OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

FACULTY OF COMPUTER SCIENCE

Communication and Networked Systems

Communication and Networked Systems

Master Thesis

Concept for a Web-of-Trust-based certificate management in RIOT OS

Adarsh Raghoothaman

Supervisor: Prof. Dr. rer. nat. Mesut Güneş
Assisting Supervisor: MSc. Frank Engelhardt

Institute for Intelligent Cooperating Systems, Otto-von-Guericke-University Magdeburg

16. August 2022

Abstract

In the realm of Internet of Things (IoT), billions of devices communicate with each other or with other devices on the Internet. IoT devices are also generally resource-constrained in terms of memory storage capacity, processing power, network bandwidth and battery power. When the devices communicate, malicious parties can sabotage the communication and cause disruption in the whole IoT environment. Therefore, security arises as a significant factor to be considered. In standard internet, security and trust among devices are provided by the centralized Public Key Infrastructure (PKI). Digital certificates are used to establish the authenticity of devices and secure the communication. PKI depends on the Certificate Authority (CA) to issue and manage the certificates. In addition, most of the IoT environments are directly dependent on cloud platforms. The cloud provides certificates for each network device and manages the devices. The sole dependency of IoT environments on the cloud and CA can be a single point of failure in case of an attack. This affects the whole IoT infrastructure, and the security and reliability of devices are compromised. This issue can be solved by using decentralized key management solutions based on Web of Trust (WoT). Devices can issue certificates for each other in a peer-to-peer manner without depending on a centralized entity like CA. The issued certificates are validated using digital signatures. When authority is transferred from a centralized platform to individual devices in the network, possible single points of failure can be avoided. However, forming trust among the devices become challenging, when the network becomes huge and complex. We need a mechanism to implement asymmetric encryption in an efficient and scalable manner. When devices in the environment do not have the trust to communicate with another node, they should find all the certificate chains along the path for forming a trust. There is much research regarding decentralized key management solutions,still, no IoT Operating Systems have a practical implementation of the concept. In this work, we first review and compare the existing decentralized key management solutions for IoT. We then analyse the requirements for such key management solutions. Then we design and implement a concept for certificate chain discovery in RIOT Operating System (OS). Finally, we evaluate the implementation based on different parameters, total duration for certificate lookup,Central Processing Unit (CPU) overhead for processing the certificates, memory usage and communication overhead.

Contents

List of Figures

List of Tables

Listings

Acronyms

AES Advanced Encryption Standard. 20

API Application Programming Interface. 3, 27

CA Certificate Authority. iii, 2, 4, 5, 10, 13

CBOR Concise Binary Object Representation. 9, 10, 20, 21, 38, 39, 41, 42

CDDL Concise Data Definition Language. 9

CoAP Constrained Application Protocol. vii, ix, 2–7, 14, 16, 19–24, 30, 41–43

CPU Central Processing Unit. iii, 2, 43

DSA Digital Signature Algorithm. 8, 9

DTLS Data Transport Layer Security. 14, 15, 17

ECC Elliptic Curve Cryptography. 9, 20, 25, 26, 33, 34, 38, 40, 41

ECDSA Elliptic Curve Digital Signature Algorithm. 8, 9, 15, 41, 42

HTTP Hypertext Transfer Protocol. 6

IoT Internet of Things. iii, 1–5, 9–17, 32, 43

JSON JavaScript Object Notation. 9

OOB Out Of Band. 24, 45

OS Operating System. iii, 2, 3, 14, 16, 44

PGP Pretty Good Privacy. 5

PKI Public Key Infrastructure. iii, 1–5, 10, 12, 13, 19

PSK Pre Shared Key. 15, 17, 19, 20, 23, 24, 29, 30, 45

RAM Random Access Memory. 33

RSA Rivest–Shamir–Adleman. 8

SHA Secure Hash Algorithms. 20–22

SoC System on Chip. 33

TCP Transmission Control Protocol. 6, 14

TLS Transport Layer Security. 14

UDP User Datagram Protocol. 6, 14, 15, 31

URI Uniform Resource Identifier. 6, 7, 19, 24

WoT Web of Trust. iii, 1–5, 12–14, 17, 43, 44

CHAPTER 1

Introduction

The primary objective of the thesis is to develop a concept for a distributed certificate chain discovery protocol. As IoT devices become ubiquitous and the data generated by these devices are transmitted over the internet, security emerges as a significant challenge. Service failures and security breaches should be prevented by establishing trust among the communication peers. This thesis involves developing a WoT-based certificate management infrastructure for IoT devices.

1.1 Motivation

The IoT connects billions of devices together over the internet, and technology is constantly evolving. According to IHS [3] the number of IoT devices worldwide will reach 75.44 billion by 2025. Almost everything around us becomes part of IoT by connecting itself to the internet either directly or indirectly through gateways. Our life becomes more comfortable, convenient and efficient with the help of IoT. The applications of IoT span over a wide range of areas like smart homes, smart cities, self-driven cars and wearables, to name a few.

The advantages that arise from the usage of these technologies are various.IoT enables automation of many processes, improving productivity and saving time and energy. The usage of resources can be optimized, which is beneficial from an economic perspective. Even though there are several benefits that we gain from IoT, there are many challenges to be tackled.

Most devices that participate in the communication are usually resource-constrained devices with limited processing power and memory capabilities. In addition, when the devices are connected to the internet, it is susceptible to attacks, and communication can be compromised. Malicious parties can interrupt the network infrastructure, causing disruptions in the system. Thus security is an important factor to be considered with ever-growing IoT technologies and products. Forming trust among the devices in the network plays an important role in considering security.

In standard Internet, PKI is used to facilitate the secure transfer of information between two entities. Digital certificates are an integral part of PKI, and trust among involving parties is formed by exchanging and verifying certificates. Digital certificates that are issued by

a centralized CA are used for verifying the authenticity of entities participating in the communication. Furthermore, public key encryption allows secure exchange of information. Every client should trust the CA and uses the CA certificate to verify the authenticity of another client. When the infrastructure is completely dependent on CA, it could be a single point of failure.

At present, as most IoT platforms and cloud-based, the digital certificates must be stored and managed centrally via the cloud interface. The cloud platform issues the certificate for devices, which must also be registered and managed centrally via the cloud. These dependencies on the cloud platform can be a single point of failure in case of an attack. This affects the security and reliability of every device which are registered with the cloud. In addition, resource-constrained devices are not able to store many certificates because of memory constraints.

In order to overcome the above-mentioned issues, decentralized key management solutions, based on WoT can be used. In WoT, nodes issue certificates for each other in a peer-to-peer manner without relying on a third party like CA as happens in PKI. Thus a non-hierarchical network of trust is formed in the environment. When trust decisions are on nodes rather than CA, making WoT is more flexible than PKI. When the networks evolve and become larger and complex, forming trust among the devices is challenging. When there are no devices with complete knowledge of the network, we need some mechanism to form trust in a distributed and scalable way. Symmetric key encryption is also not possible when the network is large. So we need to implement asymmetric encryption efficiently.

There might be some nodes in the network, which have complete information about another node. These nodes can directly form trust without the help of another party. However, establishing secure communication will be difficult when nodes have no direct trust. A node needs to find all the certificates that are along the path and verify the certificates for forming a trust.

Even though there is research on decentralized solutions for key management based on WoT, no current IoT operating system offers a practical and complete solution for the communication protocols as well as the implementation of the infrastructure for WoT based key management. This thesis proposes and implements a certificate chain discovery protocol for encryption and authentication. The certificate chain discovery problem is considered like the service discovery problem, and the CoAP service directories are used for solving the problem. The implementation runs on RIOT, an OS for the IoT. The main contributions of this thesis are:

- Review and discussion of WoT solutions for IoT .
- Design and specification of a protocol for certificate chain discovery based on CoAP.
- Implementation of the certificate chain discovery protocol in RIOT.
- Evaluation of the implementation based on the following parameters.
  - Total duration for certificate lookup
  - CPU overhead for processing the certificates
  - Memory usage of the implementation
  - Communication overhead

1.2 Thesis Structure

The remainder of this work is structured as follows: Chapter 2 provides background information on PKI, WoT, CoAP and digital certificates. In addition, an overview of the related works is also discussed. It includes the review of decentralized key management solutions for IoT and a comparison regarding the same. Chapter 3 describes the main contribution of the thesis, the implementation of a certificate chain discovery protocol for IoT using RIOT OS. The Chapter 4 elaborates on the the internal working of the implementation and the Application Programming Interface (API) provided are discussed. In Chapter 5, the implementation is evaluated, and the results are discussed. Chapter 6 provides a summary of the thesis and the next steps for improving the work.

CHAPTER 2

Background and Related Work

In this chapter, background information and related work regarding this thesis are provided. The first section presents an overview of background information for a better understanding of the implementation details. The second section presents an overview and discusses the related works.

2.1 Background

This section provides background information for a better understanding of the thesis's implementation details.Brief description about PKI, WoT, CoAP ,CoAP resource discovery, digital certificates and encryption are provided.

2.1.1 Public Key Infrastructure (PKI)

PKI facilitates the secure transfer of information over the Internet. Digital certificates and Public-Key encryption are used to encrypt the communication between two entities. In PKI, CA is a trusted third party who is responsible for signing, managing, verifying and revoking digital certificates. CA provides a digital certificate to the user u by verifying their identity and then attaching the user's public key P_u to the certificate upon certificate signing request. CA also creates a cryptographic signature for the certificate using its private key. The public key of CA is known to all users, therefore user v can verify the signature on a certificate. Upon successful verification, user v can make sure that the signature is from a trusted CA. As the user v already trust the CA and CA verified the user u,user v also trusts the user u. The public key of the user u is then used for securing the communication between v and u.

For PKI's proper working,all users must fully trust the CA. Even though PKI enables secure communication and offers verification of users, it comes with several issues. Fully trusting the CA can be a single point of failure. When CA keys are compromised, malicious parties can interfere in the infrastructure, affecting all the users who trust the CA. When adopting PKI to the IoT networks, forming trust and scalability will also be problematic.

2.1.2 Web Of Trust (WoT)

WoT, a concept from Pretty Good Privacy (PGP), is a decentralized trust model in contrast to the centralized PKI. When the PKI relies on CA to establish the authenticity between a public key and the owner, WoT establishes authenticity by issuing certificates in a peer-to-peer manner between nodes in the network, forming a non-hierarchical network of trust. The trust formed in such a network can be direct or indirect trust [4]. Direct trust is when two peers have complete knowledge about each other, so the trust is formed without the intervention of any trusted third party. Indirect trust is when two peers do not have knowledge about each other, but trust is formed via a third party.

The digital certificates containing the public key and owner information are signed for each other by nodes in the network. The key signing procedure endorses the binding between the public key and the owner. Once the certificates are issued, other nodes can validate them using the digital signature. The public key in the certificate is further used to encrypt the communication between peers. Every peer in the network has a public key and private key pair. Public keys are exchanged in the form of certificates, and the private key is kept with the owner. The owner alone has access to the private key. Information that is encrypted using a particular public key can only be decrypted using the corresponding private key. So when user u wants to communicate with user vv,user u encrypts the message with public key of user v. upon receiving the encrypted message, user v can decrypt the message using its private key. The user u also adds a signature to the message by signing using its private key. The user v then validates the signature using user u's public key. In this manner, user v can make sure that the message has not been tampered with and is also from an authentic user. If a user u does not have the key to communicate with user v, It must find the set of certificates for authenticating user v's public key. This process is called certificate chain discovery.

Let there be V number of nodes in an IoT network. Then $E \subset V \times V$ is the certificate graph for storing certification relation between nodes. If u and v are two nodes in the network and v trusts u, v issues a certificate Cert(u,v) to the node u. When a mutual certification is assumed,u also uses a certificate Cert(v,u) to node v. So if there exists a certificate chain

$$s := v_0, v_1, .., v_{\text{c-1}}, v_{\text{c}} := d \qquad v_1, .., v_{\text{c}} \in V, (v_{\text{i}}, v_{\text{i+1}}) \in E$$

from source node s to destination node d, the source node can trust the destination node.

In WoT, the trust for a particular node can vary depending upon the number of other nodes that have issued certificates for it. So there could be some nodes which are more trustworthy than others. Compared with PKI, WoT models are more flexible as the trust decisions are on the individual nodes. WoT decentralizes the trust anchor from CA to individual nodes and can be considered a decentralized PKI.

2.1.3 Constrained Application Protocol (CoAP)

CoAP is a specialized application layer protocol developed for resource-constrained devices communicating over constrained networks [5]. Constrained devices have limited RAM,

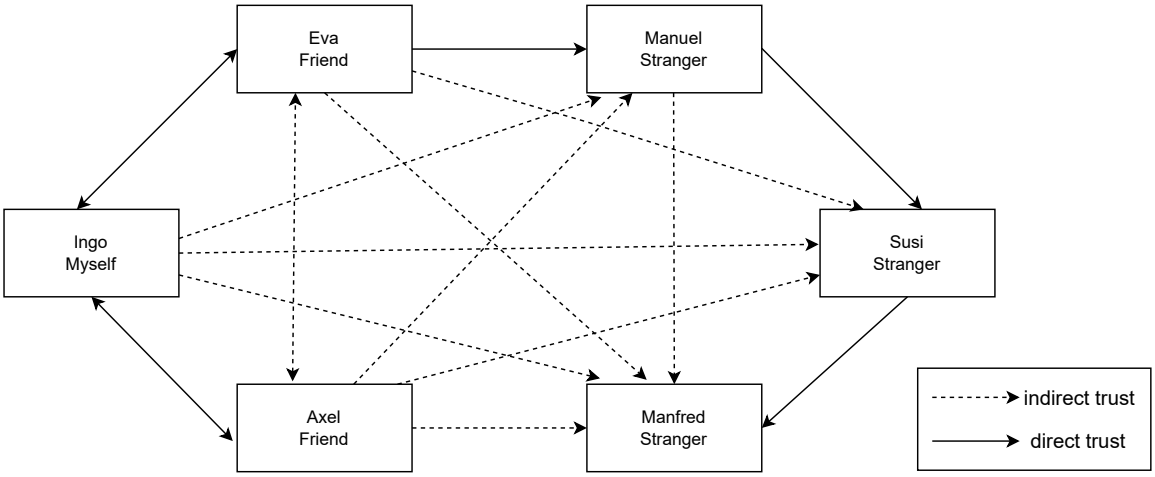

Figure 2.1: Web of Trust

ROM and processing powers. Constrained networks have low achievable throughput, high packet loss and limits on reachability over time [6].

CoAP supports the communication between constrained nodes via a request-response mechanism similar to the Hypertext Transfer Protocol (HTTP) on the world wide web. HTTP methods like GET, PUT, POST, DELETE, and error codes such as 4.04(Not Found), and 4.00(Bad Request) are also in CoAP. So CoAP can be easily integrated with HTTP for establishing communication between web and constrained environments.

Similar to HTTP, a CoAP client sends the request to the server with a method code and on a particular resource which is identified through a Uniform Resource Identifier (URI). However, a node can act as a client and server at the same time. Furthermore, the default transport protocol for CoAP is User Datagram Protocol (UDP) which is more lightweight than Transmission Control Protocol (TCP). This is because the features like message acknowledgement and congestion control in TCP are not used in UDP. So it's more suited for constrained devices.

However, CoAP has the reliability features to tackle the unreliability of UDP. CoAP provides four types of messages. Confirmable, non-confirmable, acknowledgement and reset. Every CoAP message contains a 16-bit message ID, which can be used to identify messages. Using this feature, duplicate messages can be discarded. The confirmable (CON) messages also require an acknowledgement (ACK) from the receiver. If no acknowledgement is received after a default timeout, the messages are re-transmitted with exponential back-off for a maximum number of times or till an acknowledgement is received [5]. Non-confirmable (NON) messages do not require an acknowledgement.

CoAP also supports resource discovery, where a client can learn about the resources hosted in a server without human intervention. Resource discovery is very important in machine-to-machine communications. In addition,cross-protocol proxies can be used to communicate between CoAP and other protocols. furthermore, to transmit a large amount of data which is greater than the recommended payload size of 1024 bytes, block-wise transfer can be used [7].

a) CoAP confirmable message b) CoAP non confirmable message

Figure 2.2: Example CoAP messages

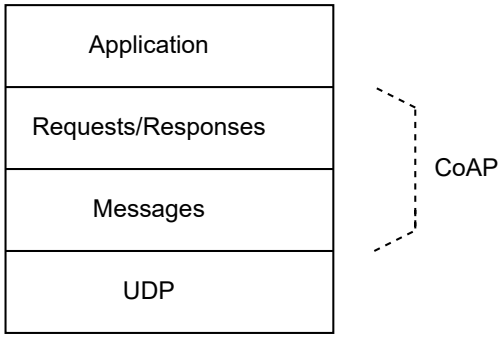

Figure 2.3: Abstract Layering of CoAP

2.1.4 CoAP Resource Discovery

Resource discovery is a useful feature provided by CoAP to learn about the resources hosted in a CoAP endpoint. It is beneficial in M2M applications where clients and server need to communicate without the intervention of a third party like humans. During discovery, the client gets the resources hosted in a server in the form of CoRE link format [8]. The server can decide which resources are discoverable and which are not. Upon receiving the request, the server sends the available resources in CoRE Link Format.

Resource discovery can be made in either unicast or multicast. If the IP address of the server is known in advance, the client can send a CoAP GET request to the "/.well-known/core" URI of the server. The server would then respond with the URI of the resources hosted in the server in CoRE link format. The client then matches the attributes in the response to fit its application. During multicast CoAP discovery, the client sends a GET request to "/.well-known/core" to the multicast address to learn about the resources in a limited range. Query parameters can be added to the request to filter the responses from multiple endpoints. For eg. specific conditions can be applied to the resource type or application-specific attributes.

2.1.5 Digital certificates and Encryption

Digital certificates are used to establish secure communication over the internet. Each endpoint generates public and private key pairs in a public key cryptosystem. Public keys are exchanged in the form of certificates, and the private key is kept secret. Public keys

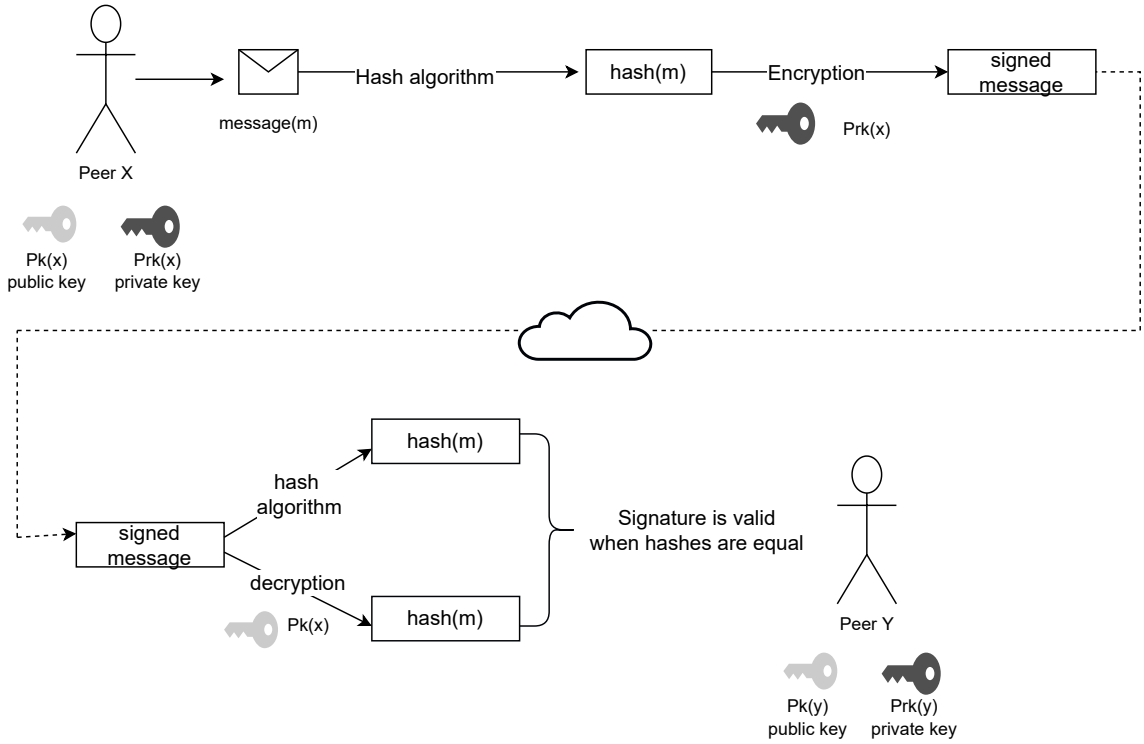

Figure 2.4: Signing and verification of messages

are used to encrypt the communication between two endpoints. Private keys are used to generate signatures. Even though there are digital signature algorithms like Digital Signature Algorithm (DSA), Elliptic Curve Digital Signature Algorithm (ECDSA) as well as Rivest–Shamir–Adleman (RSA), we focus on DSA and ECDSA.

DSA

DSA is a standard for creating digital signatures. It functions based on the framework of modular exponentiation and discrete logarithmic problems. The keys generated from DSA are difficult to compute by brute-force systems. The signature generated from a particular private key can only be validated with the corresponding public key, as shown in 2.4. In this way, the receiver can ensure the message's authenticity by verifying the signature. It also helps in non-repudiation because the sender can not claim that it has not signed the message. Message integrity can also be checked by verifying that the message has not been modified in transit. The steps involved in DSA are

- Key generation
- Key distribution
- Signing
- Signature verification

ECDSA

ECDSA is a variation of DSA based on Elliptic Curve Cryptography (ECC). It is more suited for constrained applications because of the reduced key size and the provided security. An Elliptic Curve $E(\mathrm{F_p})$ over a finite field $\mathrm{F_p}$ is defined by the following equation

$$y^2 = x^3 + ax + b \ \ (mod\ p)$$

where $a, b \in F_\mathrm{p}$ are the parameters of the elliptic curve, and the variables $x, y \in F_\mathrm{p}$ are points on the curve as well as solutions to the equation [9]. Even though different curves can be generated by varying the parameters, named curves can also be used [10].

Public key cryptosystems that rely on ECC creates public key by scalar multiplication of elliptic curve points. To calculate public key, an integer k and a point on the curve $P \in E(F_\mathrm{p})$ is chosen. In order to do the scalar multiplication, the point P is added to itself k times to get another point kP on the curve. The integer k is the ECC private key, and the point on the curve kP is the public key.

The public key kP is a point on the curve and can be represented by cartesian coordinates (x,y). The public keys can also be represented in a compressed format. The compression of public keys is explained in section 2.3.3 of standards for elliptic cryptography [9]. The basic idea is that the y coordinate value is removed from the public key during compression. An information byte is added to the key after removing the y coordinate. In the compressed format, the information byte can be either 0x02 or 0x03, depending upon the value of y. If y is odd, the information byte is 0x03. If it's even, the information byte is 0x02. If we already know the parameters of the curve, the y coordinate can be retrieved from the compressed format by using the elliptic curve equation. y value will be the square root of the elliptic curve equation mentioned above. The solution for the y coordinate will be two values. Information byte is used to identify the correct value.

More details on decompression can be found in section 2.3.4 of [9]. This property of the elliptic curve is beneficial in constrained environments. Public keys can be compressed during transmission, and upon receiving, it can be decompressed.

ECDSA is used widely in IoT devices because it has several advantages when compared with DSA. First of all, the key size of ECDSA is very small compared to DSA.ECDSA can provide the same level of security as DSA with only 256-bit keys, whereas DSA requires 3072 bits to do so. Small key size contributes to faster processing speeds in resource-constrained devices. The memory required to store the keys are also less for ECDSA.

2.1.6 CBOR and C509 certificates

Concise Binary Object Representation (CBOR) is a binary data serialization format that supports most of the common data types in internet standards and supports systems with limited memory and processing power [11]. Small code and message size also make it suitable for constrained devices. The data model used by CBOR is an extended version of JavaScript Object Notation (JSON) [12] and CBOR supports JSON syntax and data types. Although CBOR does not have any dependency on a specific schema for encoding and decoding of messages,Concise Data Definition Language (CDDL) [13] describes the CBOR structures. In order to make the data readable by humans,CBOR also has a diagnostic notation.

Field	Value
Version	CBOR int
Serial Number	CBOR unsigned big num
Issuer	CN as CBOR text string
Validity	CBOR epoch based time
Subject	CN as CBOR text string
Subject Public key	CBOR byte string
Signature algorithm	CBOR int
Signature	CBOR byte string

Table 2.1: C509 message fields

The X.509 [14] is a standard which specifies the format of certificates used in public key cryptosystems. Using digital signatures, X.509 certificate binds an entity to a public key. The certificate can be either self-signed or signed by a CA. The public key contained in the certificate can be used to encrypt the communication with an entity. The signature signed by the entity can also be verified using the public key. The X.509 certificate is specified using the ASN.1 distinguished encoding rules. The certificate contains information such as subject name, validity, signature algorithm id, public key algorithm, public key, certificate signature algorithm and signature. The latest version of X.509, version 3 also has optional extensions. If an extension is marked as critical, it must be processed by the receiver.

When adopting PKI in IoT scenarios, X.509 certificates can not be used as such because the X.509 profile [14] is not optimized for constrained environments [6]. Large certificate size is also problematic in constrained devices. So lightweight digital certificates are required to work in constrained environments. These certificates should also be compatible with the DER encoded X.509 certificates.

C.509 certificates are CBOR encoded X.509 certificates [15]. CBOR encoding reduces the size of the certificates, which is very useful in constrained scenarios. Size reduction is possible because, in C.509 certificates, static and redundant fields are omitted, elliptic curve points and timestamps are compressed. This is also beneficial when considering memory usage, power consumption, communication overhead, latency etc. The signature can be either calculated directly from the CBOR encoded data or DER encoded ASN.1 data in the X.509 certificate.

In the scope of this thesis, a subset of the fields in C.509 certificates are taken. Subject common name and public key are encoded in a CBOR array. The subject common name is encoded as a CBOR text string, and the public key is encoded as a CBOR byte string. The public key is also in the compressed format as described in section 2.1.5. Refer to table 2.1 for more information.

2.2 The DoRIoT Project

In this section, we briefly introduce the DoRIoT project [1].

We have several sensors and actuators observing an environment in a standard IoT environment. These devices are connected to the cloud infrastructure through the internet. Cloud manages the devices and stores the data transmitted from the IoT devices. Applications

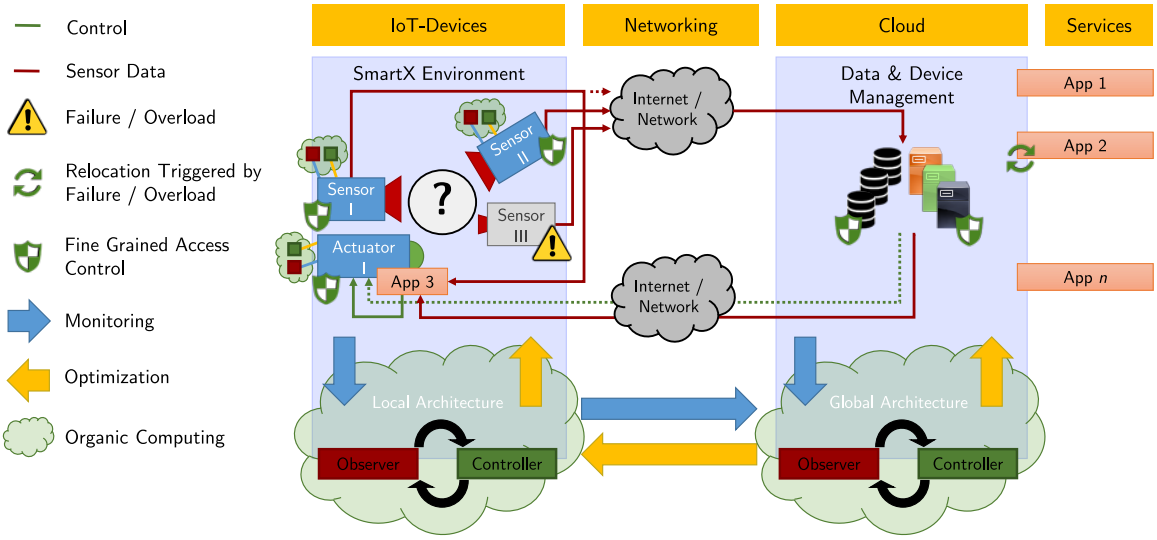

Figure 2.5: DoRIoT architecture [1]

can access these data based on their rights. Devices send data periodically or based on events to the cloud. In this manner, the applications are agnostic about the devices, and the infrastructure cannot utilize the devices' full potential. IoT devices are also static due to the unavailability of an upgrade path, and the logic running on devices is fixed. This type of infrastructure is highly dependent on the cloud.

DoRIoT overcomes these challenges by transforming the centralized architecture of IoT environments into a decentralized architecture, as shown in figure 2.5. DoRIoT contains four working areas. Runtime environment, Optimization of aggregation and disaggregation, Security and safety, Verification and validation. Organic computing makes it possible to detect failures or low service quality in advance, and appropriate countermeasures can be taken. In addition, processes running on the environment can be optimized, resulting in reduced service latency and communication costs. Optimizations also result in prolonged device run time.

DoRIoT also makes it possible to migrate services from one node to another in the network. When a particular device in the network fails, it should not affect the overall working of the environment. So the services can be migrated to any other suitable device. After migrating services, it must be ensured that they run successfully. Security and safety are important factors to be considered while migrating services. The dynamic decentralized access right management system of DoRIoT helps to overcome these problems, which was not possible before with the centralized access control systems.

Using developed concepts and tools for DoRIoT is interesting from an economic standpoint of view. There are also two new value chains that originate from the IoT environment as part of DoRIoT. Firstly premium services can be offered using the dynamic right management system. Secondly, new software functionalities can be added to existing hardware using the run time environment. This thesis is part of the DoRIoT project and aims to solve the certificate management problem in the IoT environment.

2.3 Related Work

In the IoT realm, where many devices interconnect and communicate with each other, security is an important factor to consider. When the network becomes larger, forming trust among all devices is also a huge problem. Securing the data transmitted by devices using asymmetric encryption as happening in PKI can be a solution. But this should be implemented with minimized computational and memory overhead. The solution should also be scalable. The following section discusses and compares related works regarding the adaption of asymmetric encryption techniques in IoT scenarios.

Kitada et al. [16] discuss distributed public key management systems for Ad Hoc networks in which authentication and routing layers are separated. The authentication layer selects the trusted nodes, and the routing layer sends a search packet to other nodes. Two methods are proposed for finding the certificate chains from source to destination node. The first method is called table dependent type, in which routing table information is used in finding the certificate chain. The second method is table independent type which does not consider routing tables

Table dependent type assumes that all nodes have their routing tables and its updated periodically by exchanging routing information with other nodes. A source sends a search packet which contains details about the destination node and routing information to nodes in its range. If one node already trusts the destination node, it adds its own certificate to the search packet. Otherwise, update the search packet by modifying the table information. Once the search packet is received at the destination, the destination node sends a reply message to the source node. This reply message contains all the certificates that form the certificate chain from the source to the destination node. This method also has some issues. Routing table information should be available in the authentication layer. If the routing table and network topology change very often, this method cannot be used.

On the other hand, the table independent type method sends search packet which does not contain routing information. In this method, the search packet is sent by unicasting, and it can even be used in networks where topology changes frequently. The search packet contains information about the source node, destination node, authenticating node and authenticated node. The source node sends the search packet to its trusted neighbours by unicasting. During this process source node also adds its certificate to the packet. The destination node in the search packet will be replaced by the id of the trusted node during this step. This process continues till the search packet reaches a particular node that trusts the authenticated node. At this moment, a reply packet is sent to the source node by unicasting. The reply packet contains all the packets that are needed for constructing the certificate chain.

Yasuda et al. [17] address the certificate chain discovery problems in WoT based Ad Hoc networks. A model for WoT based PKI is proposed, and a solution for the certificate chain discovery problem is also discussed. As explained earlier, the certificate chain discovery problem consists of finding a certificate chain from a source to the destination node and collecting all certificates in the chain. The problem is divided into two steps. The first step is a certificate searching phase, and the second one is a certificate collecting phase.

In the certificate searching phase, it is assumed that one certificate chain from source to destination is enough to authenticate a node. In addition, the number of hops to any other

node is also known in advance. In the searching phase, a spanning tree is created from the topology using distributed algorithms. The root of the tree will be the source node, and there will be a path from the destination node to the source node in the tree. The certificate collecting phase can be simplified to collecting the certificates from in the path from the destination node to the source node in the constructed spanning tree. So the destination node sends a packet to its parent node. Each node in the path adds its own certificate to the packet. Finally, the source node receives the packet which contains all the certificates in the certificate chain.

Ankush et al. [18] propose to implement PKI solutions for IoT with the help of blockchain. The tasks of a trusted third party, such as a centralized CA in PKI, is replaced by blockchain technology. Instead of CA, a distributed, peer-to-peer, publicly verifiable system is used. The paper discusses three different ways to adapt PKI for IoT.

The first method is using Emercoin public block chain. Name Value Storage(NVS) in this block chain is used to store ID and hash of the certificate. Upon demand, any node in the network can verify certificates by accessing the remotely trusted blockchain node, which maintains the blockchain. The second method is comparable to the first, however instead of Emercoin, Eutherium smart contracts are used to store certificates. This method provides advanced storage capabilities, and rules can be applied to the stored data structures. The third approach is based on Eutherium Light Sync mode. The third method does not need a remote blockchain node compared to the first two methods. A lightweight copy of the blockchain is stored in the IoT device instead.

The solutions put forward by Kitada et al. [16] have some issues. The protocol stops when the packet is received by every node in the network. The packets are sent via broadcast, and they contain certificates. The packet size also increases between hops because certificates are being added to the packet. This causes large communication overhead. When the number of nodes on the network increases, the communication overhead also increases. So scalability is also an issue here. The initial assumption that all node needs to have a routing table which is updated frequently is also problematic in constrained scenarios.

Even though Yasuda et al. [17] reduces communication overhead by some amount compared with Kitada et al. [16], it also has some problems. Once the search packet receives the destination,it is sent back to the source by adding certificates of the intermediary nodes. When there is large number of nodes in the network, the packet size will also be large. So scaling up the network also adds a large communication overhead. The assumption that every node in the network knows the number of hops to any other node is also problematic because of the dynamic nature of IoT networks, where nodes easily join and leave the network.

The introduction of blockchain in IoT networks also creates some issues. Whenever the blockchain size increases, the nodes in the network end up using more storage space in order to accommodate a copy of blockchain. The blockchain also requires miners who do the intensive task of adding blocks to the chain. But high computing power and memory are not available in IoT devices.

Engelhardt et al. [2] propose a decentralized key management system based on WoT approach. The certificate chain discovery problem is solved just like the service discovery problem in a semi structured approach. Moreover, it also conveyed that the scalability

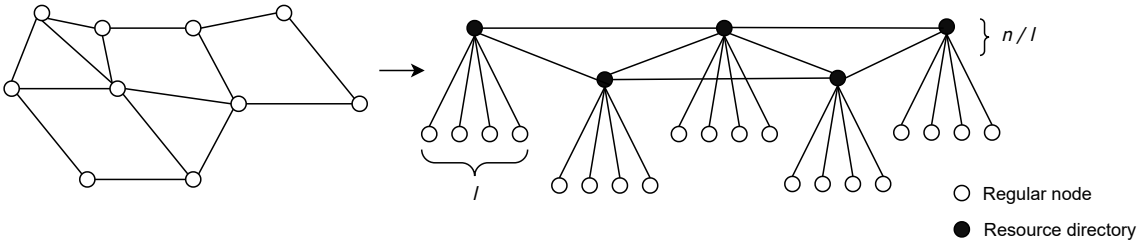

Figure 2.6: Structured network topology through resource directories [2]

issues can be solved by introducing hierarchies in the network.To be precise, hierarchies are created by resource directories [19] used for service discovery which is part of CoAP. The analysis also shows that the hierarchical WoT topologies reduce the communication overhead in IoT scenarios.

Trust in an IoT network cannot be configured in a static manner. It should be discovered automatically by the nodes in the network. This is similar to a service discovery problem where devices need to find services in other nodes in the network. So the paper put forwards to use of the same resource directories used for service discovery can solve the certificate chain discovery problem. Furthermore, to reduce the communication overhead for methods used in [17] and [16], resource directories with large memory capabilities are introduced in the network. The devices have memory in the range of giga bytes and can store large amounts of certificates. Unstructured network topologies used in [17] and [16] is transformed to a structured topology as shown in figure 2.6. n number of nodes are grouped into n / l divisions where l is the number of certificates a resource directory can store.The resource directories form trust among themselves. The network among these can be a fully connected graph or a meshed network.

All the above-mentioned related works approach the problem from a theoretical perspective, and there is no implementation based on these researches. This thesis extends the research done by Engelhardt et al. [2].We research, evaluate and implement a certificate management infrastructure that supports the topology as proposed and evaluated by Engelhardt et al. [2]. The implementation runs on RIOT OS[20].

2.3.1 Related Work in RIOT OS

In this section,we discuss the related work in RIOT OS regarding this thesis.

Ismail et al. [21] put forwards a new Data Transport Layer Security (DTLS) abstraction layer in RIOT [20] OS. This layer can be used to establish secure communication between nodes using DTLS. The modular nature of the layer enables easily switching the underlying DTLS implementation. They introduce `credman` module, which manages the credentials used by DTLS for the handshake. Also `sock-dtls` module,a socket type which uses the `credman`.

DTLS provides communication encryption similar to that of Transport Layer Security (TLS), but it is based on top of UDP instead of TCP. While application data transfer relies on UDP properties, DTLS establishes the secure handshake. The modular networking stack of RIOT makes it possible to use third-party DTLS libraries like TinyDTLS or WolfSSL for DTLS conveniently.

The DTLS credential manager stores and manages the credentials used by the encryption protocols. Different credentials can be used in the system,which are uniquely identified through `credman_type_t` and `credman_tag_t`. `sock_dtls` API can add multiple credentials to the system. If a node communicates with multiple other nodes, API handles multiple credentials used for authentication. credentials can be located in protected regions of memory, and applications must make sure that the credentials are available for DTLS. The client and server should create sockets and add credentials to the system before establishing a connection. After successful authentication and establishing a session, nodes send and receive the packets through the UDP channel.

The credentials used in this implementation can be either based on Pre Shared Key (PSK) or ECDSA. When we are using ECDSA parameters, a node should store its own public and private key, as well as the public key of the node which it intends to communicate with, in the `credman` module. So when we rely on this implementation, if a node wants to communicate with multiple nodes, it should know the public key of all other nodes in advance. Furthermore, it needs to store the public key of all other nodes in the memory. But this is problematic in constrained devices because of limited memory capabilities. In addition, there will be a problem in IoT networks with dynamic and complex topology. In an IoT environment with large number of nodes, every node should know and store the public keys of every other node in advance. When a new node enters the network, other nodes need to know and store the key of the new node in the module to communicate with the new node. Similarly, when a node leaves the network, other nodes may store the credentials even though they don't use them. So managing the credentials in a dynamic environment is also complex. There should be some mechanics to handle the dynamic behaviour of the IoT environment.Using PSK based credentials for DTLS also brings some challenges and problems. The IoT environment should be able to manage PSK for each node in the network. Nodes should also store PSK keys that are used to authenticate every other node in the network, as well as keys for nodes which could be part of the network in future. Storing keys for large number of devices contradicts with capabilities of resource-constrained devices. If the keys become compromised, malicious parties can enter the network and cause damage to all other nodes in the network.

However, with the help of resource directories and the semi-structured topology described by Engelhardt et al. [2], we can overcome these key management problems. After adapting the concept from their paper, a particular node does not need to store the certificate or keys of every other node in the network. The resource directories manage and stores certificates which contain the keys of every node in the network. A node can exchange certificates with the resource directory to register, when it joins the network. The resource directory stores the certificate of all registered nodes. When a particular node wants to communicate with a peer, node can access the certificate of the peer from the resource directory on demand. The received certificates can be further used to secure communication between peers and store the certificate if required.

CHAPTER 3

Thesis Contribution

This chapter represents the main contribution of the thesis, which is the proposal and implementation of a protocol for certificate chain discovery in RIOT OS [20]. The protocol supports the network topology proposed and evaluated in [2] and implemented as a functional library for RIOT Operating System [20].

3.1 Requirement analysis

In this section, we briefly describe and analyse the requirements needed for the certificate management library.

First of all, the implementation should be able to adapt to the dynamic nature of IoT environments. Nodes may enter and leave the network dynamically, and the nodes should be able to identify the resource directories and configure themselves automatically. This is solved by the usage of CoAP resource discovery as explained in 2.1.4. When a node enters the network, it identifies the resource directory automatically and connects with it.

The resource directory and client applications also have different requirements. The resource directory can be non constrained device with more memory capabilities [2]. So it is able to store more certificates as required. At the same time, the client is a resource-constrained device with limited memory capacity. So the client is not able to store many certificates. The library should take care of these restrictions and provide a low memory footprint on the client side.

Furthermore, provisioning the key pair to client and resource directory applications is a significant factor to be considered. This is because, not every constrained device supports the internal generation of public and private keys. We solve this issue by externally generating the key pair and distributing them to the client and resource directory with the application binary.

In addition, the provisioned key pair should only be used for a particular device and should not be associated with any other device. Using the same keys for multiple devices arises security concerns. This issue is addressed by binding a unique common name of devices with the provisioned keys.

Moreover, during certificate exchange, the integrity and confidentiality of the communication should be ensured. We use digital signatures to ensure the security of the certificate exchange. The certificates are signed using the private key of the devices such that the nodes can verify the signature using the public key on reception.

3.2 Web of Trust Concept in IoT

The core problem is establishing mutual trust among a huge set of IoT devices. RIOT already has a DTLS implementation, where we need to provide encryption keys for each individual pair of devices. The encryption keys can be either pre-shared keys or public keys. But storing keys for each connected device is not feasible in a production environment.

When we want to use asymmetric key encryption to secure communication between nodes in the IoT network, there should be some mechanism to fetch the public key of the desired communication partner. For eg. if Alice wants to communicate with Bob, Alice should fetch the public key of Bob so that the communication can be encrypted.

In the proposed scalable WoT infrastructure [2], resource directories are introduced in the network, which acts as the repository which stores the public keys of devices in the IoT network. Public keys are issued in the form of certificates to desired communication partners.

When a new node enters the network, it goes through the following steps.

1. Node registers with the resource directory
2. Node authenticates the resource directory and establishes trust. This is similar to Key signing in WoT. This can be done with hard coded PSK or root certificate. Also, Out of Band methods can also be used.
3. Resource directory stores the valid public key of the new node, and node stores the public key of the resource directory after trust establishment. The resource directory has the list of public keys of all registered nodes, while the node just needs to store the public key of the resource directory.
4. When a node wants to communicate with another node, it queries the resource directory for the public key of the desired node.
   a) If the node is already registered with the same resource directory, the resource directory can reply itself.
   b) If the node is registered with another resource directory, it must be found first. refer figure 2.6.
5. The node must verify the certificate chain of the peer node before communication. This process is called certificate chain discovery.
6. Nodes can now use the public key to secure the communication.

Even though the resource directories can form a network among themselves and form trust supporting the topology in figure 2.6, thus becoming a larger network by connecting multiple resource directories, this thesis focuses on a subpart of the network as shown in figure 3.2. For eg. When Alice and Bob enter the network, both register with the resource directory. After this process, Alice and Bob will have the resource directory's public key

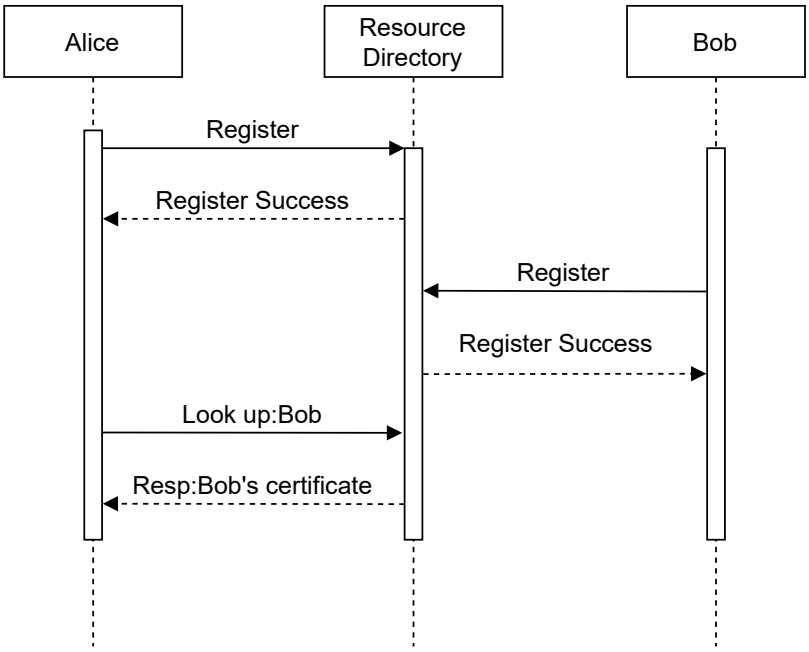

Figure 3.1: General interaction between resource directory and clients

stored. The resource directory also stores the public keys of both Alice and Bob. When Alice wants to communicate with Bob, Alice queries the resource directory for Bob's public key and receives the key in the form of certificates, as shown in figure 3.1.

3.3 Implementation

The implementation mainly consists of two phases. The first phase is a registration mechanism where clients that are part of the network topology register with the resource directory. During the registration phase, clients and the resource directory exchange their public keys

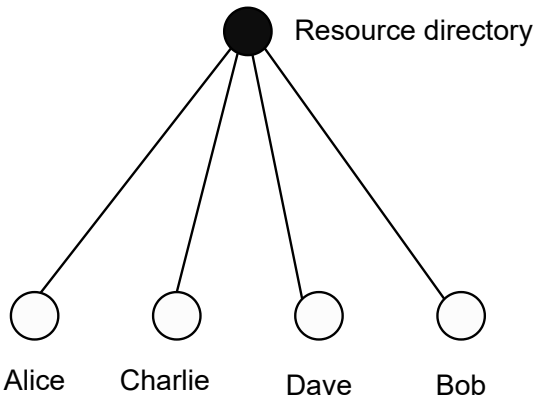

Figure 3.2: Structured network topology with a single resource directory

in the form of certificates. The resource directory stores the public keys of all clients who are registered with it.

The second phase is a lookup mechanism where a client A queries the public key of client B from the resource directory and retrieves it. The public key retrieved from this phase can be used to encrypt the further communication between client A and client B.

3.3.1 Registration interface

When a client enters the network, either it should know about the resource directory in advance, or the client should find the resource directory automatically. So in order to communicate with the resource directory, the client must know the IP address first. The client uses CoAP resource discovery to learn about the resource directory, as explained in section 2.1.4. Client sends a CoAP multicast GET request to "/.well-known/core" URI. As there might also be other CoAP servers in the network which may respond to the multicast request, we need some mechanism to filter out the responses from all the servers. So query parameters are added to the request for this purpose. To be specific, the resource type parameter "rt=wotdisc" is set. This same resource type is also set in the link parameters of the resources in the resource directory. The resource directory responds to the request with its resources, and from this response, the client learns about the resource directory. Once the resource directory's IP address is known, the client proceeds to register itself with the resource directory.

The registration interface provides mechanisms to register a client with a resource directory. After successful registration, the resource directory stores the public key of the client. The client also stores the public key of the resource directory.

An important factor that comes in to picture when exchanging certificates is the authenticity of the resource directory. Several methods can be used to verify the authenticity of the resource directory. For eg. certificates of both client and resource directory can be created from a root certificate, as in PKI, and the path to the root certificate should be verified after exchanging the certificates. Another method can be Out-Of-Band mechanisms for pairing and authenticating between client and resource directory. In the thesis, PSK are used for authenticating the resource directory. The client and resource directory have a common secret PSK. The resource directory's certificate is signed by a PSK, and upon receiving the certificate at the client side, the signature is verified using the PSK.

The sequence diagram of the registration interface is as shown in figure 3.3. The resource directory contains 2 resources as part of the registration interface. One resource is for handing the CoAP GET request from the client for the resource directory's certificate. Another resource is for handling the CoAP PUT request when the client sends its certificate.

At first, client sends a CoAP GET request to the resource directory to get the certificate of the resource directory. When the resource directory gets the CoAP GET request, the resource directory creates its certificate using the common name and public key. The certificate is in C509 format as explained in 2.1.6. In the context of this thesis, the C509 certificate contains the common name and public key of the entity. Other fields in the certificate, such as validity and signature algorithms are omitted. But can be easily extended with more fields as required. The C509 certificate contains an array of the common name as

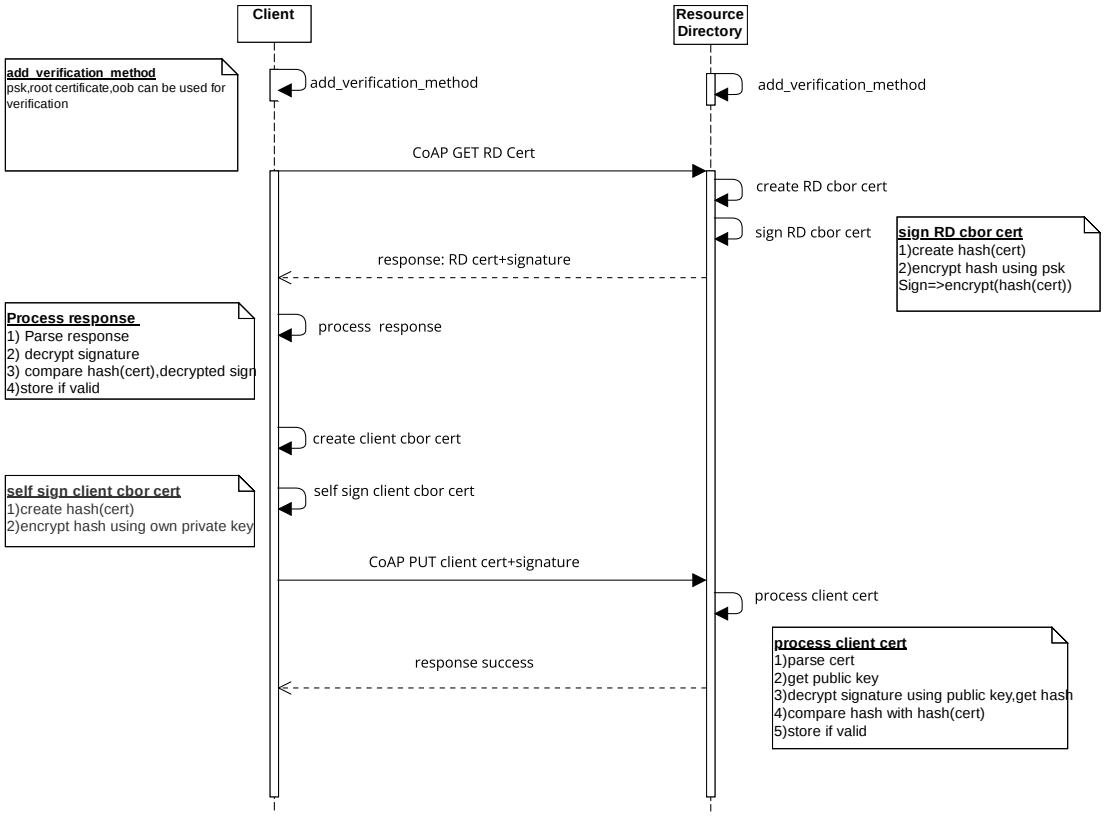

Figure 3.3: The client requests the resource directories certificate using CoAP GET.The resource directory then send its certificate to client.After successful processing of the certificate,client send its certificate to resource directory using CoAP PUT.

CBOR text string and public key as CBOR byte string. The ECC public keys are 64 bytes, but we are using a compressed version of the public key to reducing the communication overhead. The details on the compression can be found in 2.1.5.

The next step in the resource directory is to create a signature for the certificate created in the previous step. At first, the hash of the certificate is calculated using Secure Hash Algorithms (SHA) 256, which outputs a 32-byte hash. The PSK, which is a shared secret between the client and resource directory, is then used to encrypt the hash. The encryption standard used here is Advanced Encryption Standard (AES). Finally, the certificate and signature are sent to the client.

The client processes the response, once it gets a reply from the resource directory. The client parses the response and separates certificate and signature. After that, the client decrypts the signature using PSK, which is known in advance. The hash of the received certificate is also calculated using SHA 256. Decrypted signature and the calculated hash are compared, and if they are equal, we can confirm that the received certificate is valid. The client can also assure that the certificate is from a trusted resource directory because the PSK it possesses was able to decrypt the signature which was created by the resource directory using the same PSK. Once the client receives a valid certificate, it parses the C509

certificate and gets the common name and compressed version of the resource directory's public key. The client decompresses the public key and stores the common name and public key of the resource directory in a linked list.

After successful processing of the certificate from the resource directory, the client creates its own C509 certificate. The client creates a CBOR array of its common name and public key. The common name is encoded as CBOR text string. The compressed version of the public key is encoded as CBOR byte string. This process is exactly like the resource directory's certificate creation, as explained earlier. The client then generates a signature for the certificate created in the previous step. The hash of the certificate is calculated using SHA 256. Then the hash is encrypted using the private key of the client. The self-signed certificate offers a level of security to the message being transmitted. As the certificate is signed by the private key of the key, it can only be decrypted using the client's public key. The resource directory later checks the validity of the certificate by using the public of the client. Finally, the certificate and the self-signed signature are sent to the resource directory using CoAP PUT.

Once the resource directory receives the certificate from the client,it processes the certificate. The public key of the client is separated from the certificate. Then the signature is decrypted using the public key to get the hash of the certificate. The hash of the received certificate is calculated using SHA 256 and compared against the decrypted signature. If the hashes match, the resource directory can make sure that the certificate is valid. The resource directory stores the client's public key and common name in a linked list, followed by a successful response to the client.

3.3.2 Lookup interface

The lookup interface provides mechanisms to get the public key of a client, which is stored in the resource directory. Suppose client Alice wants to communicate with client Bob. Alice needs to know the public key of Bob to encrypt the communication between Alice and Bob. As we are dealing with constrained devices, which has limited memory capacity, Nodes in the network cant always store the public key of all other nodes. But the resource directory has more memory capacity, and it stores the public key of all registered clients. So client Alice looks up the certificate of Client Bob in resource directory and retrieves it. Further communication is encrypted using the public key of Bob which was retrieved from resource directory.

First of all, every client should be registered with the resource directory as explained in 3.3.1. After successful registration, the resource directory stores every client's public key and common name. In this thesis,we can assume that every clients common name is unique and only one client with a particular name registers with resource directory.

At first, clients Alice and Bob register with the resource directory. Now the resource directory contains the public keys of both Alice and Bob. When Alice wants to communicate with Bob, Alice looks ups the certificate of Bob in the resource directory. Alice sends a CoAP GET request to the resource directory. The payload contains the common name of Alice as well as Bob. Resource directory checks if both Alice and Bob are already registered with it upon receiving the lookup request. This can be checked by searching the

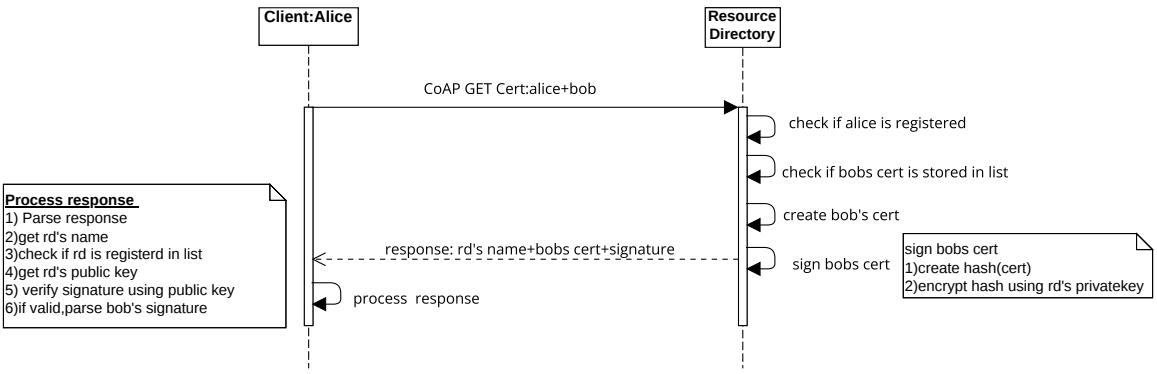

Figure 3.4: The client Alice requests the certificate of client Bob using CoAP GET.The resource directory then send its certificate to client Alice.

list using Alice and Bob's common name. If both Alice and bob were already registered, the Resource directory searches for Bob's public key in the linked list. The resource directory then creates C509 certificate of Bob using its common name and public key. Certificate creation is exactly like as explained in 3.3.1. The resource directory creates a signature for the certificate. Hash of the certificate is calculated using SHA 256 and then the hash is encrypted using resource directory's private key. Finally, the resource directory responds to the client with the resource directory's name, Bob's C509 certificate and Signature.

Upon receiving the response to the look-up request,client Alice processes the response. Alice parses the response and gets the resource directory's name. Then, it performs a name search in the linked list to check whether the directory is a known resource directory. If it's already registered, Alice gets the resource directory's public key from the linked list. Then hash of the received C509 certificate is calculated. Then the hash is verified against the signature using the public key of the resource directory. Upon successful verification Alice can assure that the certificate received from the resource directory is authentic. Alice then parses the C509 certificate to get the public key of Bob. The communication between Alice and Bob can then be encrypted using the public key.

3.3.3 Module Working

This section elaborates on the working of the certificate management infrastructure. The implementation details of the system for RIOT OS are discussed. The concept is implemented as a module for RIOT OS. The internal working of the module is abstracted and the core features are provided to the application as API. Functionalities are defined in multiple header files and separated according to the context.

The super-module `doriot_wot` contains two sub-modules.One for the resource directory and another for the client.The client module is named as `doriot_wot_cl` and the resource directory as `doriot_wot_rd`.These modules can be easily integrated into the application by adding them to the Makefile. Several build options for client and resource directory can be specified through Kconfig files. Using Kconfig makes it convenient to apply different configurations without changing the source code.

Initially, both client resource directory applications must provide the keys required for cre-

ating certificates and authentication to their respective modules. The `wot_credentials_t` structure is used for this purpose.

```
typedef struct {
    wot_key_status_t status;
    uint8_t private_key[PVT_KEY_SIZE];
    uint8_t public_key[PUB_KEY_SIZE];
    uint8_t psk_key[PSK_KEY_LEN];
} wot_credentials_t;
```

Listing 3.1: The data structure used to provision key to the module

The 32 byte elliptic curve private key and 64 byte public key should be set in this structure. These public key and private key are used in the context of certificates. The PSK in the structure is used for authenticating the resource directory during registration. After setting all the keys in the structure,it is provisioned to the module using the function `wot_provision_keys()`.

Once the keys are successfully provisioned to the module for both the resource directory and client, the resource directory should be able to listen for incoming requests from clients. `wot_rd_start()` function starts the resource directory and it listens for requests in the default CoAP port 5683.

After the resource directory is ready to receive requests from the client, the client should discover the resource directory through CoAP resource discovery. The `wot_discover_rd()` function finds the resource directory and stores the address in the module for future communication with the resource directory. Th parameter to the `wot_discover_rd()` function is a call back for discovery. The discovery call back function informs the application about the status of discovery, that is, whether the discovery was successful or not. Upon successful resource discovery, the call back function also returns the address of the resource directory to the application.

Upon successful discovery of the resource directory,client registers with it by exchanging certificates.The `wot_register_client()` function is used for this purpose.this function request the certificate of resource directory. If the certificate is successfully processed and resource directory is authenticated,client sends its certificate to the resource directory. The common name and public key of resource directory is stored in the linked list for future use. The input parameter for the `wot_register_client()` is a call back function which notifies the application whether the registration was successful or not.In similar fashion, multiple clients discover and register with the resource directory.

When a client, Alice wants to communicate with Bob, Alice requests the certificate for Bob in the resource directory.The function `wot_lookup_client()` is used to look up the certificate of a client in resource directory.If the intended client is already registered with the resource directory, it sends the certificate to requested client. The input parameters of the function are the name of the client to be looked up and a callback function for lookup. The call back function notifies the application whether the lookup was successful or failure. If the lookup was successful, a certificate structure called `wot_cert_t`is returned in the callback. The structure contains the common name and public key of the node whose certificate was received from the resource directory.

```
1 typedef struct {
2     list_node_t next;
3     char name[COMMON_NAME_MAX_LEN];
4     uint8_t pubkey[PUB_KEY_SIZE];
5 }wot_cert_t;
```

Listing 3.2: The data structure used to store information about a node

So for the resource directory,the operations can be summarized as follows:

1. provision keys to the module
2. start the server
3. register clients
4. send response to lookup requests

similarly, for the clients, the operations can be summarized:

1. provision keys to the module
2. discover resource directory
3. register with resource directory
4. look up certificates for other clients in resource directory

As the resource directory is a device with high memory capabilities, it stores the certificates of every node which are registered with it. But normally the clients are constrained devices with limited memory. It is mandatory to store the certificate of resource directory in all the registered clients.Otherwise, the clients won't be able to verify the signature for certificates that it receives from the resource directory. In the client application,client can choose whether to store or not store the certificate received as part of the lookup process. If the client does not want to store the received certificate, it's available via the call back function to the application from the module. When the client stores the certificate in the list it will be available for future use. Otherwise client needs to request the certificate from resource directory on demand.

We use Kconfig [22] to select configuration options for the applications at build time. Kconfig is used to define symbols which can specify default values. Symbols are also used to specify dependencies. With the use of Kconfig, our modules support different configurations for the applications without the need for changing the source code. User can use menu-driven user interfaces like *menuconfig* to interact with Kconfig and modify values for a particular symbol. The table 3.1 illustrates the Kconfig options available in the `doriot_wot` module.

The name of the client is configured using WOT_CL_COMMON_NAME and the name of resource directory by using WOT_RD_COMMON_NAME. Both names are string variables. The CoAP URI's, WOT_CLIENT_CERT_URI, WOT_LOOKUP_CERT_URI and WOT_RD_CERT_URI are for putting client certificate to the resource directory, requesting the look up certificate and getting certificate of resource directory respectively. Care should be taken while configuring the URI because they must be sorted by ASCII order when specifying CoAP resources at the resource directory. All three of the URI are also string values. WOT_AUTH_TYPE is used for specifying the method for authenticating the resource directory. PSK is chosen by default for verification with a default integer value of 0. Values 1 and 2 can be used choose root certificate verification and Out Of Band (OOB)

Value	Type	Description	App
WOT_CL_COMMON_NAME	string	Name of client	client
WOT_CLIENT_CERT_URI	string	URI to put client certificate	both
WOT_RD_CERT_URI	string	URI to get rd certificate	both
WOT_LOOKUP_CERT_URI	string	URI to get lookup certificate	both
GCOAP_PORT	int	CoAP port	rd
WOT_AUTH_TYPE	int	rd authentication type	both
WOT_STORE_LOOKUP_CERT	bool	store or not look up certificates	both
WOT_RD_COMMON_NAME	string	Name of rd	rd
WOT_USE_CRYPTO_CELL	bool	use hardware acceleration for ECC	both

Table 3.1: Kconfig options

verification respectively. The boolean symbol WOT_STORE_LOOKUP_CERT is used to specify whether to store or not to store the look up certificates in client. As the client could be resource constrained device with less memory, it is better to give an option like this because now client has the freedom chose if it needs to store the certificate. If the client chooses to store the certificate,it can use the certificate for future operations. Otherwise the client can request the certificates again on demand from the resource directory. The boolean symbol WOT_USE_CRYPTO_CELL is used to select the crypto hardware for ECC operations, in case the device supports it. Otherwise, the devices use crypto software for ECC signing and verification.

3.3.4 Crypto Operations

In this section, we describe the cryptography operations that are used in this module.

As security is an essential factor to be considered while exchanging certificates, we use the client and resource directory signatures to provide authentication and integrity to the certificate exchange process. The signatures are created using cryptographic operations, which are often resource intensive. This conflicts with the resource-constrained devices.

Kietzmann et al. [23] have done a comprehensive resource analysis on cryptographic operations using different methods. On a constrained node, there are three methods for enabling cryptography. Firstly we can use software libraries that deal with constrained resources. Secondly, devices that include a dedicated crypto-peripheral can be used. Finally, external crypto-peripherals can be connected to microcontrollers using a communication bus.

In this thesis, we are using asymmetric encryption standards for generating signatures. In particular, we are interested in the ECC signature generation and verification. Our module offers the flexibility to choose between crypto software libraries and peripheral crypto acceleration for elliptic curve cryptography. We use micro-ecc[1], which provides a lightweight and fast software implementation for ECC operations. We also use hardware acceleration using ARM TrustZone CryptoCell 310 [2].

From the evaluation done in [23],it is identified that the operation using crypto-peripheral

[1]https://github.com/kmackay/micro-ecc
[2]https://www.arm.com/products/silicon-ip-security/crypto-cell-300

require ≈ 20 ms complete. This performance gain is one order of magnitude less compared to crypto software. Even though there is a performance benefit when crypto-hardware is used, it ends up using more memory. The crypto cell on nRF52840 needs two separate structures to store a public and private key, which require 884 and 816 bytes, respectively. This memory requirement is very large compared to the original size of the keys. The ECC P-256 curve that we use, needs only 64 byte and 32 byte for public and private key respectively. The extra key structure size in the crypto cell is used for copying the elliptic curve parameters and mirroring the hardware state of the peripheral. Storing the curve parameters allows the crypto cell to change the curve parameters at run time. The crypto cell library also ends up using more stacks in the range of kilobytes because of the need for temporary buffers.

The micro-ecc library requires only 32 byte for the private key and 64 byte for storing the public key. This key size is same as we store in the `wot_credentials_t` and `wot_cert_t`. So whenever we need to use the crypto cell, there is an overhead of converting the keys to structures supported by the crypto cell. When compared with micro-ecc, the crypto cell on nRF52840 consumes more RAM and ROM. This is expected because the micro-ecc is designed for a low memory footprint [24].

To sum up, the crypto hardware accelerators perform better in terms of runtime and energy compared to crypto software. At the same time, crypto hardware also introduces more memory overhead. Also, not every microcontroller has inbuilt hardware accelerators. So it is up to the user to choose what method to use in the application. If the device supports hardware accelerators and is worth enough to compromise memory overhead over run time, crypto hardware can be used for ECC operations. Otherwise, crypto software can be disabled using the WOT_USE_CRYPTO_CELL option.

CHAPTER 4

Library Use

In this chapter, we describe how to use the library. The APIs are explained, and a manual for using the APIs are also provided along with some code samples. To use the module,clone the source code[1] to directory of your choice. In addition,in order to run the sample application for client and resource directory,set the `RIOTBASE` properly in Makefile and build the corresponding applications.

4.1 API Description

The following section describes the APIs, which are available to the application. The client must include `doriot_wot_cl.h` and resource directory must include `doriot_wot_rd.h` in the application to access the corresponding APIs.

4.1.1 Client API

1. `int wot_discover_rd(int (*callback)(int,sock_udp_ep_t))`
   - brief : function to find resource directory via coap resource discovery
   - parameter : callback function returning udp end point on success
   - return : integer
2. `int wot_register_client(int (*callback)(int))`
   - brief : function to register the client with rd
   - parameter : callback function returning success or failure
   - return : integer
3. `int wot_lookup_client(char *lookup_name,int(*callback)(int,wot_cert_t*))`
   - brief : function to lookup for client certificate in rd
   - parameter : name of the client to be looked up,callback function.
   - return : integer

[1] https://github.com/ad6sh/MasterThesis/tree/main/doriot_certificate_chain_discovery

4.1.2 Resource Directory API

1. `int wot_rd_start(void)`
    - brief : function to start resource directory
    - parameter : void
    - return : integer

4.1.3 Common API

1. `int wot_provision_keys(wot_credentials_t *keys)`
    - brief : function to provision keys to the module
    - parameter : structure containing keys
    - return : integer
2. `wot_cert_t *wot_cert_add(char *name,int name_len,uint8_t *pubkey)`
    - brief : function to add the certificate to list
    - parameter : name of the node, name length, public key
    - return : pointer to the stored structure
3. `bool wot_node_exists(char *name)`
    - brief : function to check if a node with a name exists in the list
    - parameter : name of the node
    - return : bool,true or false
4. `wot_cert_t* wot_cert_get(char *name)`
    - brief : function to get certificate structure of a node
    - parameter : name of the node
    - return : pointer to the structure
5. `wot_cert_t* wot_cert_del(char *name)`
    - brief : function to delete certificate structure of a node
    - parameter : name of the node
    - return : pointer to the deleted structure

4.2 User Manual

This section describes how to use the APIs to create client and resource directory applications.

4.2.1 Resource directory Application

In order to use the APIs related to the client,include the `doriot_wot_rd.h` in application header. In addition, set the path to the external library and specify the module to be used in the application Makefile as shown in 4.1.

```
USEMODULE += doriot_wot_rd
EXTERNAL_MODULE_DIRS += /path/to/module/doriot_wot
```

Listing 4.1: Adding the module in resource directory application

Firstly, create a `wot_credentials_t` structure which is used to provide the keys to the *doriot_wot_rd* resource directory module. Fill the structure with the private key, public key and PSK, followed by provisioning of the credentials to the module using the function `wot_provision_keys()`. Finally, start the resource directory using the function,`wot_rd_start()`.All the above-mentioned steps are shown in 4.2.

```
#include <string.h>
#include "doriot_wot_rd.h"

static unsigned char priv_key_rd[] = {
    0x3A, 0x8D, 0xFF, 0xFB, 0xAE, 0x7D, 0x8F, 0xA4, 0xAF, 0x3F, 0x37, 0x8E, 0x14, 0x2C, 0x60,0x2C,
    0x9C, 0xDD, 0x01, 0xE3, 0x2C, 0xD7, 0xCD, 0x3A, 0xE7, 0xF7, 0x36, 0x1C, 0xFD, 0xBF, 0x61, 0x89
};

static unsigned char pub_key_rd[] = {
    0x6E, 0x0B, 0xD3, 0xE6, 0x92, 0x58, 0xB4, 0x38, 0x82, 0xC6, 0xAE, 0x0B, 0xE1, 0x9F, 0x50, 0x4A,
    0xB2, 0x40, 0x6D, 0xE3, 0xCB, 0xC2, 0x93, 0x27, 0x4E, 0x59, 0x37, 0x36, 0xC0, 0x80, 0xC1, 0x73,
    0x06, 0xDE, 0x7C, 0x6E, 0x4E, 0xC8, 0x6B, 0xD5, 0x92, 0xDE, 0x98, 0x09, 0x1B, 0x06, 0x2A, 0x8C,
    0x68, 0x6F, 0x9E, 0xAF, 0x74, 0x47, 0x58, 0x86, 0xD8, 0x2C, 0x17, 0x68, 0xF4, 0x69, 0xB5, 0x0F
};

static unsigned char psk_key[] = {
    0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
    0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
    0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
    0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4
};

static void _set_credentials(wot_credentials_t *credentials)
{
    memcpy(credentials->private_key, priv_key_rd, sizeof(priv_key_rd));
    memcpy(credentials->public_key,  pub_key_rd, sizeof(pub_key_rd));
    memcpy(credentials->psk_key, psk_key, sizeof(psk_key));
}

int main(void)
{
    /*create structure to store credentials*/
    wot_credentials_t credentials;
    /*fill the structure with keys*/
    _set_credentials(&credentials);
    /*provision keys to the module*/
    wot_provision_keys(&credentials);
```

```
    /*start resource directory*/
    wot_rd_start();
}
```

Listing 4.2: Resource directory application

Once the resource directory is started, it can listen for registration requests and certificate lookup requests from clients.

4.2.2 Client Application

In order to use the APIs related to the client,include the `doriot_wot_cl.h` in application header. Additionally, set the path to the external library and specify the *doriot_wot_cl* module to be used in the application Makefile. This is shown in 4.3.

```
USEMODULE += doriot_wot_cl
EXTERNAL_MODULE_DIRS += /path/to/module/doriot_wot
```

Listing 4.3: Adding the module in the client application

The initial set-up of provisioning the keys to the module is exactly like we did for the resource directory. Create a `wot_credentials_t` structure and set the structure with the private key,public key and PSK followed by provisioning of the credentials to the module using `wot_provision_keys()` function.

After provisioning the keys, the client should discover the resource directory using CoAP resource discovery. The `wot_discover_rd()` function is used for this purpose. Usage of this function is described in 4.4

```
/**
 * @brief call back funtion for coap resource discovery
 *
 * @param status
 * @param remote_rd
 * @return int
 */
int discovery_callback_app(int status, sock_udp_ep_t remote_rd)
{
    switch (status) {
    case DISCOVERY_SUCCESS:
        /* resource discovery success */
        puts("resource discovery success\n");
        _print_ip(remote_rd);
        break;
    case DISCOVERY_FAILURE:
        /*resource discovery failure*/
        puts("resource discovery failure\n");
        break;
    default:
        break;
    }
    return status;
}

wot_discover_rd(discovery_callback_app);
```

Listing 4.4: Discovering the resource directory via CoAP resource discovery

The parameter to the `wot_discover_rd()` is a callback for discovery, which returns the operation's success or failure status. Upon successful discovery, the call back function returns the UDP end point of the resource directory as shown in 4.4.

If the resource discovery is successful, the client can proceed to register with the resource directory using the function `wot_register_client()`. The registration function can be called in the discovery callback function if the discovery status is a success.

```
/**
 * @brief call back function for client registration with rd
 *
 * @param status
 * @return int
 */
int registration_callback_app(int status)
{
    switch (status) {
    case REGISTRATION_SUCCESS:
        /* client succesfully registered with rd */
        puts("registration success\n");
        break;
    case REGISTRATION_FAILURE:
        /*client  failed to register with rd*/
        puts("registration failure\n");
        break;
    default:
        break;
    }
    return status;
}

wot_register_client(registration_callback_app);
```

Listing 4.5: Registering with the resource directory

The parameter to the `wot_register_client()` function is a call back for registration. The callback function returns the status of the client registration with the resource directory, which is either success or failure. The registration process is shown in 4.5.

```
int lookup_callback_app(int status, wot_cert_t *node)
{
    (void)node;
    switch (status) {
    case LOOKUP_SUCCESS:
        /* successfully received lookup certificate from rd*/
        puts("lookup success\n");
        printf("node name :%s\n", node->name);
        print_hex("node public key : ", node->pubkey, (unsigned int)PUB_KEY_SIZE)
            ;
        #if !CONFIG_WOT_STORE_LOOKUP_CERT
        free(node);
        #endif
        break;
    case LOOKUP_FAILURE:
        /*failed get lookup certificate from rd*/
        puts("lookup failure \n");
        break;
    default:
        break;
}

```

```
wot_lookup_client(name, lookup_callback_app);
```

Listing 4.6: Looking up a certificate in the resource directory

Once the registration is successful, the client becomes part of the IoT environment. Multiple clients also register with the resource directory in a similar fashion. If a client Alice wants to communicate with a client, Bob, Alice can fetch the certificate of Bob from the resource directory. The `wot_lookup_client()` is used for this purpose. There are two parameters to the `wot_lookup_client()` function. The first parameter is the name of the client to be looked up. The second parameter is a callback function for certificate lookup. The call-back function has two parameters. The first parameter is the status of the operation and the second parameter is the `wot_cert_t` structure which will contain the common name and public of the client whose certificate was requested from the resource directory. In addition, if the CONFIG_WOT_STORE_LOOKUP_CERT option is enabled, the certificate is stored in the linked list. If the user does not want to store the certificate in the list, freeing the `wot_cert_t` in the look-up call back function is recommended. The certificate lookup process is shown in 4.6.

If the client or resource directory needs to view the certificate stored in the linked list, the corresponding list functions can be used. For eg. The stored certificate structure can be fetched from the linked list using `wot_cert_get()` function as shown in 4.7.

```
int find_cert(char *name)
{
    wot_cert_t *node = wot_cert_get(name);
    if (node == NULL) {
        printf("certificate not found for :%s\n", name);
        return 1;
    }
    else {
        printf("certificate found for :%s\n", node->name);
        _print_hex("public key :", node->pubkey, (unsigned int)PUB_KEY_SIZE);
    }
    return 0;
}
```

Listing 4.7: Searching for a certificate in the list

Similarly,certificates can also be deleted from list using the `wot_cert_del()` function as shown in 4.8.

```
int delete_cert(char *name)
{
    wot_cert_t *node = wot_cert_del(name);
    if (node == NULL) {
        printf("certificate not found for :%s\n", name);
        return 1;
    }
    else {
        printf("certificate deleted :%s\n", node->name);
    }
    return 0;
}
```

Listing 4.8: Deleting a certificate from the list

CHAPTER 5

Thesis Outcome: Evaluation

In this chapter, the implementation of the thesis is evaluated based on different criteria. The objective of the evaluation is to understand how well the implementation performs on RIOT OS and what steps need to be done to increase the current implementation's performance.

5.1 Setup

This section discusses details about the evaluation environment, such as hardware and software setup. In addition, the metrics that are examined are briefly introduced.

We evaluate the implementation based on four criteria. First, a client's total duration for the certificate lookup in the resource directory is measured. Secondly, the CPU overhead for processing the C509 certificates is measured. Thirdly the memory usage required by the implementation is measured. This consists of the Random Access Memory (RAM) as well as the flash usage. Finally, the message overhead for the certificate look-up is discussed.

The hardware that we used for the evaluation is the nRF-52840 dongle, a small-cost device from Nordic Semiconductors, which supports Bluetooth 5.3, Bluetooth mesh, Thread, Zigbee, 802.15.4, ANT and 2.4 GHz proprietary protocols. The nRF-52840 System on Chip (SoC) has the ARM TrustZone CryptoCell 310 security subsystem, which provides cryptographic services.

The software used is RIOT OS of version 2022.01. Please refer to the table for more detailed information about hardware, software and the network stack 5.1. We wrote client and resource directory programs and flashed the nRF-52840 dongles. So for this evaluation, the client and resource directory are the same devices. But in the actual use case, the resource directories can also be a more powerful device with more memory and processing capabilities.

Client and resource directory modules support two options for ECC, which we use to sign and verify the certificates. The first option is using the Mirco-ECC library, a software implementation of cryptographic operations. The second option is using hardware accelerators for cryptographic operations. For example, we also use the ARM Crypto Cell in the

Parameter	**Value(s)**
Hardware	
Boards	nRF52840-Dongle
CPU	Arm® CortexTM-M4 64 MHz
Flash	1MB
RAM	256KB
TX Power	+8dBm
TX Frequency	2.GHz
Software	
IoT OS	RIOT 2022.01
Network stack	GNRC
Link layer	IEEE 802.15.4
Network Layer	IPv6 + 6LowPAN
Transport Layer	UDP
Application Layer	CoAP

Table 5.1: Evaluation parameters

nRF-52840 for signing and verification. If a device supports crypto-hardware accelerators, the device can use it for ECC operation by enabling the WOT_USE_CRYPTO_CELL configuration option in the application. However, if that is not the case, the application uses Micro-ECC by default. The evaluation is done using both crypto cell and Micro-ECC library.

5.2 Total duration for certificate lookup

In this section, the method for evaluating the total duration of the certificate look up and the results are discussed.

5.2.1 Methodology

The total duration of certificate lookup refers to the time difference between the time when the client application requests to look up the certificate in the resource directory and the client application successfully receiving the certificate of the intended client in the look-up call back function.

This duration includes the time required for the processes shown in figure 5.1. First, the total duration includes the time for preparing the request for looking up the certificate in the resource directory. Secondly, it includes the communication overhead to send the request from the client to the resource directory. Thirdly the resource directory processes the request and prepares the C509 certificate, and signs it using the resource directory's private key. Then the duration includes the communication overhead from the resource directory to the client. Finally client processes the response, verifies the signature and stores the certificate if needed.

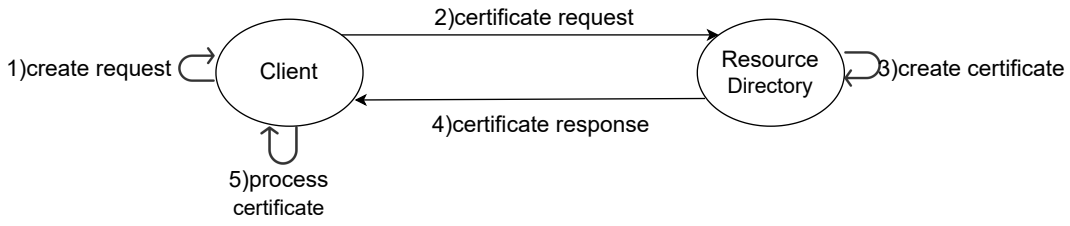

Figure 5.1: Experiment setup for timing measurements:The client creates and sends the request for certificate lookup.The resource directory creates the certificate and send the certificate as response.Client then processes the received certificate

The total duration can vary depending upon the changes in parameters described above. In addition, it also depends on the number of certificates stored in the resource directory because the resource directory needs to search through the list to find the intended client certificate. To understand how the number of certificates affects the total duration, measurements are also taken for varying numbers of certificates stored in the resource directory. So for a particular number of certificates stored in the resource directory, measurements were repeated 100 times to understand better the duration and how it is affected by the certificate search duration in the resource directory. The measurements were taken for both client and resource directory using crypto software library for signature creation and verification. Similarly, measurements were taken for client and resource directory using crypto hardware accelerators for signing and verification.

5.2.2 Result

The measured total duration for the certificate lookup are as shown in figure 5.2 and figure 5.3. The measurements were taken for varying number of certificates stored in the resource directory. The certificates stored were increased by steps of 500, and for each steps, measurements were taken. Figure 5.2 shows the total duration when both client and resource directory uses crypto-software for signing and verification. Figure 5.3 shows the total duration when both client and resource directory uses crypto-hardware acceleration for signing and verification.

It is expected that the duration will increase as the number of certificates stored in the resource directory increases. This is because of the fact that the certificate structures are stored in a linked list. The time to iterate through the linked list increases as the number of certificates stored increases. So the total duration should be proportional to the number of certificates stored.

It can be seen from the box plot that the total duration for certificate lookup increases for both cases when the number of certificates stored in the resource directory increases. For each set of measurements,the standard deviation is always close to 3ms and 4ms when using crypto-software. Similarly, when using crypto-hardware, the standard deviation is close to 1ms and 2ms.

In order to get a better understanding of total duration, measurements were repeated by keeping the number of certificates constant in the resource directory. In this measurement,

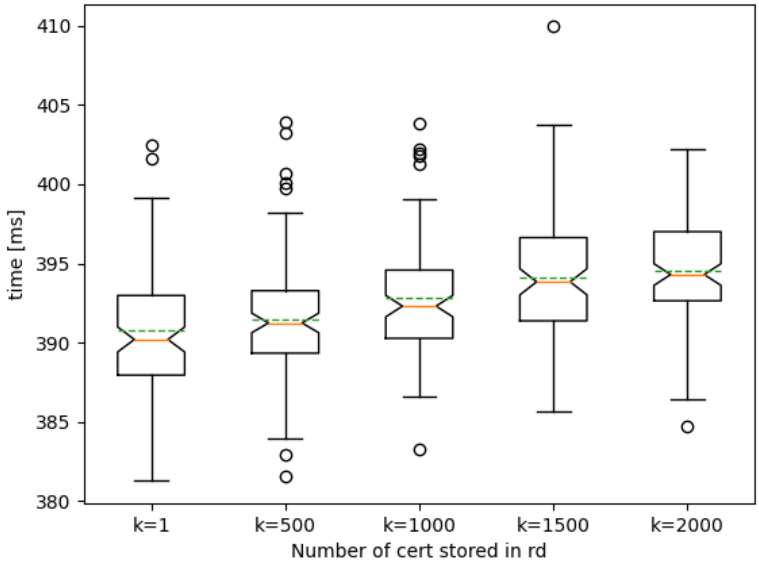

Figure 5.2: Total duration for certificate look up with crypto-software

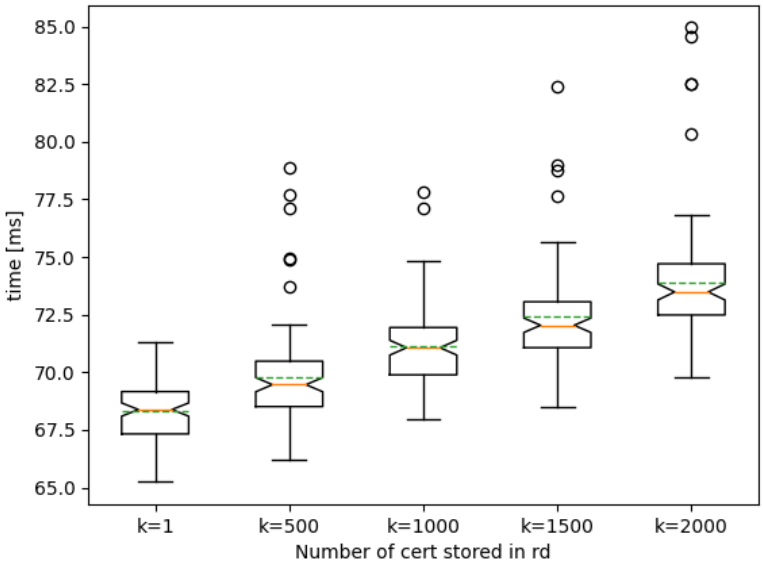

Figure 5.3: Total duration for certificate look up with crypto-hardware

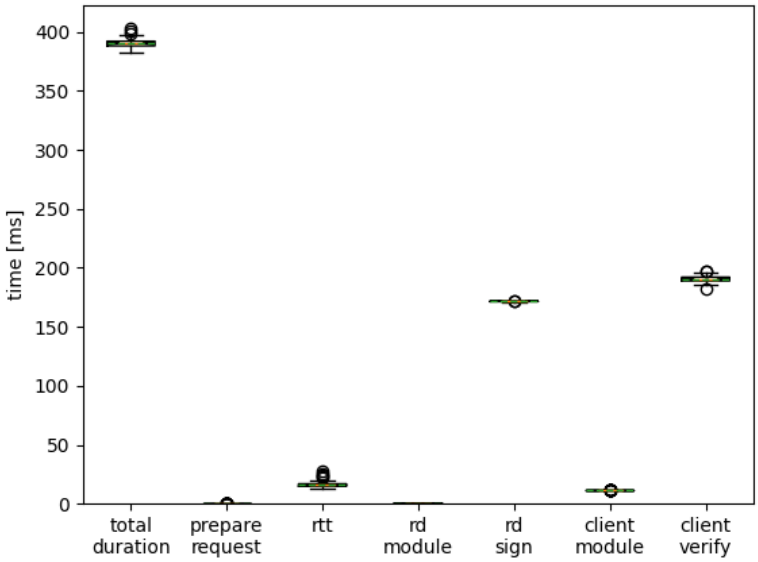

Figure 5.4: Total duration tear down with crypto-software

we calculated the time needed for each process that makes up the total duration as described in figure 5.1. The result of this experiment is shown in figure 5.4. It is identified from this experiment that, most of the time is spent on cryptographic operations. To be specific, certificate signing done by resource directory and signature verification done by client. Among the total total duration of 390 ms, signing takes up 171.5 ms, and verification takes 190.5 ms. The time to prepare look up request is 0.06 ms, and the round trip time is 16.2 ms. The resource directory takes 0.25 ms to process the request and encode the C509 certificate. Once the Client receives the certificate, it takes 11.6 ms to parse the C509 certificate.

The same experiment is repeated for the case when the client and resource directory uses crypto-hardware for signing and certification. The results are shown in figure 5.5. It can be seen from the figure that most of the time is taken up by the cryptographic operation, which is similar to the case when both client and resource directory uses crypto-software. The total duration for lookup is 68.3 ms. The resource directory takes 19.7 ms to sign the certificate and the client takes 20.7 ms to verify the signature. The time taken for all other processes is similar to that of the previous case when crypto-software is used. The client takes 0.06 ms to prepare the request and 15.96 ms for round trip time. The resource directory creates the certificate in 0.18 ms, and the client parses the response in 11.6 ms.

5.3 CPU Overhead

In this section,the method for evaluating the CPU overhead for processing C509 certificates and the results are discussed.

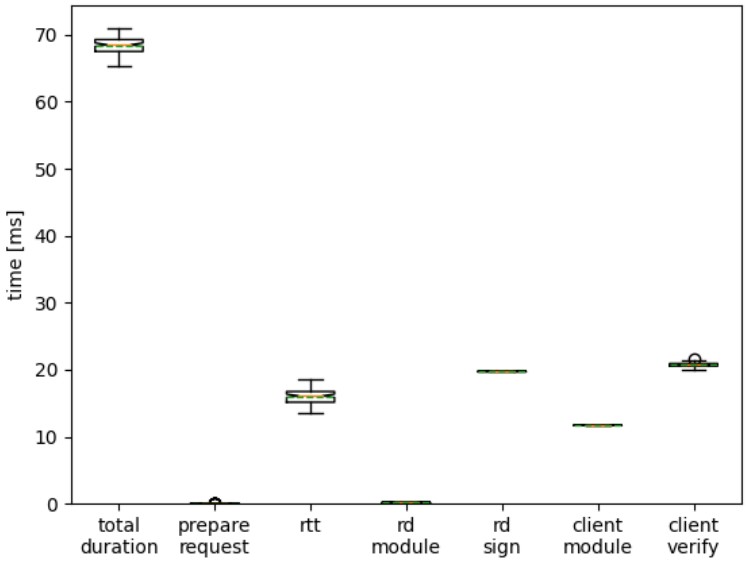

Figure 5.5: Total duration tear down with crypto-hardware

5.3.1 Methodology

In this evaluation, CPU overhead can be considered on the client as well as resource directory. On the resource directory, CPU overhead refers to the total time taken by the resource directory module to create the C509 certificate. So it is the time difference between the moment when the resource directory receives the certificate look-up request and the moment when it responds with the certificate.

The lookup request contains a CBOR map of client's name and the name of the client to be looked up. First of all, the CBOR request is parsed, and the sub-fields are separated. Resource directory checks if both of the clients are already registered with it. If yes, the resource directory iterates through the list to find the client's public key and common name. Afterwards, it encodes the details into C509 format and signs the certificate using its private key. The resource directory also includes its name in the response and sends it back. All the processes described above are considered as the CPU overhead on the resource directory side.

On the client side, CPU overhead refers to the time the client module takes to process C509 certificates. So it is the time difference between the moment when C509 is received at the coAP layer, which is then processed, and at the moment when the certificate is received to the application as certificate structure which contains the public key and common name.

Processing the C509 certificate includes parsing the CBOR lookup response, separating the sub-fields, verifying the signature using cryptographic functions, and storing the certificate. The response for the certificate look-up request contains a CBOR array of resource directories name, C509 certificate containing a common name and 33 byte compressed ECC public key and the signature signed by the resource directory. As a first step, the client

processes the response and separates the sub fields. The client then examines the validity of the received certificate. For this purpose, the client computes the hash of the C509 certificate. Then it searches the list for the public key of the resource directory using its received common name. Later on, the hash is verified against the signature using cryptographic operations. Upon successful verification, client stores the certificate. While storing the certificate, the client name and the public key is separated from C509 certificate. The public will be in compressed format. So it is also decompressed before storing in the list. CPU overhead refers to the time taken for the whole process explained above.

Either we can use the crypto-software library to verify the signature or the crypto-hardware accelerators. Measurements are repeated 100 times and compared to get a better understanding of CPU overhead. The measurements were taken for both client and resource directory using the crypto-software library for signature verification and signing. Similarly, measurements were also taken for both client and resource directory using crypto hardware accelerators for signing and verification. Switching between these two modes was done using the WOT_USE_CRYPTO_CELL option.

5.3.2 Result

The results of the analysis can also be taken from the Total duration figures 5.4 and 5.5 because it also contains the CPU Overhead. In the box plot, client-module refers to the time taken for the client to parse and CBOR response and store the certificate. It does not include the time taken for signature verification. client-verify refers to the time taken for signature verification. rd-module refers to the time for the resource directory to process the request and create the C509 format. rd-sign refers to the time taken to sign the certificate.

When crypto software is used 5.4 on the resource directory, the module takes 0.25 ms for creating the certificate and 171.53 ms for signing the certificate. So in total the CPU overhead is 171.78 ms. However, when crypto hardware acceleration is used 5.5, the signing time reduces to 19.73 ms and the total overhead becomes 19.91 ms.

When the crypto software is used for verification on the client,the signature verification takes on average 190.57 ms to complete. The time to process the CBOR response takes on average, 11.67 ms. So it takes 202.14 ms to completely process and verify the C509 certificate on nRF52840 dongle. If the crypto hardware accelerator is used for verification, the signature verification takes, on average 20.75 ms to complete. The time to process the CBOR response is similar to that of the previous case and takes on average 11.67 ms. So it takes around 32.42 ms to completely process and verify the C509 certificate on nRF52840 dongle.

It is also identified that most of the CPU time is consumed by cryptographic operations on the client and resource directory. Verification needs more time compared to signing in both modes of crypto operations.

5.4 Memory Usage

In this section,we discuss the method for evaluating the memory usage and the evaluation results.

Component	Client	Resource Directory
sys	85.7 KiB	69.3 KiB
cpu	8.5 KiB	8.3 KiB
core	4.7 Kib	7.2 KiB
drivers	1.6 KiB	1.6 KiB
boards	0.1 KiB	0.1 KiB
newlib	10.4 KiB	7.9 KiB
CBOR	1.8 KiB	2.3 KiB
µECC	5.3 KiB	5.3 KiB
doriot_wot	3.2 KiB	2.3 KiB
app	1.1 KiB	0.2 KiB
Total	122.4 KiB	104.5 KiB

Table 5.2: Memory Usage for Client and resource directory using crypto Software

5.4.1 Methodology

The memory consumed by the certificate management module is evaluated in terms of RAM as well as flash usage. To do this, the application is built for nRF-52840 dongle. In order to get better insights into memory usage, the python tool cosy [1] is used. Cosy performs a deeper analysis of the binaries to get detailed information on what is taking up the memory. Cosy is also capable of breaking down the binary up to compile units. In this manner, we get to know how each file contributes to the binary file size. Cosy also outputs how much data is used by text, data, and bss segments.

5.4.2 Result

The results of the memory usage evaluation are as shown in table 5.2 and table 5.3. The flash memory usage of client and resource directory applications when both uses crypto software for ECC operations are shown in table 5.2. Similarly the table 5.3 shows the flash memory usage when the client and the resource directory uses crypto hardware of ECC operations.

The *doriot_wot* client module uses 3.2 KiB of flash in both cases and *doriot_wot* resource directory module uses 2.3 KiB of flash. Using Crypto-hardware has a significant effect on flash usage. Crypto hardware alone takes up 35.7 KiB of memory. But when only crypto software is used, it consumes only 5.3 KiB of flash.

The RAM usage consumed by the client and resource directory mainly depends on the number of certificate structures stored. The structure `wot_cert_t` 3.2 is used to store the common name and uncompressed ECC public key of nodes in the network. The common name takes 8 bytes maximum, and the public key takes 64 bytes. In addition, the `wot_cert_t` contains `list_node_t`, which is used to reference list. The size of `list_node_t` is 4 bytes. So in total, we need 76 bytes to store information about a node in the network. When a resource directory needs to store information about 100 nodes, it requires 7600 bytes from the RAM.

[1] https://github.com/haukepetersen/cosy

Component	Client	Resource Directory
sys	63.8 KiB	69.3 KiB
cpu	7.6 KiB	14.9 KiB
core	2.9 Kib	7.2 KiB
drivers	1.6 KiB	1.6 KiB
boards	0.1 KiB	0.1 KiB
newlib	10.3 KiB	7.9 KiB
CBOR	1.8 KiB	2.3 KiB
µECC	2.0 KiB	2.0 KiB
Cryptocell	35.7 KiB	35.7 KiB
doriot_wot	3.2 KiB	2.3 KiB
app	0.9 KiB	0.2 KiB
Total	129.9 KiB	143.6 KiB

Table 5.3: Memory Usage for Client and resource directory using crypto Hardware

Component	Size(Bytes)
Client Common name	8 (maximum)
RD Common name	8 (maximum)
ECC Public Key	33
ECC Signature	64
CoAP headers	9
CBOR tags	10
Total	132

Table 5.4: Communication overhead

5.5 Communication Overhead

5.5.1 Methodology

The communication overhead caused by the certificate exchange depends on the size of the exchanged certificates and the overhead from CoAP headers. In the scope of this thesis, the C509 certificate contains only the common name of the client and the compressed version of the ECC public key. The signature generated by ECDSA also contributes to the overhead. But the certificate may also contain other fields like version number, serial number, validity, subject public key algorithm, signature algorithm, extensions etc. The look-up response in our implementation also contains the common name of the resource directory. This common name is important to the response because it is used to check if the response is from a trusted resource directory so that client can search for its public key in the list.

5.5.2 Result

The results of the communication overhead analysis are shown in table 5.4. The communication overhead during look up process depends on the size of CBOR message and CoAP headers. The C509 certificate contains 33 byte public key and name of the client. The common name length can vary depending upon the client. The Signature generated from ECDSA is of size 64 bytes. Also, the response contains the name of the resource directory, which can also vary depending upon the application. In addition,as every data field is encoded as CBOR data types, the CBOR tags for identifying the data types also contribute to the communication overhead. Finally, the CoAP layer adds header for identifying the content type and format. So according to table 5.4, 132 bytes are required for certificate exchange, if the common names of both parties are 8 bytes.

CHAPTER 6

Conclusion

This chapter summarizes the thesis and provides insights into future works. The initial goal of the thesis is described, and the contributions are summarized. The key points from the evaluation results are also briefly described.

6.1 Summary

In this thesis, a concept for a WoT based certificate management is proposed and implemented. Most of the IoT platforms available today are cloud-based. Provisioning of the edge devices must be done via cloud interface, and the certificates used for authentication and encryption must be managed by the cloud platform. The dependency on the cloud platform can be a single point of failure. Decentralized solutions based on WoT approach can be used to resolve the issue. But there are no communication protocols and key management infrastructure supporting this approach. So the goal of the thesis was to formalize and implement a concept for distributed certificate chain discovery protocol.

Firstly, we conducted a literature survey regarding WoT solutions for IoT and compared the related research. Even though there are many solutions proposed, the implementation part was missing. Also, when the devices in the network are constrained in resources such as memory, CPU and power, care should be taken to make the key management infrastructure lightweight. The solution also should be easily scalable and work with low communication overhead. So we implemented the key management infrastructure supporting the research described and evaluated by Engelhardt et al. [2]. The WoT topology introduced in their research reduces the communication overhead in IoT environments.

Secondly, we specified a protocol for certificate chain discovery, supporting the hierarchical WoT topology as proposed and evaluated in [2]. The protocol is based on CoAP, and the devices in the network exchange the certificate in C509 format. The network contains a resource directory, normally a non-constrained device, which stores the certificates of the nodes in the network. When a node enters the network, it finds the resource directory via CoAP resource discovery. Then the node registers with the resource directory. During registration, the node and resource directory exchange their certificates. In a similar fashion, other nodes in the networks also register with the resource directory. When a node wants

to communicate with another node in the network, it looks up the corresponding node's certificate from resource directory. In both registration and certificate look-up phases, cryptographic signatures are used for authentication and integrity checking certificates.

Thirdly we implemented the protocol for certificate chain discovery in RIOT OS. The concept is implemented as a module for RIOT OS. The *doriot_wot* module contains two submodules, one for the client and another for the resource directory. Applications can include the corresponding modules in the Makefile and utilize the functionalities. Application configurations can be easily modified using the Kconfig file by changing appropriate options.

Finally, we evaluated the solutions based on various parameters. We have measured the Total duration for certificate lookup, CPU overhead for processing and creating the C509 certificates, Memory usage and Communication overhead. The total duration of certificate look-up was found to be increasing as the number of certificates stored in the resource directory increased. This is expected behaviour since we are storing the certificate structures, containing the public key and common name, in a linked list in the resource directory. So as the list grows, the time to iterate through the list will also increase, increasing the total duration for certificate look-up. When generating and processing the C509 certificates, most CPU overhead is for cryptographic operations such as signing and verifying. So we provided two methods for cryptographic operations, one using crypto software and another using crypto hardware. From our measurements, it is evident that the crypto hardware implementation is much faster compared to crypto software. On the contrary, the memory usage for implementation with crypto hardware is higher than that of crypto software. The communication overhead mainly depends upon the size of the C509 certificates. In this thesis, the C509 certificate contains only the common name and public key of nodes. So the communication overhead is kept at a minimum. But a certificate may contain many other fields, which increases the size of the certificate.

The initial goal of the thesis was to develop a concept for WoT based certificate chain discovery protocol in RIOT OS. So with this thesis, we have implemented the concept, and it's working successfully on RIOT OS.

6.2 Future Work

In this section, the next steps for improving this thesis are discussed.

To improve upon this thesis, the first step would be to consider the whole hierarchical topology evaluated in [2]. This thesis focuses on a sub-section of the topology with a single resource directory and multiple clients connected to it. There could be multiple resource directories in the network, which can form a network among themselves. The protocol can be extended to support multiple resource directories and nodes connected to it. So the protocol becomes more scalable.

Secondly, the C509 certificates used in this certificate only contain the common name, public and signature of a particular node. But an X509 certificate may contain much other information such as version number, serial number, validity, Subject public key info, extensions etc. So as a next step, the C509 certificates can be extended to support more information.

Another important factor to be considered is the authentication of resource directories dur-

ing registration. In the current implementation, we are using PSK for authenticating resource directories. The signature of the resource directories certificate is created using PSK, using which the client also validates the certificate. But another authentication mechanism can also be implemented for this purpose. For eg. authentication using root certificates or using some OOB methods.

Finally, the resource directory functionalities could also be further implemented in a non-constrained device which can act as a storage unit for certificates of the nodes in the network.

Bibliography

[1] DoRIoT: Dynamic runtime for organically (dis-)aggregating IoT-processes. http://doriot.net/. lst accessed July 29, 2022.

[2] Frank Engelhardt and Mesut Güneş. Combined certificate and resource discovery for dynamically (dis-)aggregating iot processes. In Ralf H. Reussner, Anne Koziolek, and Robert Heinrich, editors, *INFORMATIK 2020*, pages 1215–1224. Gesellschaft für Informatik, Bonn, 2021.

[3] IHS. Internet of things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions). https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/, 2016. Accessed June 14, 2019.

[4] Florina Almenárez-Mendoza, Andrés Marín-López, Celeste Campo, and Carlos García. Ptm: A pervasive trust management model for dynamic open environments. *1st Workshop on Pervasive Security, Privacy and Trust in Conjuntion with Mobiquitous*, 01 2004.

[5] Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). RFC 7252, RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7252.txt.

[6] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. RFC 7228, RFC Editor, May 2014. http://www.rfc-editor.org/rfc/rfc7228.txt.

[7] C. Bormann and Z. Shelby. Block-wise transfers in the constrained application protocol (coap). RFC 7959, RFC Editor, August 2016.

[8] Z. Shelby. Constrained restful environments (core) link format. RFC 6690, RFC Editor, August 2012. http://www.rfc-editor.org/rfc/rfc6690.txt.

[9] Elliptic Curve Cryptography, Standards for EfficientCryptography Group, ver. 2. <https://secg.org/sec1-v2.pdf>, May 2009.

[10] S. Turner, D. Brown, K. Yiu, R. Housley, and T. Polk. Elliptic curve cryptography subject public key information. RFC 5480, RFC Editor, March 2009.

[11] C. Bormann and P. Hoffman. Concise binary object representation (cbor). STD 94, RFC Editor, December 2020.

[12] T. Bray. The javascript object notation (json) data interchange format. STD 90, RFC Editor, December 2017.

[13] H. Birkholz, C. Vigano, and C. Bormann. Concise data definition language (cddl): A notational convention to express concise binary object representation (cbor) and json

data structures. RFC 8610, RFC Editor, June 2019.

[14] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile. RFC 5280, RFC Editor, May 2008. `http://www.rfc-editor.org/rfc/rfc5280.txt`.

[15] John Preuß Mattsson, Göran Selander, Shahid Raza, Joel Höglund, and Martin Furuhed. CBOR Encoded X.509 Certificates (C509 Certificates). Internet-Draft draft-ietf-cose-cbor-encoded-cert-03, Internet Engineering Task Force, January 2022. Work in Progress.

[16] Y. Kitada, K. Takemori, A. Watanabe, and I. Sasase. On demand distributed public key management without considering routing tables for wireless ad hoc networks. In *6th Asia-Pacific Symposium on Information and Telecommunication Technologies*, pages 375–380, 2005.

[17] Hisashi Mohri, Ikuya Yasuda, Yoshiaki Takata, and Hiroyuki Seki. Certificate chain discovery in web of trust for ad hoc networks. In *21st International Conference on Advanced Information Networking and Applications Workshops (AINAW'07)*, volume 2, pages 479–485, 2007.

[18] Ankush Singla and Elisa Bertino. Blockchain-based pki solutions for iot. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 9–15, 2018.

[19] C. Amsüss, Z. Shelby, M. Koster, C. Bormann, and P. van der Stok. Constrained restful environments (core) resource directory. RFC 9176, RFC Editor, April 2022.

[20] RIOT: The friendly Operating System for the Internet of Things. `https://riot-os.org`. last accessed March 01, 2022.

[21] M. Aiman Ismail and Thomas C. Schmidt. A DTLS abstraction layer for the recursive networking architecture in RIOT. *CoRR*, abs/1906.12143, 2019.

[22] The Linux Kernel Development Community.Kconfig Language. `https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html`. last accessed April 04, 2022.

[23] Peter Kietzmann, Lena Boeckmann, Leandro Lanzieri, Thomas C. Schmidt, and Matthias Wählisch. A performance study of crypto-hardware in the low-end iot. Cryptology ePrint Archive, Paper 2021/058, 2021. `https://eprint.iacr.org/2021/058`.

[24] Tjerand Silde. Comparative study of ecc libraries for embedded devices. `https://tjerandsilde.no/files/Comparative-Study-of-ECC-Libraries-for-Embedded-Devices.pdf`,last accessed July 20,2022.

I herewith assure that I wrote the present thesis titled *Concept for a Web-of-Trust-based certificate management in RIOT OS* independently, that the thesis has not been partially or fully submitted as graded academic work and that I have used no other means than the ones indicated. I have indicated all parts of the work in which sources are used according to their wording or to their meaning.

I am aware of the fact that violations of copyright can lead to injunctive relief and claims for damages of the author as well as a penalty by the law enforcement agency.

Magdeburg, November 10, 2022

(Adarsh Raghoothaman)