UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Lightweight cryptography for IoT devices

Pedro Miguel Coelho Rosa

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutor José Manuel da Silva Cecílio
Prof. Doutor André Nuno Carvalho Souto

2022

Acknowledgments

I would like to thank my advisers Prof. José Cecílio and Prof. André Souto for all the guidance, patience and help provided. Without them, it would have not been possible for this work to reach its end. I also want to thank to the PhD student Zygimantas Jasiunas for the assistance provided for the experimental part of the energy consumption calculation.

I want to thank the most important people, not only during this year, but throughout the 22 years of my life, my family. Without them my academic life would never have gone the same way and I would not have been able to finish my master's degree. Finally, I would like to thank my friends who have supported and helped me during this year.

Aos meus pais

Abstract

Lightweight cryptography is a field that has been growing fast recently due to the demand for secure Internet of Things (IoT) applications. These algorithms provide security for computational power, memory, and energy-constrained devices. In this work, we propose a new protocol based on lightweight cryptography algorithms that enables the generation and distribution of keys for symmetric systems to be used in private communications on a wireless sensor network (WSN). The proposed protocol is designed to work in multi-hop communication networks, where nodes out of range of the Base Station can be part of the network, offering the same security mechanisms that a node in the communication range of the Base Station has. Experimental results and a detailed comparison with other architectures show how fast and energy-efficient the protocol is, while ensuring a high level of authenticity, confidentiality and integrity.

Keywords: Lightweight, Cryptography, IoT, WSN, Diffie-Hellman

Resumo

A inovação tecnológica possibilitou que novos dispositivos se liguem entre si, criando um ecossistema que traz inúmeras funcionalidades e vantagens no dia a dia, criando a Internet das Coisas (IoT). Atualmente, cerca de 46 mil milhões de dispositivos IoT são usados, devendo esse número chegar a 125 mil milhões até 2030. Este número crescente de dispositivos aumenta a quantidade de dados que são transportados em redes abertas como a Internet. A sua proteção é um desafio atual, presente na investigação científica da área e representa um desafio crítico para IoT: aumentar a segurança e privacidade dos dados. A falha na utilização de criptografia adequada e outros métodos de segurança pode levar a graves consequências, como acessos não autorizados e manipulação de dados. Os dispositivos IoT têm normalmente poucos recursos, com limitações a nível de processamento, memória e com desafios a nível de consumos energéticos, em parte porque são geralmente colocados em ambientes isolados. O uso de algoritmos de criptografia específicos é por isso essencial, uma vez que os esquemas tradicionais de criptografia exigem recursos que não estão normalmente disponíveis em equipamentos IoT.

O objetivo principal deste trabalho de mestrado é desenvolver uma Arquitetura de Distribuição de Chaves para gerar chaves criptográficas simétricas entre nós da rede, numa comunicação *multi-hop*. Para atingir esse objetivo, os nós previamente autenticados podem autenticar outros nós que se queiram juntar á rede. Depois da autenticação, a *gateway* gera uma chave de sessão com o esse nó usando o protocolo *Diffie-Hellman* e um algoritmo de criptografia "leve" (adequado ao limite de recursos dos dispositivos). Levando em consideração o cenário de aplicação do projeto AQUAMON e os seus requisitos, vários esquemas de distribuição de chaves e algoritmos de criptografia leve, da competição NIST, foram estudados para encontrar um método adequado para distribuir os parâmetros necessários entre os nós e a *gateway*. Com base nos dez algoritmos finalistas da NIST, projetamos um novo protocolo de autenticação para gerar chaves de criptografia simétricas entre nós numa comunicação *multi-hop*. Ao mesmo tempo, o protocolo fornece integridade usando um algoritmo de assinatura digital. A nossa proposta de distribuição de chaves envolve duas fases principais: autenticação de nós e geração da chave de sessão. Para maior comodidade e aumento da vida útil do esquema, apresentamos também outros mecanismos necessários como a renovação de chaves de sessão. Para o correto funcionamento da arquitetura, cada nó deve ter uma chave de rede padrão,

um algoritmo de criptografia, um conjunto de chaves armazenadas num arquivo, um valor primo p e g para o protocolo de troca de chaves *Diffie-Hellman* e, finalmente, um algoritmo de assinatura digital para assinar e verificar as mensagens. Todas as mensagens na arquitetura, incluindo os processos de acordo das chave de sessão, serão cifradas com o algoritmo de criptografia leve. Qualquer algoritmo de criptografia pode ser usado na arquitetura desde que as chaves sejam do mesmo tamanho que as definidas na nossa arquitetura e suportados pelos dispositivos envolvidos.

A fase um da arquitetura consiste na autenticação do nó. A *gateway* ou um nó já autenticado irá autenticar o nó que está a tentar juntar-se á rede. A troca de mensagens nesta fase consiste no nó tentar provar que é legítimo, mostrando que possui o conjunto correto de parâmetros, como a chave de rede, uma chave correta do conjunto de chaves (que será usada para cifrar menssagens antes de uma chave de sessão ser estabelecida) e responder corretamente ao desafio dado. A segunda fase da arquitetura é a geração da chave de sessão. Depois de um nó ser autenticado, apenas a *gateway* pode gerar a chave de sessão com o nó (com ou sem a ajuda de um nó intermédio). Para isto, é usado o protocolo de acordo de chaves *Diffie-Hellman*. Os parâmetros são cifrados com a chave do conjunto de chaves decidido na autenticação sendo, portanto, resiliente a ataques *man-in-the-middle*. Desde que o nó tenha as variáveis corretas para esta fase, ele poderá calcular a chave comum de 256 bits com o *gateway*, permitindo que comuniquem com segurança. As chaves de sessão devem ser renovadas periodicamente para aumentar o tempo de vida da rede e a segurança geral. Dependendo de cada cenário e ambiente, a periodicidade de renovação das chaves é diferente e deve ser repensada. Para renovar a chave, repetimos o processo na fase 2, mas desta vez, todos os parâmetros são cifrados com a chave de sessão antiga. Este processo deve ser iniciado pelo nó. No entanto, se não for esse o caso, a própria *gateway* pode solicitá-lo, removendo o nó da rede se não houver resposta. A arquitetura apresentada fornece um esquema de autenticação e confidencialidade, porém, nenhum mecanismo de integridade real é utilizado. Assinaturas digitais podem resolver esse problema, garantindo que nenhuma mensagem na rede seja modificada. Para implementar um mecanismo de assinatura digital, os nós devem calcular o par de chaves privada/pública e enviar a chave pública cifrada para o *gateway*, como qualquer outra mensagem na arquitetura. Os nós usam a chave privada para assinar as mensagens enquanto que a *gateway* usa a chave pública para verificar as mensagens assinadas enviadas por ele. O par de chaves privada/pública também é renovado com a chave de sessão, se necessário.

A proposta foi implementada em linguagem de programação C. A implementação consiste em dois programas: *Gateway* para a estação base da arquitetura e os Nós para todos os dispositivos leves que vão comunicar com o *gateway* de forma direta ou indireta. Para simular um cenário da vida real onde vários nós se interligam á *gateway*, usamos *threads* da biblioteca *pthread*. Uma vez que um nó solicita conexão com o *gateway*, ela

cria uma *thread* onde executa todas as operações e processos da arquitetura para o nó específico.

Também é apresentada uma análise de segurança dos métodos utilizados na arquitetura, nomeadamente a diferença entre o protocolo *Diffie-Hellman* usual e a nossa versão cifrada que previne alguns ataques conhecidos como *man-in-the-middle*. Por fim, mostramos a eficácia das chaves de criptografia geradas considerando o seu tamanho e o seu tempo de resistência contra ataques.

Apresentam-se ainda informações estatísticas dos algoritmos executados num Raspberry Pi 3 B+ e num Desktop com um processador i5-12400F. Também é possível ver resultados da solução como tempos de execução de todas as operações: solicitação de autenticação, geração do desafio, resposta do desafio, verificação do desafio, cálculo de valores públicos de *Diffie-Hellman* e chaves de sessão. A abordagem proposta foi simulada em dois cenários diferentes: comunicação entre os nós e a *gateway* diretamente, e comunicação entre os nós e o *gateway* com a ajuda de um nó intermédio. O consumo de energia também foi medido executando a arquitetura no Raspbery Pi 3 B+. A arquitetura gasta pouca energia, apenas cerca de 0,03W por segundo, ou 30 milijoule no cenário descrito. Consideramos a possibilidade de existir picos de energia durante a renovação das chaves dos nós, porém, tal não foi verificado, comprovando que os cálculos necessários para a geração das chaves são muito leves. Isto permite-nos diminuir o intervalo da renovação da chave, se necessário, sem comprometer significativamente o consumo de energia.

Por fim, foi feita uma comparação detalhada da arquitetura com esquemas semelhantes num conjunto diversificado de parâmetros como: consumo de recursos, confidencialidade, integridade, escalabilidade, renovação de chaves, resistência de captura e *forward and backward secrecy*. Ao pontuar cada parâmetro com um valor alto, médio ou baixo e justificar a nossa pontuação, a arquitetura mostrou ser uma boa solução para redes com vários nós operacionais numa vasta área.

Palavras-chave: Criptografia, Internet das coisas, Redes, Diffie-Hellman

Contents

List of Figures

List of Tables

List of Acronyms

AD Associated data

AES Advanced Encryption Standard

ARX Addition Rotation XOR

CPU Central Processing Unit

DH Diffie-Hellman

DOS Denial of service

DSA Digital signature algorithm

ECC Elliptic curve

ECDSA Elliptic curve DSA

FN Feistel Network

GFN Generalized Feistel Network

IETF Internet Engineering Task Force

IoT Internet of things

IP Internet Protocol

Light-SAE Lightweight Secure Access Enhancement for Multihop IoT Networks

LWC Lightweight cryptography

MAC Message authentication code

MIM Man-in-the-Middle

NIST National Institute of Standards and Technology

NLFSR Nonlinear-feedback Shift Register

RAM Random Access Memory

ROM Read-only Memory

RSA Rivest–Shamir–Adleman

SKEW Self Key Establishment Protocol for Wireless Sensor Networks

SPN Substitution-permutation Network

TCP/IP Transmission Control Protocol/Internet Protocol

WSN Wireless Sensor Network

Chapter 1

Introduction

In a world full of digital technology, the Internet of things (IoT) has allowed countless devices to connect over the network, creating an ecosystem that brings numerous advantages to our daily lives [5]. IoT uses smart devices and the internet to develop new solutions. Even if it goes unnoticed, these devices can be found everywhere. There are numerous other areas where IoT is used, such as smart cities, transportation, sales, farming and tourism [62]. The main goal of these networks is sensing and transferring data to a base station or to the cloud to be processed and stored [51].

Currently, approximately 46 billion IoT devices are being used, and this number is expected to reach 125 billion by 2030 [31]. With such an increase, networks are bound to become more complex with the amount of data that will circulate. A substantial part of this data needs to be protected, mainly confidential information, bringing to light a critical challenge in IoT: security and privacy.

1.1 Motivation

Most devices in IoT do not use secure encryption or authentication algorithms in their communications [30]. Failure to use proper encryption and other security methods can lead to severe consequences such as unauthorized access and manipulation of data.

IoT devices tend to be small and are usually placed in constrained environments [28], and therefore are very resource-limited in terms of processing power, memory and energy. Hence, using encryption algorithms specific to such devices is necessary since traditional system encryption schemes usually require unavailable IoT resources.

It is of extreme importance that IoT applications provide confidentiality, in order to prevent private data leaks and unauthorized access. Authenticity and Integrity are also extremely important, specially on sensor networks where, eventually, these sensors are placed on public places. Device authentication and the data flow should be managed carefully with correct mechanisms that fit these low resource devices.

Consequently, the scientific community started to consider Lightweight Cryptography

(LWC) [28], aiming to develop cryptographic primitives and algorithms that can execute faster, requiring less memory footprint and a very little amount of energy. The main goal is to achieve the best trade-off between security, cost (in terms of resources), and performance [28].

National Institute of Standards and Technology (NIST) has initiated the process to evaluate and standardize lightweight cryptography algorithms suitable for the constrained environments of IoT [50]. The process started with 57 submissions, and by the time of writing is in the final stage with ten candidates (ASCON [26], ELEPHANT [13], GIFT-COFB [7], Grain128-AEAD [34], ISAP [25], Photon-Beetle [8], Romulus [33], Sparkle [10], TinyJambu [63], and Xoodyak [23]).

However, using the correct encryption algorithms is not the only problem in some IoT applications. Before each device can encrypt and communicate the data, they must establish encryption keys among themselves using key distribution schemes.

1.2 Objectives

The main goal of this thesis is to develop a Key Distribution scheme called Lightweight Secure Access Enhancement for Multihop IoT Networks (Light-SAE) able to generate encryption keys among nodes in a multi-hop communication. To achieve this goal, already authenticated nodes will be used to authenticate other nodes trying to join the network. By doing this, we are not only allowing for nodes out of range of a gateway to get authenticated and join the network the same way close ranged nodes do, we are alleviating computational processing needed by the gateway. The gateway will then generate a session key with the node using the Diffie-Hellman (DH) Key exchange protocol and a lightweight encryption algorithm.

Multi-hop communication is a type of communication that allows devices to communicate over large areas by using other nodes in the middle as bridges to retransmit the messages. Previous research shows that this type of communication could also reduce energy consumption of the devices, prolonging the network lifetime in a Wireless Sensor Network (WSN) [66]. Allowing devices to communicate and establish all cryptography primitives necessary using this technology is a must for WSN where nodes are scattered over a large area and are not able to communicate directly with others.

This work was done in the context of AQUAMON project [18]. Taking into account the application scenario of the project and its requirements, several key distribution schemes were studied to find an adequate method to distribute the parameters needed between the nodes and the gateway so they can communicate privately. However, those schemes showed limitations concerning the resources available and the network topology that can be used to build the monitoring system. Based on the NIST ten finalist algorithms, we designed a new authentication protocol to generate symmetric encryption keys

among nodes in a multi-hop communication. At the same time, the protocol provides integrity by using a digital signature algorithm (DSA).

During and after establishing the encryption keys, devices need a lightweight encryption algorithm to encrypt all the data and parameters. For that, different methods of lightweight encryption were studied and listed some characteristics and specifications of NIST ten finalist algorithms. We had chosen two of them to adapt and implement on the project where our solution was tested.

In summary, the main objectives of this work are:

- Design a key distribution scheme that supports a multi-hop communication between nodes and a gateway, providing authenticity, confidentiality and integrity.
- Study lightweight encryption algorithms to find the most suitable ones for our solution and use them for the encryption of whole data and all parameters needed during session key establishment.
- Design and implement an architecture scheme to test the performance of a real scenario network provided by the AQUAMON project.

1.3 Contributions

In this work we offer a detailed study of NIST finalist lightweight cryptography algorithms by providing some of their specifications along with execution time results to help choose the most appropriate for applications.

We propose and develop a multi-hop Key Distribution scheme called Light-SAE that allows for the authentication and generation of session keys between a main station and each sensor node. By using a lightweight cryptography algorithm, a digital signature algorithm and a set of exchanged messages to prove the authenticity of the devices, we ensure the confidentiality, authenticity and integrity in our solution.

We also compare our solution to other existing schemes in the literature by presenting results and evaluate them in several parameters such as resource cost, confidentiality, integrity, scalability, key freshness, capture resistance and finally, forward and backward secrecy.

1.4 Thesis Outline

This document is organized in the following way:

- Chapter 2 – Introduction of important and necessary background information for the understanding of the research work such as: constrained devices, cryptography, lightweight cryptography, key distribution in WSN and digital signatures.

- Chapter 3 – This chapter contains some related work on Lightweight cryptography, focusing on NIST finalist algorithms and key distribution approaches in WSN with positives and negatives of each architecture.
- Chapter 4 – Describes the proposed multi-hop key distribution solution. It is divided into several steps/stages and mechanisms needed to handle different authentication steps providing robustness against attacks.
- Chapter 5 - Some details of the implementation of our proposed work are discussed in this chapter namely how devices perform certain steps of the scheme, important functions and variables used, structure of the messages and other important mechanisms such as key renewal.
- Chapter 6 - A security analysis is presented where we describe how our scheme defends against known attacks. We also show through some calculations how safe our generated keys are.
- Chapter 7 - Presents the obtained results of the chosen cryptography algorithms and digital signature algorithm to be used in the protocol. Then, results of our proposed solution processes are presented in single-hop and multi-hop communication scenarios. They were tested on two different machines with different capabilities and resources.
- Chapter 8 - Concludes the thesis by summarizing what was achieved in this work as well as future work possibilities.

Chapter 2

Background

In this section, we present the background needed for this research work, such as constrained devices in IoT, concepts of traditional and lightweight cryptography with a tour on characteristics of NIST finalists and finally, key distribution schemes.

2.1 Constrained Devices

IoT applications encompass many small devices and objects that can connect themselves, directly or indirectly, to the internet. An IoT application usually consists of two kinds of devices: sensor nodes and a gateway [49]. Sensor nodes are used to sense environmental conditions such as temperature, humidity, sound and vibrations. Constrained devices usually work under severe resource-constrained environments such as low computational power and memory, limited battery and insufficient wireless bandwidth [15]. These devices are mostly used to collect specific data from natural ecosystems or inside buildings. Their final goal is to send this information to a base station.

Base station or Gateway are relatively powerful devices with good processors and no constraints on power and memory. These devices are the ones that later route the information gathered by the nodes and sensors to the servers [49] as presented in Figure 2.1.

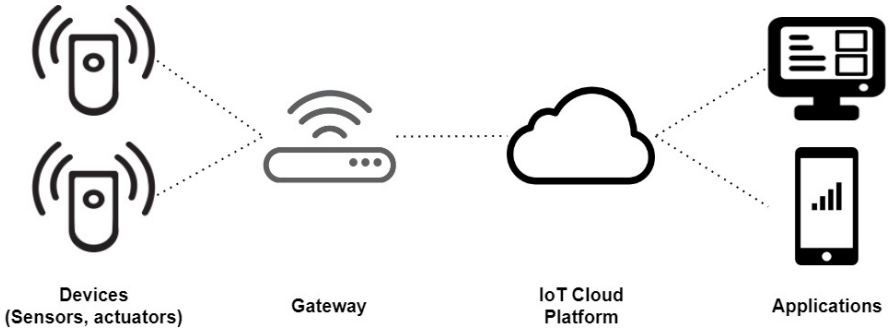

Figure 2.1: IoT Architecture

Internet Engineering Task Force (IETF) classified constrained devices into three categories to simplify the great variety of devices that can connect to the internet [49]:

- Class 0: Devices in this category are the most constrained in processing capabilities and memory, with less than 10 KiB of Ram and 100 KiB of Flash. Because of their weak nature, secure communication is sporadic and difficult to implement among such devices.

- Class 1: These devices have more processing power and memory than the previous. They can support lightweight security protocols for secure communication between themselves and the gateway.

- Class 2: This last class belongs to the less constrained devices. They can perform as good as mobile phones and support most protocol stacks. However, devices in this class should still be energy efficient for high interoperability.

2.2 Cryptography

Cryptography is a technique to achieve confidentiality of data and protect information. It is used daily by billions of people around the world [47]. The emergence of online commerce and social networks increased the data produced by organizations [60]. Some data must be protected, otherwise it could cost a company millions. Busafi and Kumar [2] refer to data security as the highest priority when it comes to using encryption algorithms as this data has to reach the intended users safely and unchanged by any malicious eavesdroppers.

Cryptographic algorithms can be classified as Symmetric or Asymmetric cipher. Symmetric cipher uses the same key for encryption and decryption of data by both parties (sender and receiver). In contrast, asymmetric cipher implies the usage of two different keys for the two operations. Most of the times, the public key is used for the encryption, while the private key is used for the decryption [42]. However the inverse use can happen, to prove the authenticity of who is sending the data for example. Since Asymmetric ciphers use public keys, which are known to the public, and private keys are only known to the user, there is no need to distribute the private one before transmission, reducing the risk of getting them exposed. However, asymmetric ciphers are computationally intense since they are based on complex computational calculations and therefore require a lot of resources to be implemented. Kumar et al. [42] estimate that Asymmetric encryption schemes are almost 1000 times slower when compared to Symmetric encryption schemes.

2.3 Lightweight Cryptography

The standard cryptography algorithms cannot be used on IoT architectures due to the constrained properties of the latter ones. The main challenges when implementing traditional cryptography algorithms on IoT architectures are [48]:

- Limited memory to store and run applications (RAM, ROM);
- Low computing power to process data;
- Usage of batteries for power;
- Small physical area to deploy.

To achieve secure applications, it was necessary to create encryption algorithms dedicated to such constrained devices [28]. Lightweight cryptography aims to develop cryptography primitives and algorithms that can execute faster, have little memory footprints and consume very little energy. Therefore, the three main characteristics of LWC algorithms are physical cost (physical space occupied, memory demand and energy consumption), performance (latency and speed), and security (block and key length and other parameters) [61]. LWC algorithms achieve the first two characteristics by using simple round functions on tiny blocks ($\leq$64bit) and relatively small keys ($\leq$80bit). The last characteristic but not the least significant one, security is achieved by adopting several modes of operation such as Block Cipher (which offers six internal structures like SPN, FN, GFN, ARX, NLFSR and Hybrid), Stream Cipher and Hash functions. Block cipher refers to ciphers where fixed-size blocks are processed simultaneously, as opposed to a stream cipher, which encrypts data one bit at a time.

Several research papers have already established a comparison [30] [61] [57] comparing software and hardware implementations of LWC algorithms. However, the most recent research on the field has been submitted about the NIST competition algorithms. In particular [30] and [46] compare and evaluate all of these algorithms from the competition (round 2) in terms of throughput and speed (clock-cycles/byte), execution time (ms), CPU usage, RAM usage and energy consumption. All of these works and analyses will be useful for researchers in choosing suitable algorithms for real scenarios and architectures with their specific limitations.

2.4 Key distribution in WSN

Cryptographic algorithms need encryption keys to encrypt and later decrypt information. For that, nodes and the base station in a WSN, must agree on these keys in such a way that no adversary can get them, compromising the whole network [41]. Key management is the core of secure communication in a WSN [55]. Keys should be generated or distributed and

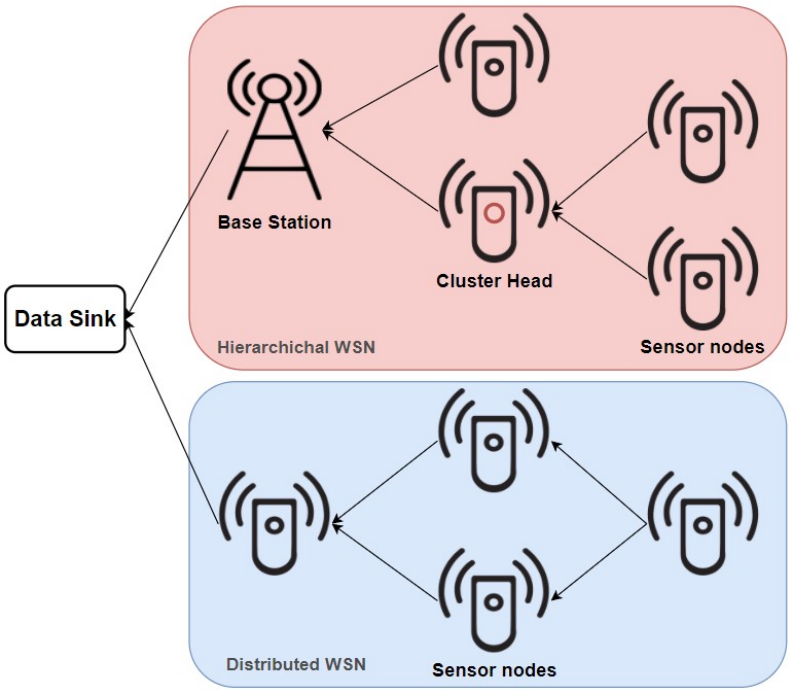

Figure 2.2: Types of Wireless Sensor Networks

then stored in a way that no outsider can access it since their discovery would compromise the communication and data integrity. The entire scheme should include a refresh and update process of the keys from time to time and should satisfy the three groups of metrics mentioned by Marcos A. Simplício et al. [56]: efficiency, security, and flexibility. There are different ways of implementing such schemes. For instance, a single key could be used for all nodes, making the process easier. However, if one node gets compromised, so is the entire network. On the other hand, having a different key for every node makes the whole key management process harder if the WSN consists of a large number of nodes [41]. There is no one-size-fits-all solution for key distribution problems in WSN [17].

In [32], it is possible to see a table with advantages and disadvantages between the usage of Single network-wise key and Pair-wise key.

2.4.1 Key distribution in Distributed WSN

In distributed WSN architectures, shown in Figure 2.2, there is no fixed infrastructure, and the network topology is not known before deployment. In these networks, nodes scan their surroundings to find their neighbouring nodes [17]. Distributed WSN sensor nodes can use either pre-distributed, dynamically generated pair-wise or group-wise key schemes. Key predistribution is the distribution method of keys onto nodes before deployment in WSN. As the name suggests, this method stores keys into the nodes before deploying the network. Key predistribution requires little overhead in the communication and almost

no processing overhead since the keys are already pre-loaded in the nodes [52]. Pair-wise key distribution is an approach where every node in the sensor network shares a unique symmetric key with every other node [17]. Other similar approaches are Random key pre-distribution discussed in [27] and location-based key management [20]. The first consists in deploying a random subset of keys from a key pool to each node before deployment of the network. The second uses grid information of the network area to determine the location of the nodes and improve the key connectivity. The idea is to have each sensor share a pair-wise key with its closest neighbors [17].

2.4.2 Key distribution in Hierarchical WSN

A Hierarchical WSN is shown in Figure 2.2. In this type of network, there is a hierarchy among the nodes based on their capabilities. Devices usually consist of base stations, cluster heads and sensor nodes [17]. Cluster heads are trusted nodes that can act as the key server as well as keep some data temporarily until it needs to be sent to the base station. This is usually done in order to alleviate overhead in the base station in networks with a high number of sensors. Nodes and the base station work the same way as distributed WSN. Specific devices can then send the data to the data sink which refers as the destination for the data flow.

2.5 Digital signatures

Digital Signature is a mathematical scheme which ensures the integrity of a conversation by giving authenticity to the digital message along with its sender [39]. This type of encryption technique is perfect for schemes that require data integrity during the exchange of messages between several parties. A digital signature consists on attaching a unique code that acts as a stamp on the message. This code can be generated by hashing the message with the signer user private key.

Currently, industries use this technology to improve document integrity in several areas [45]:

- ID cards – Many workers in several corporations use smart cards to identify themselves. These are physical cards that are endowed with their digital signature and can be used to either mark presence in an area at a certain time, or enter a restricted area of a building.
- Financial services – As expected, financial sectors use digital signatures for various important documents like contracts, loan processing and insurance documentation.
- Cryptocurrencies – This technology is a very important part of cryptocurrencies like bitcoins to authenticate the blockchain. It is also used to show ownership or participation in transactions of these currencies.

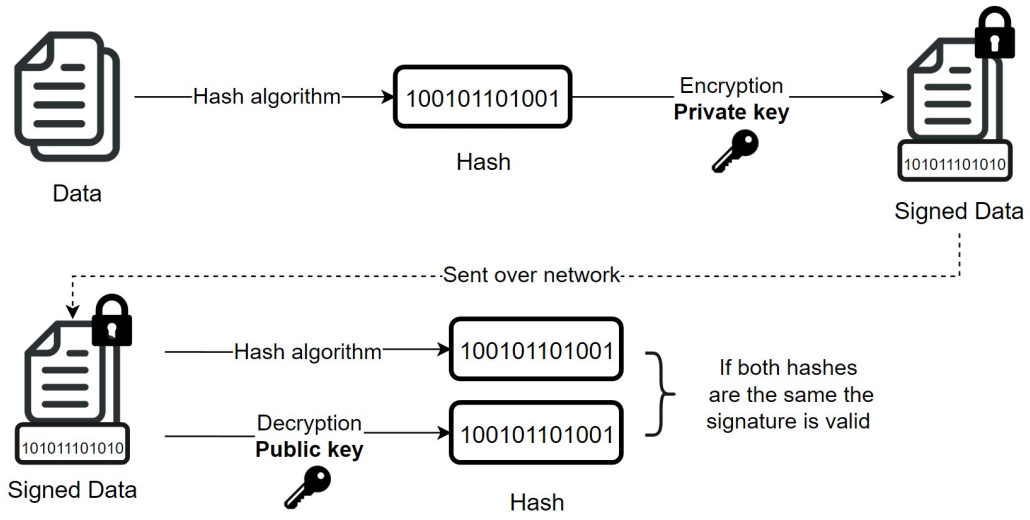

Figure 2.3: Digital Signature Diagram

There are usually three distinct phases in digital signature schemes:

- Private/public key pair generation – In these schemes, usually the signer generates this key pair. The private key is only known to the user/device that will be signing and sending the message, while the public key will be used to verify the message and must be previously sent to the user/device meant to receive the messages.
- Signature generation – Here, the signer will generate a hash of the message and encrypt it with the private key, after that, it sends the digitally signed data to the correct recipient.
- Signature verification – In this last step, the recipient must use the public key to verify the message, along with generating the hash of the message using the hash algorithm. If these two hashes are the same, it means the message is correct and was not modified in any way.

In Figure 2.3 it is possible to see how the whole process described above works. The main reason why these methods use hashes is because their value is unique to the hashed data, meaning that any changes, even in a single character, will result in a totally different value [45]. During the signature verification, if the verification of the hash does not match the second computed hash of the same data, it proves that the data was changed after the signature.

Chapter 3

Related Work

3.1 Lightweight cryptography

Since most current cryptography algorithms were designed for desktop/server environments, many of these algorithms do not fit into constrained devices. Since 2017, the National Institute of Standards and Technology (NIST) has initiated a call, evaluation and standardization process of lightweight cryptography algorithms suitable for IoT constrained environments [50]. The process started with 57 submissions, and it is currently in the final stage with ten finalists: ASCON [26], ELEPHANT [13], GIFT-COFB [7], Grain128-AEAD [34], ISAP [25], Photon-Beetle [8], Romulus [33], Sparkle [10], TinyJAMBU [63], and Xoodyak [23]. Table 3.1 summarizes some characteristics of these ten algorithms, such as the mode of operation which was briefly introduced in the previous chapter, if the algorithms provide authenticated encryption and hash, the recommended or mandatory key size and nonce, the block size and number of rounds, and finally if the algorithm is targeted for software or hardware.

Name	Language	Mode of operation	Authenticated encryption	Hash	Key	Nonce	Block size	Rounds	Target
ASCON [26]	C	Permutation	Yes	Yes	$\leq$160	128	64,128	18,20	Software
ELEPHANT [13]	C	Permutation	Yes	Yes	128	128	64,128	80	Soft/Hard
GIFT-COFB [7]	C	Block cipher	Yes	No	128	128	128	40	Hardware
Grain-128 [34]	C/C++	Stream cipher	Yes	No	128	96			Software
ISAP [25]	C	Permutation	Yes	Yes	128	128		31,33 48,56	Hardware
PHOTON-B [8]	C	Permutation	Yes	Yes	128	128	Arbitrary	12	Hardware
Romulus [33]	C	Block cipher	Yes	Yes	128	128	128	40	Hardware
SPARKLE [10]	C	Block cipher	Yes	Yes	128,196 256	256	16,24,32	4	Hardware
TinyJAMBU [63]	C	Block cipher	Yes	No	128,196 256	96	20	n	Hardware
Xoodyak [23]	C/C++	Permutation	Yes	Yes	$\leq$180	128		12	Hardware

Table 3.1: NIST finalist algorithms specifications

Fotovvat *et al*. [30] made a comparative analysis of the 32 algorithms (from Round 2) that were presented for IoT sensors where different cases were tested. TinyJAMBU was the lowest energy-consuming algorithm on a Raspberry Pi 3B. This algorithm has

high expectations with small and very organized code and minimal execution time. ELEPHANT showed worse results on a Raspberry Pi 3B in terms of execution time but was very similar in RAM and CPU usage compared to TinyJAMBU. Based on this conclusion, we chose the TinyJAMBU algorithm to be used in this proposal, where all messages will be encrypted using it.

TinyJAMBU [63] was developed by Hongjun Wu and Tao Huang. It is a variant of the JAMBU [13], a block cipher authenticated encryption scheme based on key permutation. The algorithm uses a 128, 196 or 256-bit key permutation, a state size of 128 bytes, and a message block size of 32 bytes. It also uses a nonlinear feedback shift register to update the state.

TinyJAMBU encryption method has four stages:

- Initialization - consists of the randomization of the state using the key and the nonce;
- Process associated data - As the name suggests, processes the full blocks of associated data (AD) and the partial blocks is XOR to the state;
- Encryption - This is where the plain text gets processed to get the ciphertext;
- Finalization - After the encryption, the algorithm generates a 64 bit authentication tag to perform the verification process on the decryption later.

The decryption process is very similar, but this time there is a verification step after the finalization to compare the previous tag with a newly generated one after the decryption process. The authors of TinyJAMBU state that associated data plays the same role as the nonce. When the same nonce but different associated data are used for a key, it is equivalent to the use of a new and unique nonce. If the same nonce and associated data are reused for a key, TinyJAMBU provides strong protection of the secret key and provides strong authentication security, but provides weak protection of the plaintext.

ELEPHANT [13] is an authenticated encryption scheme created by Tim Beyne *et al*. It is based on a nonce-based encrypt-then-MAC construction, and the mode of operation is permutation-based. The ELEPHANT scheme consists of three instances: Dumbo, where the permutations are smaller with 160 bits and is better suited for hardware. Jumbo with a 170-bit permutation, provides higher security under the same conditions. Finally, the Delirium instance achieves the best level of protection with 200-bit permutation and is stated to be better for software use but still performs well in hardware.

ASCON [26] is a lightweight encryption algorithm designed by Christoph Dobraunig *et al*. and can be used for lightweight hashing and encryption. ASCON has a low-degree substitution box that allows implementations with small overhead in both hardware and software. It uses a single lightweight permutation with a sponge based mode of operation

and an SPN-based (substitution–permutation network) round transformation that consists of 3 steps. The authors claim that all ASCON families provide 128-bit security. However, for this claim to be valid, the implementation must ensure that the nonce is never repeated for encryption with the same key. Decrypted plaintext should only be released after verifying the tag is completed. ASCON leaks the plaintext length like some encryption algorithms since it is the same as the ciphertext length (excluding the Tag length). So, if the confidentiality of the plaintext length is a requirement, users must compensate with padding.

Upon selecting the lightweight cryptography algorithm that are compatible with IoT devices, we will review the work concerning device/node authentication and message integrity. We will discuss the key distribution mechanisms for wireless sensor networks and the digital signature mechanisms that can be used in constrained devices.

3.2 Key Distribution in WSN

SKEW is a Self Key Establishment Protocol for Wireless Sensor Networks that manages encryption keys with less storage, communication, key transmission frequency and computational overheads in comparison with similar protocols [54]. The protocol can be applied in distributed and hierarchical WSN. If the first option is used, all nodes encrypt messages with a group key that can be refreshed periodically. So, each node that generates a group key broadcasts it to all nodes that can receive key refreshing messages. The approach used for Distributed WSN is clustering, consisting of transforming a distributed WSN into a hierarchical one. The base station selects the best nodes for coverage as cluster heads, encrypts the messages with a group key, and broadcasts them to all the nodes in its range.

SKEW decreases the overhead by combining key refreshing with usual network messages. Since information required for new key generation is piggybacked on transmitted messages, it is said that communication overhead for key distribution is reduced by 50% compared to other architectures. Only one message is required to establish a pair-wise key and one message to establish a group key. Keys are not directly transmitted within messages, only information about how they may be generated. Therefore it is less probable for keys to be disclosed. All the messages sent in our solution are encrypted, achieving a good level of confidentiality. As for integrity, no mechanism is mentioned. Even if packets always go encrypted, this means that an attacker could change the contents of messages in the network without being noticed. The architecture experiments proved scalability, the number of nodes was incremented to 32, 48, 64 and 100 nodes. In each step, all nodes were able to use broadcasted instruction for key refreshment. All nodes can generate a new version group key and broadcast it to other nodes. It is stated that the number of generated keys in SKEW is comparatively low because only nodes that

receive new encrypted messages must do refreshing. SKEW approach detects when an attacker tries to acess information in RAM, resetting if needed. However, if a node gets compromised, information in executive code and non volatile memory can be stolen.

A.B. Feroz Khan and G. Anandharaj present AHKM [29]. This scheme consists of a hash key based key management scheme with a multi-hop approach. It uses a one-way hash function for the establishment of a secure key between one-hop and multi-hop nodes. The network model is a hierarchical WSN, so it consists of a base station, cluster heads and sensor nodes. Each node starts with an initial key K_i that is used to generate a master key. This master key will be used to calculate a pair-wise key with a neighbouring node so they can communicate with each other. The key generation is based on a hash chain implemented with a one-way hash function so each cluster can hold at least one key among n keys generated by the hash function. Newly joined nodes get authenticated by the base station (one-hop neighbour) and periodically send "hello" messages to confirm their presence. They get removed from the network if no confirmation is obtained.

Based on pair-wise and group key management schemes, the work in [55] proposes an alternative method where a membership authentication mechanism is added to the key pre-distribution process. A control center is used to schedule the entire WSN communication. The scheme consists of several phases: Handshake process between 2 nodes that want to establish communication; Authentication and distribution phase where the control center authenticates both nodes; finally, the Communication phase, where both nodes calculate the session key to proceed with secure communication. It is also stated that the scheme resists capturing attacks more efficiently and performs better than similar schemes. However, when it comes to real-life scenarios, it is hard to ensure the credibility of the control center.

The control center authenticates the sensors by comparing the parameters known only by itself and the object. A timestamp prevents replay attacks from achieving a secure authentication process. The scheme achieves a good level of confidentiality and integrity by using a hash function to authenticate messages. Sensors can randomly join and leave the network. As long as the sensor passes the authentication of the control center when joining, it can obtain the session key, proving good scalability. As for key refreshing, the method contains a heartbeat feature, and if a node is captured, the sensor heartbeat feedback will exceed the time interval set so that the control center will revoke the key. This mechanism also makes it so if a node is captured, the session key gets revoked. If the adversary tries to rejoin the network, it won't be able to authenticate itself or use the previous session key.

Vipin Kumar *et al*. [41] propose a decentralized scheme for homogeneous cluster-based architecture. The scheme uses a hash chain to generate a key. Nodes use the strategy proposed in [9] to find common keys. The base station sends a hash chain and random keys to the nodes, and each node hashes the keys X amount of times, X being

their identifier. For example, nodes A and B ($ID_A > ID_B$) send their keys to one another, and the one with a lower identifier has to hash them ($ID_A - ID_B$) more times to get the same key. However, the proposed architecture does not guarantee that every node will find a common key using the hash chain. If so, the nodes use a recursive function and a preloaded seed before deployment to generate a common key. This way, the nodes will communicate with each other using this pair-wise key.

As for storage overhead the architecture is said to store only 10 to 20 keys and for expenditure of energy, it is stated that for most cases only one message per node is required (if recursive formula is used, more energy is required). The architecture does not present execution times making it difficult to further compare resource costs with other proposals. The architecture achieves a high level of confidentiality as all parameters are sent encrypted with a timestamp. To further maintain the integrity of the messages, the hash value of the message is also sent. As for scalability, if a new node is added to the network, a set of keys from a key pool is loaded to the node so it can find common keys with its neighbours. Keys can also be refreshed (for a specific node or all nodes). To do this, the base station sends a new parameter that will be used to generate the key. If a chain is compromised, so is the node. There is also a trade-off between computation and resilience by increasing the length of the chains. The architecture avoids node replication attacks since the adversary would need to have all the keys and assign himself the set of keys to the other nodes. Finally, the scheme allows for node replacement while keeping the same key as the older node which could be a problem if an outsider could take possession of the node.

Shahwar Ali *et al.* introduced a new cryptographic approach for data security in WSN [4]. The approach uses the low-energy adaptive clustering hierarchy protocol (a.k.a. LEACH) for data routing and Diffie-Hellman for key generation. Diffie-Hellman uses asymmetric encryption, meaning it gives a high-security level. However, since it is not resilient to man-in-the-middle attacks, the authors created an improved version called SMDH. The approach uses a hash function, and only the legitimate receiver of the parameters can calculate that hash. Every value transmitted over the network is hashed and can only be calculated if the receiver has the correct method and values. The authors also offer experimental results based on multiple parameters such as encryption, decryption, response time, and computational cost.

Benamar Kadri *et al.* proposed an efficient key management scheme for hierarchical wireless sensor networks in [6]. The method proposed is a simplified public-key infrastructure based on the authenticity of the base station as a secure entity. The base station and cluster heads are crucial to securing data transmission and establishing a session key with the nodes in its cluster. After all the proceedings, each sensor shares a symmetric key with its cluster head and the base station.

The scheme is based on symmetric key encryption, making it more efficient regarding

resource consumption. The entire handshake process is stated to have an energy cost of about 30 milijoule. Unfortunately however, no more details on performance are given, making it difficult to understand the amount of resources the architecture actually uses. Using handshakes, the base station shares with each sensor over the network a symmetric cryptographic key used for encrypting ordinary traffic over the network for data confidentiality. The architecture has an authentication system for all devices, and the integrity of messages is secured by a message authentication code (MAC) encrypted with the session key. To maintain integrity of messages, the protocol proposes to use a message authentication code MAC encrypted with the session key. As for scalability, in order for a sensor to join, the administrator must load the public key of the base station on the node. A node can then proceed with the handshake on a cluster head. As usual in these architectures, a key refreshing mechanism should be used to increase the life expectancy of these networks. The key update is launched by the cluster head using the same hand shake (to establish a new session key with the base station) The cluster head then encrypts the new key with the old one to send it to the other nodes within its cluster. No acknowledge is sent by the nodes. This scheme also seems to have forward secrecy since keys gets refreshed from time to time, this means that if a node leaves the network, after the refresh he won't have the valid key anymore. However, nothing about backward secrecy is mentioned and nodes might be able to rejoin using the same key (if its still valid). Authors mention that capture attack is out of scope of their architecture.

An improved energy-efficient key distribution and management scheme is proposed in [19]. The architecture has less overhead than other schemes because there is no requirement of encryption and decryption of the parameters. As for energy, the authors carried out an experiment with 1000 nodes (5, 10 cluster heads) and a runtime of 80 seconds. Each cluster head had an initial 10 J of energy. The residual energy (energy left after experiments) was around 6.5 J in cluster heads. In this architecture, cluster heads do almost all the work, alleviating the workload of the base station. After establishing keys, all information goes encrypted throughout the network. For the parameter exchange to establish such keys however, no encryption is used. As the authors state, it is not required because of the polynomial component, which makes it hard to compute the keys. No integrity mechanism is mentioned by the authors. As the parameters to establish keys goes without any encryption, messages could possibly be modified as the attacker wants without any detection. The authors carried out simulations on the scheme with 250, 500, 750 and 1000 nodes, proving that the architecture is scalable. However, as the quantity of nodes increases the system fails to scale as more memory will be required to store the keys. Nodes can refresh the keys using a random number sent by the gateway and bitwise XOR operation. In the end, all nodes will use the new keys for encrypted communication and discard the random number. The architecture is resilient to node capture attacks as long as the number of captured nodes does not exceed a certain number threshold.

3.3 Digital Signatures

In [35], Jansma and Nicholas present a performance comparison on some known public key cryptosystems used in digital signatures, such as Rivest–Shamir–Adleman (RSA) and elliptic curve (ECC). The paper provides a thoroughly explanation on how each of the schemes generate the key pair, the signature and how the verification process works. Source code for an easy implementation of these mechanisms is also provided. After implementing each algorithm and comparing the experimental running times, it was suggested by the authors that RSA key generation is significantly slower than ECC for key sizes above 1024 bits. However, for devices that do not need to generate the keys for each use (each message), RSA is still a very good option. As for the signature generation, both schemes seem to be very close to each other, with ECC winning for very large key sizes above 15360 bits. Finally, RSA took the win in the signature verification procedure, having much smaller times compared to ECC. It was easily concluded that for applications requiring message verification more often than signature generation, RSA could be a better choice.

William Stallings describes three digital signature algorithms approved and standardized by NIST in [58] witch are the DSA, elliptic curve DSA (ECDSA) and several versions of RSA-based digital signature schemes. Before DSA, the author presents simpler versions of that algorithm, like the Elgamal DSA which is an asymmetric key encryption algorithm based on the Diffie–Hellman key exchange. Schnorr DSA is also based in discrete logarithms and minimizes the message-dependent amount of computations required to generate the signature as it does not depend on the message and could be done in the idle period of the processor. All the DSA algortihms have been widely implemented and with the sufficient key size, can provide any security level desired. Finally, a comparison of the algorithms is given comparing different key sizes to the security each can provide.

Attila A. *et al.* [64] propose a lightweight multi-time digital signature scheme called SEMECS. The proposed scheme is highly efficient due to minimal and fast processes to generate a 32-byte signature, which translates to energy efficiency in constrained processors that are used in IoT applications. The scheme also uses a compact 32-byte private key and requires little storage. The authors provide a security analysis where they exploit the fact that SEMECS uses multiple-time signatures, meaning it has higher security for a limited number of times. Comparison aspects of the scheme with others are also provided, such as Signature generation time, Verification time, private/public key sizes and energy usage.

Craig Costello and Patrick Longa present a lightweight digital signature scheme called SchnorrQ [22], which combines the well-known Schnorr scheme and elliptic curves from FourQ [21]. FourQ is a high security and performance elliptic curve achieving a 128-bit security level. Results show that FourQ is around five times faster than methods offered by NIST, like the NIST P-256 curve and Ed25519 [11]. SchnorrQ offers extremely fast and

high secure 64 bytes digital signatures using a 32 bytes public/private key pair. Authors in [44] and [64] provide results of implementations of SchnorrQ in microcontrollers and compares it with other known schemes such as μKummer, proving to be twice as fast as that scheme and about four times faster than the fastest Ed25519 implementations in the literature. SchnorrQ efficiency translates not only to reduced latencies but also significant savings in energy, making this signature scheme an excellent way to achieve integrity in lightweight implementations and low-power applications such as IoT.

Chapter 4

Proposed Work

In this section we present our Key Distribution scheme, Light-SAE, and all of its features and mechanisms.

Light-SAE involves two main phases: node authentication and session key generation. For comfort, we also present other necessary mechanisms like session key renewal.

Table 4.1 summarizes the variable definitions needed to understand the proposed solution.

Abbreviation	Meaning
DK	Default Network Key
PK	Pool of keys
K_i	i-th key of Pool of keys
$R1$	Random value 1
$R2$	Random value 2
X	Second parameter of challenge
SK	Session key
f	Encryption function
f_{K_i}	Encryption function with K_i as key

Table 4.1: Variable definition

For the sake of simplicity, we consider nodes A, B and a gateway. Our solution is shown in Figure 4.1. It aims to allow the distribution of keys in a multi-hop way, so that node B out of range of the base station (gateway) can communicate using an intermediate and already authenticated node A. We assure that each node should have a Default Network Key DK, an encryption algorithm f, a pool of keys PK, a prime value p and g for the DH key exchange protocol and finally, a digital signature algorithm to sign or verify the messages. Every message exchange in this scheme, including establishing the key, will be encrypted with a lightweight encryption algorithm. ELEPHANT Dumbo version or TinyJAMBU 256-bit version from the NIST lightweight cryptography competition will be used in the scheme. However, any encryption algorithm may be used as long as the keys are the same size. Both algorithms present a well-organised code, showed promising

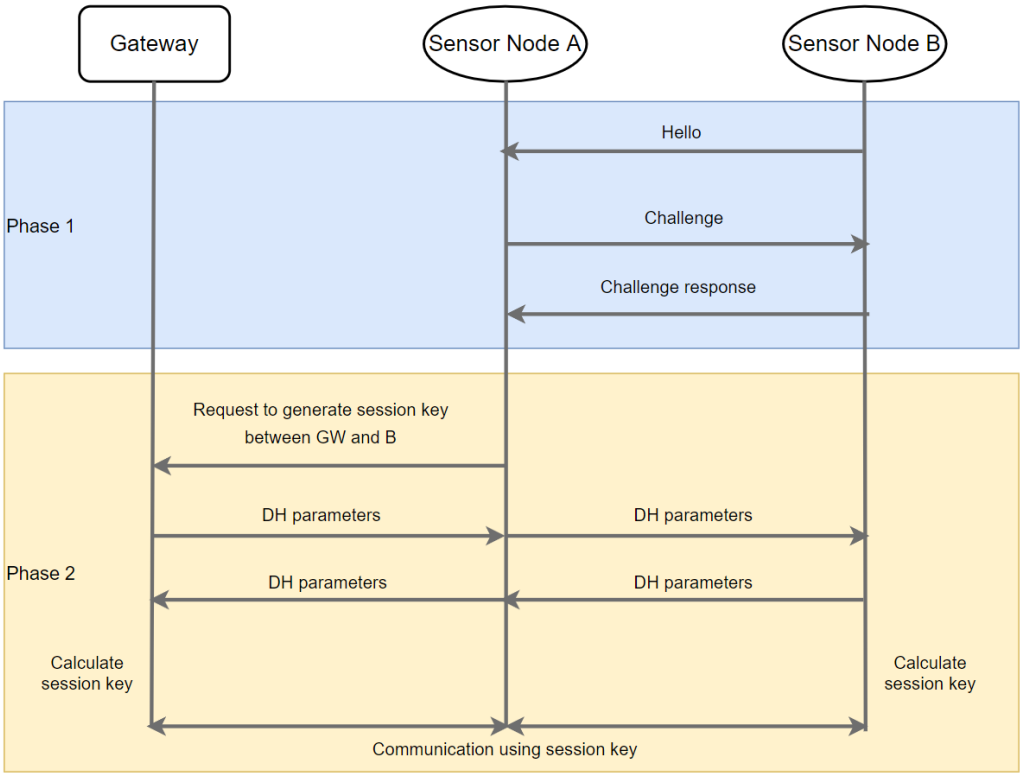

Figure 4.1: Proposed solution

results in previous studies mentioned in previous sections and better suit our implementation scenario.

In the proposed work, nodes will be performing several processes in different phases and hence acknowledgements steps are necessary at certain points of the protocol. For this purpose, different kinds of messages with different IDs are needed. When nodes or even the gateway receive them, they know what needs to be done. The types of messages are as follows:

- *Authentication message* - These are messages sent by nodes trying to join the network to already authenticated nodes or the gateway. Contents of the message are the node ID and a Key K_i;
- *Authentication response message* - These are the response to the authentication messages and are sent by authenticated nodes or the gateway. Contents are a challenge meant to be answered by the node seeking authentication;
- *Challenge response message* - As the name suggests, it consists of the challenge response and is meant to the node/gateway that sent the challenge in the first place;
- *Request to gateway message* - This message is sent by authenticated nodes to the gateway. It is a request for the gateway to initiate the session key generation process

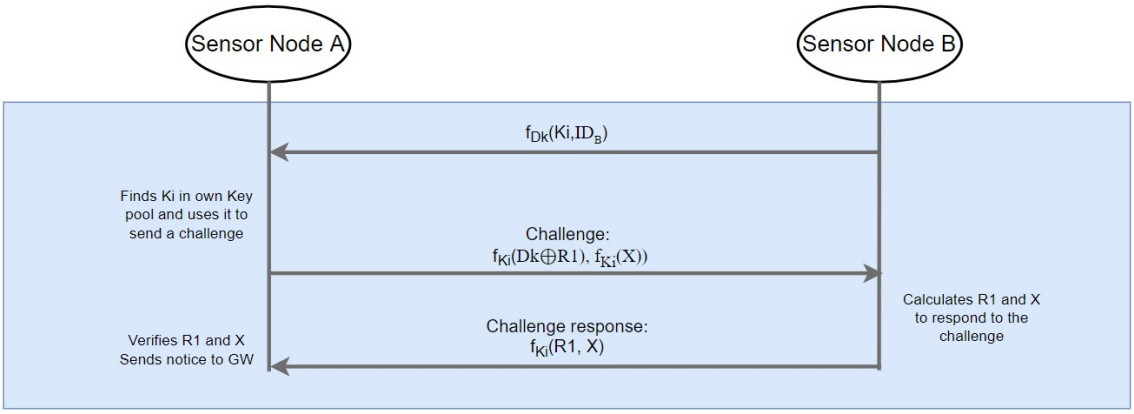

Figure 4.2: Node Authentication

with the recently joined node through himself. The contents of the message are the node ID that wants to join and the K_i used during its authentication.

- *Public value message* - Every device eventually sends this message. The contents of this message is the parameter needed (public value) by the other device to generate the common session key. Nodes can send other parameters with the public value in this message such as its own ID for verification purposes explained later and a public key from the digital signature algorithm;
- *Session key renewal message* - This message is sent by the nodes once per interval of time. It consists of the same contents as the *public value message* and it is meant for the gateway to calculate a new session key with the new public value sent.
- *Data message* - Message containing collected data by the nodes.
- *Key refresh time message* - This is a message sent by the gateway. It tells the node it is time to refresh the session key in case the node did not begin the process itself.
- *Acknowledge message* - Message sent by the gateway to the node in order to acknowledge receiving of the new public value from the *session key renewal message*.

We call readers attention for the fact that only authenticated nodes can send encrypted data to the gateway if it joins the network and is authenticated.

4.1 Phase 1 - Node Authentication

After a node out of range of the gateway joins the network, it sends an *authentication message* to neighbour nodes that will have to authenticate it. This message contains a random key K_i from the pool of keys KP and its own ID, encrypted with the encryption

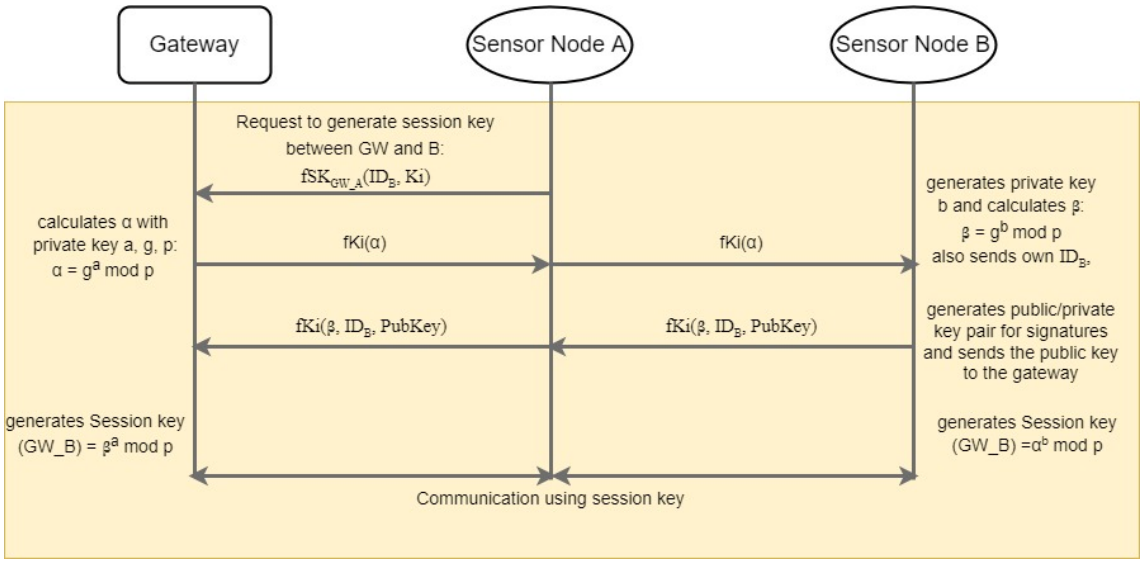

Figure 4.3: Session Key Generation

algorithm using as key the DK. The pool of keys are in a file poolOfKeys.txt that is created and managed by an administrator. Then, node A receives this message, decrypts it with DK and checks whether K_i is in its own KP. If it does not find it, it puts the node ID on their own blacklist. If Node A finds K_i, it will send a challenge to node B using this key to verify the identity of the newly joined node B. The challenge consists of: $f_{K_i}(DK \oplus R1, f_{K_i}(X))$ where $R1$ is a random value generated by node A. X can be different values, from a variable that characterizes the node that sent the challenge with its Mac $(f_{K_i}(MacA))$, a variable that uses a timestamp $(f_{K_i}(Timestamp \oplus ID_A))$ or a simple random variable ($R2$). After node B receives the challenge, it has to calculate $R1$ and X so it can send the challenge response: $f_{K_i}(R1, X)$. In Figure 4.2 it is possible to see the entire node authentication process. After the process above, node A sends a request for the gateway to generate a session key with node B.

4.2 Phase 2 - Session Key Generation

After node A (and possibly another node) confirms the identity of node B, it sends a *request to gateway message* to the base station. This request is sent encrypted with a session key between the gateway and node A: $f_{SK_{GWA}}(ID_B, K_i)$. The ID of node B is also sent so the gateway can later confirm it is really exchanging information with node B. To create the session keys between nodes and the gateway, the Diffie-Hellman protocol is used as shown in detail in Figure 4.3. The parameters are encrypted with the K_i node A sent in the request, so it is resilient to man-in-the-middle attacks (MIM). Every node in the network should have a g and a prime p value required by the DH protocol. This way, nodes only have to compute a private key in order to generate the session key. After

the device calculated its own public value it will send it to the other device so both can calculate the 256-bit session key used by the lightweight encryption algorithms. In this stage, the node also generates a public/private key pair to be used in digital signatures, the public key is sent to the gateway since it will be needed to verify the signatures from that point forward.

With this solution, the gateway will have session keys with every node in the network and all the communication is encrypted using these session keys and encryption algorithm.

4.3 Session Keys Renewal

Session keys should be renewed periodically to increase the network lifetime and general security. Depending on each scenario and environment, the period necessary to renovate the keys is different and should be thought out. To renew the key, we repeat the process in Phase 2. But this time, all the parameters go encrypted with the old session key instead of K_i. This time the gateway does not need to send the parameter α from the DH protocol since it is a constant value throughout the entire scheme (because its private key does not change without administrator interference). Node B starts by generating another parameter β with a distinct and newly generated private key (using a timestamp value for example). After computing the parameter β, it sends to the gateway, so both of them can generate a new session key. After the gateway receives the message and can generate the session key, it sends an *acknowledge message* to node B, making sure the new session key can be generated and updates on both sides. Only then they discard the old session key and start using the new one to exchange messages. If for any reason however, the node does not send the public value, the gateway must initiate the key renewal process itself by sending a *key refresh time message*. After some time, if the node does not send the new public value, it is removed from the network by the gateway. In this process, the private/public key pair used in the signatures is also renewed by the node and the new public key is sent to the gateway along with the new public value β.

4.4 Message treatment

As different types of messages are sent in the network from the start of the authentication until the end of the session key generation and future data, devices must know how to proceed when receiving them. For this, a switch is used as it executes a block of code depending on a message type variable. For example, if the message is an *authentication message*, which was explained before, the program will execute the function “authenticationResponse”, explained later in section 5.

During the process of authentication and session key generation, the gateway will receive several messages from few nodes and will need to use the correct key to decrypt

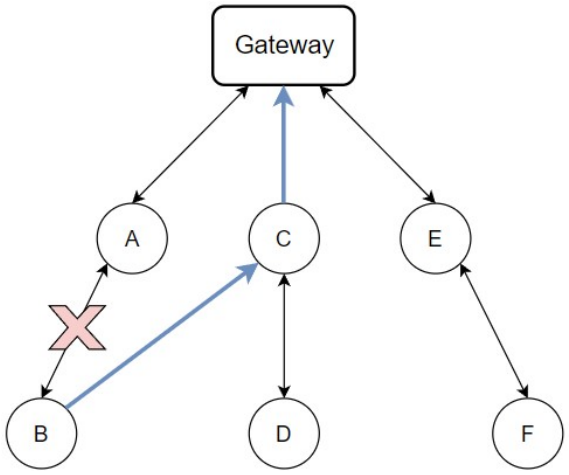

Figure 4.4: Neighbouring node communication

them. For example, when the gateway receives a request from a node A to generate the session key with a node B (*request to gateway message*), it will need to decrypt the request with node A session key, and decrypt/encrypt the public values with the established K_i so node B can correctly decrypt the message in order to generate the session key. To do this we use a message type parameter on each message. This way the devices can check the type of the message they receive and use the correct key for it. Using the previous example, the gateway would see that the message coming from A is a request to generate a session key with another node (*request to gateway message*), so the key needed is the session key with authenticated node A. For the next message, as it is a *public value message*, the gateway must decrypt with the K_i used by node A and node B during the authentication.

4.5 Neighbours communication

Using the previous scenario where node B uses node A to communicate with the gateway, assume that node A goes offline for some reason, as shown in Figure 4.4. If this happens, node B will not communicate with the gateway anymore. To solve this problem, nodes need to communicate with their neighbouring nodes to find another path until finally reaching the gateway. Instead of using node A, node B would use node C or any other node in its range as an intermediate to talk to the gateway. In this case, node B would find node C and re-established communication with the gateway through it, as we can see in Figure 4.4 represented with the blue arrows, the new route. We call the reader's attention to the fact that there is no need for node C to authenticate node B again.

4.6 Digital Signature

The solution proposed in this Thesis provides a scheme with good authentication and confidentiality. However, no real integrity mechanism is used. Digital signatures can solve this missing feature, making sure no message in the network is tampered. In our proposal, each device private key is calculated randomly and used for the DH parameters and session key generation. To implement the digital signature mechanism, nodes must calculate the private/public key pair used to generate and verify the signatures and send the public key to the gateway along with the public value, using the *public value message*. The nodes would use the private key to sign the messages while the gateway would use the public key received from the node, in order to correctly verify the signed messages sent by them. The private/public key pair would also be refreshed along with the session key if required. In this scenario however, the gateway cannot sign messages for the nodes to verify them. This means that if we wanted to send messages from the gateway to the sensor nodes, integrity would be missing in those messages.

4.7 Protocol implementation

We have resorted to the use of pseudo-code in order to describe the interactions between devices in Light-SAE in a scenario where a node B is out of range of the gateway. We present the pseudo-code for node Authentication and Session key generation for each agent involved in these processes.

4.7.1 Node Authentication

Algorithm 1 describes the code of a node (B) that initiates the process of joining the network. It starts by picking a random key K_i from the Key pool PK (Key collection should be in a file, so it is more flexible and easier to change it later). After having the key, it sends an *authentication message* to authenticated nodes (or gateway if it is in range) containing the K_i and the own ID: M$\{f_{DK}(K_i, ID_B)\}$. The node will now wait to receive a response from the authenticated node. Such message consists of a challenge. The node will decrypt it and calculate the values $R1$ and X so that it can answer to the challenge with the message: M$\{f_{K_i}(R1, X)\}$.

Algorithm 1 Node trying to join the network

$K_i = Kp[rand()\%size]$
decrypts M using K_i
$R1 = (DK \oplus R1) \oplus DK$
sends challenge response $(R1, X)$

Besides the normal functions of our nodes, like reading and sending data, nodes already authenticated are also permanently waiting to receive messages from other nodes trying to join the network. After receiving the message, it checks if the K_i sent by the node is in the Poll of Keys PK as we can see in Algorithm 2.

Algorithm 2 Authenticator node

```
flag = 1
decrypt(M) with DK
for i < size do
    if PK[i] = K_iB then
        flag = 0
        K_iA = K_iB
        break
    end if
end for
if flag = 1 then
    insert ID_B to blacklist
end if
R1 = rand()
X = rand()
challenge = f_{K_i}(DK ⊕ R1, X)
decrypt(M) with K_i
if R1_A = R1_B and X_A = X_B then
    send request to Gateway
else
    insert ID_B to blacklist
end if
```

If the key is valid, the node keeps the K_i to encrypt further messages until the session key gets generated. If not, the node ID is added to a blacklist for future reference. The node will now create and send a message consisting of a challenge to further authenticate the node trying to join the network: M$\{f_{K_i}(DK \oplus R1, X)\}$. Then, the node has to wait for the challenge response. After receiving, it decrypts the message and checks if $R1$ and X are correct, placing it on the blacklist if that is not the case. If everything is correct, the node will send a *request to gateway message* (encrypted with the session key between both of them) containing K_i used in the process above and the ID of the node so that the gateway can compare it later: M$\{f_{SK_{GWA}}(ID_B, K_i)\}$. This request for the gateway is meant to initiate the session key generation process with the joining node.

4.7.2 Session Key Generation

Algorithm 3 presents the code of a node, already authenticated, in the process of generating the session key. After receiving the public value from the gateway, it starts by generating a private key b to compute the value β with that private key and the hard coded

values g and p. It also generates a private/public key pair that will be used to sign and verify message signatures. The node sends the message M: $\beta, publicKey$ to the gateway. Finally, it calculates the session key with the public value α that was received.

Algorithm 3 Authenticated node that needs a Session Key

decrypts M with K_i
$charb = rand()$
$char\beta = g^b modp$
generates $private/public$ key pair
encrypt($\beta, publicKey$) with K_i
send $\beta, publicKey$ to Gateway
$charSK = \alpha^b modp$

The gateway, after receiving the *request to gateway message* (containing the K_i and the node ID) from the node that authenticated the new node, proceeds to execute Algorithm 4. It starts by decrypting the message and saving K_i and ID from the node. After that, it calculates the value α with the hard coded values "g", "p" and own private key "a". The gateway sends the message Mα to the node. The gateway now waits for the node to send a *public value message*. Once it does, it decrypts it and saves the $publicKey$ to verify the signatures later on. It also checks if the ID_B, sent from the node B along with the public value, matches the one sent by node A, who previously has authenticated it (this process is skipped if node B was authenticated directly by the gateway and not by an intermediate node A). If the ID matches, the gateway proceeds to generate the session key, allowing them to communicate privately.

Algorithm 4 Gateway

decrypt M with SK_{GWA}
$char\alpha = g^a modp$
encrypt(α) with K_i
send α to node
decrypts M with K_i
if $ID_B(sentbyA) = ID_B(sentbyB)$ **then**
    $SK_{GWB} = \beta^a modp$
else
    insert ID_B to blacklist
end if

Chapter 5

Implementation

In this section, the implementation of the proposed work is discussed, presenting the methods and approaches used. The entire solution was developed in the programming language C. The implementation consists of two programs: Gateway for the base station of the architecture, and nodes for all the lightweight devices that will communicate with the gateway directly or indirectly. The entire code of our solution is available in GitHub.

In order to implement a real life application where several nodes connect to the gateway, we have used threads from the pthread C library to create a multi-thread server. Once a node requests connection to the gateway, it will create a thread where it will execute all the operations and processes from the architecture of that specific node as it is possible to see in Figure 5.1.

5.1 Important variables

As previously mentioned, there are seven main "variables" for the architecture to work properly. First, the default network key is hard coded on every node meant to join the network as well as in the gateway. As for the pool of keys, a linked list is used with the following structure, where data is the K_i and the key is its index:

```
struct pool {
   char* data;
   int key;
   struct pool *next;
};
```

Firstly, the gateway and the node have to read every 256-bit key K_i from the file poolOfKeys.txt and store them into the list with an insert function. The nodes in the network trying to get authenticated have to do the same. After that they can generate a random number and use it as index to get a random K_i from the list. As for devices authenticating others, they simply receive the K_i from a node (in the first message of the

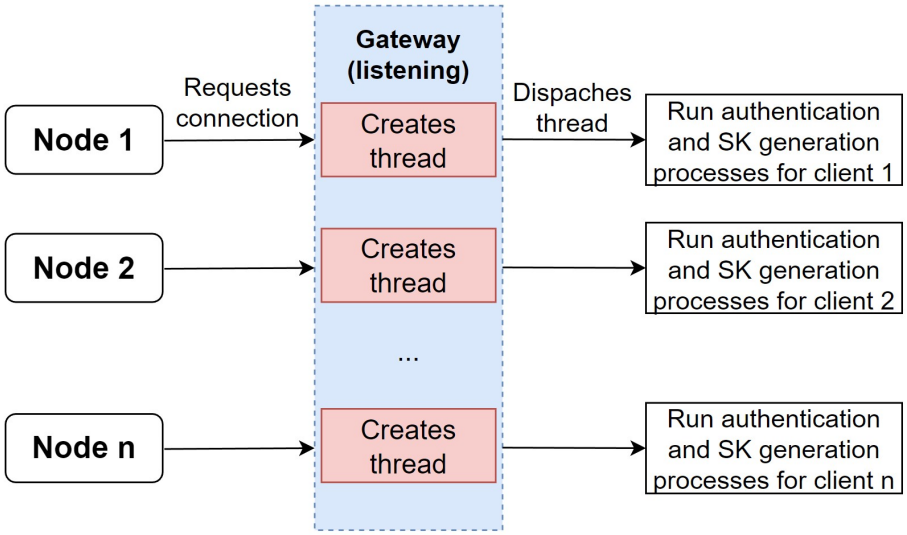

Figure 5.1: Nodes joining the network thread diagram

scheme, *Authentication message*) and searches for it in its list, saving it to encrypt further architecture messages until a session key is generated or placing the node ID in a blacklist if the key is not found in the list. Bellow, it is represented how the keys should be in the file poolOfKeys.txt, with a key per line.

```
8297166255472A9104928374FC63F2941C382C72319D4719023AF7923F59823A
912374F65129948179F3047578332143AF987921873123B90123DF128E109283
2ADB74D178263B217309850C875F62741231FB41219AC9081289DC98762A23EB
4023948294729691273 4C14033235323BA987AFF9871221ADE987D9F977D7862
03D0578BAFF3D9D912D8E3DDF59489F021AAD791D23791AB79CDD7398274CA05
```

After reading all the keys, they are placed in the list by index, like the following:

```
[(4,03D0578BAFF3D9D912D8E3DDF59489F021AAD791D23791AB79CDD7398274CA05)
(3,4023948294729691273 4C14033235323BA987AFF9871221ADE987D9F977D7862)
(2,2ADB74D178263B217309850C875F62741231FB41219AC9081289DC98762A23EB)
(1,912374F65129948179F3047578332143AF987921873123B90123DF128E109283)
(0,8297166255472A9104928374FC63F2941C382C72319D4719023AF7923F59823A)]
```

The encryption algorithm is a very important aspect in Light-SAE. TinyJAMBU 256 bit version from the NIST lightweight cryptography competition was used in our solution. Every message in it, with the exception of the first one (sent by the node seeking authentication) is encrypted with the algorithm. For this, devices use the function encrypt, inputing the plain text and receiving the cipher text. Devices can then send this cipher text to the other one as well as the cipherSize variable. This variable can reveal the size of the plaintext since its always 8 more bytes than the plaintext size. However, since the other device will need this variable to correctly use the decrypt function and it has no way to calculate it with the cipher alone, we have to send it over. The Diffie-Hellman prime values G and P are all hard coded on all devices meant to join the network just like

the default network key. For the session key generation process, these values must be the same in both devices, otherwise the calculation of the session key will fail. Finally, each device must have the digital signature algorithm to be used in order to sign and verify the messages in the scheme.

5.2 Main functions

In this subsection it is presented the main functions used in the solution. The gateway program has less functions and a relatively smaller code size compared to nodes, since these could act as nodes requesting authentication, or nodes authenticating other devices. However, as expected in such architectures, the gateway still does the more demanding computations and processes while saving information from all nodes. The gateway also has to deal with all the data collected by the nodes by processing and storing them.

authenticationResponse - This function is used by already authenticated devices. It receives the first message, the *authentication message* (containing the K_i and node ID) from nodes requesting authentication. It starts by finding the K_i in its key pool or placing the node in the blacklist if the key is not valid. If the key K_i is valid, then it proceeds to calculate the challenge. Firstly it generates a random 256 bits hexadecimal string $R1$, that is then XORed with the default key. Finally, it generates a random 256 bits hexadecimal string $R2$ (value X) and concatenates it with the previous XOR, obtaining the challenge. After this, the node encrypts it and sends to the node requesting authentication.

respondingChallenge - This function is used by nodes and consists in the necessary proceedings to answer the challenge correctly. It receives the challenge and splits into parts. The second part simply has to be decrypted in order to obtain $R2$. As for the first part, the node has to XOR it with the default network key in order to obtain $R1$, which was generated by the node authenticating it. After both of these processes are done, the node concatenates $R1$ and $R2$ and encrypts it to send it back.

checkingChallenge - As the name suggests, this function verifies if the challenge was correctly answered. It's a function used for devices authenticating nodes. It receives the challenge response by the node and consists in simply comparing the $R1$ and $R2$ sent by the node, with the $R1$ and $R2$ previously generated by itself. If it's incorrect the node is inserted in the blacklist. This function is slightly different from the gateway and the nodes. If the challenge was correctly answered and the gateway is the one authenticating it, this function calls for the calculate public value function described next. However, if a node is the one authenticating the other node, it generates a *request to gateway message* consisting of the K_i used and the ID of the node requesting authentication so it can send it to the gateway.

calculatingPubValue - This function receives as input a *request to gateway message* in a multi-hop scenario, or *public value message* in single-hop scenario. This function

uses g and p parameters as well as the private key to generate the public value meant for the other device. As the calculations require the operation of power with relatively big numbers, we use modular exponentiation to do the calculations. The output of this function is the public value, so the other device can use it to calculate the session key.

generateSessionKey - Like the previous function, this one executes the same calculations, but instead of using g and p to calculate the public value, it uses the public value from the other device to calculate the correct session key. Like before, modular exponentiation is used for the operation, obtaining a 256-bit key.

5.3 Message structure

The message structure consists on 7 variables: messageType, senderID, ID, cipherSize and cipher, signature and signaturePublicKey, as shown bellow.

```
struct message_struct {
    unsigned int messageType;
    unsigned char senderID[6];
    unsigned char ID[6];
    unsigned char signaturePublicKey[33];
    unsigned char signature[64];
    unsigned int cipherSize;
    unsigned char cipher[CRYPTO_BYTES];
} message;
```

The last two variables consist of actual data meant to be sent to the other node and its size, which is needed to correctly decrypt the message. The variable $senderID$ is used so the gateway knows who sent the message. For example, if an authenticated node A with the ID 12345 wants to send a message to the gateway, the ID 12345 will go on that parameter. Once the gateway sees who is sending the message, it acts accordingly (authenticates the node, stores data, or generates public value for a node authenticated by an intermediate node). The variable ID is used for nodes out of range of the gateway to send their own ID, so the gateway can compare that ID with the one that the intermediate node sent (node that authenticated the out of range node). Next, the $messageType$ is used as mentioned previously on the message treatment section, so the gateway can see what type of message it is receiving so it can use the correct key to decrypt it. Finally, the $signature$ is the signed message used so the other device can verify the sender and the $signaturePublicKey$ is the public key generated along with the private key by the node meant to be used by the gateway when verifying the signatures.

5.4 Session keys storage

The gateway will generate session keys with all the nodes on the network. For future communication, it also has to keep them organized so it knows which one to use later to decrypt messages from every node. To do this, a linked list is used. After generating the session key with the node, it also gets stored in a linked list as well as the node ID and a timestamp, with the following structure:

```
struct node {
    long int timestamp;
    char* data;
    int key;
    struct node *next;
};
```

The timestamp is used to keep track on how long the session key was generated. After pre-established time, if a node does not initiate the session key renewal process, the gateway can initiate it or remove the node from the network if it keeps not responding. When the gateway receives a message from a node, it checks the node ID and searches for the respective session key in the list. After the session key is renewed, the gateway simply replaces the old session key with the new one in the list.

5.5 Blacklist

Just like the liked list with the session keys, the blacklist works the same way but with only one parameter, the node ID. When a node sends an *authentication message* to join the network, the first thing the device that is authenticating the new node does is check if the node ID is in this blacklist. If it is not in the last, it proceeds with the next checks. A node can be placed in the blacklist for 3 reasons:

- If the K_i the node sent does not match any K_i in the file poolOfKeys.txt;
- If the challenge response is not correct;
- In the scenario where a node A authenticated a node B, when the ID (of the node authenticated B) sent by the node A to the gateway does not match the ID that the node B sent to the gateway itself.

Once a node is in a device blacklist and tries to join the network again, the device notices and stops communication with it immediately.

This way, each node and the gateway will have its own blacklist, meaning that if a malicious node tries to join the network via a node A and fails to do so, it can try to join

via another node since they did not share the blacklist. In summary, a malicious node can try to join via every node in his proximity a single time. This is not a problem since the computing processing needed to place the node in the blacklist is similar to checking if he is in the blacklist. However, it could make sense to have a shared blacklist in some WSN with lots of nodes close to each other as long as the complexity of the update and fault tolerant mechanisms on the blacklist are not too high.

5.6 Session key renewal

Both devices should know when to renovate the keys. For this, once the session key is generated, so is a timestamp (timestamp 1) that will get stored together with the key. In order to renewal the key, a separate thread will be running with a function called "timeToRenew" where it will spend most of its time sleeping. However, from time to time, it will create/update a new timestamp (timestamp2) in order to check if it is time to renew the key. For example, if the time set for the sleep is 5 seconds, and the key must be renew every 20 seconds, the thread will wake up after 5 seconds and do the following calculation: timestamp2 - timestamp1. When this result is superior or equal to 20 seconds, the session key renewal operation is called as it is possible to see below.

```
void* timeToRenew(void *sockt){
  gettimeofday(&timestamp,NULL);
    while(1){
        sleep(5);
        gettimeofday(&timestamp2,NULL);

        int dif = timestamp2.tv_sec - timestamp
        .tv_sec;

        if(dif >= keyRenewalTime){
          Key renewal calculation
          ...
        }
    }
}
```

As mentioned before, if the node does not initiate this process for any reason, the gateway will send a *key refresh time message* to the node in order for it to initiate the process. Just like the thread explained above, the node also has one that waits for this message from the gateway that triggers the key renovation.

5.7 SchnoorQ

In our solution, we choose the SchnorrQ digital signature algorithm from the FourQ elliptic curve for our implementation. As mentioned in a previous section, the node is the one

generating the private/public key pair that will be used for the process, to do so, it uses the $FullKeyGeneration$ function. It starts by generating a random 32-byte private key $SecretKey$, after that, it calculates the corresponding 32-byte public key $PublicKey$ and sends it to the gateway in the *public value message*.

```
SchnorrQ_FullKeyGeneration(SecretKey, PublicKey);
```

After the correct device meant to receive the node messages has the $PublicKey$, the node can start signing messages with the $Sign$ function. In order to generate the Signature, the function should receive as inputs both keys, the specific message and its size.

```
SchnorrQ_Sign(SecretKey, PublicKey,
Message, MessageSize, Signature);
```

Finally, once the signature reaches the correct device (gateway), it can use the $Verify$ function that receives the $PublicKey$ (sent form the node), the Message (that was already decrypted by the gateway) and its size, the signature and a variable "Valid" that will be used to easily check if the verification was successful.

```
SchnorrQ_Verify(PublicKey, Message,
MessageSize, Signature, &valid);
```

SchnorrQ uses a hash function to compute a digest of the message and calculate its signature, however, as we are using an encryption algorithm, it does not make sense to calculate the hash of the encryption, specially if the messages are small. So, in order to reduce overhead, using the encryption instead of the hash to calculate the signature is a good approach.

Chapter 6

Security Analysis

In this chapter, a security analysis of the methods used in our solution is discussed, such as the difference between normal Diffie-Hellman and our modified version of it as well as how it prevents some known attacks. Finally, we show the effectiveness of the generated encryption keys considering their size and how they are calculated using the brute force attack.

6.1 Modified Diffie-Hellman

Diffie-Hellman is a key exchange algorithm that does not provide confidentiality. As such, it is usually used within other protocols. The values calculated within this protocol are shared over unreliable networks where an attacker could be listening and therefore, modify them. In order to achieve one of the most important security aspect, confidentiality, all parameters calculated in DH protocol are sent encrypted in our scheme as mentioned previously. The encryption helps prevent MIM attacks, which will be explained in more detail later. In Table 6.1 it is possible to see the main differences in the original DH, our version of it and the method proposed in [4] called SMDH aforementioned in the Related Work section.

Parameter	Diffie-Hellman	Light-SAE	SMDH
Asymmetric	Yes	Yes	Yes
Energy efficient	Yes	Yes	Yes
Confidentiality	No	Yes	Yes
MIM attack	Yes	No	No

Table 6.1: Modified DH differences

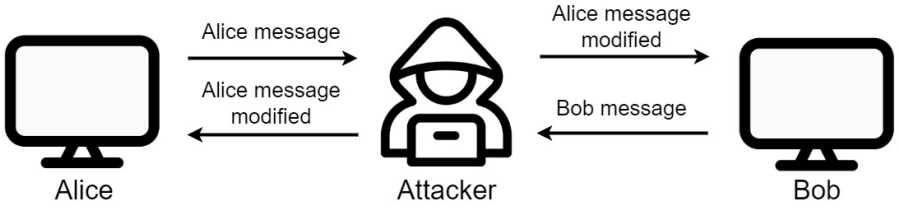

Figure 6.1: Man-in-the-middle attack

6.2 Attacks analysis

A more detailed discussion about various types of attacks and how our solution prevents them is presented in this section.

6.2.1 Man-in-the-middle attack (MIM)

MIM is a known attack where usually, as the name suggests, an adversary acts as the middle man in a "conversation" between two parties while both are unaware of its existence. The adversary has the ability to intercept messages and modify them for personal advantage. As seen in Image 6.1, Alice is trying to send a message to Bob, however the attacker intercepted it and modified the contents of it. After that, it can do the same with Bob messages. This way, an outsider has entire control of the conversation as well as the data that goes through it. This type of attack has severe impact on the network as the content of the messages may contain confidential information [37]. As mentioned previously, DH is not protected against this attack. As both parties need to send their own public Values to each other in order to calculate the key, the attacker in the middle could intercept Bob public value, calculate his own (attacker has to have the G and P value for this) and send it to Alice, generating a key between Alice and the attacker. However, in our scheme we encrypt this information with a key K_i. Only if the attacker has access to the network key, will he be able to eventually figure out what K_i will be used between devices before generating the session key, making him able to intercept, decrypt and modify the contents of the message (like the public value sent) before the SK generation process.

6.2.2 Brute force attack

Brute force is one of the most well known attack which relies on computing power rather then cleverness of the attacker [16]. This attacker uses trial-and-error to find passwords or encryption keys. In other words, the adversary tries every possible key combination until the correct one is found. Depending on the size of the key, the time needed to successfully discover the key may be seconds up to years.

Let us consider that a machine or group of machines could run through 2^{30} (1.07 billion) keys per second (this number varies depending of how powerful the machines are

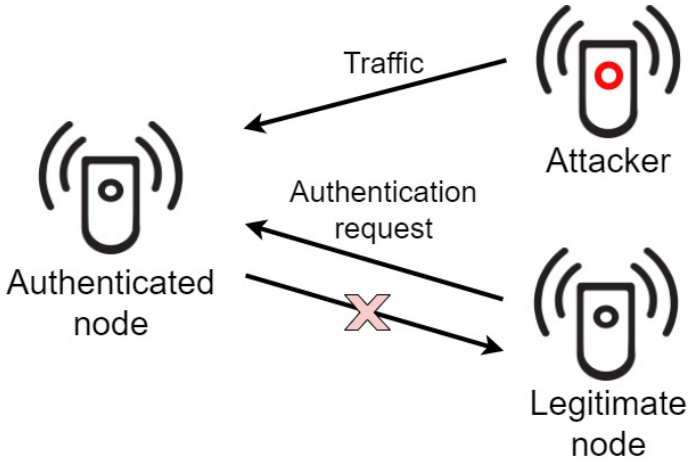

Figure 6.2: Denial-of-service attack

and the number of cores being used together in order to find the right key). In order to see how many seconds it would take to brute force a key depending of its size, we can use the following equation 6.1:

$$BruteForceTime(s) = \frac{2^{KeySize(bits)}}{2^{30}} \tag{6.1}$$

If we calculate the time needed for a 256-bit key, which is the key size used by AES256, it would take 2^{256-30} seconds or approximately $3.42 * 10^{60}$ years to discover the key. This value however is obtained by calculating every possible key, meaning it represents the maximum time it would take for a key of such size. The average would be half of the time since the key could be either on the lower half or higher half of the key space. This means that in average, it would take 2^{226-1} seconds or approximately $1.71 * 10^{60}$ years since each additional bit in the key size doubles the key space. To put it into prospective, the lifetime of the universe is around 13.7 billion years. This means that it would take $1.24 * 10^{50}$ times that amount on average to find the correct key through brute force attack.

6.2.3 Denial of service (DOS) attack

In DOS attacks, the adversary floods the target with traffic and information in order to cause a crash. In both scenarios, the main goal is to deprive and refuse a service from the target to legitimate users.

In Figure 6.2 it is possible to see a scenario of this attack in our implementation. Because of the traffic that the attacker is “spamming” the authenticated node, it can not respond to the authentication request from a legitimate node trying to join our network. However, like previously mentioned in our scheme, the blacklist prevents this attack to a level. As the attacker will not be able to authenticate correctly, it will be placed in the

blacklist, thus any messages sent by him will be ignored. This message verification takes very little computational power so it will be hard to affect the normal functionality of a node or the gateway.

However, there is a known problem with this method. A possible attack that could be executed is changing the ID from the malicious node multiple times. This way, the blacklist would be useless. We thought about some solutions involving variables characterizing each device precisely, like the MAC address. However, if the authenticating device does not have any confirmation of which of these are valid or invalid, the problem would persist since there is no way of confirming them.

A simple solution we thought of could be the usage of a safelist. The network administrator would specify which MAC (or other variables) are valid to join the network in that list. If this approach was chosen, the first verification by the authenticating devices would be to check if the variable sent by the node is in the safelist, discarding him otherwise. It would be necessary for an administrator to decide which nodes could join the network before deployment. However, this method brings some disadvantages, such as lost scalability, unless this list is modifiable and updated on devices while the network is running.

Christos Gkountis *et al.* proposes a lightweight scheme [36] which is based on a set of rules to efficiently characterize packets sent to a network switch and distinguish them as malicious or legitimate. These rules are used to decide if traffic should be forwarded or dropped. The authors evaluated their proposal in terms of bandwidth consumption, number of entries in the flow table and CPU utilization. Besides very brief peaks on the CPU, the proposal proves to be a good method to mitigate this issue and maintain the lightweight aspect of such networks.

6.2.4 Nodes being curious

In these types of networks, where some nodes have important roles such as authenticating nodes and labeling them as “authentic”, there is sometimes the worry of cases where nodes are “curious”. In our solution, only nodes who were authenticated by the gateway or an authenticated node are given the title of being authentic. After this process is completed, the nodes remain operational without any sort of interruption like an update. With this, we make sure the node was in no way altered by any outsider, and it is in fact, since it joined the network, one of our authentic nodes.

It is important to notice that even if an authenticated node is able to authenticate others, they are never supposed to generate the session key with them. That is solely the job of the gateway. However, since the key generation process messages goes encrypted with the K_i, the authenticated node could very well decrypt it, and change the parameters at will, acting itself as the middle man. Like it was stated before, as long as the network makes sure a node does not leave the network and rejoins as authentic, or stops for an update

(since it could be modified by an outsider), a node will not modify messages meant for the gateway from other nodes.

6.3 Security level of the generated keys

As mentioned in [40], in Diffie-Hellman, the difficulty of "cracking" or in other words, finding out the shared key that was established through the exchange, scales directly with the size of the primes used. However, one additional bit on the primes is not equal to one additional bit of "key strength" in AES keys for example. The reason behind it is because not every number in that interval is a prime. If we take a look at AES-256 for example, we expect the key to be a string of 256 random bits, meaning that the key could be any of the 2^{256} possibilities. Yet, if we use 256-bit primes in DH means, there will be much fewer than those 2^{256} possibilities. Therefore, if we aim for the same "key-strength" as any AES versions, we need to use larger primes, so that we have a number of possible primes is in the range of that AES version.

As we want to maintain the lightweight aspect of our solution and use NIST encryption algorithms, we use small primes with 256-bits in order to obtain a 256-bit key. However, as shown in the calculations bellow, even at this size, the scheme is secure for a good amount of time. If the key is calculated with an 256-bit prime, that means we are working with values in the "realm" of 2^{256} or approximately 1.2×10^{77} possible numbers. Among all these values, we can do an estimate calculation of how many primes there are using the Prime Number Theorem [65] in equation 6.2.

$$\pi(x) = \frac{x}{\log_2 x}. \tag{6.2}$$

There are approximately 1.5×10^{75} or 2^{250} possible primes for the calculation of the public value. If we estimate how long it would take to brute force such a value with a rate of 2^{30} trials per second, it would take approximately 2^{220} seconds or 5.3×10^{58} years to discover the secret, which is an unimaginable amount of time.

Brute force is obviously not the best attack for methods with large numbers and there are much effective attacks that could possibly reveal the secrets way faster, however it is a good method to have a basic understanding of how secure the keys are and compare them to other known schemes. That is why Light-SAE offers a lightweight key refresh mechanism that can be adjusted to any network so there is no way of using such attacks in the time interval that the key is being used.

Chapter 7

Results

The proposed solution and the algorithms were tested in 2 different machines: a Desktop with a six-core i5-12400F CPU @2.50GHz, NVIDIA GeForce RTX 3070 8GB and 16GB of RAM DDR4; and on a Raspberry Pi 3 Model B+ with a 1.4GHz 64-bit quad-core processor. This chapter presents the results obtained from our tests. It is possible to see results of our solution like execution times of all operations such as: Authentication request, challenge generation, challenge response, challenge verification, calculation of DH public values and session keys, along with the processes used in the digital signature algorithm. Energy consumption of the architecture in single-hop and multi-hop communication is also presented. Finally a detailed comparison of our solution is presented with several other similar schemes.

7.1 Encryption algorithms

Bellow, there is a list of variables that both algorithms have as default. Some of these can be tuned to fit personal usage.

- CRYPTO_KEYBYTES: 16 - Allows for keys of 32 bytes in both algorithms. TinyJAMBU does not allow higher values. However, increasing this value on ELEPHANT works just fine, changing the ciphertext (size stays the same) and increasing execution time.

- CRYPTO_NPUBBYTES: 12 - Allows for nonces up to 24 bytes in both algorithms. While on TinyJAMBU, the increase of this value does not change the resulting ciphertext, on ELEPHANT, this value can be increased to a maximum of 20, allowing the usage of nonces 40 bytes long. The authors of ELEPHANT recommend leaving the variable since it could lead to errors.

- CRYPTO_ABYTES: 8 - Allows up to 8 bytes of associated data in both algorithms. The default value of this variable can be increased from 8 to 128 bytes, but the

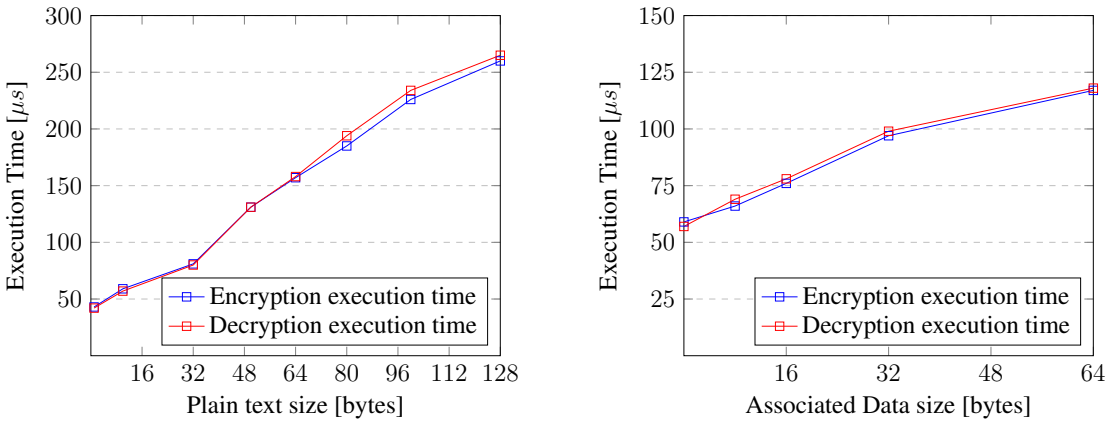

Figure 7.1: TinyJAMBU average execution time dependence of Plain text and Associated Data size in Raspberry

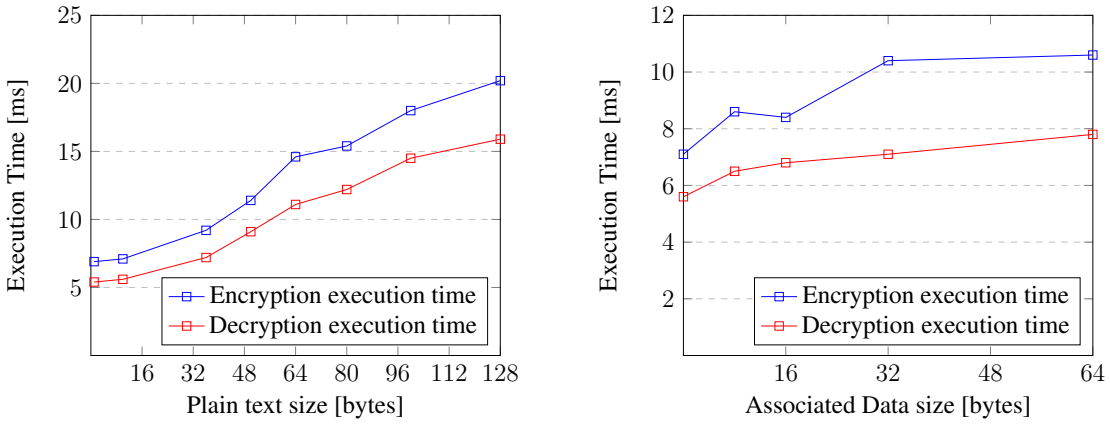

Figure 7.2: ELEPHANT average execution time dependence of Plain text and Associated Data size in Raspberry

algorithms will increase the execution time. If associated data increases above the default value, the cipher size increases when the ELEPHANT algorithm is used.

In Figure 7.1 it is possible to see how the plain text and associated data size affects the execution time on TinyJAMBU in both encryption and decryption processes. The same tests can be seen in Figure 7.2 for ELEPHANT. The results were achieved by calculating the average of ten runs in each setting. Both algorithms come with a default maximum plain text size of 64 bytes. However, it can be changed, and as shown in the graphics, it was tested from 1 to 128 bytes. Both algorithms end up with the same size of cipher text, always adding 8 bytes to the original plain text size. TinyJAMBU presents much better results, taking around 150 microseconds to encrypt a message of 64 bytes while ELEPHANT takes 12 milliseconds. In this case, TinyJAMBU is about 80 times faster than ELEPHANT. This could be due to TinyJAMBU being targeted explicitly to hardware,

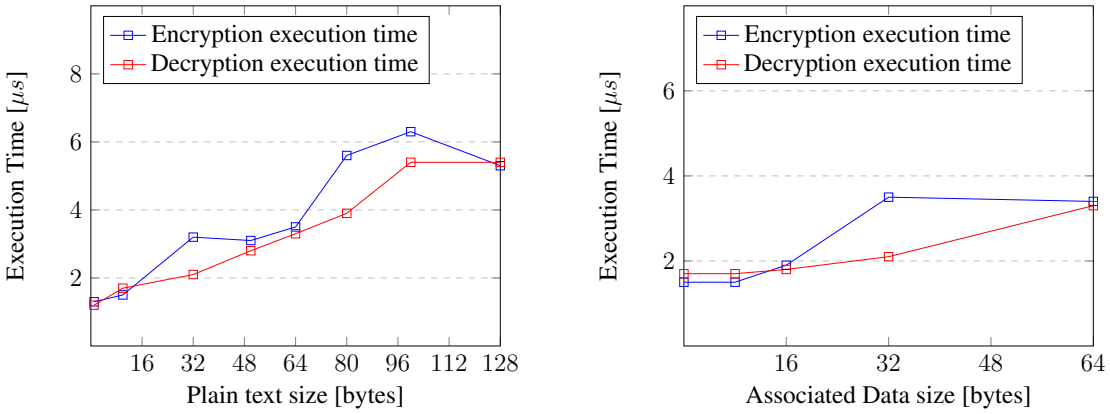

Figure 7.3: TinyJAMBU average execution time dependence of Plain text and Associated Data size in Desktop

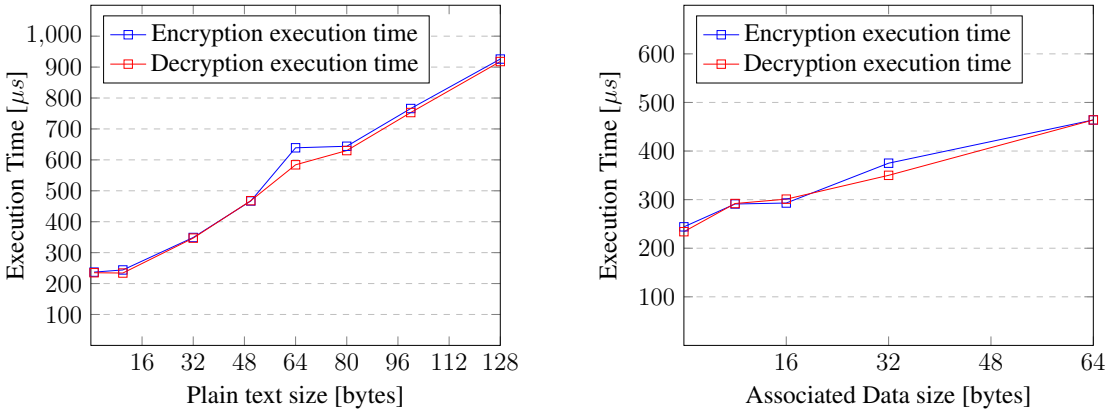

Figure 7.4: ELEPHANT average execution time dependence of Plain text and Associated Data size in Desktop

while ELEPHANT is targeted to both software and hardware. The estimated execution time (ExTime) on a Raspberry 3 of TinyJAMBU can be calculated depending on the plain text on Equation 7.1 or depending on the associative data size on Equation 7.2. Their coefficient of determination (R-Squared) is also presented:

$$ExTime(\mu s) = 1.6 \times PlainTextSize + 50, R^2 = 0.9909 \tag{7.1}$$

$$ExTime(\mu s) = 0.9281 \times AD + 60.725, R^2 = 0.9732 \tag{7.2}$$

Tests with the nonce showed that TinyJAMBU does not allow a nonce higher than the default 24 bytes, while in ELEPHANT, this variable can be increased up to 40, but a decrease in performance should be expected. Overall, nonce does not affect the execution time if used as intended by the authors, from 1 to 24 bytes.

The algorithms allow up to 8 bytes of associated data as default. If this variable is kept that way, it does not influence execution time on ELEPHANT as it does in TinyJAMBU. TinyJAMBU allows up to 128 bytes of associated data, but ELEPHANT only allows up to 64 bytes. Interestingly, especially on Raspberry Pi, the execution time increases proportionally to the AD in TinyJAMBU. However, it increases logarithmically on ELEPHANT. The estimated execution time on a Raspberry 3 of ELEPHANT can be calculated depending on the plaintext in Equations 7.3 (encryption) and 7.4 (decryption) or depending on associated data in Equations 7.5 (encryption) and 7.6 (decryption):

$$ExTimeEnc(ms) = 0.1133 \times PlainTextSize + 6.2095, R^2 = 0.9794 \tag{7.3}$$

$$ExTimeDec(ms) = 0.0902 \times PlainTextSize + 4.8346, R^2 = 0.9819 \tag{7.4}$$

$$ExTimeEnc(ms) = -0.0012 \times AD^2 + 0.1326 \times AD + 7.1738, R^2 = 0.9307 \tag{7.5}$$

$$ExTimeDec(ms) = -0.0005 \times AD^2 + 0.0628 \times AD + 5.7985, R^2 = 0.9435 \tag{7.6}$$

The same tests were conducted on a more robust machine as mentioned above. The results can be seen in Figures 7.3 and 7.4. The execution times decreased as expected since it is a more powerful machine with more resources. However, the difference in numbers between both algorithms did not. Their difference increased compared to the results on the Raspberry. TinyJAMBU seems to be around 200 times faster than ELEPHANT to encrypt/decrypt a message 64 bytes long.

Besides the two algorithms used to implement our solution, we also evaluated some of the other NIST finalists such as: ASCON, GIFT-COFB, Grain-128AEAD, ISAP, Romulus and SPARKLE. We ran the algorithms 10 times in order to obtain the average of those runs, using a plaintext of 32 digits and 128 bits key, measuring the execution time of the encryption and decryption process along with a representation on how bigger the cipher text is compared to the original plain text. Table 7.1 presents the results in microseconds on the Desktop while Table 7.2 presents the results in microseconds in the Raspberry 3.

Algorithm	Encryption	Decryption	Cipher Size
TinyJAMBU	3.1	2.9	+8
ELEPHANT	479.3	478.6	+8
ASCON	94.8	37.9	+16
Grain-128AEAD	359.8	328.1	+8
ISAP	36	14.8	+16
Romulus	145.4	131.7	+16
SPARKLE	7.6	4	+16

Table 7.1: NIST LWC algortihms execution times(μs) in Desktop

Algorithm	Encryption	Decryption	Cipher Size
TinyJAMBU	97.3	95.1	+8
ELEPHANT	13284.2	12940.9	+8
ASCON	1458.5	1139.2	+16
Grain-128AEAD	9469.7	8805.7	+8
ISAP	2376.7	1551.2	+16
Romulus	4387.4	4100.1	+16
SPARKLE	85.4	76.8	+16

Table 7.2: NIST LWC algortihms execution times(μs) in Raspberry 3

7.2 Proposed Solution

In this section, we present all the results of tests made on the network such as execution times of Light-SAE and the digital signature algorithm and energy consumption. The simulation setup is also described as well as the network settings used.

7.2.1 Environmental setup

The proposed approach was simulated in 2 different scenarios: Communication between nodes and the gateway directly, and communication between nodes and the gateway with

the help of an intermediate node. All the results presented were calculated by making 10 nodes join the network until session keys were established in order to calculate the average execution time of all processes. Encryption, decryption and times until the messages are successfully received by the devices were ignored for these experiments. In order to get as close as possible to the scheme execution times, both programs (gateway and nodes) were run in isolated cores in a virtual machine with 4 cores on the Desktop, and in the Raspberry Pi 3. Core 2 was dedicated only for the gateway, while core 3 was only for nodes.

7.2.2 Network

During the implementation of the architecture, all communication was made using Transmission Control Protocol (TCP/IP) sockets. The gateway socket listens on a particular port at an Internet Protocol (IP) address, while other socket (nodes) reaches out to the other to form a connection. Function read and write are used to send and receive the message over the network. To simulate nodes connecting directly to the gateway, we use the gatewayPort. For nodes out of range of the gateway, we connect them to other nodes using the port nodePort (that already authenticated node can be listening and waiting for connections). The reasoning behind it is to test the architecture processes in a multi-hop communication, by setting the specific ips that each client must connect itself to. For example, a client 1 will communicate with the gateway and get authenticated by it while a client 2 will forcibly be connected and authenticated by an authenticated node.

7.2.3 Execution time

To measure the total time of our solution, we measured each process separately: Authentication request, challenge generation, challenge response, challenge verification, calculating public value and session key generation in both single-hop and multi-hop scenarios.

In Figures 7.5 and 7.6 it is possible to see the execution times of all the processes on the node and the gateway since the beginning of the authentication until the establishment of the session key. This demonstrates the results in the node to gateway direct communication scenario, meaning no intermediate node takes part during the whole process. As predicted, because of the difference of resources in both machines, the execution times on the Desktop are much lower than in the raspberry. The most demanding process is by far the calculation of the public value and signature key pair generation on the recently joined node. The reason for this is because the signature keys take a long time to generate as it will be shown in more detail later on. The total execution time are 169 microseconds on the Desktop and 2600.7 microseconds (2.6 milliseconds) in the raspberry.

In Figure 7.7, 7.8 and 7.9 we present the gateway and nodes execution times of the scheme in a scenario where an intermediate node authenticates the node trying to join

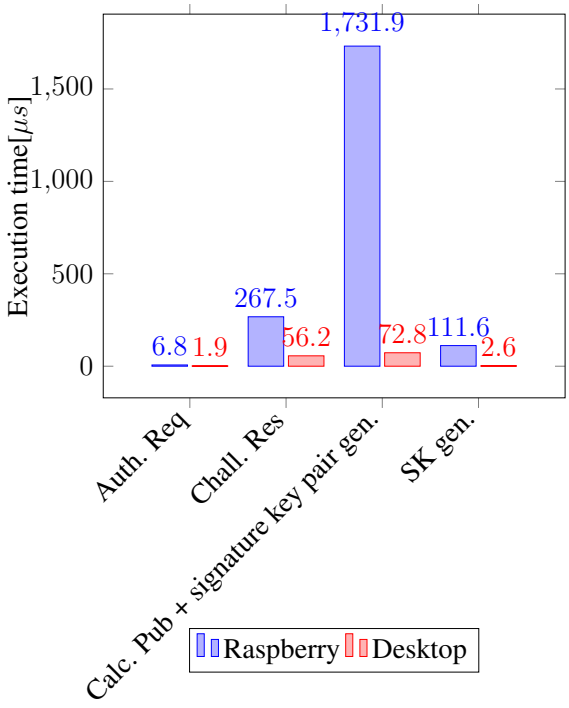

Figure 7.5: Node processes execution times in single-hop communication

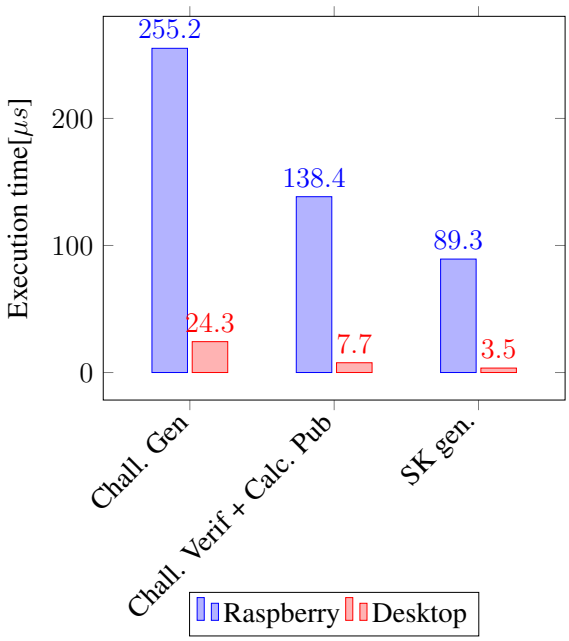

Figure 7.6: Gateway processes execution times in single-hop communication

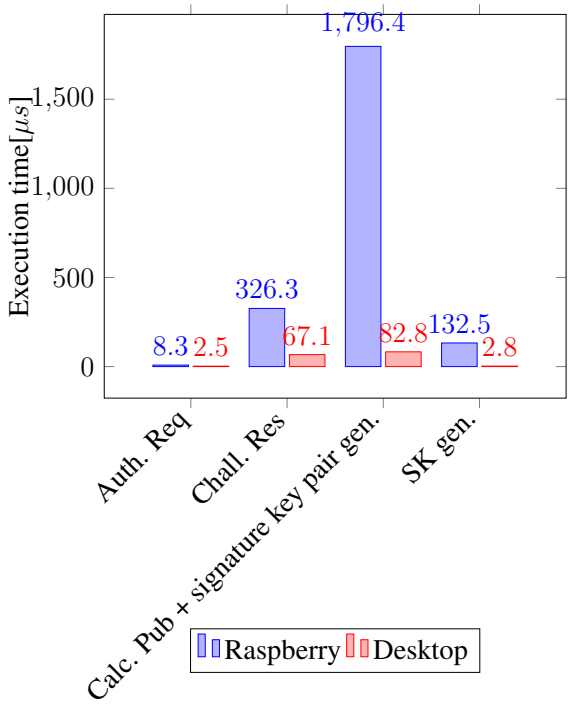

Figure 7.7: Node B processes execution times in multi-hop communication

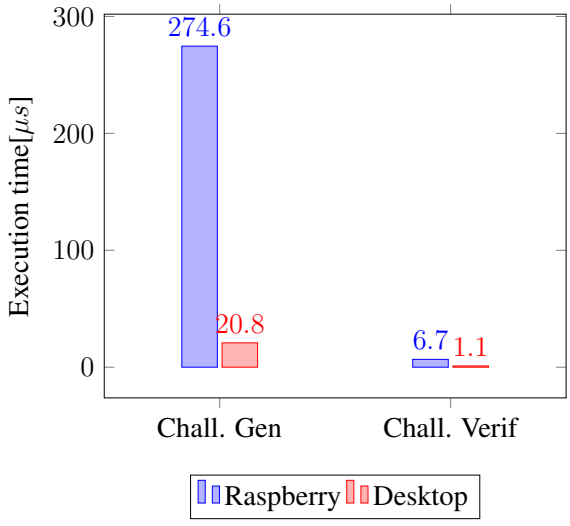

Figure 7.8: Node A processes execution times in multi-hop communication

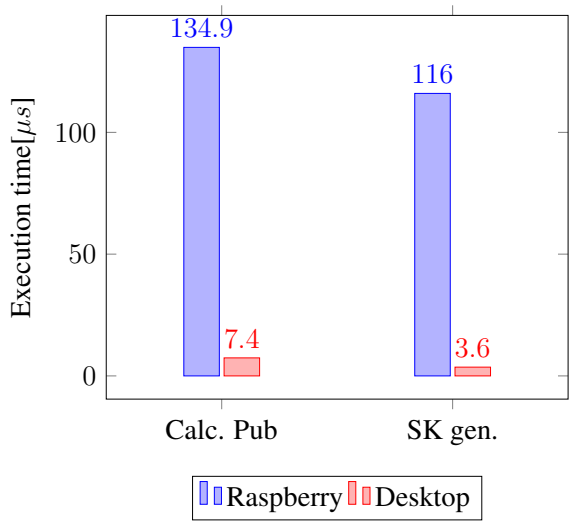

Figure 7.9: Gateway processes execution times in multi-hop communication

and later helps redirecting messages so node B can communicate and generate a session key with the gateway. The purpose of using an intermediate node is not only to allow devices further away from the gateway to communicate with it, but also to alleviate some computational overhead on the gateway. As shown in the graphics, in this scenario the gateway only has to worry about two processes (calculating public value and session key generation), reducing the possible overhead in networks with a high number of nodes. The total execution time are 188.1 microseconds on the Desktop and 2795.7 microseconds (2.8 milliseconds) in the raspberry.

7.2.4 Session key renewal

As mentioned previously, session key renewal will consist on the node generating a new Diffie-Hellman public value with a new random private key in order to calculate a new session key. This means that the node will have to do those two operations, while the gateway simply has to calculate the session key after receiving the public value from the node.

7.2.5 Energy consumption

Idle			Direct node			Indirect node		
Average	Min	Max	Average	Min	Max	Average	Min	Max
2.87	2.70	3.00	2.89	2.70	3.10	2.93	2.70	3.10

Table 7.3: Power measurement in Raspberry PI 3 B+ (W)

To measure the energy consumption of our solution, we ran it on the Raspbery Pi 3 B+. We took data from 2 different scenarios: A node communicating directly with the

gateway, sending periodic messages every 10 seconds and renovating the session key every 60 seconds. An intermediate node doing the proceedings described above (for itself) as well as re-transmitting messages from another node back to the gateway. In Table 7.3 it is possible to see the power measurements in watts (average, minimum and maximum) of both scenarios, as well as the idle state of the device. Unfortunately, with the hardware available to perform the tests, we could only record decimal variations in the value. However, by doing the average of the values obtained over a period of time, we could get a closer guess of what the device spends in each scenario over time. As expected, the solution presented spends little energy, only around 0.03W per second, or 30 milijoule in the described scenario. We were also expecting to see peaks of energy during key renovation of the nodes, however, it was not possible to see those, proving that the calculations necessary for the generation of the key are very lightweight. This allow us to decrease the interval of key renewal if necessary without energy trade off.

7.2.6 SchnoorQ

Process	Desktop	Raspberry PI 3
Key pair generation	51.4	840
Signature generation	23.8	794
Signature verification	<1	<1

Table 7.4: Schnoor processes execution time (μs)

Just like the encryption algorithms and the processes on Light-SAE, we workbenched the Digital signature algorithm used in our scheme. We ran each presented process 10 times in order to calculate the average execution time in microseconds. As it is possible to see in Table 7.4 the heavier process is the generation of the key pair taking more than twice the time compared to the Signature generation in the Desktop. However, when it comes to the Raspberry, the difference between both processes is very small. The standard deviation in the key pair and signature generation on the Desktop is 8.1018 and 5.1146 respectively and 77.37 and 97.30 on the Raspberry. The verification process takes less than a microsecond to finish on both devices, proving that SchnoorQ is a good algorithm to use in architectures where a device must verify lots of messages from various sources.

7.2.7 Schemes comparison

In Table 7.5 we compare various similar schemes to our own. Similar to what the authors on [55] did, we have scored with high, medium or low several parameters such as: Resource cost, confidentiality, integrity, scalability, key freshness, capture resistance and forward and backward secrecy. The papers [55], [41], [54], [6] and [19] were already

Scheme	Resource cost	Confidentiality	Integrity	Scalability	Key freshness	Capture resistance	Forward and backward secrecy
Light-SAE	Medium	High	Yes	High	Medium	Medium	High
Hao S. [55]	Medium	High	Yes	High	Medium	High	High
Vipin K. [41]	Medium	High	Yes	High	Medium	High	Medium
Mohsen S. [54]	Low	Medium	No	High	High	Medium	No
Kadri B. [6]	Low	Medium	Yes	Medium	Medium	Low	Medium
Chakavarika [19]	Low	Medium	No	High	Medium	Medium	No
Morshed A. [1]	High	Low	Yes	Medium	Low	Medium	Low
Yinghong L. [43]	Medium	Low	Yes	High	Low	Medium	Low
Zhou J. [38]	Low	Medium	Yes	Medium	Medium	Low	High
Bowen S. [59]	Low	Medium	Yes	High	High	Low	High

Table 7.5: Schemes comparison table

presented in the related work section as well as the reasoning behind their evaluation. As for the architectures on [1], [43], [38] and [59], they are presented and evaluated in more detail in [55]. We tried to compare all the architectures the best way possible with the information available and with a starter basis on said paper.

The reasons we scored our solution like presented are the following: For our solution, nodes only have to store at most 2 keys, the session key with the base station and a K_i used for authentication (that can later be deleted). As for energy, tests present a usage of 30 milijoule on a specific scenario described above, with a low interval of messages being sent. Execution times are also presented for all of our solution processes. On a Raspberry Pi, the longest process by far, takes arround 1.8 milliseconds. Light-SAE uses Diffie-Hellman, consisting in big operations, these take a bit more resources than schemes that distribute the keys. However, since the keys that are generated are 256 bits in size, it takes neither much time nor resources compared to other methods, having the benefit of not passing session keys through the network.

As for confidentiality, the gateway authenticates the sensors by comparing several parameters known only to valid nodes. Authentication parameters are calculated randomly by the gateway. Eavesdroppers or man-in-the-middle attacks are prevented by using encrypting the Diffie-Hellman parameters that goes through the network. Furthermore, if a key somehow got captured, it would only affect a single node since each of them has a unique session key with the gateway.

The solution presented ensures integrity of the messages with the usage of the digital signature algorithm SchnorrQ. If a message gets captured during transmission, an attacker can not modify or tamper with the contents of it without the gateway noticing since the verification will fail.

Sensors can join the network as long as the authentication process runs smoothly. Authentication may happen directly with the gateway (if near it) or using and intermediate authenticated node (if out of range of the gateway), ensuring a good degree of scalability.

The session keys get refreshed from time to time. The process should be initialized

by the node, however, since the gateway stores a timestamp when the session key was established, if a node does not initiate the process, the gateway itself will request for a session key renewal. If no response is given by the node, it gets removed from the network.

As for capture resistance, if a node shuts down, restarts, updates or gets replaced, it has to go through the authentication process. Only nodes with all the correct parameters can pass this stage. This ensures that no outsider can capture a node and takes possession of it while remaining in the network.

Finally, our solution ensures forward and backward secrecy. Sensors who disconnect, loses the session key. Once they rejoin the network, they need to be authenticated and establish a new one.

Chapter 8

Conclusion

Most devices in IoT applications do not use secure encryption or authentication algorithms in their communication. Failure to use these security methods can lead to severe consequences such as unauthorized access and manipulation of data.

The presented work focuses on the proposal and development of a Lightweight Key Distribution solution for WSN called Light-SAE. It supports multi-hop authentication and communication between resource constrained devices and a base station. Devices are able to get authenticated and generate a common key with a gateway even if out of its reach. We use the latest NIST LWC algorithms to encrypt all the data and parameters that go through the network, however any encryption algorithm can be used with our solution as long as the key size matches the ones defined, which in our case, was 256-bits. Encrypted Diffie-Hellman is used to generate the 256-bit session keys unique to every node as well as a renewal mechanism to update them after a necessary amount of time. A modified version of the known Schnorr algorithm, SchnorrQ was used to generate the signatures and the verification of them to ensure no message was modified by any outsider. With all these mechanisms we have achieved a good key distribution solution that provides authenticity, confidentiality and integrity.

We provided experimental results that prove how fast Light-SAE processes are as well as efficient in the used energy. Various tests we also made on the finalists encryption algorithms from the NIST competition in order to find the most suitable combination of variables to be used. A detailed comparison of the scheme was made with similar schemes in a diverse set of parameters proving to be a good solution for networks with various operating nodes in a vast area.

8.1 Future work

With Light-SAE, the gateway will coordinate all the key generation and the data received from the nodes. However, this could cause an overhead in the gateway in scenarios where the network has a large amount of nodes. A solution to this problem and possible future

work for this proposal would be allowing the gateway to select some nodes as cluster heads. This way, cluster heads would generate the session keys for the neighbouring nodes that are assigned to them and receive their data. The gateway would eventually receive this data stored in the cluster heads.

Nowadays, the confidentiality of data is crucial even for IoT. Therefore there are scenarios where the gateway will need to send confidential information to the nodes. In this case, as mentioned before, no integrity would be supported since in our current proposal only the nodes sign messages for the gateway to verify them. However, making the opposite also true could bring advantages to networks that require the gateway to send confidential information to sensor nodes.

Implementing the safelist, consisting of a list with legitimate nodes' information (such as their ID or MAC), would mitigate a possible attack where malicious nodes repeat the authentication with several IDs. However, we call the reader's attention to the fact that each node, after getting authenticated, would need to receive this safelist to authenticate future nodes correctly. Also, to maintain the solution's scalability, the safelist should be modifiable during the network operation and updated in each authenticated device.

Bibliography

[1] Morshed Aski. Akbar, Haj Seyyed Javadi. Hamid, and Shirdel. Gholam Hassan. A full connectable and high scalable key pre-distribution scheme based on combinatorial designs for resource-constrained devices in iot network. *Wireless Personal Communications*, 114, 10 2020.

[2] Samya Al Busafi and Basant Kumar. Review and analysis of cryptography techniques. In *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*, pages 323–327, 2020.

[3] Kurdistan Ali and Shavan Askar. Security issues and vulnerability of iot devices. 5:101–115, 02 2021.

[4] Shahwar Ali, A Humaria, M Sher Ramzan, Imran Khan, Syed M Saqlain, Anwar Ghani, J Zakia, and Bander A Alzahrani. An efficient cryptographic technique using modified diffie–hellman in wireless sensor networks. *International Journal of Distributed Sensor Networks*, 16(6):1550147720925772, 2020.

[5] Parvaneh Asghari, Amir Rahmani, and Hamid Haj Seyyed Javadi. Internet of things applications: A systematic review. *Computer Networks*, 148, 12 2018.

[6] Kadri B., Moussaoui D., Feham M., and A. Mhammed. An efficient key management scheme for hierarchical wireless sensor networks. 4(6), 2012.

[7] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift-cofb v1.1. *NIST*, 05 2021.

[8] Zhenzhen Bao, Avik Chakraborti, Nilanjan Datta, Jian Guo, Mridul Nandi, Thomas Peyrin, and Kan Yasuda. Photon-beetle authenticated encryption and hash family. *NIST*, 05 2021.

[9] Walid Bechkit, Yacine Challal, and Abdelmadjid Bouabdallah. A new class of hash-chain based key pre-distribution schemes for wsn. *Computer Communications*, 36(3):243–255, 2013.

[10] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschadl, Amir Moradi, Léo Perrin, Aein Rezaei Shahmirzadi, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang1. Schwaemm and esch: Lightweight authenticated encryption and hashing using the sparkle permutation family. *NIST*, 05 2021.

[11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Cryptology ePrint Archive, Paper 2011/368, 2011.

[12] Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. Dumbo, jumbo, and delirium: Parallel authenticated encryption for the lightweight circus. *IACR Transactions on Symmetric Cryptology*, 2020(S1):5–30, Jun. 2020.

[13] Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. Elephant v2. *NIST*, 05 2021.

[14] Andrey Bogdanov, Miroslav Knežević, Gregor Leander, Deniz Toz, Kerem Varıcı, and Ingrid Verbauwhede. spongent: A lightweight hash function. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 312–325, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[15] Carsten Bormann and Carles Gomez. Terminology for constrained-node networks (7228bis), 10 2016.

[16] Leon Bošnjak, J. Sres, and B. Brumen. Brute-force and dictionary attack on hashed real-world passwords. pages 1161–1166, 05 2018.

[17] Seyit Camtepe and Bulent Yener. Key distribution mechanisms for wireless sensor networks: a survey, 01 2021.

[18] António Casimiro. Aquamon - dependable monitoring with wireless sensor networks in water environments, 2018. Accessed: 2021-10-10.

[19] Chakavarika, Tafadzwa, Gupta, Shashi, and Chaurasia Brijesh. Energy efficient key distribution and management scheme in wireless sensor networks. *Wireless Personal Communications*, 97:1–12, 11 2017.

[20] Jaewoo Choi, Yonghyun Kim, JuYoub Kim, and Taekyoung Kwon. A study of location-based key management using a grid for wireless sensor networks. *Journal of the Korea Institute of Information Security and Cryptology*, 25:759–766, 08 2015.

[21] Craig Costello and Patrick Longa. Fourq: four-dimensional decompositions on a q-curve over the mersenne prime. 06 2015.

[22] Craig Costello and Patrick Longa. Schnorrq: Schnorr signatures on fourq. July 2016.

[23] Joan Daemen, Seth Hoffert, Silvia Mella, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Xoodyak, a lightweight cryptographic scheme. *NIST*, 05 2021.

[24] Jonathan de Carvalho Silva, Joel J. P. C. Rodrigues, Antonio M. Alberti, Petar Solic, and Andre L. L. Aquino. Lorawan — a low power wan protocol for internet of things: A review and opportunities. In *2017 2nd International Multidisciplinary Conference on Computer and Energy Science (SpliTech)*, pages 1–6, 2017.

[25] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. Isap v2.0. submission to nist. *NIST*, 2019.

[26] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. *NIST*, 05 2021.

[27] Wenliang Du, Jing Deng, Y.S. Han, Shigang Chen, and P.K. Varshney. A key management scheme for wireless sensor networks using deployment knowledge. In *IEEE INFOCOM 2004*, volume 1, page 597, 2004.

[28] Carlos Santos Fernandes. *Choosing the Future of Lightweight Encryption Algorithms*. IST, 2018.

[29] A.B. Feroz Khan and G. Anandharaj. Ahkm: An improved class of hash based key management mechanism with combined solution for single hop and multi hop nodes in iot. *Egyptian Informatics Journal*, 22(2):119–124, 2021.

[30] Amir Fotovvat, Gazi M. E. Rahman, Seyed Shahim Vedaei, and Khan A. Wahid. Comparative performance analysis of lightweight cryptography algorithms for iot sensor nodes. *IEEE Internet of Things Journal*, 8(10):8279–8290, 2021.

[31] Nick G. How many iot devices are there in 2021?, 2021. Accessed: 2021-10-13.

[32] Amit Kumar Gautam and Rakesh Kumar. A comprehensive study on key management, authentication and trust management techniques in wireless sensor networks. *SN Applied Sciences*, 3(1):1–27, 2021.

[33] Chun Guo, Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. Romulus v1.3. *NIST*, 2019.

[34] Martin Hell, Thomas Johansson, Alexander Maximov, Willi Meier, Jonathan Sonnerup, and Hirotaka Yoshida. Grain-128aeadv2 - a lightweight aead stream cipher cover sheet. *NIST*, 2019.

[35] Nicholas Jansma. Performance comparison of elliptic curve and rsa digital signatures. 05 2004.

[36] Uzair Javaid, Ang Kiang Siang, Muhammad Naveed Aman, and Biplab Sikdar. Mitigating lot device based ddos attacks using blockchain. In *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems*, CryBlock'18, page 71–76, New York, NY, USA, 2018. Association for Computing Machinery.

[37] Dr Javeed and Umar MohammedBadamasi. Man in the middle attacks: Analysis, motivation and prevention. *International Journal of Computer Networks and Communications Security*, 8:52–58, 07 2020.

[38] Zhou Jian, Sun Liyan, Duan Kaiyu, and Wu Yue. Research on self-adaptive group key management in deep space networks. *Wireless Personal Communications*, 114, 10 2020.

[39] Ravneet Kaur and Amandeep Kaur. Digital signature. In *2012 International Conference on Computing Sciences*, pages 295–301, 2012.

[40] T. Kivinen and M. Kojo. More modular exponential (modp) diffie-hellman groups for internet key exchange (ike), 2003. Accessed: 2022-04-27.

[41] Vipin Kumar, Navneet Malik, Gaurav Dhiman, and Tarun Kumar Lohani. Scalable and storage efficient dynamic key management scheme for wireless sensor network. *Journal of Ambient Intelligence and Humanized Computing*, 07 2021.

[42] Yogesh Kumar, Rajiv Munjal, and Harsh Sharma. Comparison of symmetric and asymmetric cryptography with existing vulnerabilities and countermeasures. *International Journal of Computer Science and Management Studies*, 11, 10 2011.

[43] Yinghong Liu and Yuanming Wu. A key pre-distribution scheme based on subregions for multi-hop wireless sensor networks. *Wireless Personal Communications*, 5:1–20, 11 2019.

[44] Zhe Liu, Patrick Longa, Geovandro C. C. F. Pereira, Oscar Reparaz, and Hwajeong Seo. Fourq on embedded devices with strong countermeasures against side-channel attacks. *IEEE Transactions on Dependable and Secure Computing*, 17(3):536–549, 2020.

[45] Ben Lutkevich, Vicki-Lynn Brunskill, and Peter Loshin. Digital signature, 2021. Accessed: 2022-05-05.

[46] Kamyar Mohajerani, Richard Haeussler, Rishub Nagpal, Farnoud Farahmand, Abubakr Abdulgadir, Jens-Peter Kaps, and Kris Gaj. Fpga benchmarking of round 2 candidates in the nist lightweight cryptography standardization process: Methodology, metrics, tools, and results. Cryptology ePrint Archive, Report 2020/1207, 2020. https://ia.cr/2020/1207.

[47] Abdalbasit Mohammed and Nurhayat Varol. A review paper on cryptography. In *7th International Symposium on Digital Forensics and Security*, pages 1–6, 06 2019.

[48] Bassam J. Mohd, Thaier Hayajneh, and Athanasios V. Vasilakos. A survey on lightweight block ciphers for low-resource devices: Comparative study and open issues. *Journal of Network and Computer Applications*, 58:73–93, 2015.

[49] Nagasai. Classification of iot devices, 2017. Accessed: 2021-10-28.

[50] National Institute of Standards and Technology. Lightweight cryptography, 2017. Accessed: 2021-09-24.

[51] Yasaroglu Pinar, Abduljabbar Zuhair, Alotaibi Hamad, Akcam Resit, Kadavarthi Shiva, and Abuzaghleh Omar. Wireless sensor networks (wsns). In *2016 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–8, 2016.

[52] VíICTOR ARINO PÉREZ. Efficient key generation and distributionon wireless sensor networks, 2013.

[53] Vincent Ricquebourg, David Menga, David Durand, Bruno Marhic, Laurent Delahoche, and Christophe Loge. The smart home concept : our immediate future. In *2006 1ST IEEE International Conference on E-Learning in Industrial Electronics*, pages 23–28, 2006.

[54] Mohsen Sharifi, Saeid Pourroostaei Ardakani, and Saeed Sedighian Kashi. Skew: An efficient self key establishment protocol for wireless sensor networks. In *2009 International Symposium on Collaborative Technologies and Systems*, pages 250–257, 2009.

[55] Hao Shi, Mingyu Fan, Yu Zhang, Maoyang Chen, Xingyu Liao, and Wenqiang Hu. An effective dynamic membership authentication and key management scheme in wireless sensor networks. In *2021 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6, 2021.

[56] Marcos A. Simplício, Paulo S.L.M. Barreto, Cintia B. Margi, and Tereza C.M.B. Carvalho. A survey on key management mechanisms for distributed wireless sensor networks. *Computer Networks*, 54(15):2591–2612, 2010.

[57] Saurabh Singh, Pradip Sharma, Seo Moon, and Jong Park. Advanced lightweight encryption algorithms for iot devices: survey, challenges and solutions. *Journal of Ambient Intelligence and Humanized Computing*, pages 1–18, 05 2017.

[58] William Stallings. Digital signature algorithms. *Cryptologia*, 37(4):311–327, 2013.

[59] Bowen Sun, Qi Li, and Bin Tian. Local dynamic key management scheme based on layer-cluster topology in wsn. *Wireless Personal Communications*, 103, 11 2018.

[60] Sandeep Tayal, Nipin Gupta, Pankaj Gupta, Deepak Goyal, and Monika Goyal. A review paper on network security and cryptography. *Advances in Computational Sciences and Technology*, 10(5):763–770, 2017.

[61] Vishal A. Thakor, Mohammad Abdur Razzaque, and Muhammad R. A. Khandaker. Lightweight cryptography algorithms for resource-constrained iot devices: A review, comparison and research opportunities. *IEEE Access*, 9:28177–28193, 2021.

[62] Upasana. Real world iot applications in different domains, 2022. Accessed: 2022-05-14.

[63] Hongjun Wu and Tao Huang. Tinyjambu: A family of lightweight authenticated encryption algorithms. *NIST*, 03 2019.

[64] Attila A. Yavuz and Muslum Ozgur Ozmen. Ultra lightweight multiple-time digital signature for the internet of things devices. 2019.

[65] D. Zagier. Newman's short proof of the prime number theorem. *The American Mathematical Monthly*, 104(8):705–708, 1997.

[66] Jun-Lin Zhang and Ling Nie. Energy efficiency of multi-hop communication in wireless sensor network. 01 2016.

Appendix A

Solution code

A.1 Code structure

The code can be found on GitHub at https://github.com/jmcecilio/IoT_Authentication. It is possible to see the code organization in more detail in Figure A.1. There are two different versions available: x32 for 32-bits devices; x64 for 64-bits devices. On a Raspberry Pi 3, x32 version was used while on the desktop and Raspberry Pi 4, the x64 version was used. The structure of the programs is the same for both versions, consisting in the Gateway (for the base station), Client (for nodes) and Client2 (which purposely talks to intermediate clients and not the gateway). Each program consists of a set of c and h files to present an easy to read and organized code.

The main files are *gateway.c*, *client.c* and *client2.c* which is where all the main functions of each program are for the correct operations of the solution. The *encrypt.c* file, along with *api.c* and *crypto_aead.h* are for the LWC algorithm being used, TinyJAMBU. The folders FourQ, random and sha512 are for the digital signature algorithm being used, SchnorrQ. The most important file is the *schnorrq.c* which consists of the algorithm main function presented before.

Finally, all three programs have a *main.c* which is where all the functions are called accordingly for the correct operation of Light-SAE.

A.2 Compile and Run

In order to compile each program the command "*make -B*" should be used. To run the gateway, the command "*./gateway*" should be used. For the nodes, "*./client*" or "*./client2*" must be used respectively for each type of client followed by a 5 digit ID (for example "*./client 11111*").

To test single-hop communication, the gateway should be running. After that, multiple clients can be run from the *implementationClient* program as long as different IDs are used. To test the multi-hop communication, a client from the *implementationClient*

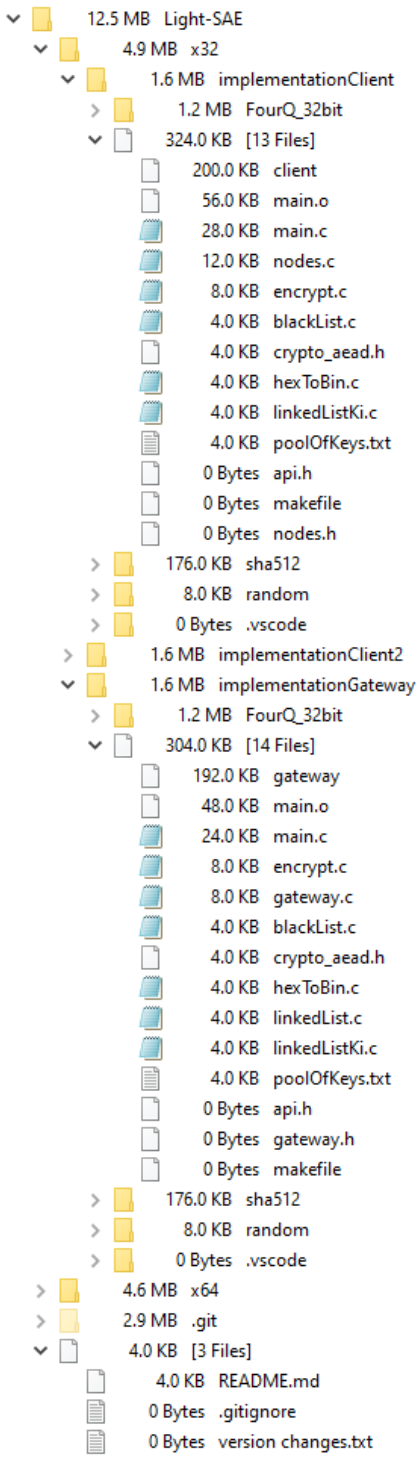

Figure A.1: Light-SAE tree size

program should be running so it can serve as an intermediate node for others. After that, multiple clients can be run from the *implementationClient2* program.

A.3 Possible experiments

While running the solution, some simple experiments can be done in order to test the implemented mechanisms.

- To make the node stop the key renewal mechanism and test the gateway requesting it instead, change the variable "*GWRenewRequestTester*" to 1 (in client or client2).
- Try stoping a working node in order to see the gateway removing him after a bit more than the *KEY_RENEWAL_TIME* since last renewal.
- Try adding an already existing node in the network to see that it wont be accepted. The existing node may rejoin the network after it is removed by the gateway.
- The digital signature algorithm may be tested by modifying the message before the verification in the gateway.