounds will not run, and instead the network nodes will waste energy on useless rounds where no real node wishes to join. A mitigation strategy to this is to let the network only schedule rarely, either by increasing the pre-join period, or dynamically changing the pre-join based on successful and failed joins.
- All authenticators listen for incoming supplicant connections during ECDHE

rounds, and have no limit to the number of faulty connections. This means an attacker can keep sending messages with correct replay counters but faulty ECDSA signatures, and since ECDSA verification consumes much power, it can cause the battery of network nodes to run out prematurely. In Chapter 3, we discuss mitigation strategies for this problem.

In Chapter 4, we discuss our applications' security requirements. We state that we wish for data confidentiality, data authenticity, and replay protection to be ensured. The last three of the above limitations are solely related to Chaos's degraded performance and does not touch our requirements. The first issue is a serious limitation that may compromise the security of the whole network. Although utilizing a network-wide key has serious security implications, it was a conscious choice in the design stages of this thesis and came forth as a consequence of the limited resources that we are dealing with. There are ways to mitigate its severity, such as early detection and exclusion through PKC.

We also consider that the issue of a leaked network key is not easily solved by relying even on scheme other than network-wide key, such as the pair based link key scheme we discuss in Chapter 3, since even one compromised key is detrimental, due to the nature of Chaos's all-to-all communication, and the security fix relies on detecting compromise and revoking the corresponding key. Then the issue is the same and solution is the same for our network-wide key scheme. This might be better solved with more powerful computers that rely more heavily on asymmetric key cryptography.

6.5.2 Implementation Limitations

List of known implementation vulnerabilities:

- We do not take into account that nodes may reboot, and what this does to a node. For instance, the ECDHE based key exchange anti-replay-counter is only stored in volatile memory, and is consequently deleted after a reboot. After reboot and the node has once again joined the network, an attacker can send old join messages to the rebooted node which will accept them as correct. This attack will not enable the attacker to join the network, but consumes more power and energy from the authenticator than scrambled fake data would. Storing the anti-replay counters in non-volatile memory would solve this issue, and since the TelosB has flash memory this should be easy.
- Currently, all nodes receive the trusted third party's private certificate for creating a per node private certificate, and then the trusted third party's private key is discarded, while the public part of it is kept. Although we delete the private part, a tenacious attacker might be able to obtain it nonetheless. This goes against our design goal of keeping the core trust element off the network, and must be fixed by hard coding each node's private and public certificates into the motes directly.
- Currently, we deactivate frame counters due to unresolved synchronization problems.

- We have not yet implemented any checks for buffer overflows in our mutual authentication and key exchange.
- Chaos does not currently discern between link layer authenticated and unauthenticated messages. This means network nodes synchronize to unsecured link layer messages coming from an unknown source, such as is the case in the pre-join rounds. The solution to this is to handle these messages differently than verified network messages, and not adjust variables such as slot number and round number if the packet containing these is not network authenticated. ECDSA signing each link layer frame is a viable solution due to Chaos's real time requirements and the high cost of ECC operations.
- Lastly, there is always the risk of security bugs.

These issues represent the current state of the applications. They have severe security implications if remained unsolved. Luckily, all but the last are straight forward to fix. As for the last issue, security bugs are always an uncertainty, and especially so in this case, since the code has not yet been properly peer-reviewed, or ran in any automated test frame work.

7
Conclusion

In this chapter, we conclude the thesis work and finish with a discussion regarding how well it fits into the current state in the area of WSNs, as well as theorize regarding future improvements.

7.1 Conclusion

In this thesis, we present a set of solutions for securing the communication of the Chaos network primitive. Although the solutions rely on pre-existing hardware support for link-layer security, the thesis's main contribution is how to make dynamic join work in conjunction with the low latency, high performance, all-to-all communications characteristics of the Chaos protocol.

7.2 Future Work

Our solution with PKC does not provide all security features mentioned in Chapter 4, but our solution is in our opinion adaptable enough and enables the possibility of developing further security features. Some areas of interest are:

- In this thesis, we were only able to rely on approximations for the OpenMote, since Chaos is not ported to it. Porting Chaos to the OpenMote would be of most interest.
- Key establishment between joining nodes and the existing network can be optimized. Our design and implementation serve as a proof of concept, and functionalities such as the splitting of large messages can be moved down from the application layer into a Chaos library so that it only takes one Chaos round to disseminate a large payload, instead of multiple.
- There is much room for improvements with regards to how nodes couple. At this moment, supplicants choose to couple with a random neighbouring authenticator, which means it is sometimes sub-optimal. There is room for research in letting authenticators select an optimal coupling configuration for the network, based on signal strength, number of present motes, etc, and then reach consensus regarding this in a low-latency manner.
- Currently, the scheduling has problems and can be improved much as well. For instance, the proper way to schedule a join after a new node has attained the network key from the network, is perhaps for it to modify a field in the Chaos header that signals this, and so that the Chaos initiator can then schedule the join application at an optimal time.

- The ECC library can also be improved upon either by optimizing reoccurring functions for the platform, or through hardware acceleration such as how it is done on the OpenMote.
- The nodes' certificates are unique per node. In the case of a detected node capture or similar case, it is possible to revoke this node's certificate and establishing new keys through PKC so that the node gets ostracized from the network. Future work in this area would be interesting.
- Further research in detecting corrupted/overtaken/compromised nodes/key leakage/tampering of nodes is interesting and detection of compromised security is needed for the countermeasures we discuss in this thesis to take place.
- Since our secure join and authentication network library works with multiple parallel sessions, there may be multiple neighbouring ECDHE sessions interfering with each other on the radio. Utilizing different radio communication channels, either static or dynamic, can mitigate the interference on each neighbouring session and improve link layer reliability and overall performance significantly in densely populated networks.

This list presents just a portion of the possibilities related to Chaos. Although the majority of the points above are only related to Chaos, areas such as compromise detection is interesting for all WSNs. Furthermore, solving and mitigating the implementation and design limitations mentioned in Chapter 6 is an important task that is well suited for future master's thesis students.

Bibliography

[1] Ortal Arazi, Hairong Qi, and Derek Rose. "A public key cryptographic method for denial of service mitigation in wireless sensor networks". In: *Sensor, Mesh and Ad Hoc Communications and Networks, 2007. SECON'07. 4th Annual IEEE Communications Society Conference on*. IEEE. 2007, pp. 51–59.

[2] Rolf Blom. "An optimal class of symmetric key generation systems". In: *Advances in cryptology*. Springer. 1984, pp. 335–338.

[3] Contiki. *Contiki Scheduling*. URL: `https://github.com/contiki-os/contiki/wiki/Processes`.

[4] Joan Daemen and Vincent Rijmen. *AES Proposal: Rijndael*. 1999.

[5] Whitfield Diffie and Martin E Hellman. "New directions in cryptography". In: *Information Theory, IEEE Transactions on* 22.6 (1976), pp. 644–654.

[6] Qi Dong, Donggang Liu, and Peng Ning. "Providing DoS resistance for signature-based broadcast authentication in sensor networks". In: *ACM Transactions on Embedded Computing Systems (TECS)* 12.3 (2013), p. 73.

[7] Wenliang Du et al. "A key management scheme for wireless sensor networks using deployment knowledge". In: *INFOCOM 2004. Twenty-third AnnualJoint conference of the IEEE computer and communications societies*. Vol. 1. IEEE. 2004.

[8] Wenliang Du et al. "A pairwise key predistribution scheme for wireless sensor networks". In: *ACM Transactions on Information and System Security (TISSEC)* 8.2 (2005), pp. 228–258.

[9] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. "Contiki-a lightweight and flexible operating system for tiny networked sensors". In: *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE. 2004, pp. 455–462.

[10] Adam Dunkels et al. "Protothreads: simplifying event-driven programming of memory-constrained embedded systems". In: *Proceedings of the 4th international conference on Embedded networked sensor systems*. Acm. 2006, pp. 29–42.

[11] Milica Pejanović Đurišić et al. "A survey of military applications of wireless sensor networks". In: *Embedded Computing (MECO), 2012 Mediterranean Conference on*. IEEE. 2012, pp. 196–199.

[12] Morris Dworkin. *Recommendation for block cipher modes of operation. methods and techniques*. Tech. rep. DTIC Document, 2001.

[13] Laurent Eschenauer and Virgil D Gligor. "A key-management scheme for distributed sensor networks". In: *Proceedings of the 9th ACM conference on Computer and communications security*. ACM. 2002, pp. 41–47.

[14] Federico Ferrari et al. "Efficient network flooding and time synchronization with Glossy". In: *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*. IEEE. 2011, pp. 73–84.

[15] Nils Gura et al. "Comparing elliptic curve cryptography and RSA on 8-bit CPUs". In: *Cryptographic hardware and embedded systems-CHES 2004*. Springer, 2004, pp. 119–132.

[16] IEEE. *IEEE 802.15.4*. URL: `https://standards.ieee.org/getieee802/download/802.15.4-2011.pdf`.

[17] Texas Instruments. *CC2420 Radio Datasheet*. URL: `https://inst.eecs.berkeley.edu/~cs150/Documents/CC2420.pdf`.

[18] Vivek Kapoor, Vivek Sonny Abraham, and Ramesh Singh. "Elliptic curve cryptography". In: *Ubiquity* 2008.May (2008), p. 7.

[19] Neal Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of computation* 48.177 (1987), pp. 203–209.

[20] Konrad-Felix Krentz and Christoph Meinel. "Handling Reboots and Mobility in 802.15. 4 Security". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM. 2015, pp. 121–130.

[21] Konrad-Felix Krentz, Hosnieh Rafiee, and Christoph Meinel. "6LoWPAN security: adding compromise resilience to the 802.15. 4 security sublayer". In: *Proceedings of the International Workshop on Adaptive Security*. ACM. 2013, p. 1.

[22] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[23] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling. "Chaos: Versatile and efficient all-to-all data sharing and in-network processing at scale". In: *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. ACM. 2013, p. 1.

[24] Krijn Leentvaar and Jan H Flint. "The capture effect in FM receivers". In: *Communications, IEEE Transactions on* 24.5 (1976), pp. 531–539.

[25] Helger Lipmaa, Phillip Rogaway, and David Wagner. "Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption". In: *National Institute of Standards and Technologies*. Citeseer, 2000.

[26] An Liu and Peng Ning. "TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks". In: *Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on*. IEEE. 2008, pp. 245–256.

[27] Victor S Miller. "Use of elliptic curves in cryptography". In: *Advances in Cryptology—CRYPTO'85 Proceedings*. Springer. 1985, pp. 417–426.

[28] Geoff Mulligan. "The 6LoWPAN architecture". In: *Proceedings of the 4th workshop on Embedded networked sensors*. ACM. 2007, pp. 78–82.

[29] NIST. *ECC Light Certificate*. URL: `https://github.com/nist-emntg/ecc-light-certificate`.

[30] OpenMote. *OpenMote 2538*. URL: `http://www.ti.com/lit/ds/symlink/cc2538.pdf`.

[31] Dr G Padmavathi, Mrs Shanmugapriya, et al. "A survey of attacks, security mechanisms and challenges in wireless sensor networks". In: *arXiv preprint arXiv:0909.0576* (2009).

[32] Al-Sakib Khan Pathan, Hyung-Woo Lee, and Choong Seon Hong. "Security in wireless sensor networks: issues and challenges". In: *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*. Vol. 2. IEEE. 2006, 6–pp.

[33] Pawani Porambage et al. "Certificate-based pairwise key establishment protocol for wireless sensor networks". In: *Computational Science and Engineering (CSE), 2013 IEEE 16th International Conference on*. IEEE. 2013, pp. 667–674.

[34] Maneesha V Ramesh, Sangeeth Kumar, and P Venkat Rangan. "Wireless Sensor Network for Landslide Detection." In: *ICWN*. Citeseer. 2009, pp. 89–95.

[35] Shahid Raza, Thiemo Voigt, and Vilhelm Jutvik. "Lightweight ikev2: A key management solution for both the compressed ipsec and the ieee 802.15. 4 security". In: *Proceedings of the IETF workshop on smart object security*. Citeseer. 2012.

[36] Shahid Raza et al. "Lithe: Lightweight secure CoAP for the internet of things". In: *Sensors Journal, IEEE* 13.10 (2013), pp. 3711–3720.

[37] Dale Skeen and Michael Stonebraker. "A formal model of crash recovery in a distributed system". In: *Software Engineering, IEEE Transactions on* 3 (1983), pp. 219–228.

[38] Group D. Stebila. "Elliptic curve algorithm integration in the secure shell transport layer". In: *RFC 5656* (2009).

[39] Ellen Stuart, Melody Moh, and Teng-Sheng Moh. "Privacy and security in biomedical applications of wireless sensor networks". In: *Applied Sciences on Biomedical and Communication Technologies, 2008. ISABEL'08. First International Symposium on*. IEEE. 2008, pp. 1–5.

[40] TelosB. *TelosB Datasheet*. URL: `http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf`.

[41] Deepika Verma, Rekha Jain, and Anurag Shrivastava. "Performance Analysis of Cryptographic Algorithms RSA and ECC in Wireless Sensor Networks". In: *IUP Journal of Telecommunications* 7.3 (2015), p. 51.

[42] Yong Wang, Garhan Attebury, and Byrav Ramamurthy. “A survey of security issues in wireless sensor networks”. In: *IEEE Communications Surveys and Tutorials* 8 (2006), pp. 2–23.

[43] Geoffrey Werner-Allen et al. “Deploying a wireless sensor network on an active volcano”. In: *Internet Computing, IEEE* 10.2 (2006), pp. 18–25.

[44] Kin Choong Yow and Amol Dabholkar. “A light-weight mutual authentication and key-exchange protocol based On elliptical curve cryptogaphy for energy-constrained devices”. In: *International Journal of Network Security & Its Applications (IJNSA)* 2.2 (2010).