Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Information Security and Communication Technology

Krishna Shingala

An alternative to the Public Key Infrastructure for the Internet of Things

Master's thesis in Communication Technology
Supervisor: Danilo Gligoroski, Katina Kralevska, Torstein Heggebø
June 2019

NTNU
Norwegian University of
Science and Technology

NORDIC®
SEMICONDUCTOR

NTNU – Trondheim
Norwegian University of
Science and Technology

An alternative to the Public Key Infrastructure for the Internet of Things

Krishna Shingala

Submission date:	June 2019
Responsible professor:	Danilo Gligoroski, IIK, NTNU
Supervisor:	Danilo Gligoroski, IIK, NTNU
Co-Supervisor:	Katina Kralevska, IIK, NTNU
Co-Supervisor:	Torstein Heggebø, Nordic Semiconductor ASA

Norwegian University of Science and Technology
Department of Information Technology and Electrical Engineering

Title:	An alternative to the Public Key Infrastructure for the Internet of Things
Student:	Krishna Shingala

Problem description: Internet of Things (IoT) enables participation of constrained devices on the Internet. Limited resources, bandwidth, and power on the devices have led to new protocols. Some examples of IoT driven and driving protocols are:

- MQTT, CoAP that are application protocols for IoT;
- 6LoWPAN enables efficient support of IPv6 on low power lossy networks;
- CBOR enables concise data formatting; and
- DTLS enables secure channel establishment over unreliable transport like the UDP.

Security is one of the key factors for the success of IoT. TLS/DTLS secures the channel between the servers and the devices. Confidentiality is an important aspect of such a secure channel. Establishing the identity of an entity another. Identity and Access Management (IAM) refers to identity based management of access rights.

IoT services use the new application protocols but reuse the security architecture from the already existing web service. The security of all services on the Internet today stand on the Public Key Infrastructure (PKI). The Certificate Authority (CA) provides the root of trust.

The PKI is designed, improved, and extended till date for communication between a web service and browser client operated by a user. The IoT devices neither have a user interface or the resources of a PC. These devices are not general-purpose clients that connect to many services. Hence, many features of PKI may be ill-suited for IoT.

The master thesis questions the relevance of PKI for the IoT. Efficiency and security are the two main aspects evaluated. In addition, a study conducted to explore any alternative trust models that do not need a certificate authority.

Note: The title of the thesis is updated based on feedback received during the presentation of the results at the faculty. The problem description and the scope remain unchanged.

Responsible professor:	Danilo Gligoroski, IIK, NTNU
Supervisor:	Danilo Gligoroski, IIK, NTNU
	Katina Kralevska, IIK, NTNU
	Torstein Heggebø, Nordic Semiconductor ASA

Abstract

Identity and Access Management (IAM) is an integral part of the security solution for IoT. Today, the cloud service providers determine the IAM schemes, and the constrained IoT devices implement them. The IoT services adapt to use an IoT application layer protocol like the MQTT; however, for IAM, the authentication of these services is identical to the web services and mandates the support for Public Key Infrastructure (PKI) on constrained clients that are no user-operated browsers.

On a constrained device, the verification of the X.509 certificate chain is resource intensive and requires information like the time - information that may be missing or unavailable on the device. Further, today, the resourceful and human-aided browsers struggle to effectively handle the security exceptions triggered by expired, revoked, and malicious X.509 certificates. These challenges only compound with sleepy, battery powered, and remotely operated devices with no possibility for human inspection.

PKI may not just be inefficient and demanding but can overwhelm a minimally equipped IoT device with a large and ever-expanding umbrella of trust. An IoT device connects to only a handful for services in its entire lifetime and needs limited trust. PKI may, therefore, be ill-suited for IoT. A study of the existing alternatives to the PKI provides no solution suitable for use in IoT. Besides, some alternatives like the OAuth-based federated identity, rely on PKI for service authentication.

We present a novel trust model, Vriksh: The Tree of Trust (VTT), tailored for use in IoT. This model aims to provide an embedded device-friendly entity authentication and limit the trust peripheries. With VTT, trust trees group the identities with equal access rights in the system using Merkle trees. We prototype the use of VTT with TLS raw public keys to compare the energy and resource efficiency of VTT with PKI on an embedded platform.

With VTT, we show new, efficient solutions for emerging use cases are possible without the invention of new cryptographic primitives or modifications to the existing ciphers in TLS. However, to establish VTT as an alternative to PKI, the verification of the proposed revocation methods for VTT and independent security reviews are essential. Moreover, PKI

with its wide-scale use, enjoys the privilege of constant improvement with extensive use, scrutiny, many attacks, and known vulnerabilities.

In the absence of ready alternatives, we make deployment recommendations for the use of PKI in IoT based on findings from this thesis. A question raised in the thesis, however, remains unaddressed for PKI - How to handle certificate expiry and revocation in remotely operated IoT devices? A critical topic to be pursued in the future to secure IoT.

Preface

The Masters and this thesis is an attempt to break free from a preoccupation - What motivates humans, in particular, me?

Nordic Semiconductor ASA and Norway offers a secure life. The realization of this secure life came the hard way. Acknowledgment of complacency had to come first. In India, all the education and Bachelors was the stepping stone towards a job, financial independence, and security. I concluded that the motivation is directly proportional to insecurity. Moreover, security has no relation to fulfillment.

The conclusion conflicted with the natural urge to evolve, to be fulfilled. I wondered what next? The appeal of research is high; the opportunity and support offered by the life in Norway conducive. The industry experience helped reveal enough unaddressed matters demanding attention and research. Factors that motivate me to take up part-time Masters alongside my employment at Nordic Semiconductor ASA.

The pleasure of study without pressure is a gift. A gift contributed to by many - Nordic Semiconductor ASA, NTNU, and the Norwegian taxpayers. Deep gratitude for the vast access to knowledge and an opportunity to be curious.

The curse of this gift is - the only limiting factor in my research, and all my endeavors is me. My imagination, aspiration, and capacity determine the benchmarks. Fortunately, I have always found support to live the curse - at work, at university, from friends and family!

So, the revised life equations are:

1. Security can create opportunity.
2. Motivation is proportional to the product of curiosity and opportunity.
3. We seek to be fulfilled. Hence the motivation is to be fulfilled.

So this leads to the new preoccupation is: Why do humans seek fulfillment?

Disclaimer: any information in the preface is personal, including and particularly the life equations.

Acknowledgment

Heartfelt gratitude to my supervisors, Danilo Gligoroski, Katina Kralevska, and Torstein Heggebø. They have encouraged me and provided timely, critical, yet constructive feedback through their review.

I have been fortunate to receive constant support throughout my part-time studies from Torstein as a boss, and Nordic Semiconductor ASA as an employer. The many colleagues at Nordic, through their interest and co-operation in my studies, have transformed my achievement into our collective achievement. No expression of gratitude exists for such inclusiveness.

Finally, many thanks to my friends and family that kept me human and sane through the overload of study during employment.

Contents

List of Figures

List of Tables

List of Acronyms

3GPP The 3rd Generation Partnership Project.

6LoWPAN IPv6 over Low-Power Wireless Personal Area Networks.

ACE-OAuth Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework.

AES Advanced Encryption Standard.

ASN.1 Abstract Syntax Notation One.

AWS Amazon Web Service.

BLE Bluetooth Low Energy.

CA Certificate Authority.

CASL Compute all, store less.

CBOR Concise Binary Object Representation.

CoAP Constrained Application Protocol.

CRL Certificate Revocation List.

CSR Certificate Signing Request.

CWT CBOR Web Token.

DAA Direct Anonymous Attestation.

DER Distinguished Encoding Rules.

DNS Domain Name Server.

DTLS Datagram Transport Layer Security.

ECC Elliptic Curve Cryptography.

ECDSA Elliptic Curve Digital Signature Algorithm.

EC-PAKE Elliptic Curve - Password Authenticated Key Exchange.

EPID Enhanced Privacy ID.

EST Enrollment over Secure Transport.

HPKP HTTP Public Key Pinning.

HTTP Hypertext Transfer Protocol.

IAIK The Institute of Applied Information Processing and Communications.

IAM Identity and Access Management.

IBC Identity Based Encryption.

IdP Identity Provider.

IETF Internet Engineering Task Force.

IM Subscriber Identification Module.

IMSI International Mobile SUbscriber Identity.

IoT Internet of Things.

IP Internet Protocol.

ITU International Telecommunication Union.

ITU-T ITU Telecommunication Standardization Sector.

JWT JSON Web Token.

MITM Man-in-The-Middle.

MQTT Message Queuing Telemetry Transport.

NIST National Institute for Standards and Technology.

NoT Network of Things.

NTP Network Time Protocol.

OCSP Online Certificate Status Protocol.

PAKE Password Authenticated Key Exchange.

PBC Pairing-Based Cryptography.

PKCS Public Key Cryptography Standards.

PKI Public Key Infrastructure.

PKIX Public Key Infrastructure X.509 Certificate.

PPK Power Profiler Kit by Nordic Semiconductor ASA.

PSK Pre-Shared Key.

RoT Root of Trust.

RSA Rivest–Shamir–Adleman.

RTC Real Time Clock.

SACL Store all, compute less.

SCEP Simple Certificate Enrollment Protocol.

SCPKI Smart Contract Based Public Key Infrastructure.

SDO Secure Device Onboarding.

SDSI Simple Distributed Security Infrastructure.

SHA Secure Hash Algorithm.

SLCO Store less, compute one.

SNI Server Name Indication.

SNTP Simple Network Time Protocol.

SPKI Simple Public Key Infrastructure.

SSH Secure Shell.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

TOFU Trust On First Use.

TPM Trusted Platform Module.

UDP	User Datagram Protocol.
VTT	Vriksh: The Tree of Trust.
WoT	Web of Trust.
WPA2	Wi-Fi Protected Access II.
WPA3	Wi-Fi Protected Access III.
ZKP	Zero Knowledge Proof.

Chapter 1

Introduction

1.1 Scope

The term Internet of Things (IoT) encompasses a vast variety of use cases. The simple and loose criteria of a constrained device connected to the cloud over the Internet can be met by a wearable, connected car, smart home appliance, asset tracker, predictive maintenance sensor and many more. Such a generalization is excellent for conceptualization and can identify common requirements for all the use cases. However, such generalization can mask the critical differences between the use cases and may result in inapt solutions.

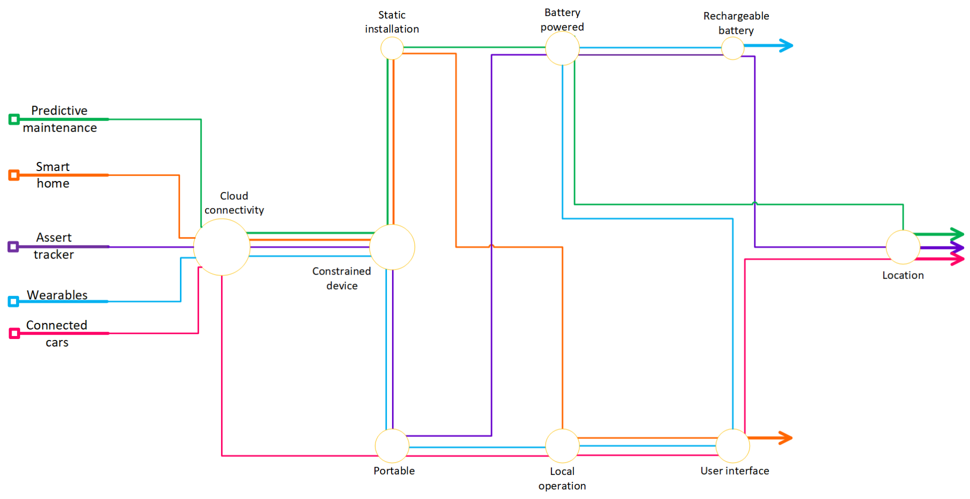


Figure 1.1: Some IoT use cases depicted as tracks and few requirements as junctions. All tracks diverge after the start at the 'Cloud Connectivity' junction.

Figure 1.1 depicts the commonality and differences between some of the common IoT use cases. In the figure, the size of the requirement junctions are proportional to the number of use case tracks passing it. The use cases should have 'Constrained Device' as a common requirement junction, not just 'Cloud Connectivity'. However, this is not the case. The definition of a Constrained Device can vary a lot. Here, Linux and user operated smart cars are not considered constrained. IETF in [BEK14] defines the terminology for constrained devices and classifies them based on energy, data size, and code size.

We cut short the discussion on the use case specific variations in requirements in Figure 1.1 to focus on the one common requirement - 'Cloud Connectivity'. We refine the scope further with Cloud Connectivity for devices that can run TLS, however, with constraints. IETF in [BEK14] classifies such devices as Class 2 devices.

1.2 Motivation

Identity and Access Management (IAM) is critical for the security of the Internet of Things (IoT) enabled systems. Insecure identity schemes can become launchpads to large exploits that can compromise the credibility of the system. Most identity schemes for the IoT devices employ the public key cryptography. Transport Layer Security (TLS) client certificates, JSON Web Token (JWT), and federated identity are common device authentication schemes.

In all the schemes, the device authenticates the server via the X.509 certificate chain during the TLS handshake. The certificate is trusted and verified via the root Certificate Authority (CA). This root CA must be pre-installed on the devices. The service authentication in IoT is identical to the authentication of a web service.

X.509 and PKI based authentication in IoT has practical challenges. Exchange of certificate chains per session demand radio resources, and power. To verify an entity an X.509 certificate, the prerequisites are knowledge of time and identity of the trusted anchors - the root CAs. The verifier stores root CA's certificates persistently and securely. Secure nonvolatile memory is very limited on many IoT device. PKI based authentication requires signature verification, ANS.1 and DER decoding of fields in the X.509 certificate for each certificate in the certificate chain. These are resource intensive operations.

Handling certificate expiry requires time information and user interface; both may be available on a constrained device. Today, a browser notify the user of revoked, expired, or malicious certificates to invoke user action. This strategy is unavailable to remotely operated IoT devices lacking a user interface. There exists no good recommendations and standards for handling these exceptions for IoT.

Most IoT service providers offer only X.509 certificate for server authentication. All IoT devices, therefore, must support PKIX to authenticate servers they connect.

Such a mandate, despite the limited resources, the missing user interface, and the inability to detect and report malicious certificates in a remote IoT device.

Another problem for IoT is the PKI architecture that allows spreading a wide umbrella of trust at the expense of poor visibility. The trust anchor does not know the entities endorsed by it. Such an architecture already adversely affects the users of existing web services. The constraints of an IoT device only worsen the situation. A server compromise that goes undetected has large-scale impact. A compromised CA has a larger impact, even if detected. The impact spans across services and systems. Recovering from the effects of such compromise is tedious even with human-aided browsers and operating systems.

A fundamental difference to consider between an IoT device and a browser is that a device connects to only a handful of services on the Internet in its entire lifetime. A browser, on the other hand, must offer its user the ability to access any existing and future web services. A browser, therefore, benefits as it well-utilizes the wide trust offered by the PKI. For, the minimally equipped IoT devices, the wider the umbrella of trust is, the larger are the possibilities of exploits. IoT makes case for tight and limited trust bounds.

Use of public key cryptography is the right choice for secure and scalable IoT solutions. In the absence of efficient and well-suited alternate trust models, the use of public key cryptography implies the use of X.509 and PKI.

IoT changes the landscape of assumptions, requirements, and constraints. Is PKI appropriate, relevant, and efficient for these new landscapes? Given that IoT devices connect only to a handful of services in their life cycle, is the umbrella of trust offered by a CA in a PKI too wide? Does such a wide trust make the constrained devices more vulnerable? The thesis focuses on exploring answers to these questions.

1.3 Objective and Methodology

The objective of this thesis is to find an entity authentication scheme suitable for the new challenges and requirements of IoT. PKI provides the trust model for all existing IAM in use. Therefore, we first, we study PKI in light of the constraints of an embedded IoT device. The study aims to establish the effectiveness and efficiency of PKI for IAM in IoT. This study described in Chapter 2, is biased towards constrained devices, as these are the weakest and most vulnerable links in the security chain.[1]

[1]It could be argued that human users are more vulnerable than any constrained machine, as they are prone to social engineering and very many mistakes. We leave this discussion out for another day.

The challenges with use PKI in constrained devices begins the hunt for a suitable alternative to PKI in Chapter 3. Efforts in this direction lead to a novel alternative trust model, the Vriksh: The Tree of Trust (VTT), proposed in the thesis. The alternative described in Chapter 4 designed to make public key cryptography usable in remotely operated IoT devices with no user interface. It is important to emphasize that this is no attempt to dethrone PKI in existing, well-functioning web services. Rather, to address new use cases that cannot be well-addressed with PKI.

A proof-of-concept and a limited prototype of VTT is built and presented in Chapter 5. The motivation for the prototype is to gather enough information to compare the use of VTT and PKI with TLS. Existing open source libraries portable on embedded platforms **mbed TLS** are used to build the prototype. We use OpenSSL to generate the necessary credentials. The code and scripts for the prototype are available on GitHub and necessary references provided in the chapter. The license on the default mbed TLS Apache 2.0 license.

In Chapter 6, we provide an evaluation of VTT and PKI on Cortex-M4 based embedded platform. Appendix Chapter A contains the embedded applications to measure energy consumption ECDSA verify and SHA-256. Notice that we measure the performance of the individual cryptographic primitives used with PKI and VTT. For energy consumption of the radio, we extrapolate the information gathered in the prototype with the model-based estimates. We finally combine the various results to compare PKI and VTT on the constrained platform.

With Chapter 7, we shift focus to some of the many ignored aspects and highlight the limited scope of the work in the thesis. Here, we acknowledge that many factors guide the security choices of a system. Further, we use the results from Chapter 6 to make concrete recommendations for the use of PKI in IoT. Chapter 8 contains some noteworthy observations, conclusions, and possible future work. We include an initial draft of a conference paper in Appendix B, the paper is a result of work done in this thesis.

Chapter 2
Background

Most web services today are access by human users via a browser client on a PC or mobile phone. User name and password authenticate the user. OAuth [Har12] and OpenId Connect [Ope05] enable Single Sign-On and federated identity. A web service issues user authorized, short-lived, and limited access grants to another service via JSON Web Token (JWT) [JBS15]. RFC7523 [JCM15] defines the use of JWT in OAuth. Regardless of the type of user and client authentication, the services typically rely on the X.509 certificate and the PKI to authenticate the server. The server provides its certificate during Transport Layer Security (TLS) handshake. HTTP [FGM+99] is the typical application layer protocol. RFC6818 [Yee13] defines HTTP over TLS. Figure 2.1 depicts a typical web service.

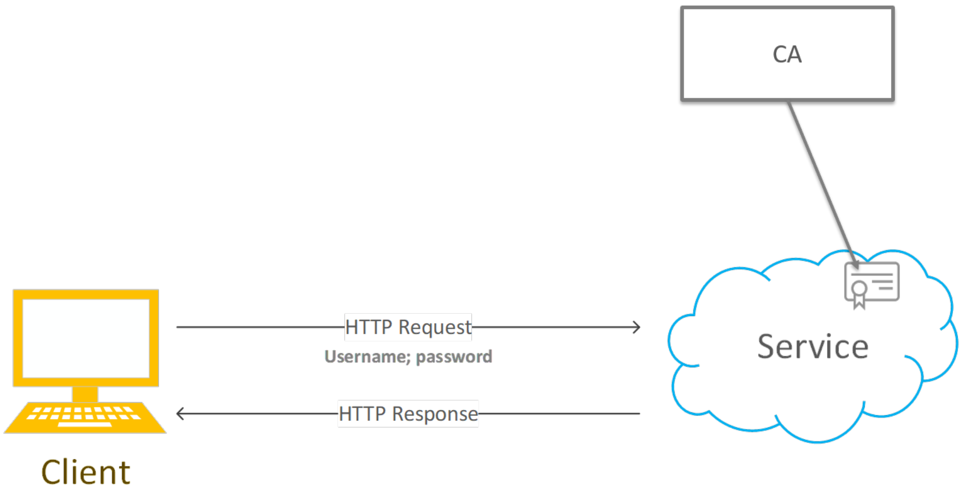


Figure 2.1: Authentication of a web service with X.509 certificate chain. The Web Service is authenticated before the client provides its user name and password.

For IoT, HTTP is resource intensive. New constrained device-friendly application protocols like CoAP [SHB14], and MQTT [BG14] are designed and deployed. Today, most cloud service providers support MQTT as the default IoT protocol.

New IAM schemes for IoT devices are emerging. For example, Google IoT Cloud [Cor19] uses a JSON Web Token (JWT) generated by the device to identify the devices. Internet Engineering Task Force (IETF) is drafting OAuth for use in constrained devices [SSW$^+$19]. All these schemes rely on TLS. The authentication of the IoT service is identical to any web service - X.509 certificate of the server provided during TLS handshake.

2.1 Trust Management with PKI

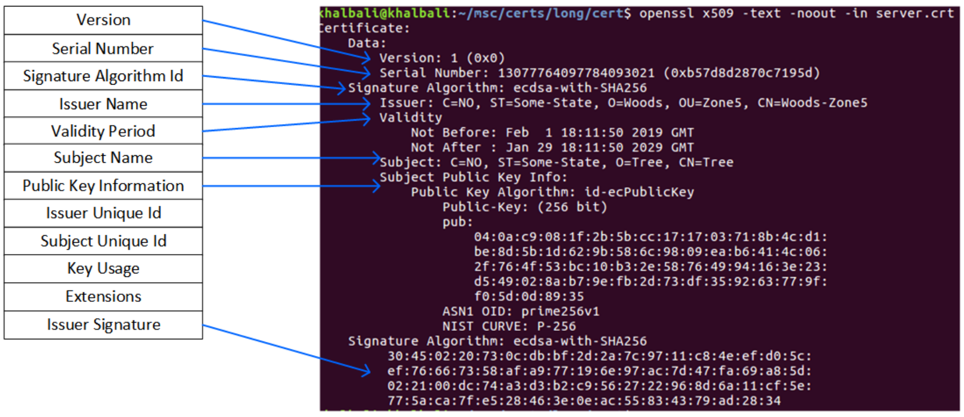
This section provides an overview of entity authentication and certificate management with PKI.

2.1.1 X.509 Certificate

X.509 certificate [CSF$^+$08] defines the public key format. The X.509 certificates bind a public key to an entity on the Internet. The entity, referred to as a subject in the certificate, can be an organization, a server, or a person. An X.509 certificate provides a digital identity to its subject. The digital identity is time limited. The certificate may include additional usage restrictions. The binding of the key with the subject is verified and certified by a certificate authority. The certificate, therefore, also identifies the issuer. The issuer stamps its certification on the binding with its digital signature. This digital signature must be included in the certificate for the certificate to be considered valid. Any entity that uses X.509 certificate proves possession of the private key corresponding to the public key in the certificate. Typical schemes used for proof of possession are signature schemes.

Figure 2.2 depicts a sample X.509 certificate in version 1 format and fields mandatory for version 3. The information in the X.509 certificate is ASN.1 DER encoded. Abstract Syntax Notation One (ASN.1) is a formal interface description language and in use in cryptography, telecommunications, and other uses. The ITU-T X.690 standard defines Distinguished Encoding Rules (DER) encoding rules for ASN.1. The public keys and the signatures in the certificate are base64 encoded. RFC4648 [Jos06] defines the base64 encoding rules.

A special case of X.509 certificate is the self-signed certificates. Here, the subject signs its bindings. Root CA certificates are typically self-signed. Figure 2.3 depicts a sample self-signed root CA certificate, used for the evaluation of PKI in this thesis. Notice that the subject and the authority key identifiers are identical. The hash of the public key is the identifier.



```
khalbali@khalbali:~/msc/certs/long/cert$ openssl x509 -text -noout -in server.crt
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number: 13077764097784093021 (0xb57d8d2870c7195d)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=NO, ST=Some-State, O=Woods, OU=Zone5, CN=Woods-Zone5
        Validity
            Not Before: Feb  1 18:11:50 2019 GMT
            Not After : Jan 29 18:11:50 2029 GMT
        Subject: C=NO, ST=Some-State, O=Tree, CN=Tree
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:0a:c9:08:1f:2b:5b:cc:17:17:03:71:8b:4c:d1:
                    be:8d:5b:1d:62:9b:58:6c:98:09:ea:b6:41:4c:06:
                    2f:76:4f:53:bc:10:b3:2e:58:76:49:94:16:3e:23:
                    d5:49:02:8a:b7:9e:fb:2d:73:df:35:92:63:77:9f:
                    f0:5d:0d:89:35
                ASN1 OID: prime256v1
                NIST CURVE: P-256
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:20:73:0c:db:bf:2d:2a:7c:97:11:c8:4e:ef:d0:5c:
         ef:76:66:73:58:af:a9:77:19:6e:97:ac:7d:47:fa:69:a8:5d:
         02:21:00:dc:74:a3:d3:b2:c9:56:27:22:96:8d:6a:11:cf:5e:
         77:5a:ca:7f:e5:28:46:3e:0e:ac:55:83:43:79:ad:28:34
```

Figure 2.2: X.509 Certificate Structure. On the left, the fields of the structure mandatory for version 3 of X.509 certificates are defined. Certificate verification includes validating various fields of the certificate, not just signature verification.

The certificate in Figure 2.3, with the basic constraints field, sets CA to true to indicate that the certificate holder can issue certificates to others. Figure 2.2 does not include this field, indicating that the certificate holder does not have the right to issue new certificates. All certificates contain mandatory and optional extensions based on the intended usage and the choice of deployment in the system. Many fields must be verified in a valid certificate and usage restrictions imposed accordingly. Ignoring such fields can lead to access escalation and exploits.

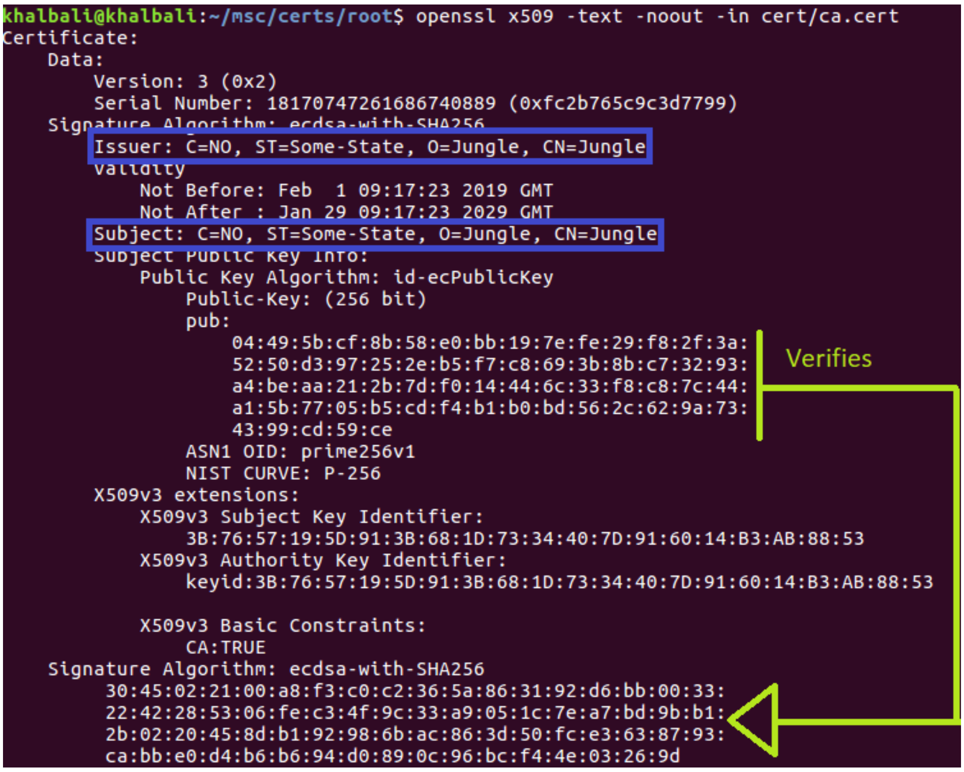
2.1.2 Certificate Issuance

Any entity that requires an X.509 certificate generates a Certificate Signing Request (CSR). This Certificate Signing Request (CSR) describes the entity and its public key. PKCS#10 standard [NK00] defines the format of a Certificate Signing Request. Figure 2.4 shows a sample CSR used in this thesis.

The issuer CA must verify the contents of each CSR before signing it. Notice that unlike an X.509 certificate in Figure 2.2, the CSR in Figure 2.4 does not contain any issuer information. Hence, any CA contacted with the CSR may sign it. Enrollment over Secure Transport (EST) [PYH13] and SCEP [PNV11] are two online enrollment protocols to request a CA to sign a CSR.

2.1.3 Certificate Validation

The root CA is the trust anchor in PKI. Figure 2.5 depicts a sample certificate chain with one intermediate CA and one entity. The end entity certificate is used to prove the digital identity of the entity on the Internet. The verifier of this digital identity



```
khalbali@khalbali:~/msc/certs/root$ openssl x509 -text -noout -in cert/ca.cert
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 18170747261686740889 (0xfc2b765c9c3d7799)
    Signature Algorithm: ecdsa-with-SHA256
        Issuer: C=NO, ST=Some-State, O=Jungle, CN=Jungle
        Validity
            Not Before: Feb  1 09:17:23 2019 GMT
            Not After : Jan 29 09:17:23 2029 GMT
        Subject: C=NO, ST=Some-State, O=Jungle, CN=Jungle
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:49:5b:cf:8b:58:e0:bb:19:7e:fe:29:f8:2f:3a:
                    52:50:d3:97:25:2e:b5:f7:c8:69:3b:8b:c7:32:93:
                    a4:be:aa:21:2b:7d:f0:14:44:6c:33:f8:c8:7c:44:
                    a1:5b:77:05:b5:cd:f4:b1:b0:bd:56:2c:62:9a:73:
                    43:99:cd:59:ce
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        X509v3 extensions:
            X509v3 Subject Key Identifier:
                3B:76:57:19:5D:91:3B:68:1D:73:34:40:7D:91:60:14:B3:AB:88:53
            X509v3 Authority Key Identifier:
                keyid:3B:76:57:19:5D:91:3B:68:1D:73:34:40:7D:91:60:14:B3:AB:88:53

            X509v3 Basic Constraints:
                CA:TRUE
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:21:00:a8:f3:c0:c2:36:5a:86:31:92:d6:bb:00:33:
         22:42:28:53:06:fe:c3:4f:9c:33:a9:05:1c:7e:a7:bd:9b:b1:
         2b:02:20:45:8d:b1:92:98:6b:ac:86:3d:50:fc:e3:63:87:93:
         ca:bb:e0:d4:b6:b6:94:d0:89:0c:96:bc:f4:4e:03:26:9d
```

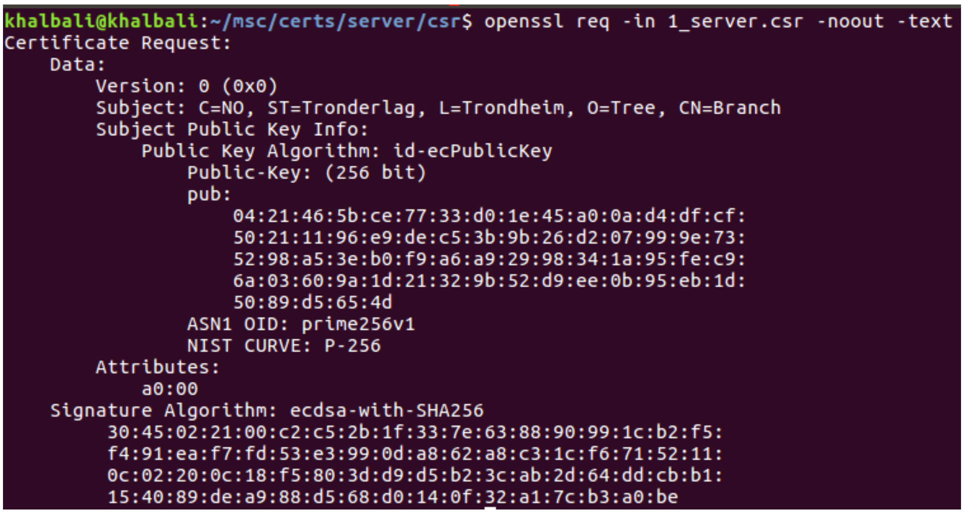
Figure 2.3: A sample self-signed X.509 version 3 certificate. The issuer and the subject are the same. The public key in the certificate verifies the signature in it.

trusts the root CA. The root CA certificate is expected to be pre-installed for the validation to succeed.

Many global and national organizations play the role of a CA. Around 150 root CA certificates are on browsers. Operating Systems also maintain their list of trusted root CA. **Root store** refers to such a bundle of trusted root CA certificates. On browsers, any update to the root store typically results in a firmware upgrade of the browser software.

2.1.4 Expiry and Revocation

Certificates expire when the validity period indicated in the certificate is reached. Knowledge of time is needed to detect expired certificates. A change in the subject's organization and detection of security breech are some of the many revocation reasons of a certificate.



```
khalbali@khalbali:~/msc/certs/server/csr$ openssl req -in 1_server.csr -noout -text
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=NO, ST=Tronderlag, L=Trondheim, O=Tree, CN=Branch
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:21:46:5b:ce:77:33:d0:1e:45:a0:0a:d4:df:cf:
                    50:21:11:96:e9:de:c5:3b:9b:26:d2:07:99:9e:73:
                    52:98:a5:3e:b0:f9:a6:a9:29:98:34:1a:95:fe:c9:
                    6a:03:60:9a:1d:21:32:9b:52:d9:ee:0b:95:eb:1d:
                    50:89:d5:65:4d
                ASN1 OID: prime256v1
                NIST CURVE: P-256
        Attributes:
            a0:00
    Signature Algorithm: ecdsa-with-SHA256
         30:45:02:21:00:c2:c5:2b:1f:33:7e:63:88:90:99:1c:b2:f5:
         f4:91:ea:f7:fd:53:e3:99:0d:a8:62:a8:c3:1c:f6:71:52:11:
         0c:02:20:0c:18:f5:80:3d:d9:d5:b2:3c:ab:2d:64:dd:cb:b1:
         15:40:89:de:a9:88:d5:68:d0:14:0f:32:a1:7c:b3:a0:be
```

Figure 2.4: A sample certificate signing request requested by the entity 'Tree'. The requesting entity signs the request with its private key. The issuer verifies the signature in the request with the public key in the CSR.

Certificate Revocation List (CRL) [CSF+08] is periodically updated and published by the CA to inform users of revoked certificates. To know the revocation status of a certificate, the users must search the serial number of the certificate in issuer's CRL. Delta CRL may be requested to optimize the search. With Online Certificate Status Protocol (OCSP) [SMA+13], the status of the certificate of interest can be queried to avoid searching in CRL.

The certification status checks require dedicated protocols on the side of TLS that use the certificates for authentication. Therefore, it is not adequate to implement TLS to authenticate an entity. Browsers, to protect their users, detect revoked and expired certificates for every TLS connection. Browsers generate user notifications on such security exceptions to get user input to continue or abort the TLS connection.

Also, it is important to note there is no way automated or digital method to detect malicious servers or use of certificates. Human users manually report suspicious use of a certificate. The Certificate Authority publish an email address to contact and request revocation of certificates. After reporting, a response time of 24 hours to investigate and revoke malicious certificates are typical.

2.1.5 PKI based entity authentication

Any prover, to authenticate itself, presents the X.509 certificate chain leading to the root CA to the verifier. The verifier is assumed to trust the root CA. Therefore,

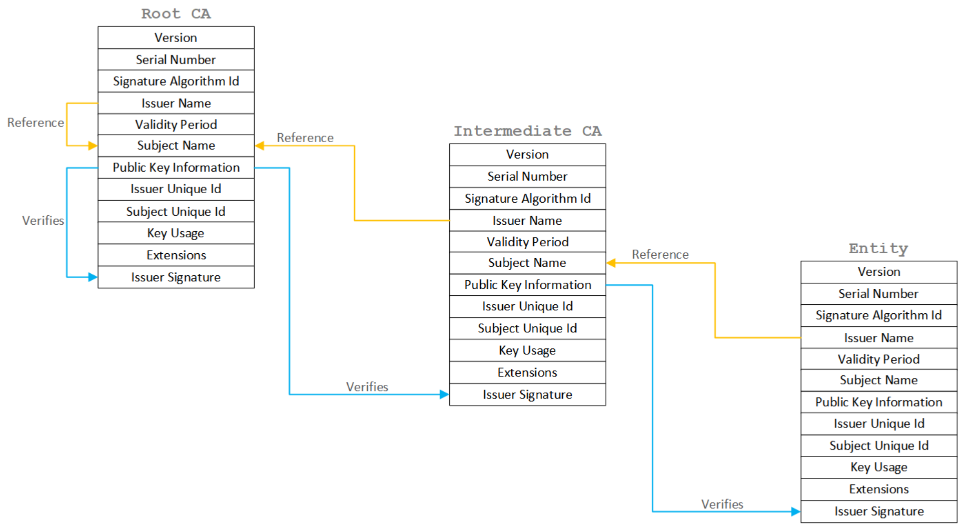
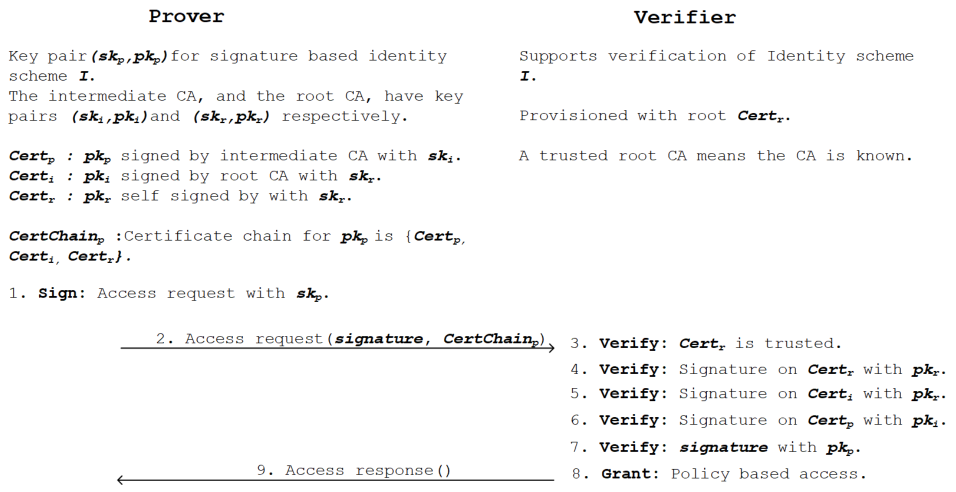


Figure 2.5: A sample certificate chain with self-signed X.509 version 3 certificate. The issuer and the subject are the same. Hence, the public key in the certificate verifies the signature in the certificate.

entity authentication with PKI requires validation and verification of each subject that propagates trust from the root CA to the prover.



Key pair (sk_p,pk_p) for signature based identity scheme ***I***.
The intermediate CA, and the root CA, have key pairs (sk_i,pk_i) and (sk_r,pk_r) respectively.

$Cert_p$: pk_p signed by intermediate CA with sk_i.
$Cert_i$: pk_i signed by root CA with sk_r.
$Cert_r$: pk_r self signed by with sk_r.

$CertChain_p$:Certificate chain for pk_p is {$Cert_p$, $Cert_i$, $Cert_r$}.

1. **Sign**: Access request with sk_p.

Supports verification of Identity scheme ***I***.

Provisioned with root $Cert_r$.

A trusted root CA means the CA is known.

2. Access request(***signature***, $CertChain_p$) $\longrightarrow$

3. **Verify**: $Cert_r$ is trusted.
4. **Verify**: Signature on $Cert_r$ with pk_r.
5. **Verify**: Signature on $Cert_i$ with pk_r.
6. **Verify**: Signature on $Cert_p$ with pk_i.
7. **Verify**: ***signature*** with pk_p.
8. **Grant**: Policy based access.

$\longleftarrow$ 9. Access response()

Figure 2.6: PKI based entity authentication with a signature scheme as the identity scheme. Authentication with PKI requires verification of the entire CA chain.

Figure 2.6 depicts a normative entity authentication using the PKI and X.509

certificates. The setup assumes one intermediate CA. In practice, there could be none or many. The signature scheme used by the CAs and the prover need not be the same. We use a common scheme for simplicity. The figure simplifies the contents of the certificates and verification of each certificate in the chain. In practice, each certificate contains many ASN.1 encoded mandatory, optional, and extension fields. Subject name, issuer name, issuer signature, and expiration date are few examples of such fields. The verifier must validate each of these fields before verifying the signature. Refer [CSF^{+}08] for details.

2.2 PKI Vulnerabilities and Attacks

In this section, we consider some of the vulnerabilities and attacks on the PKI. The scope is limited to most relevant to the subject of the thesis, and therefore, the list is not comprehensive. Wikipedia [con19b] provides a comprehensive history of risks and vulnerabilities of PKI since the year 1995. The most recent entry in the list is from the year 2017, a man-in-the-middle attack on a Brazilian Bank.

Figure 2.7 depicts trust propagation from the trust anchor to various entities on the Internet. An important feature of the PKI is its scalability - a single root CA can support many organizational trust structures and users without impacting its existing users. A downside of this feature, however, is that the root CA has no visibility of entities anchoring their digital identity on it. To elaborate, in Figure 2.7, the root CA, does not have any visibility into the number of certificates issued by the Organizational CAs. Therefore, the lack of visibility propagates along with trust.

All entities in the certificate chain, including the root CA, are vulnerable to attacks. As explained, in Section 2.1.3, the verification of the certificate requires verification of the certificate chain up to the root CA. The certificates do not carry any information about the depth of their certificate chains. Moreover, the verifier may not check the usage fields of the certificate. These factors can lead to the issue of malicious certificates and unauthorized certificate chain extensions by compromised parties.

2.2.1 CA Private Key Compromise

Reference [MS11] categorizes attacks resulting from compromised private key of a CA. The private keys are typically secured using a hardware security module and good security measures taken to ensure the use of the hardware security module. These measures may give an impression that CA private key compromise unlikely. However, they are not. One of the most publicized cases is that of DigiNotar [vDM13], a defunct Dutch certificate authority. The hardware security module was left plugged into a system on the network. A network-exploit resulted in the issuance of around

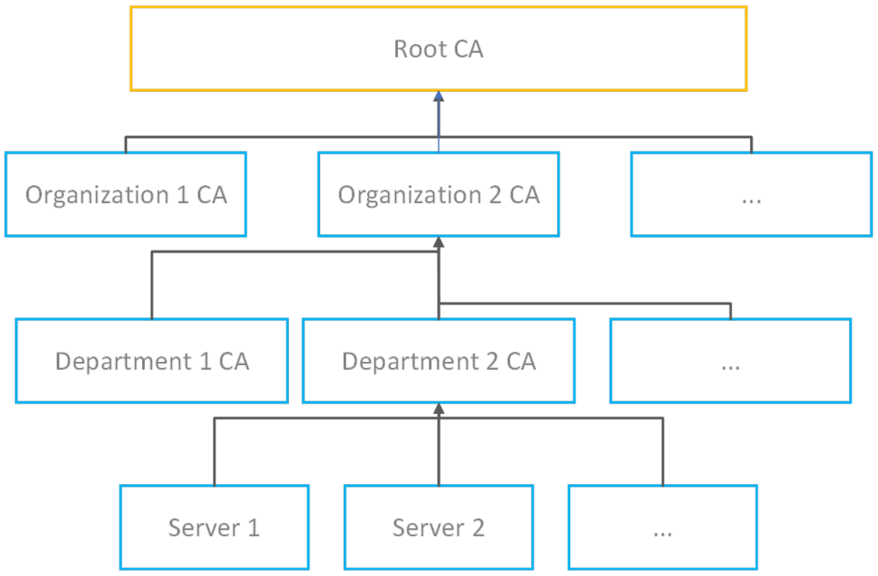


Figure 2.7: A sample PKI setup to demonstrate trust propagation from an issuer to its subjects. The issuer has visibility only of its immediate subjects.

500 bogus certificates. The bogus certificates were mostly used to impersonate service by Google, like the Gmail [Wik19c]. The bogus certificates were used to launch man in the middle attacks and spy on certain individuals of interest. The attack was large enough to revoke the trust in DigiNotar as a Certificate Authority.

Higher up the CA in the certificate chain, higher is the incentive to attack it. For example, in Figure 2.7, an attack on the root CA allows attacking domains across organizations.

2.2.2 Man in the middle attacks

Section 2.1.1 and Section 2.1.3 emphasize that entity authentication involves more than verification of the signature in the X.509 certificate of the entity. Overlooking the verification of additional fields in the certificate can result in Man in the middle exploits. Reference [MS11] provides an example of such an attack. Unverified **Basic constraints** field of the certificate results in bogus certificates for a domain passed on as valid.

Ensuring the hostname matches the subject name in the certificate is critical as well. Ignoring this can result in man in the middle attacks by adversaries that do not own the domain.

Today, browsers have patched most known vulnerabilities with the use of PKI. However, when deploying PKI to IoT, these attacks are worth revisiting.

2.2.3 Revocation

The default interval for publishing updated CRL is one week providing a window of opportunity to adversaries. During this period, an adversary can exploit users as revoked certificates continue to be trusted. Also, revocation checks require relying on an additional online service. A denial of service attack on such a service may imply the users decide to connect to web services regardless of revocation status, or, not having access to any service at all.

2.3 Use of PKI in IoT

As mentioned at the beginning of this chapter, for IoT, new application layer protocols are invented to be constrained device-friendly. However, the security architecture for IoT services remains identical that of web services. Notice that Figure 2.8 looks identical to a web service depicted in Figure 2.1, except for the application protocol.

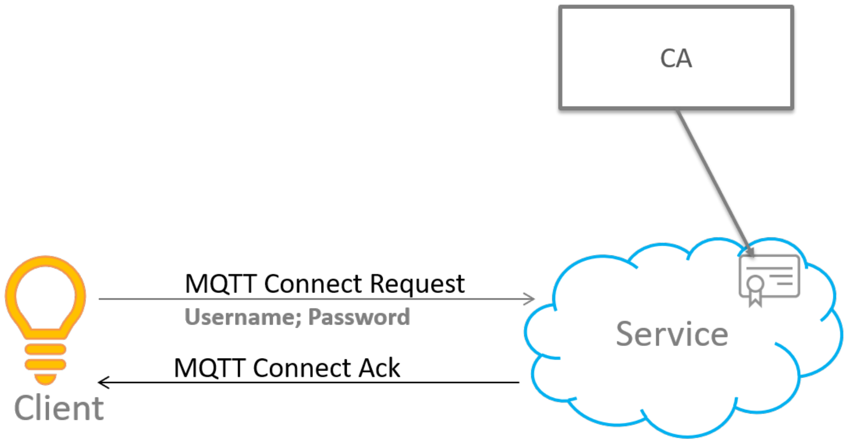


Figure 2.8: Authentication of an MQTT based IoT service using X.509 certificate chain. Here, the light bulb is symbolic of a constrained IoT Device. The IoT Service is authenticated using TLS before the client.

The need for uniquely identifying an IoT device and the lack of user interface on the device to input user name and password is acknowledged. Many schemes are in use today for servers to authenticate devices. Cloud service providers define these schemes. A comparison of JWT based authentication scheme with mutual authentication scheme is available at [Shi19a]. Google has defined the JWT based scheme for IoT. Amazon Web Service (AWS) mandates X.509 certificates for all clients at TLS handshake. These are two of the many possible mechanisms. Table 2.1 summarizes IAM schemes of some notable cloud providers.

As evident from Table 2.1, all the major IoT service providers rely on PKI for authentication of the server. Therefore, an embedded device must support X.509 certificate verification and install necessary root CA certificates.

Cloud Service Provider/ Authentication	**Client**	**Server**
Amazon Web Service	X.509 Certificate	X.509 Certificate
Google IoT Core	JSON Web Token	X.509 Certificate
IBM Watson	X.509 Certificate Access Token	X.509 Certificate
Microsoft Azure Cloud	X.509 Certificate, Access Token	X.509 Certificate

Table 2.1: IoT Client and Server Authentication schemes deployed by major cloud service providers. All cloud service providers offer the same mechanism for authentication of the service, TLS Server Certificate.

2.4 PKI Challenges for IoT

2.4.1 Time

Most embedded devices may not have a good way to implement time information. The battery-operated devices may enter power saving mode often to conserve energy and hence, may not be able to track time. Such devices may have the possibility to query time when online using the Network Time Protocol (NTP) [MMBK10] and Simple Network Time Protocol (SNTP) service [Mil06]. Such online services, however, are vulnerable to attack themselves. Threat models and security measures for time synchronization services are available at [Miz14]. Therefore, the dependency on time to detect expired certificates creates new attack surfaces on embedded devices. On PCs, user's observation of time and feedback to correct any suspicious time information serves as a defense against network manipulated time.

IoT devices with no user interface may need to implement multiple time synchronizations to gain confidence in date and time information in use. GPS can provide time information independent of NTP, hence a platform may support cross-verification of date and time. However, such automated countermeasures are expensive and resource hungry and not adequate. Therefore, validation of **Not Before** and **Not After** certificates fields in Figure 2.2 and Figure 2.3, is not reliable on IoT devices.

2.4.2 Revocation

An embedded devices must implement an additional service to query the revocation list or the status of a certificate received over TLS. An additional service implies additional resources on the system - more code and data memory, one more socket of the limited concurrent sockets managed by an embedded IP stack, and contention over the limited bandwidth with the other services. Moreover, a long certificate chain may imply querying the status of each certificate with the issuer. Besides, the unavailable or unreliable time information on the device makes revocation management unreliable.

Revocation is already a challenge for the browsers connecting to the existing web services. Browsers implement countermeasures for like HTTP Pinning [EPS15] and OCSP stapling [Pet13] in addition to protocols to check revocation status. HTTP pinning is available only for HTTP based clients. Hence, are unavailable for IoT that uses protocols like MQTT and CoAP.

2.4.3 Root Store

Embedded devices have little persistent storage as compared to PCs and mobile phones. Therefore, storing certificates of all global CA is not possible for an embedded device. Besides, as already mentioned, these devices connect to only a handful of services in their entire life cycle. Therefore, these devices do not need to implement a large root store like the browsers and the operating systems. Moreover, trusting many CA's expands the scope of vulnerabilities for an embedded device.

The current model of deployment for IoT devices is to provision root certificates based on the target eco-system. However, once provisioned, updating the root store to close any vulnerabilities may be a challenge. No good measures to address this aspect exist. Firmware update may be an option.

2.4.4 Lack of User Interface

Today, on PCs, the user decides the course of action to handle security exceptions firmware upgrades. They are aimed with many notifications and warnings to educate themselves towards a sound decision. Despite the warnings, the user has the final word, and cannot be forced to abort connections with malicious sites or install updates. For IoT devices that are only operated remotely and have no user interface, reporting security exceptions of the remote service to the service may be of little use. No standards exist to report the use of malicious certificates in devices with no user interface.

A 2016 study and analysis [Tae16] of 80 million certificates captured through network scan shows that 65% of certificates in use are invalid certificates. Further, the invalid certificates are reissued, meaning that invalid certificates would continue to be the silent majority on the Internet. The study traces the invalid certificates to specific categories of Internet users - devices like printers, modems, and VoIP phones. Is this a hint that some Internet participants cannot cope with the demands of the PKI?

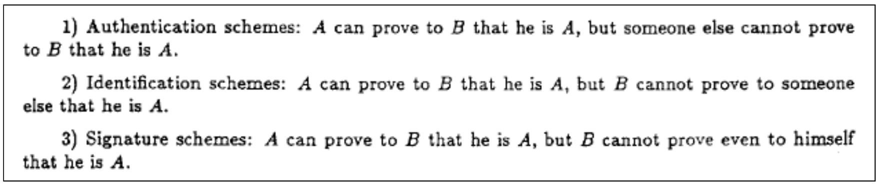
Chapter 3

Existing Alternatives to PKI

In this chapter, we discuss and assess alternative trust models to PKI for entity authentication. The assessment gives preference to usability in constrained embedded devices. This preference is not put before security. The usability, here, encompasses at least two aspects - efficiency of use, and limited user interface on the device. Before we study the alternatives, however, we define entity authentication and consider some relevant threats.

Entity Authentication

Bellare and Rogaway [BR94] point out that there is no common definition for entity authentication and provide the following informal definition. "Entity authentication is the process by which an agent in a distributed system gains confidence in the identity of a communication partner." Fiat and Shamir in [FS87] create three categories of protection when proving digital identity. Figure 3.1 is the snapshot of these three definitions from [FS87].



1) Authentication schemes: A can prove to B that he is A, but someone else cannot prove to B that he is A.

2) Identification schemes: A can prove to B that he is A, but B cannot prove to someone else that he is A.

3) Signature schemes: A can prove to B that he is A, but B cannot prove even to himself that he is A.

Figure 3.1: How to Prove Yourself: Classification of Identity schemes from [FS87].

In Figure 3.1, the authentication schemes guard A and B against external threats. To put this in the current context, IoT device and the cloud services do not compromise each other. Stolen remote devices that can compromise the cloud service's identity is undesirable. Only signature schemes provide the property of non-repudiation. With Identity schemes, a verifier can generate counterfeit transcripts by choice of challenge and response. For life-critical applications like insulin delivery, non-repudiation may be important.

In Section 3 of this chapter, we look at threats to identify the right level of protection needed for entity authentication. An IoT device may speak to third-party services and therefore may not have the degree of trust required with Authentication Schemes defined by Fiat and Shamir in Figure 3.1.

Mutual Authentication

Entity authentication of cloud service by the IoT device is as critical as the entity authentication of the device by the service. However, we do not use the term mutual authentication as the formal definition in [BR94] as it requires the use of a single scheme, that is, a common method to authenticate each other. The industry practice, however, is to use different schemes to authenticate clients and servers.

IoT Threats and Risks

NIST, in its special publication Network of Things (NoT) [Jef16] defines the common primitives to build a distributed system, including IoT. One of the aims of these definitions is to analyze security-risk trade-offs. The five primitives defined in [Jef16] are listed below with a short explanation in the current context.

– `Sensor`. Constrained IoT devices that measure or control mechanical, electrical, chemical, optical properties. Transmission of measurement data occurs on a `Communication Channel`.
– `Communication Channel`. Wired or wireless communication interface for information transfer between other primitives.
– `Aggregator`. An edge-node for data collection from clusters of sensors. The data may be transformed before forwarding to other components.
– `eUtility`. Composite of current and future cloud service components. Database, web hosting services, and management portals are some examples of `eUtility` components. `eUtility` is computationally more capable than `Aggregator`.
– `Decision Trigger`. Rules to invoke actions based on data events. Decision triggers may provide immediate feedback or generate predictive information.

Figure 3.2 depicts the interaction between the various primitives. Here, to study the threats and attacks, we use only the `Sensor`, `Communication Channel`, and the `Aggregator` primitives. We include the `Adversary` primitive to explain the attacks better. An `Adversary` is any entity with malicious intent, internal or external to the NoT.

In [DZH18], the authors provide a comprehensive list the security requirements and possible attacks. Of the various security objectives, entity authentication and non-repudiation are security objectives relevant to the current scope. Some relevant attacks are similarly listed below. To explain the attacks in context, we use primitives from [Jef16].

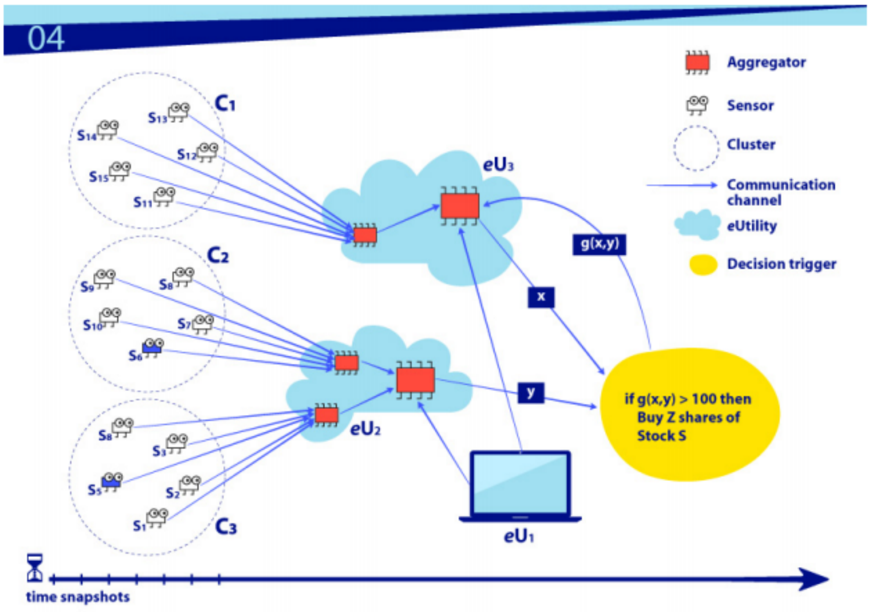


Figure 3.2: A sample NoT model to demonstrate the interactions between the primitives. The Sensor primitive symbolizes a constrained device and communicates with the Aggregator primitive. In this thesis, the Aggregator symbolizes the cloud service endpoint that all devices communicate with over a secure channel.

- **Replay attack.** The `Adversary`, with the intent to mislead, or gain unauthorized access, records one or more sessions on the `Communication Channel` between a `Sensor` and `Aggregator` to later play it again to one one of the parties.
- **Stolen Verifier.** If the `Aggregator` stores the device authentication information, then the `Aggregator` once stolen, or compromised, can compromise all the `Sensor`.
- **Stolen Device.** If `Sensor` stores authentication information of the `Aggregator`, then a stolen `Sensor` can compromise the `Aggregator`. We take the liberty to include unwiped secrets compromised from disposed, end of lifecycle devices in this category.
- **Man-in-the-middle.** `Adversary` intercepts the `Communication Channel` between the `Sensor` and `Aggregator` with the intent to eavesdrop, amend, append and/or withhold data.
- **Privileged-insider attack.** A trusted user, typically an administrator the NoT system turns into an `Adversary`. The `Adversary` uses the knowledge of the system to alter system behavior or hold the system hostage illegitimately.
- **Impersonation attack.** The `Adversary` successfully assumes the identity of a legitimate entity in the system with the intent to compromise the data integrity of the system. Sybil attacks may be considered a subcategory. Here, the `Adversary`

assumes many identities in the system.

Many of these attacks can be partially or entirely mitigated using strong Identity schemes. The Identity and signature schemes from Figure 3.1 do not require the verifier to act in good faith. A downside of the primitives identified by NIST is that it does not include any primitives for third-party trust anchors like CA that are critical for identity verification of the primitives.

At each session, introducing an element of freshness on identity verification can mitigate **Replay Attacks**. Further, curtailing damages of the compromise of one entity, to that entity alone can ensure small, constrained devices do not become launchpads for bigger, system-wide compromise. However, to limit the scope of our analysis to the relevant attacks for this thesis, the passive attacks that compromise user privacy are not discussed.

3.1 Symmetric Keys

Password or PIN-based authentication is in wide use today. Authentication in the wireless networks and user authentication by web services over the Internet is based on pre-shared key. Wi-Fi Protected Access II (WPA2) [Gro04] is the default authentication protocol for IEEE 802.11 based Wi-Fi networks. The current generations of 3GPP standards - 2G, 3G, and 4G, all use symmetric keys for authentication of modules connecting to the cellular network. Bluetooth Low Energy (BLE) uses a maximum of 6-digit PIN in its 4.0 specification [Blu10]. TLS supports pre-shared key based ciphers since its inception. For TLS bulk encryption, all ciphers use symmetric keys. The symmetric key is established during the session and depends on the cipher. Public key cryptography is too expensive for bulk data and hence used only during handshake for authentication and key agreement. In HTTP, user authentication is password based. Most layers of the Internet stack - network, transport, and the application use symmetric keys. Given this, it is time to address why the thesis insists on the use of public key cryptography for IoT.

Figure 3.3 demonstrates that the sender does not verify the identity receiver. The receiver relies on the knowledge of the key to ensure the message originated from a legitimate sender. The receiver never explicitly verifies the identity of the sender. Symmetric keys are not designed for the unique identification of entities. Further, comprised secret compromises all the entities that share the common secret. A typical technique to limit compromise to use pairwise secrets. An example use of pairwise keys is in cellular networks. The network providers distribute the SIM cards with the shared secret known to the IM and the Home Subscriber Register. The IM identified by IMSI is authenticated to have the secret knowledge on each network attach request. Use of a common sequence number is a common mitigation against replay attacks. IM is expected to store the shared key in tamper-proof hardware.

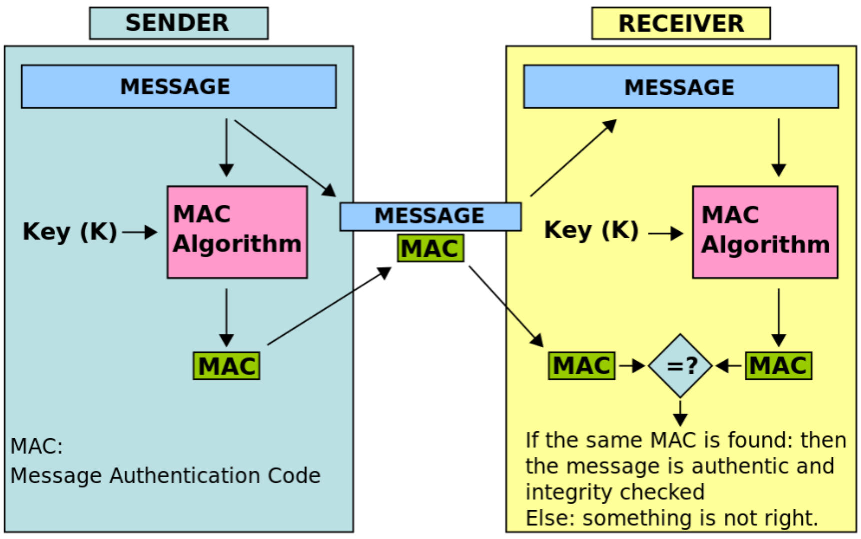

SENDER
MESSAGE
Key (K)
MAC Algorithm
MAC
MAC:
Message Authentication Code
MESSAGE
MAC
RECEIVER
MESSAGE
Key (K)
MAC Algorithm
MAC
=?
MAC
If the same MAC is found: then the message is authentic and integrity checked
Else: something is not right.

Figure 3.3: Message Authentication: Knowledge of a shared secret is to verify if the message arrived from a trusted sender. Source [Wik19e].

Symmetric keys are vulnerable to online and offline dictionary attacks. Weak passwords further reduce the strength of symmetric keys. Cracking Wi-Fi password and authentication in university courses is a fun way to grasp the vulnerabilities of passwords. Dragonfly Key Exchange [Har15] introduced in WPA3 resist some of these attacks. Dragonfly Key Exchange is an instance of the Password Authenticated Key Exchange (PAKE), where the password is used to derive the parameters for key exchange. ITU standard [oI07] provides recommendations for the use of PAKE. Thread Network [Gro17] uses EC-PAKE for adding new devices to the network. PAKE can ensure strong security even with weak passwords and invalidates brute force attacks. For applications with many participants to gain access to a shared resource, PAKE enables a user-friendly yet secure solution.

Based on the classification provided by Fiat and Shamir in Figure 3.1, PAKE is an Authentication scheme. Despite the protection against weak passwords, compromised passwords can enable new rogue devices to gain access to a shared resource. Stolen Device, Stolen Verifier, and Privileged-insider attacks remain unmitigated even with PAKE.

Symmetric keys, though simple and computationally fast, are not designed for unique identification of entities. Non-repudiation cannot be achieved the key to prove and verify the identity is the same. Key distribution, renewal, and revocation remain key challenges with symmetric keys.

3.2 Simple Public Key Infrastructure

Authors of Simple Public Key Infrastructure (SPKI) in [Ell99] and [EFL$^+$99] argue that the binding of globally unique subject name with its public key with X.509 certificates is of little use. SPKI certificates use Simple Distributed Security Infrastructure (SDSI) [Riv99] names, defined in the local scope instead of globally unique identifiers. SPKI advocates binding the public key with granted authorization access to it. Therefore, the SPKI certificates identify permissions to access resources and invoke actions in the system. SPKI certificates are expressed in LISP format. Unlike the X.509 certificates, SPKI certificates contain sensitive information about the system, and hence considered private. SPKI certificates must be easy to parse and accurately express the authorization. The aim is to enable embedded devices like smart cards to participate in the system and avoid errors from elaborate, ambiguous authorization. SPKI and SDSI introduce group certificates to define consistent authorization for multiple keys. Short-lived certificates for fine grain access control are possible to implement with SPKI.

Similar to X.509 certificates, SPKI certificates have a validity period, and CRLs manage certification revocation. SPKI proposes positive and negative online validation of entities, instead of checking only for revoked entities. Commercial third-party CA has no role in SPKI, and therefore, it is undefined as who issues certificates and revocation lists. Wikipedia suggests deployment of SPKI in private systems. Further, the inability to monetize SPKI and lack of business case to develop any SPKI tools. Today, there is no implementation of SPKI available with TLS. Reference [Vid05] provides a formal analysis of SPKI with TLS. Authors in [KL07] propose a model to enable access control between peers with SPKI. A central server manages certificates and authorization in the system.

Many of SPKI requirements resonate with IoT requirements. For example, simple certificates, redundant global names, and, group-based access control. However, like the PKI, the solution depends on auxiliary services like the time, online revocation status. SPKI specifications remain experimental draft since 1999. No effort has been made to address known issues identified in the drafts. For example, the delegation of trust without resources owners knowledge, and no mechanism to control the depth of delegation remain open. To elaborate on the impact of these issues in IoT, let us consider an example. A firmware upgrade server with authorization to upgrade firmware to IoT devices may delegate the authority to another service. This service can further delegate the authorization. Each delegator adds its signature to the delegated certificate. An IoT device may have to accept firmware from a strange entity after all the signatures in the SPKI certificate verify. Recall that PKI suffers a similar loss of visibility with trust propagation. SPKI has repackaged the issues with PKI for IoT, not addressed them.

3.3 Federated Identity

As mentioned in Chapter 2, OAuth is widely used for web services today. ACE-OAuth [SSW^{+}19] brings OAuth to the constrained world. Figure 3.4 depicts the basic protocol between the client (constrained device), an Authorization Server, and resource server. The response rejecting an access request due to inadequate authorization may include the address of the authorization server. All clients communicate with the Authorization Servers using CoAP. By default, clients authenticate an Authorization Server using their X.509 certificates during DTLS handshake. [SE17] defines authentication methods for the client. Resource Servers are typically authenticated using PKIX certificates TLS/DTLS handshake. The access claims may be expressed in JWT [JBS15] or the more concise form CWT [JWET18].

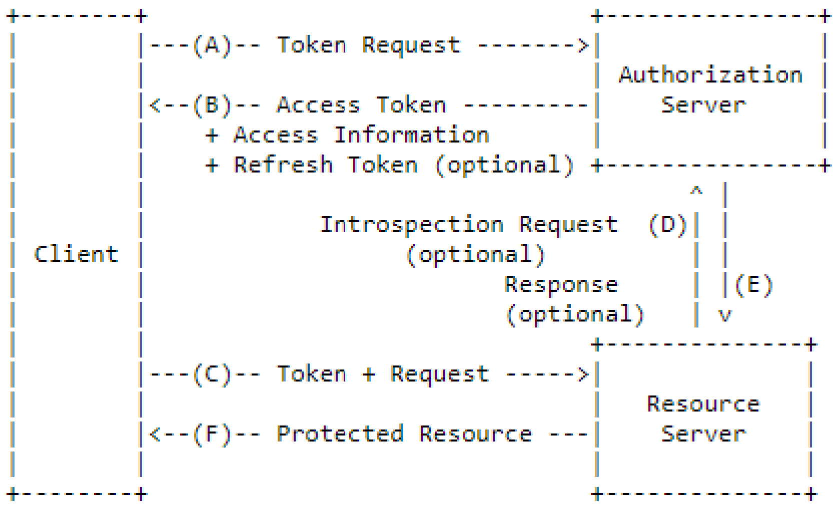

Client
Authorization Server
---(A)-- Token Request ------->
<--(B)-- Access Token ---------
+ Access Information
+ Refresh Token (optional)
Introspection Request (D)
(optional)
Response
(optional)
(E)
---(C)-- Token + Request ----->
<--(F)-- Protected Resource ---
Resource Server

Figure 3.4: ACE-OAuth: Basic protocol. Source [SSW^{+}19]. The authorization server is typically pre-provisioned in the client.

For federated identities with ACE-OAuth, Authorization play the role of IdP. The constrained devices prove their identity to a designated IdP. The IdP vouches for the devices to other service providers (resource servers). Reference [FA18] measures an OAuth-based federated Identity scheme on an embedded platform.

Federated identities can simplify credential management for the constrained devices with single credentials to authenticate towards many services. Further, an IdP can provide access revocation services. The catch, however, is that the framework relies on TLS and PKI authentication to authenticate the servers, including the IdP. Therefore, ACE-OAuth is not independent of PKI.

3.4 Web Of Trust

In human societies, existing friends introduce one to new friends. With time and experience, the friend circle is expanded and trimmed. Web of Trust (WoT) emulates this in the digital space. WoT, used in OpenPGP, allows a trusted user to introduce a stranger. Trust from know entity to the unknown is propagated using digital signatures. Physical, face to face verification of key, and thorough inspection of associated attributes is required before signing the certificate of the new user. RFC4480 [SGKR06] defines the certificate and signature formats. With WoT each user builds and maintains its trusted set of certificates. Certificates can expire and include validity field expressed in time. Therefore, with time, the trust circle expands and contracts.

Could WoT be used with IoT? One device inducing another device into the system is not far fetched. The resurrecting duckling [Sta02] draws a similarity between a duckling and an IoT device. A just-hatched duckling would become the ward of any creature it first sees. This guardianship typically belongs to the mother duck, but not necessarily. An IoT device is designed to be as impressionable as a duckling with the intent to initiate the device into the target eco-system. A dedicated provisioning device or an already initiated device may take on the role to join new devices in the eco-system. Therefore, such a device could share its trust circle with a new device.

OpenPGP is a network of equals - each person has the same rights to send emails, attachments, and files to one another. In IoT, not all participants are equally capable, and hence, functionality and access rights cannot be the same. IoT benefits from a layered architecture with clear access policies. A device-to-device network automatically extending trust and expanding the circle of trust of services may compromise the security of the entire system. Usually, initiating a new device in the system requires a secure environment, and so does extending trust. Such a secure environment is not possible with wireless networks. Human-aided provisioning does not scale. Also, storing cross-signed certificates from individual entities with their access right may be inconvenient for all entities, especially the constrained devices.

A constrained IoT device has limited persistent memory to store trusted certificates and connects to a handful of services. WoT if used for IoT needs modifications to layer trust and add mechanisms to restrict trust propagation based on access policies. Besides, the complex question of efficient trust revocation remains.

3.5 Pairing Based Encryption

Pairing-Based Cryptography (PBC) pairs elements between two groups to a third to construct a cryptographic system. Three-party one-round Diffie–Hellman key agreement [Jou00] secure is one of the first use of PBC. IBC, rooted in PBC, has become a research topic of its own. PBC maps are in general expensive, with not many efficient implementations available. However, PBC based research as an alternative to PKI is popular and looks promising. Therefore, this is covered here for the sake of completion and future possibilities.

3.5.1 Certificate-less Public Key Cryptography

Certificate-less Public Key Cryptography [ARP03], as the name suggests, enables public key cryptography without the use of certificates. IBC is extended to include authentication. The solution relies on a trusted key center to transmit private keys to registered users. User keys are derived from their identities and the private key of the trust center. Similarly, the public key of an entity is generated using the public key of the trust center and the identity of the entity. Topics of key regeneration, expiry, and revocation are untouched.

3.5.2 Milagro

Milagro [Bud16] aims to make public key cryptography available in TLS without need for any central authority. PBC based ciphers with distributed authorities dissimilar but compatible with Bitcoin blockchain. Figure 3.5 depicts a sample system with key share distributed to cloud service provider and the customers of the service provider in addition to a distributed trust authority provider. The client and server entities in the system are depicted to derive their keys/identities on interaction with all of the trust authorities.

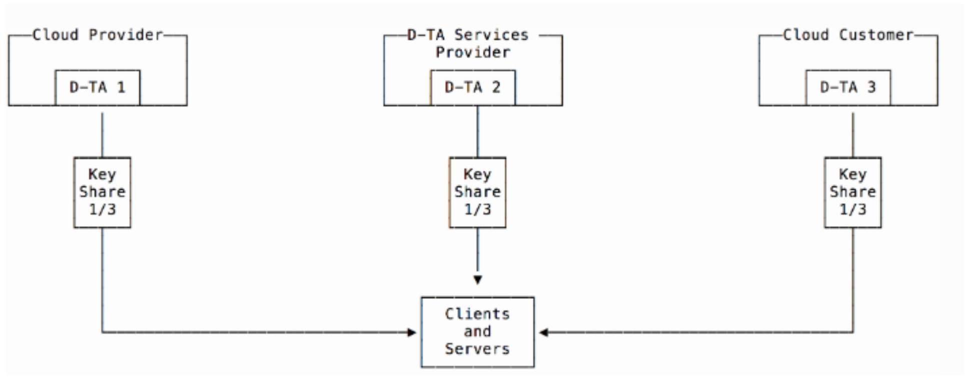


Figure 3.5: Milagro: Distributed Trust Authority. The stakeholders in the system distribute the role of trust authority among themselves. Source [Bud16]

.

An incubator project [McC16] is created to which has remained dormant since its inception. The white parer [Bud16] suggests the use of ZKP-based multi-factor authentication for user authentication of cloud services. The target platforms are PC and mobile phones. Not many details are available at this point to evaluate this proposal for use in IoT.

3.6 Enhanced Privacy ID

Enhanced Privacy ID (EPID) [BL12] proposes an Identity scheme using Direct Anonymous Attestation (DAA) [Ern04]. Under the strong RSA assumption, EPID and DAA offer remote and anonymous attestation. EPID is deployed with recent Intel-based laptops to identify legitimate products in the field.

In DAA, a remote attestation service verifies an anonymous signature to authenticate legitimate TPM hardware. The use case for DAA is to prove the authenticity of products that integrate TPM without compromising the user's privacy. DAA enables signature based revocation under one condition - extraction of the private key from the TPM module. The verifier can check if a signature arrived from a revoked private key. EPID extends DAA to enable revocation even without the knowledge of the private key.

EPID is deployed with recent Intel-based laptops. Intel's Secure Device Onboarding (SDO) [Int18] enables provisioning of legitimate IoT devices. EPID embedded in a TPM identifies a device. The solution focuses on authentication of devices to an attestation service. The attestation service can provision cloud-service and user-specific credentials into the device. Authentication of cloud services by the devices is left to the cloud service. Though the on-boarding is secure; this does not ensure a secure cloud communication.

The attestation service is similar to an IdP discussed in Section 3.3. Authentication of IdP service is achieved with TLS and PKIX. With SDO, this aspect is unclear and unknown. Authentication of the remote service is necessary to mitigate MITM attacks. An adversary, at the very least, can eavesdrop and delay messages.

3.7 Trust On First Use

Trust On First Use (TOFU) proposes binding the entity with the public key it presents on the first contact. On subsequent connections, the entity is verified to present the same public key. Digest of the public is used to ensure the entity uses the same key at every session.

Secure Shell (SSH) [YL06] uses this model with public key based authentication. Reference [WK16] suggests the use of TOFU with OpenPGP to mitigate Man-in-The-Middle (MITM) attacks. HPKP [EPS15] uses TOFU to detect CA compromise and impostor services. The HTTP client maintains the fingerprint of the server's certificate and generates user alerts if an anomaly in server certificate is detected.

The author of [Cer18], with not many details, shares the idea of applying TOFU in IoT. The intention is to allow an IoT device to generate its own private and public keys, and register the public key on first use. The registration of the public key is expected to occur in a secure environment. On registration, the public key is mapped to an IP address, or, similar identifier. The mapping enables a verifier to ensure the device uses its registered public key.

TOFU is vulnerable to MITM attacks. Use of TOFU with easily impressionable IoT devices imply undetected adversaries and devices compromised to impostor services. DNS poisoning in the network would allow an adversary to imprint their keys on the device. The legitimate service may then be talking to IoT device via the impostor service that can easily manipulate the communication and trigger undesirable decisions in the system. No methods for key expiry and revocation imply long lasting exploits in the system.

3.8 Blockchain

Blockchain is the trendy new solution for all problems, including IoT. Authors in [YWN$^+$18] propose the use of blockchain to create a *Trustchain* between various actors in the system. The trust, here, means reliability and integrity of data and device ownership. Secure device life cycle management is emphasized to be critical to achieving this goal of trust. The implementation in [YWN$^+$18] is based on Hyperledger Fabric [Hyp19a]. Supply chain and device ownership management are prominent use cases of Hyperledger Fabric. PKI and X.509 certificate provide entity authentication Hyperledger Fabric. TLS is optional for inter-node communication.

Authors of [PTM$^+$18] surveys various blockchain technologies for use in IoT. Most literature today focuses on tamper-proof, reliable data from sensors that can be sold by owners for analytics, and concerns on privacy with little focus on constraints of an IoT device. No benchmarks exist on the participation cost incurred by a constrained device. Also, any security compromises arising with participation in unauthenticated networks are not analyzed.

Smart Contract Based Public Key Infrastructure (SCPKI) [AB17] suggests blockchain based certificate management of entities. The design proposed is based on Ethereum [Eth15]. One motivation for SCPKI is to save any price paid for trust to a central authority like the Certificate Authority (CA). The other is the slow process to introduce new trust anchors (root CA certificate) in operating systems and browsers.

Hyperledger Indy [Hyp19b] and Sovrin [Sov19] aim to provide self-sovereign digital identities to people, organizations, and Internet-enabled things. Prominent features of this solution include decentralized digital identities and minimal disclosure to the verifier with Zero Knowledge Proof (ZKP). Certain volunteer organizations referred to as *Stewards* host and validate the blockchain. The Hyperledger Internship Program [Hyp19c] includes a topic to provide a reference implementation on a constrained device. Hence, no benchmarks are available. Also, it is unclear if a user interface is mandatory for a device to hold a self-sovereign identity.

3.9 TLS: Raw Public Keys

IETF defines the use of raw public keys with TLS [WTG+14] with the intent to make public key cryptography constrained device-friendly. CoAP [SHB14] mandates the use of raw public keys to secure communication between a server and a client. DTLS implementations like TinyDTLS [Ber16b] exist for use in constrained embedded devices. IETF with [SE17] drafts the use of raw public keys for client authentication for ACE-OAuth.

[WTG+14] suggests out-of-band mechanisms for trust and access management. Therefore today, there exists no trust model proposed to know if a public key identifies a legitimate authorized entity with certain access rights. Moreover, no model exists to manage key expiry and revocation.

3.10 Summary

Many alternatives to PKI exist, but none that address the challenges of IoT. Some, like the raw public keys, are well-suited building blocks for IoT and not a complete solution. While others like SPKI integrate identity with access, but leave the trust model open.

Federated identities using OAuth shift the paradigm of IAM management. Despite the many merits, it relies on PKI for server authentication. Hence it is no alternative to PKI as the open issues with use PKI on IoT remain.

PBC based schemes are expensive for use in IoT devices and require a central trusted server to create identities. Access management, key expiry and revocation with PBC is an open task for security architects and engineers. TOFU and WoT complicate trust and key management for constrained devices with no user interface.

Blockchain-based solutions, suffer some or all of the following drawbacks for IoT. The blockchain based solutions rely on PKI for Identity management, TLS for authentication and confidential communication, address limited use cases suited for smart properties like the cars and have no benchmarks for use in the constrained device.

We see that the alternative approaches considered are either not independent of PKI, or not suitable for IoT. We therefore, propose a new approach in the Chapter 4. This novel alternative to PKI borrows the concept of local domains from SPKI and the building block of a Blockchain - Merkle Tree.

Chapter 4

Vriksh: The tree of trust

Access to a resource, an action, or a privilege in the system is the motivation to prove the digital identity. Identity Schemes provide necessary primitives to establish digital identity. Under the guidance of Kerckhoffs' Principle [Wik19d], security systems have evolved to make mechanics of any security scheme public knowledge. The secret key is the one private asset. System resources are private assets and available only based on system policy. Given this, any entity may implement an Identity Scheme to gain access in the system. The knowledge of enrolled entities and their identities is a prerequisite for identity-based access. Trust adds this layer of knowledge and is essential for entity authentication.

The novel Vriksh: The Tree of Trust (VTT), provides an alternative to PKI model of trust. VTT weaves identity, trust, and access management together with two primary goals:

1. Propose an embedded device-friendly use of public key cryptography for identity and access management in private IoT deployments.
2. Limit the periphery of trust to only the relevant entities within the closed system.

The proposed trust model is called the **Vriksh: The Tree of Trust (VTT)**. In Sanskrit, Vriksh means a tree. Merkle tree is the basis of the proposal, and hence, the name. Merkle tree is a cryptographic data structure and finds applications requiring data integrity. Ralph Merkle first proposed the Merkle tree for a one time use digital signature scheme, see [Mer79].

A wide range of applications use Merkle trees. We mention some of the notable applications here. Merkle tree is one of the building block of the Bitcoin blockchain, [Nak08]. Certificate transparency, [Lau14], enables transparent audit logs for X.509 certificates issued by CA. The solution proposes append-only logs built with Merkle trees. Secure device block, [HWF15], uses a Merkle tree to protect data at rest stored across files.

4.1 Overview

Vriksh: The Tree of Trust (VTT) proposes the use of Merkle trees for access based trust management. All entities under an access role, with equal access rights, are grouped in a common trust tree. The trust tree is a Merkle tree, identified by it's Merkle root. The end node or the leaf of the trust tree is the identity of an individual entity. Any entity in the system shall be a part of exactly one trust tree. Figure 4.1 shows a sample trust tree of five entities with equal access rights.

A trust tree is similar to an access role, in access management, but not synonymous. The access role defines the access rights and policies to govern access to one or more resources. Access roles define policies to manage resources; trust trees enforce them. Trust trees may have shorter life-spans than access roles. There may be more than one trust trees attached to the same role but only one access role defined per trust tree. We elaborate access role and trust trees further in Section 4.7. Section 4.8 provides methods to update trust trees.

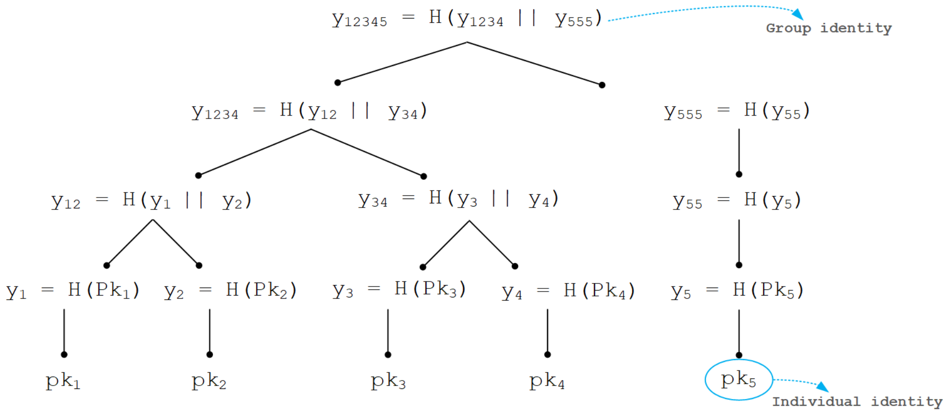


Figure 4.1: VTT: A sample trust tree identified by the Merkle root y_{12345}.

The trust tree in Figure 4.1 has five members, each identified by a public key $\boldsymbol{pk}$. $\boldsymbol{y_{12345}}$, the Merkle root, and identifies the tree. The cryptographic hash function $\boldsymbol{H}$ determines the size of $\boldsymbol{y_{12345}}$. The operation $||$ indicates concatenation of inputs. Any entity in the system must belong to a trust tree and is always identified by the individual identity and the group identity. Such an identification is analogous to a person's identification with a first and last name. Entity authentication is explained in Section 4.6.

4.2 Why use a Merkle Tree?

Any tree structure, including the Merkle Tree, is organic and can be easily be updated to include more entities in the system. Expanding the trees is possible without breaking the existing authentication in sub-tree. The system can intentionally employ this property of the Merkle to create logical partitions in the system. The logical partitions could be aimed to create small trust trees to aid constrained devices, and/or, to facilitate geographical, location separation of remotely connected components in the system.

4.3 Setup

Let $\boldsymbol{d}$ be a constrained device in a group of many devices $\boldsymbol{D}$. Each device in $\boldsymbol{D}$ can connect to the remote service $\boldsymbol{S}$. $\boldsymbol{S}$ consists of load-balanced many servers. Each server represented as $\boldsymbol{s}$. Connection to the service $\boldsymbol{S}$ implies a connection to only one of the individual servers. A server $\boldsymbol{s}$ can manage simultaneous connections with many devices in $\boldsymbol{D}$.

All participants in the system, each device $\boldsymbol{d}$ in $\boldsymbol{D}$ and each server $\boldsymbol{s}$ in $\boldsymbol{S}$, have unique key pair ***(sk, pk)*** of a cryptographic identity algorithm $\boldsymbol{C}$. [1] Here, ***sk***, is the secret key and the ***pk*** is the public key.

All entities must belong to a trust tree. A Merkle tree groups the identities of all relevant entities to form a trust a tree. The Merkle root of the tree identifies the trust tree. A cryptographic hash function $\boldsymbol{H}$ is used to hash the public keys and to create the Merkle tree. [2] Figure 4.1 depicts a sample trust tree with five entities.

4.4 Tree size and depth

The size of the tree determines it's depth. The size, here, refers to the number of end or leaf nodes in the tree. For example, the size of the tree in Figure 4.1 is 5. In Figure 4.1, the depth of the tree is 3.

If $\boldsymbol{m}$ is the number of end nodes in the tree, and then the depth of the tree, $\boldsymbol{n}$, is given by (4.1).

$$n = \lceil log_2 m \rceil \tag{4.1}$$

[1]Generation of keys and authentication are determined by the algorithm used. Any identity algorithm may be used for the proposed scheme. Therefore, choice and details of algorithm are not discussed here.

[2]The exact size of hash output, and the hash algorithm can be chosen based on the use case.

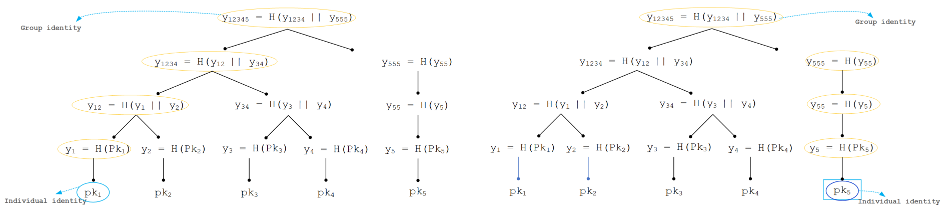
4.5 Tree membership

The membership of an entity requires ensuring that the computed Merkle root or the group identity matches the stored root for the Merkle Tree. Three possible are possible for establishing membership in a Merkle Tree based on the trade-off between computation and storage are described here.

The choice of algorithm for tree membership is local to the verifier and not visible to the prover of membership.

4.5.1 Store all, compute less

With Store all, compute less (SACL), complete Merkle tree is stored. Here, complete implies storing hash values of the all end nodes, the intermediate nodes, and the root of the tree. Storing the raw values of the ends nodes is not necessary. This method is analogous to the authentication path described in [Mer79] for signature scheme. Here, the number of transmissions needed to verify the signature establishes the efficiency of the scheme. Reference [DPP16] suggest further efficiency improvisations to the scheme.



(a) Authentication of Entity with siblings (b) Authentication of Entity with no siblings

Figure 4.2: Merkle Tree Authentication with SACL method. The number of hash computations needed to establish membership is identical for an entity with and without siblings.

Figure 4.2 depicts the number of hashes required when the complete tree, including the intermediate hashes, is pre-computed and stored. Here, for a tree of depth $\boldsymbol{n}$, the maximum number of hashes $\boldsymbol{c_{max}}$, needed for authentication is given by equation (4.2) and the maximum storage size, $\boldsymbol{s_{max}}$, needed in bytes, given by (4.3).

$$c_{\text{max}} = n + 1 \tag{4.2}$$

$$s_{\text{max}} = h_{\text{s}} * (2^{(n+1)} - 1) \tag{4.3}$$

In Figure 4.2, the depth of the tree is $\boldsymbol{n}$ is three. For membership verification in the tree, by (4.2), we compute maximum of four hashes. From (4.3), the maximum storage space needed for a hash of 256-bits ($\boldsymbol{h_s}$ = 32 bytes) is 480 bytes.

4.5.2 Compute all, store less

With CASL, only the leaf nodes and the Merkle Root. Here, only the hashes of the end nodes and the Merkle root hash is stored. Here, we trade more computation power and hence energy for storage. For this method, for a tree of depth $\boldsymbol{n}$, the maximum number of hashes and the maximum storage size are (4.4) and (4.5), respectively.

$$c_{\max} = (2^{(n+1)} - 1) \tag{4.4}$$

$$s_{\max} = h_{\mathrm{s}} * (2^n + 1) \tag{4.5}$$

For the tree in Figure 4.1, the depth of the tree, n is 3. With this method, the maximum number of hashes needed to verify is 16. The maximum storage needed for a $\boldsymbol{h_s}$ = 32 bytes, is 288 bytes.

4.5.3 Store less, compute one

For very constrained devices, further optimization is possible. Here, the verifier can store only the hashes of the end entities and the root of the Merkle tree. During verification, the identity of the entity to be authenticated is hashed. The membership algorithm is reduced to search in look up in the end entities entry of the look-up table. In other words, the maximum number of hashes, $\boldsymbol{c_{max}}$ in (4.6), needed for authentication is one for any depth of the tree. This method is referred to as the SLCO. The maximum storage size, $\boldsymbol{s_{max}}$ is given by (4.3).

$$c_{\max} = 1 \tag{4.6}$$

$$s_{\max} = h_{\mathrm{s}} * (2^n + 1) \tag{4.7}$$

This method has limited use for client devices that, before requesting a connection to a server, know which group of servers it wishes to connect to. Therefore, it can limit the search to the database identified the Merkle Tree root. In addition, since the intermediate hash values are not stored, or computed, authentication of subtrees is not possible for such clients.

4.5.4 The trade-off

The space-time trade-off may not be significant enough for small trees. However, with larger trees, the cost of storage may be significant to consider the trade-off. As depicted in table 4.1, the storage requirement doubles with every increment of the depth of the tree regardless of the algorithm. The cost of computing each hash on target can aid the choice of method used for tree membership.

SACL vs. SLCO

One might wonder why the SACL method is considered when SLCO is an efficient variant. However, SACL verifies data-at-rest tree. Hence, may be important for systems that do not have secure storage in hardware. Also, the SLCO can authenticate in subtrees. SLCO, therefore, has limited use and fine-tuned for very constrained devices. SLCO, if not used under the condition of prior knowledge of group identity of the prover, can lead to access privilege escalation, and hence, is not secure.

Depth/Cost	m_{max}	SLCO		SACL		CASL	
		c_{max}	s_{max}	c_{max}	s_{max}	c_{max}	s_{max}
3	8	1	288	4	480	15	288
4	16	1	544	5	992	31	544
5	32	1	1056	6	2016	63	1056
6	64	1	2080	7	4064	127	2080

Table 4.1: Merkle Tree: Resource requirements for membership verification. The depth of the tree and the membership verification method determine the resource requirements.

4.6 Identity and Authentication

VTT requires every participant in the system to a part of a trust tree. Authentication involves verification of both the group identity and the individual identity. Preferably, in that order. Verification of the two identities is detailed below.

1. Group identity. The group identity depicted in Figure 4.1 is verified by membership proof of the individual identity in the Merkle tree. Section 4.5 describes methods for membership proof.
2. Individual identity. The identity verification procedure is determined by the Identity Scheme. Any Identity Scheme of choice may be used with VTT. VTT recommends the use of a common scheme for all the members in the tree. The group identity should be verified before the individual identity. As it is typically cheaper to verify the group identity as compared to the individual identity.

To elaborate with an example, consider the trust tree in Figure 4.1. Let us assume represents the trust tree for remote service S with five load balanced servers s_1, s_2, s_3, s_4, s_5. A client, to authenticate the service, must be provisioned with the server's trust tree in 4.1. Only one of the servers will communicate with the client at any given time. Let us say the server s_5 connects to the client. The client must authenticate the service by verifying:

1. The public key, pk_5, is a member of the Merkle tree y_{12345}.
2. The server has the secret key corresponding to the public key pk_5. The cryptographic algorithm C determines the exact verification algorithm.

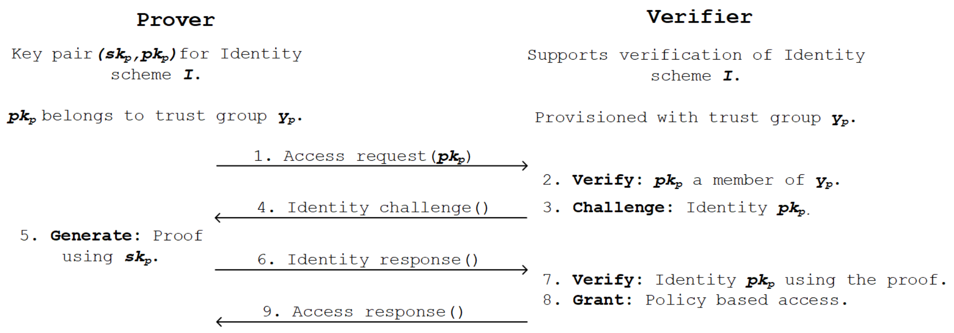

Prover
Verifier
Key pair (sk_p, pk_p) for Identity scheme I.
Supports verification of Identity scheme I.
pk_p belongs to trust group y_p.
Provisioned with trust group y_p.
1. Access request(pk_p)
2. Verify: pk_p a member of y_p.
4. Identity challenge()
3. Challenge: Identity pk_p.
5. Generate: Proof using sk_p.
6. Identity response()
7. Verify: Identity pk_p using the proof.
8. Grant: Policy based access.
9. Access response()

Figure 4.3: A normative protocol to demonstrate entity authentication of VTT manged entities. For authentication, both the group identity y_p and the individual identity pk_p are verified.

Figure 4.3 depicts VTT based entity authentication. The prover identified by pk_p, in trust tree y_p is authenticated on the access request. The membership verification in step 3 is a function of the depth of the Merkle tree, as described in Section 4.5. The protocol depicts a successful run of the protocol. The protocol could fail at any stage, and access request will fail. Note that the protocol messages exchanged, their format and count depend on the choice of application layer protocol, transport protocol, and the Identity Scheme. The normative protocol depicted in Figure 4.3 is tailored to emphasize verification of the group identity before the individual identity.

Figure 4.4 depicts a sample run with signature-based Identity Scheme. With the signature scheme, there is no challenge needed from the verifier. Therefore, the verifier does not have to be online. Regardless of the Identity Scheme used, and if the verification happens online or offline, the group identity verification remains the same. The group identity is always verified by membership in the Merkle tree.

Most use cases require mutual authentication, that is the prover must be sure that it is talking to the right verifier. Mutual authentication is a measure to prevent man in the middle attacks. Mutual authentication is not depicted in Figure 4.3 and Figure 4.4 for simplicity.

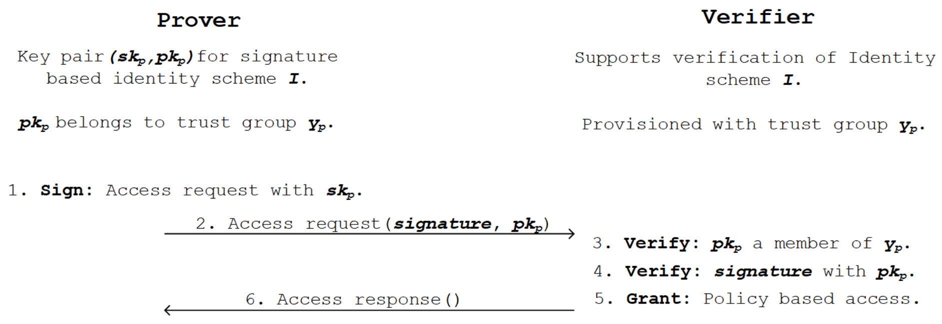

Prover
Key pair (sk_p,pk_p) for signature based identity scheme I.
pk_p belongs to trust group y_p.
1. Sign: Access request with sk_p.
2. Access request(signature, pk_p)
6. Access response()
Verifier
Supports verification of Identity scheme I.
Provisioned with trust group y_p.
3. Verify: pk_p a member of y_p.
4. Verify: signature with pk_p.
5. Grant: Policy based access.

Figure 4.4: VTT based normative entity authentication using a signature scheme for identity. With the signature scheme, the verifier may be offline. The verifier must verify both the group identity y_p and the individual identity pk_p.

Figure 4.5 depicts a normative mutual authentication between a client and a server. In any system, servers and clients have different access rights and therefore, belong to different trust trees. Notice that in the figure the client and the server are members of different trust trees identified by y_c and y_s respectively. The client verifies pk_s to be a part of the trust tree y_s. Similarly, the server verifies the client to belong to y_c and also the signature using the pk_c.

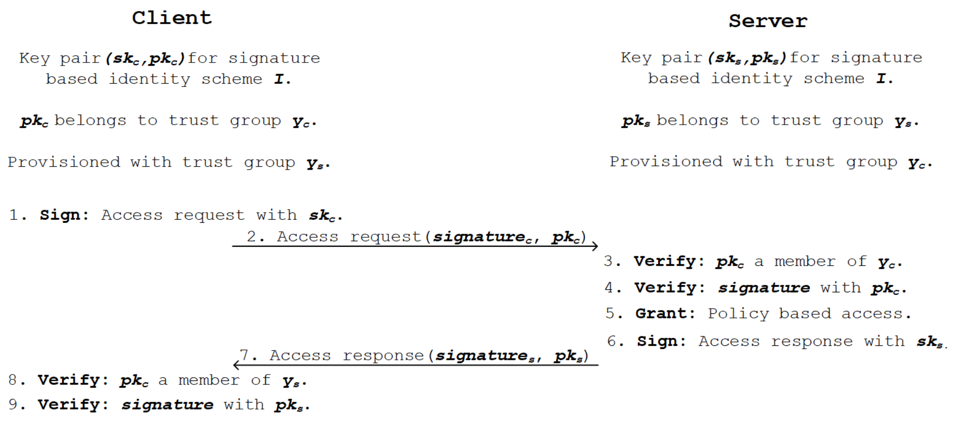

Client
Key pair (sk_c,pk_c) for signature based identity scheme I.
pk_c belongs to trust group y_c.
Provisioned with trust group y_s.
1. Sign: Access request with sk_c.
2. Access request(signature_c, pk_c)
7. Access response(signature_s, pk_s)
8. Verify: pk_c a member of y_s.
9. Verify: signature with pk_s.
Server
Key pair (sk_s,pk_s) for signature based identity scheme I.
pk_s belongs to trust group y_s.
Provisioned with trust group y_c.
3. Verify: pk_c a member of y_c.
4. Verify: signature with pk_c.
5. Grant: Policy based access.
6. Sign: Access response with sk_s.

Figure 4.5: VTT based normative mutual authentication protocol run between a client and a server. The server and the client belong to different trust trees.

In Figure 4.5, for simplicity, the client and the server use the same Identity Scheme I. The Identity Scheme for each tree and access role used may be different. Regardless of the Identity Scheme for the individual identity, verification of identity remains the same. Group identity is always verified by the membership of an individual's identity in the trust tree corresponding to the access role. Also, the choice of hash function

H to construct each trust tree can vary with each tree. However, for simplicity and consistency, in this thesis, this is not considered or discussed.

4.7 The garden of trust

Identity and Access Management (IAM) involves defining and managing the access policies for each resource in the system. Access policies are attached to each entity in the system to determine it's access in the system. Every access request must ascertain the identity of an entity requesting access to enforce the corresponding access policy. Access roles enable governing entities within a role with common policies.

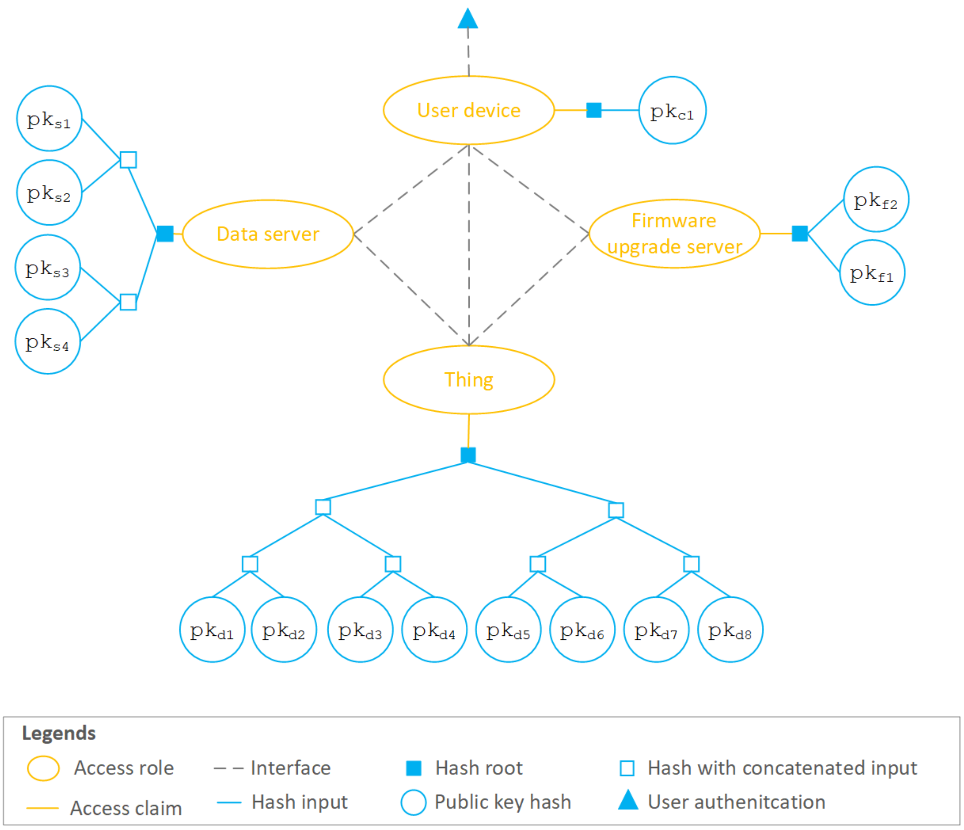


Figure 4.6: VTT: Sample Identity and Access Management. Each access role has one trust tree attached to identify the entities with the access rights defined by the access role.

VTT proposes an access role-based approach for IAM. It is typical for IAM to define many roles for granular access management. Notice that in Figure 4.6, two server roles are defined. Only the firmware upgrade servers can push new firmware to the devices grouped under the access role 'Things'. Data servers do not enjoy this right. Since the data servers and firmware upgrade servers have different access rights in the system, these are a part of separate trust trees.

With VTT, all entities must be a member of a trust tree. An entity is added to a trust tree corresponding to it's access rights in the system. Each trust tree is attached to an access role. Each member of the trust tree has equal access rights in the system. There can be more than one trust tree attached to an access role; however, only one access role per trust tree.

The distinction between trust tree and access roles enables the system to separate the entities with certain access rights from the access rights. Trust trees determine and manage entities that exercise the access rights by the access role. While the access roles define and manage the rights and policies to access the system's resources.

Trust tree based management of entities is discussed in Section 4.8. In Figure 4.6, for simplicity, each access role has only one trust tree attached to it. There could be more. In the figure, the interface defines the interaction methods, if any, between access roles. The entities in the trees execute the interaction methods.

The sample access management in Figure 4.6 depicts trust trees of different sizes, age, and each with a role. Hence, IAM with the model implies gardening of these trust trees through their life cycle. As detailed in Section 4.8, the trust trees are born, live, and die. The death of a tree may be due to age (expiry) or sickness (revocation).

4.7.1 The trust periphery

With trust comes vulnerability. The VTT aims to limit the trust periphery, and hence the vulnerability. In VTT, trust is not extended to all entities within the system. The interactions and interfaces between access roles determine the trust periphery. Any entity trusts only trust trees attached to other access roles that it interacts with. To clarify with an example, consider the access roles and interfaces in Figure 4.6. Notice that in the figure, data server and the firmware upgrade server do not have any interfaces defined. This implies no access claims are made by any entities in these access roles to resources in each other's realm.

4.8 The living tree

All sections until now assume that the trust trees exist. We now discuss the formation and management of trust trees.

VTT assumes that at the initial setup of the system, all entities in the system, their identities, access rights, and hence, access roles are known. Therefore, at setup, initial trust trees can be formed. For example, based on the load and availability requirements of the system, the number of servers can be determined. Similarly, the types of sensors in the system, and their count can also be determined based on requirements.

In VTT, each trust tree in the system is of different size, function, and life span. However, all trust trees are assumed to be updated since the initial setup. The scheme makes two different recommendations for updating the tree. The recommendations are based on the resourcefulness of the entities that form the trust trees. For example, a gateway device, implementing a proxy server cannot belong to the trust tree of load balanced servers. As such a gateway, though plays a server, it neither equal in rights and resourcefulness.

4.8.1 Autonomous trees

For trees with resourceful and always available members of the system, like the servers, the tree could self-managed, and hence autonomous. VTT proposes the use of threshold ring signature to update a trust tree. The ring signatures must be generated by the entities of the trust tree. Therefore, a tree can be updated and maintained by the entities within the tree. Much research exists on ring signature and threshold signature. Authors in [OTYO18] have proposed the use of a ring signature for the update problem. Similarly, reference [BSS02] describes the application of a threshold signature for ad hoc groups. Multi-signature transactions in Bitcoin, [Con19a], use the threshold of k-out-n signatures to validate them.

VTT recommends the use of a common threshold and a minimum threshold of 60%. The exact threshold needed is system dependent and defined as applicable. The minimum is to ensure majority approved updates to the tree. To elaborate with an example, to update a server trust tree with 8 members, 6 out of the 8 servers must approve the update. The assumption, here, is that not all the servers in the system are compromised at the same time. The uncorrupted majority replaces any malicious minority servers. If the system is compromised such that no majority can be reached, then the system must be set up again.

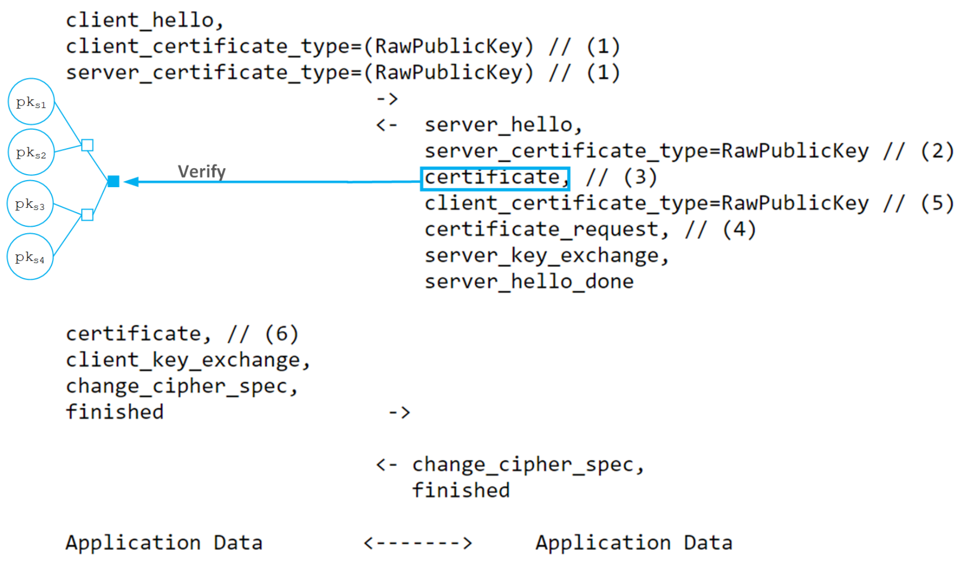
This member majority approved update of the tree is used for system expansion, replacement of faulty components, etc. as well. Any new servers added to the system must get the approval of the majority of the existing servers. Digital schemes may be the most practical way to indicate digital approval.

4.8.2 Non-autonomous trees

The threshold based scheme is not practical for sleepy constrained devices. Getting all devices to collaborate within a time window may be a challenge. User-initiated updates either via user web portals, and/or other dedicated management devices, may be a suitable approach to update trust trees of constrained devices. In this case, the tree is updated by other entities, and hence, such trees are considered to be non-autonomous. Any updates to the tree, however, should be verifiable. Here too, threshold signatures may be used.

4.9 Use of VTT with TLS

We have looked at the novel scheme in detail to understand trust and access management from a theoretical perspective. To use VTT in a practical application, we now discuss it's integration with TLS. As discussed in Section 2.3, TLS is in wide use and default mechanism for server authentication. Therefore, the integration of any new scheme with TLS provides a good basis to evaluate the scheme.



```
client_hello,
client_certificate_type=(RawPublicKey) // (1)
server_certificate_type=(RawPublicKey) // (1)
                                      ->
                                      <-   server_hello,
                                           server_certificate_type=RawPublicKey // (2)
                                           certificate, // (3)
                                           client_certificate_type=RawPublicKey // (5)
                                           certificate_request, // (4)
                                           server_key_exchange,
                                           server_hello_done

certificate, // (6)
client_key_exchange,
change_cipher_spec,
finished                               ->

                                      <- change_cipher_spec,
                                         finished

Application Data         <------->       Application Data
```

Figure 4.7: TLS raw public keys and VTT based entity authentication. The server certificate, a raw public key, must belong to the server trust tree for successful authentication.[3]

In Section 3.9, we considered the use of raw public keys for TLS defined in RFC7250 [WTG+14] and evaluated it to be a building block; a building block to make public key cryptography efficient on a constrained device, but lacking a trust model. VTT fills in this gap and provides a workable scheme. For the use of VTT with TLS, the thesis proposes the use of public key expressed in raw format to be the individual identity of all entities. The group identity would then be the Merkle root of all the raw public keys that identify individual members of the tree.

We use the TLS handshake depicted in Figure 4.7 to explain the use of VTT for entity authentication. For the figure, the handshake exchange is copied from [WTG+14] and VTT specific depictions. In the figure, the client first verifies the raw public key of the server to be a member of the server trust tree. In other words, the client verifies the servers group identity. On successful verification, proceeds with the handshake according to the methods already defined for the cipher with TLS. Therefore, to use VTT with TLS, no new methods or alterations are needed.

The server can similarly authenticate the client using the VTT and raw public keys. Alternatively, JWT based or other eco-system specific schemes may be used. It is possible to use VTT with JWT and any other Identity Scheme. However, these are not elaborated to limit the scope of the thesis.

4.9.1 Client Initiated Communication

The design of TLS secure communication between client and servers in a client-server model; a client initiates TLS session with a server to access one or more specific resources in the system. For example, the client may check the availability of new firmware with the firmware upgrade servers. Therefore, it is reasonable to assume that the client already knows the trust tree to authenticate the server.

4.9.2 Server Name Indication

Server Name Indication (SNI), a TLS extension defined in RFC6066 [Eas11], enables the client to identify the service of interest to the server. Current practice is to use the hostname of the service. With VTT, the group identity of the service, the Merkle root, is indicated SNI extension. The client then authenticates the server raw public key to be a part of the indicated trust tree only. Any other entity not belonging to the indicated trust, even if known to the client, will fail the group identity verification. In this event, the client would abort the connection with the reason 'Unknown certificate'.

Chapter 5

Prototype

This chapter documents the prototype phase of the thesis. The prototype demonstrates the use of VTT with TLS raw public keys, as described in Section 4.9. The prototype is built to provide:

1. a limited proof of concept for VTT, and
2. data to compare PKI with VTT.

For VTT, the prototype limits the scope to allow TLS connections from members of hard-coded, pre-provisioned trust tree and reject any connections from non-members. For PKI, no development is needed. We only capture of data captured needed for the evaluation phase. In addition, the implementation is tested for correctness; however, it is neither robust nor production-ready.

The high-level requirements that guide the design and implementation choices are listed below.

1. **Portability.** Portability of prototype to embedded platforms. The prototype is hence implemented in C. [1]
2. **Reusability.** The prototype is built with existing components already in use for embedded platforms.
3. **Test and validation.** The prototype is correct even if not robust.

To build the prototype, the development model used is the Incremental and Iterative approach [con15] as it seems essential and instinctive to:

1. Develop in small incremental steps.
2. Iterate the plans and requirements.

The implementation increments used for the prototype are described in Section 5.4. Any assumptions made are stated in Section 5.1, and all components and tools are described in Section 5.4. Justification for choices is provided where necessary.

[1] The TLS server role is not ported to embedded platform, hence could be implemented in any high level language. However, C is used for TLS server as well as most TLS implementations are implemented in C. This also enables reusing implementation common to server and client roles.

5.1 Assumptions

The assumptions made for implementation are:

1. DTLS and TLS. Any evaluation of TLS applies for DTLS and vice versa. [2] So, it is adequate to implement and evaluate one of them.
2. TLS Role. Embedded devices play the TLS client role and are resource constrained. Cloud services play the TLS server role and are very resourceful.
3. Cryptographic Hardware acceleration. Use of hardware acceleration of cryptographic primitives is not considered for the evaluation. [3]
4. Application layer protocols for IoT, example MQTT, CoAP, are not relevant for the current scope. They have no impact on TLS and VTT based authentication.

5.2 Components and Tools

5.2.1 TLS Library

Multiple open source implementations of TLS available. Wikipedia [Wik19b] provides a comprehensive list of most TLS implementations along with feature and cipher based comparison. Portability is also compared. Of the many implementations of TLS, we limit ourselves to three - OpenSSL, mbed TLS, TinyDTLS[4].

TinyDTLS is written to have a minimal footprint on embedded devices, with just the features needed to support two ciphers mandated by CoAP in RFC7252 [SHB14]. TinyDTLS is now available as a part of the Eclipse IoT project at [Ber16a]. TinyDTLS is one of the first implementation of Raw Public Key for TLS.

OpenSSL [The] is the most widely used TLS library for Desktop applications. OpenSSL provides a wide variety of command line tools and applications for key generation, certificate signing, hashing, etc. OpenSSL includes pre-compiled server and client applications that can be invoked using the command line. ASN.1 and X.509 certificate parsing and decoding tools are also included in the library.

mbed TLS library [ARM] is a popular choice for TLS implementation on embedded platforms. The library supports a wide variety of TLS features and ciphers and includes many test applications. It also includes clear documentation for configuring and porting the library.

[2]TLS assumes reliable transports, for example TCP. DTLS is an adaptation of TLS for use over UDP-like unreliable, connection-less transports. DTLS addresses issues typical to an unreliable transport -packet loss, out-of-order packet reception. DTLS accounts for fragmentation of packets typical to connection-less transports. DTLS does not introduce any new cipher suites.

[3]Reception and parsing of certificates are not hardware accelerated. Verification of certificates chains use hash and signature verification algorithms. Hence, hardware acceleration may be a beneficial, but left out of scope.

[4]Wikipedia [Wik19b] does not list TinyDTLS as a TLS implementation

For the prototype implementation, we choose the mbed TLS library. We use OpenSSL as a Desktop tool to generate credentials and test. The factors that influenced our choice are listed below.

1. Support for Raw Public Key.
   TinyDTLS implements PSK and raw public key but does not support X.509 certificates. Some clones add limited support for this feature.
   mbed TLS implements PSK and X.509 certificates and has an open pull request [Kah15] to add support for the raw public key.
   OpenSSL supports PSK and X.509 certificates. There is no support for the raw public key yet.
2. Portability to embedded platforms. OpenSSL is not well suited for Cortex-M devices. TinyDTLS is optimized for resource-constrained devices. mbed TLS is configurable for limited features on embedded platforms.
3. Test and support. mbed TLS is used in commercial products and has a very wide variety of unit tests and test applications with a large, active community. TinyDTLS enjoys smaller adoption and community base as compared to mBedTLS.
4. Future feature extension. To continue any work on the topic at a future date, the flexibility to extend the prototype beyond the use of limited primitives is necessary. TinyDTLS supports very limited crypto toolbox - SHA-256, ECDSA, ECC curve secp256r1 and AES in CCM mode. mbed TLS provides a much wider range.

5.2.2 Merkle Tree

We evaluate two C libraries for the Merkle tree.

1. IAIK Merkle tree library. The IAIK library [HW15] uses SHA256 as the hash function to build the Merkle tree.
2. Vector Gua's Merkle tree library. This [Guo12] library constructs the Merkle tree using MD5[5].

For the prototype, we use the IAIK library. Factors that influence our choice are listed below.

1. Code and repository structure. The IAIK library has better structure, with usage example in the form of tests. The library identifies and documents the configuration parameters like the tree size, the size of the hash function size, etc.
2. Test and use. The IAIK is tested and in use in another project; this gives better confidence in the implementation. The IAIK library is used in the secure

[5]The prototype replaces the default hash implementation in the library with the mbed TLS SHA-256 function. Therefore, the hash function used in the library is not used as an elimination criterion.

device block library [HWF15] to ensure confidentiality and integrity protection of data-at-rest.

5.2.3 Wireshark

Wireshark [Wir06] is a popular tool for network packet capture and analysis. It supports many protocols, including TLS. During the development of the prototype, Wireshark is used for debugging, verifying, and analyzing the TLS handshake messages exchanged between a server and a client.

5.2.4 Python

Python [Tea05] is a high level, interpreted programming language. Python scripts have been used here for two purposes.

1. Automation. Setup required creating CA, Server, and Client credentials and signing them where applicable. Automated scripts ensure an efficient and consistent process. The scripts also document the process used. The scripts execute OpenSSL shell commands to generate ECC keys, create certificate signing request Sing certificates, create raw public key and generation of the Merkle tree of these raw public key digest. OpenSSL is invoked in the scripts for most cryptographic operations.
2. Verification. For correctness, verification of the Merkle tree created by the C library is necessary. Python scripts enable quick independent implementation of the Merkle tree for the purpose.

All scripts used are available included at [Shi19d].

5.3 Actors

For the prototype, we create limit ourselves to a minimal set of actors. These are enumerated below.

1. Certificate Authority (CA).
2. Servers.
3. Clients.
4. Adversary.

Servers and clients belong to separate trust trees and play the role of TLS server and client, respectively. By NIST definitions discussed in Section 3, the servers represent the `Aggregator` primitive, and the clients represent the `Sensor` primitive. The adversary is essential to ensure the prototype can detect non-members, and take necessary action. CA are included for the comparison of VTT with PKI. PKI cannot function without a CA.

5.4 Development Increments

The development increments used for prototype implementation are described here. The incremental build of the prototype provides better clarity on scope and outcome at each step. The order of the increments does not necessarily define or determine the chronology of execution. For ease of development and debug, the development environment used is Linux. The developed code, as it is in C, is portable to an embedded platform. In Chapter 6, we use mbed TLS port to an embedded platform for performance evaluation. However, in the prototype phase, we use only the Linux environment on a PC.

5.4.1 Credential Set Up

For the prototype, we need keys for each actor in the system. An important aspect during the evaluation is the data exchanged and energy usage for PKI and VTT. For this, we need to define the evaluation set up for PKI and VTT. The scope of this increment is to generate ECC keys and certificates to be used for prototype and evaluation.

The process of key generation and certificate signing is automated using Python and OpenSSL. Two certificate chains are used for evaluation of PKI. A **Long Certificate Chain** as depicted in Section 5.1, and, a **Short Certificate Chain** as depicted in Section 5.2. The choice of certificate chains is based on observation of industry practices. In particular, the choice is influenced by two major cloud service providers for IoT, Google, and Amazon. Google uses a long certificate chain while Amazon uses a shorter one.

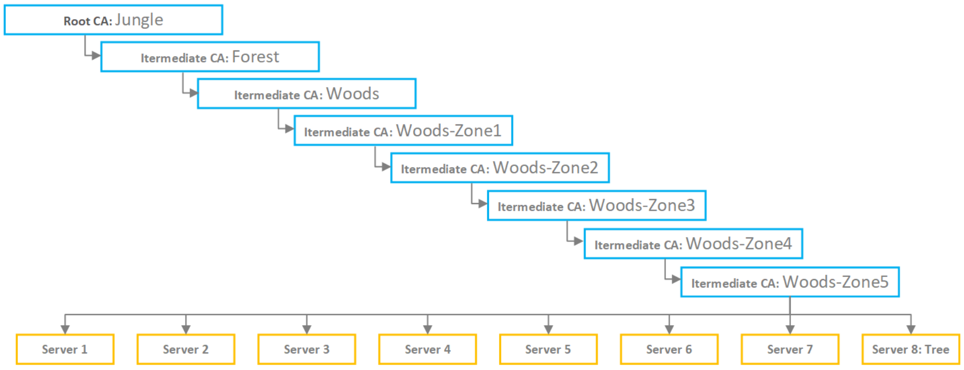

Root CA: Jungle
Itermediate CA: Forest
Itermediate CA: Woods
Itermediate CA: Woods-Zone1
Itermediate CA: Woods-Zone2
Itermediate CA: Woods-Zone3
Itermediate CA: Woods-Zone4
Itermediate CA: Woods-Zone5
Server 1
Server 2
Server 3
Server 4
Server 5
Server 6
Server 7
Server 8: Tree

Figure 5.1: CA setup for a long certificate chain used to evaluate PKI.

In Figure 5.1 and Figure 5.2, CA's are depicted in blue. The end entities, the servers are depicted in yellow. The boxes indicate X.509 certificates. All certificates are signed using **ECDSA_SCEP256r1_SHA256**. The justification of choice is provided in Section 6.1.1 and Section 6.1.2. The subject names used for each entity are test names chosen for normative purposes.

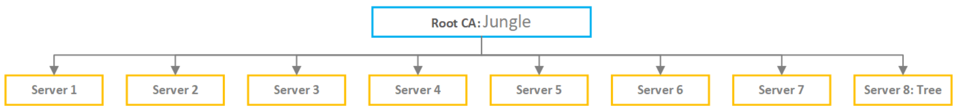


Figure 5.2: CA setup for a short certificate chain used to evaluate PKI.

We set up 8 servers based on input from deployment choices made by colleagues in the industry. A system with 8 servers can handle 100,000 incoming requests per second. However, this aspect is not independently researched. Credentials for 128 clients are generated; however, only one of them is used. Client authentication using trust tree is not evaluated as the servers are resourceful. An adversary is needed to ensure no malicious party is able to participate in the system.

At the end of this increment, the following setup is available at [Shi19e].

1. ECC private and public key pair for all entities.
2. Self-signed certificate for the root CA and the adversary.
3. Two certificate chains for server and clients.

    a) Short certificate chain as depicted in Figure 5.1 and

    b) Long certificate chain as depicted in figure Figure 5.2.

4. Public keys in raw public key format for servers, clients, and the adversary.
5. Merkle trees generated in Python to cross-check the C implementation.

The Merkle tree setup is to create a trust tree for the eight test servers, as depicted in Figure 5.3. The raw public keys itself are not a part of the Merkle tree, and are not stored in the prototype, only their digests are. Hence these are depicted in yellow in the figure.

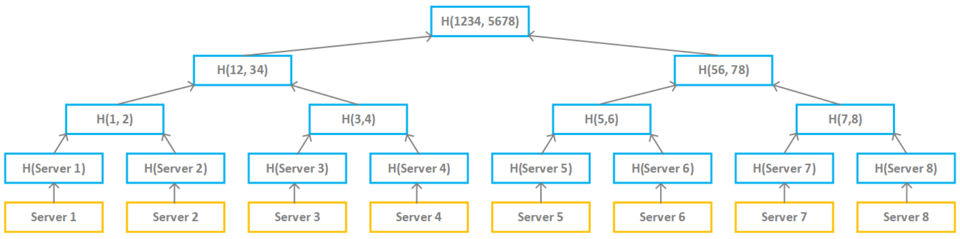


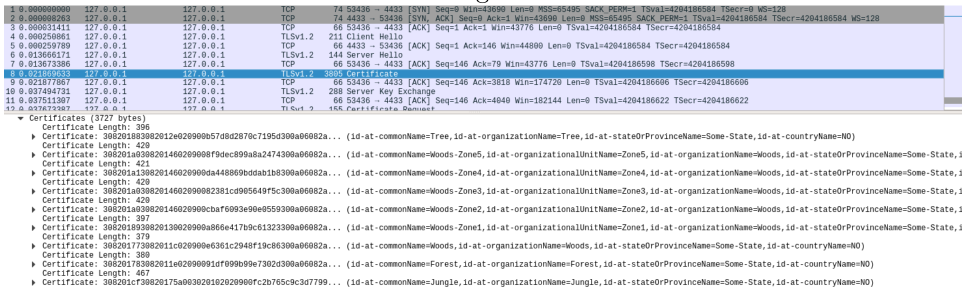
Figure 5.3: The trust tree for server constructed using the SHA-256 digest of raw public keys of eight test servers for prototype and evaluation.

5.4.2 X.509 certificates with PKI authentication

In this increment, the client and the server applications included in the mbed TLS library are used with the prototype configurations at [SmTc19]. The configuration is to test and develop the prototype on Linux. The ssl_server2.c and ssl_client2.c are the application run for testing.

This increment servers two purposes:

1. Ascertains that the TLS engine, and the client and the server applications are in working state.
2. Provides the data for short and long certificate chains to evaluate PKI.



```
 1 0.000000000   127.0.0.1      127.0.0.1      TCP        74 53436 → 4433 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=4204186584 TSecr=0 WS=128
 2 0.000008263   127.0.0.1      127.0.0.1      TCP        74 4433 → 53436 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=4204186584 TSecr=4204186584 WS=128
 3 0.000031411   127.0.0.1      127.0.0.1      TCP        66 53436 → 4433 [ACK] Seq=1 Ack=1 Win=43776 Len=0 TSval=4204186584 TSecr=4204186584
 4 0.000250861   127.0.0.1      127.0.0.1      TLSv1.2   211 Client Hello
 5 0.000259789   127.0.0.1      127.0.0.1      TCP        66 4433 → 53436 [ACK] Seq=1 Ack=146 Win=44800 Len=0 TSval=4204186584 TSecr=4204186584
 6 0.013666171   127.0.0.1      127.0.0.1      TLSv1.2   144 Server Hello
 7 0.013673386   127.0.0.1      127.0.0.1      TCP        66 53436 → 4433 [ACK] Seq=146 Ack=79 Win=43776 Len=0 TSval=4204186598 TSecr=4204186598
 8 0.021869633   127.0.0.1      127.0.0.1      TLSv1.2  3805 Certificate
 9 0.021877867   127.0.0.1      127.0.0.1      TCP        66 53436 → 4433 [ACK] Seq=146 Ack=3818 Win=174720 Len=0 TSval=4204186606 TSecr=4204186606
10 0.037494731   127.0.0.1      127.0.0.1      TLSv1.2   288 Server Key Exchange
11 0.037511307   127.0.0.1      127.0.0.1      TCP        66 53436 → 4433 [ACK] Seq=146 Ack=4040 Win=182144 Len=0 TSval=4204186622 TSecr=4204186622
12 0.037673387   127.0.0.1      127.0.0.1      TLSv1.2   155 Certificate Request
  ▾ Certificates (3727 bytes)
      Certificate Length: 396
    ▸ Certificate: 308201883082012e020900b57d8d2870c7195d300a06082a... (id-at-commonName=Tree,id-at-organizationName=Tree,id-at-stateOrProvinceName=Some-State,id-at-countryName=NO)
      Certificate Length: 420
    ▸ Certificate: 308201a030820146020900 8f9dec899a8a2474300a06082a... (id-at-commonName=Woods-Zone5,id-at-organizationalUnitName=Zone5,id-at-organizationName=Woods,id-at-stateOrProvinceName=Some-State,i
      Certificate Length: 421
    ▸ Certificate: 308201a130820146020900da448869bddab1b8300a06082a... (id-at-commonName=Woods-Zone4,id-at-organizationalUnitName=Zone4,id-at-organizationName=Woods,id-at-stateOrProvinceName=Some-State,i
      Certificate Length: 420
    ▸ Certificate: 308201a03082014602090082381cd905649f5c300a06082a... (id-at-commonName=Woods-Zone3,id-at-organizationalUnitName=Zone3,id-at-organizationName=Woods,id-at-stateOrProvinceName=Some-State,i
      Certificate Length: 420
    ▸ Certificate: 308201a030820146020900cbaf6093e90e0559300a06082a... (id-at-commonName=Woods-Zone2,id-at-organizationalUnitName=Zone2,id-at-organizationName=Woods,id-at-stateOrProvinceName=Some-State,i
      Certificate Length: 397
    ▸ Certificate: 30820189308201300209 00a866e417b9c61323300a06082a... (id-at-commonName=Woods-Zone1,id-at-organizationalUnitName=Zone1,id-at-organizationName=Woods,id-at-stateOrProvinceName=Some-State,i
      Certificate Length: 379
    ▸ Certificate: 308201773082011c020900e6361c2948f19c86300a06082a... (id-at-commonName=Woods,id-at-organizationName=Woods,id-at-stateOrProvinceName=Some-State,id-at-countryName=NO)
      Certificate Length: 380
    ▸ Certificate: 308201783082011e02090091df099b99e7302d300a06082a... (id-at-commonName=Forest,id-at-organizationName=Forest,id-at-stateOrProvinceName=Some-State,id-at-countryName=NO)
      Certificate Length: 467
    ▸ Certificate: 308201cf30820175a003020102020900fc2b765c9c3d7799... (id-at-commonName=Jungle,id-at-organizationName=Jungle,id-at-stateOrProvinceName=Some-State,id-at-countryName=NO)
```

Figure 5.4: Successful TLS Handshake with test CA Setup and the long certificate chain depicted in Figure 5.1.

Figure 5.4 depicts a successful handshake with when the server uses the long certificate chain created in Section 5.4.1 and mbed TLS is configured according to prototype requirements. The mbed TLS configuration includes only the cryptographic primitives needed for verification of the certificate and using the **TLS_ESDA_ECDHE_AES**. The necessary platform defines for time, and file system access for Linux are defined.

5.4.3 Raw public keys with no authentication

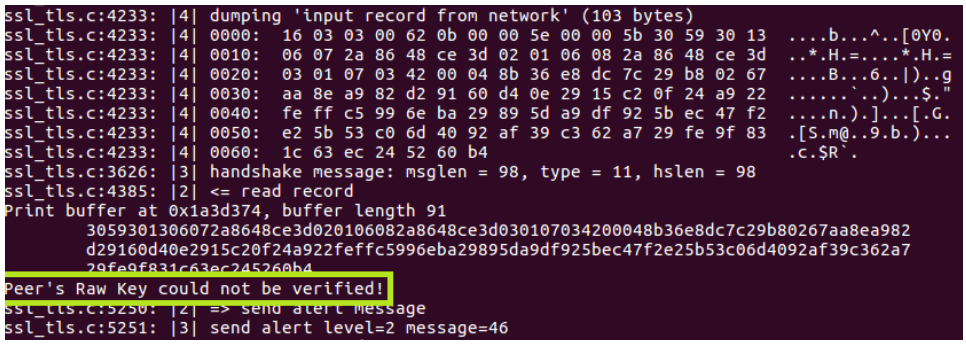
The raw public key pull request for mbed TLS [Kah15], is more than three years old. mbed TLS development has witnessed many new features, bug fixes, and restructure. The main objective of this increment is to rebase the pull request such that a common master is used for raw public keys and X.509 certificates.

This increment resulted in a new pull request [Shi19c] towards the existing pull request implementing the raw public key [Kah15].

5.4.4 Raw public keys with digest-based authentication

The original raw public key implementation in the pull request for mbed TLS, [Kah15] does not provide any verification callback to the application. This implies that all entities that offer a raw public key are accepted. Therefore, to authenticate entities using VTT, additional hooks are necessary. In this increment [Shi19f] we add an application registered verification callback to verify the public key of the peer during the handshake. The application, via the return value of the callback, may indicate the mbed TLS library to continue or abort the ongoing TLS connection.

To verify the feature, we use a pre-provisioned SHA-256 digest in the client application. The client on every TLS connection computes the SHA-256 digest of the server's raw public key in the verification callback. The computed digest is compared with the expected pre-provisioned value. If the values match, then, the connection continues, else, the connection is aborted.



```
ssl_tls.c:4233: |4| dumping 'input record from network' (103 bytes)
ssl_tls.c:4233: |4| 0000:  16 03 03 00 62 0b 00 00 5e 00 00 5b 30 59 30 13  ....b...^..[0Y0.
ssl_tls.c:4233: |4| 0010:  06 07 2a 86 48 ce 3d 02 01 06 08 2a 86 48 ce 3d  ..*.H.=....*.H.=
ssl_tls.c:4233: |4| 0020:  03 01 07 03 42 00 04 8b 36 e8 dc 7c 29 b8 02 67  ....B...6..|)..g
ssl_tls.c:4233: |4| 0030:  aa 8e a9 82 d2 91 60 d4 0e 29 15 c2 0f 24 a9 22  ......`..)...$."
ssl_tls.c:4233: |4| 0040:  fe ff c5 99 6e ba 29 89 5d a9 df 92 5b ec 47 f2  ....n.).]...[.G.
ssl_tls.c:4233: |4| 0050:  e2 5b 53 c0 6d 40 92 af 39 c3 62 a7 29 fe 9f 83  .[S.m@..9.b.)...
ssl_tls.c:4233: |4| 0060:  1c 63 ec 24 52 60 b4                             .c.$R`.
ssl_tls.c:3626: |3| handshake message: msglen = 98, type = 11, hslen = 98
ssl_tls.c:4385: |2| <= read record
Print buffer at 0x1a3d374, buffer length 91
        3059301306072a8648ce3d020106082a8648ce3d030107034200048b36e8dc7c29b80267aa8ea982
        d29160d40e2915c20f24a922feffc5996eba29895da9df925bec47f2e25b53c06d4092af39c362a7
        29fe9f831c63ec245260b4
Peer's Raw Key could not be verified!
ssl_tls.c:5250: |2| => send alert message
ssl_tls.c:5251: |3| send alert level=2 message=46
```

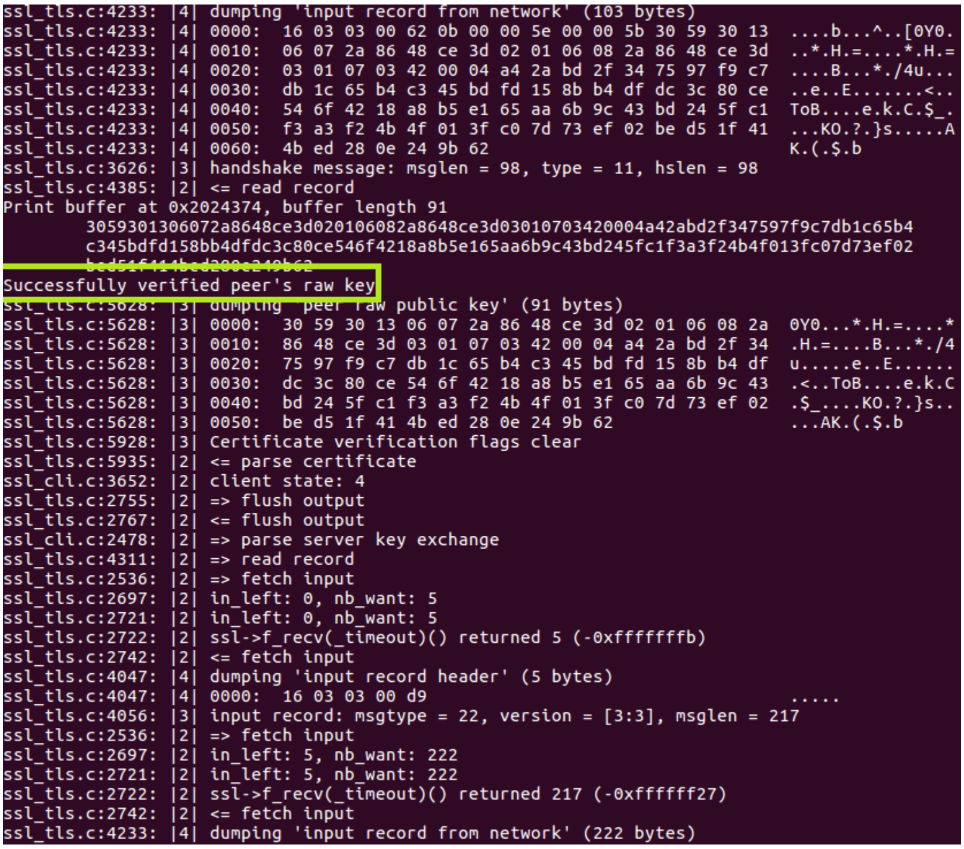
Figure 5.5: The client aborts the TLS connection with the server on a mismatch of computed SHA-256 digest of the server's raw public key with the expected. The expected value of the is pre-provisioned digest in the client.

Console output for acceptance of a trusted key and rejection of an adversary or non trusted key based on SHA-256 digest is depicted in Figure 5.6 and Figure 5.5, respectively. The client aborts TLS handshake if the digest does not match the expected. The aborted handshake sequence appears identical to Figure 5.7 to an onlooker.

5.4.5 Raw public keys with VTT based authentication

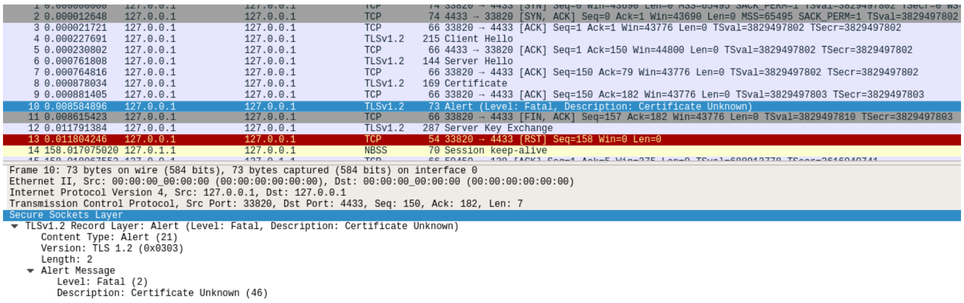
This increment [Shi19b] completes the proof of concept for the use Vriksh: The Tree of Trust (VTT) with TLS for server authentication by the client, as depicted in Section 4.7. The trust tree used to authenticate the raw public key of the server is as depicted in Figure 5.3. Any non-member raw public is rejected, as depicted in Figure 5.7.

The membership verification algorithm used here is SACL. This method described in Section 4.5.1. The proposal of use of Server Name Indication (SNI) to identify the trust tree of interest is not implemented in the prototype due to lack of time.



```
ssl_tls.c:4233: |4| dumping 'input record from network' (103 bytes)
ssl_tls.c:4233: |4| 0000:  16 03 03 00 62 0b 00 00 5e 00 00 5b 30 59 30 13  ....b...^..[0Y0.
ssl_tls.c:4233: |4| 0010:  06 07 2a 86 48 ce 3d 02 01 06 08 2a 86 48 ce 3d  ..*.H.=....*.H.=
ssl_tls.c:4233: |4| 0020:  03 01 07 03 42 00 04 a4 2a bd 2f 34 75 97 f9 c7  ....B...*./4u...
ssl_tls.c:4233: |4| 0030:  db 1c 65 b4 c3 45 bd fd 15 8b b4 df dc 3c 80 ce  ..e..E.......<..
ssl_tls.c:4233: |4| 0040:  54 6f 42 18 a8 b5 e1 65 aa 6b 9c 43 bd 24 5f c1  ToB....e.k.C.$_.
ssl_tls.c:4233: |4| 0050:  f3 a3 f2 4b 4f 01 3f c0 7d 73 ef 02 be d5 1f 41  ...KO.?.}s.....A
ssl_tls.c:4233: |4| 0060:  4b ed 28 0e 24 9b 62                             K.(.$.b
ssl_tls.c:3626: |3| handshake message: msglen = 98, type = 11, hslen = 98
ssl_tls.c:4385: |2| <= read record
Print buffer at 0x2024374, buffer length 91
        3059301306072a8648ce3d020106082a8648ce3d03010703420004a42abd2f347597f9c7db1c65b4
        c345bdfd158bb4dfdc3c80ce546f4218a8b5e165aa6b9c43bd245fc1f3a3f24b4f013fc07d73ef02
        bed51f414bed280e249b62
Successfully verified peer's raw key
ssl_tls.c:5628: |3| dumping 'peer raw public key' (91 bytes)
ssl_tls.c:5628: |3| 0000:  30 59 30 13 06 07 2a 86 48 ce 3d 02 01 06 08 2a  0Y0...*.H.=....*
ssl_tls.c:5628: |3| 0010:  86 48 ce 3d 03 01 07 03 42 00 04 a4 2a bd 2f 34  .H.=....B...*./4
ssl_tls.c:5628: |3| 0020:  75 97 f9 c7 db 1c 65 b4 c3 45 bd fd 15 8b b4 df  u.....e..E......
ssl_tls.c:5628: |3| 0030:  dc 3c 80 ce 54 6f 42 18 a8 b5 e1 65 aa 6b 9c 43  .<..ToB....e.k.C
ssl_tls.c:5628: |3| 0040:  bd 24 5f c1 f3 a3 f2 4b 4f 01 3f c0 7d 73 ef 02  .$_....KO.?.}s..
ssl_tls.c:5628: |3| 0050:  be d5 1f 41 4b ed 28 0e 24 9b 62                 ...AK.(.$.b
ssl_tls.c:5928: |3| Certificate verification flags clear
ssl_tls.c:5935: |2| <= parse certificate
ssl_cli.c:3652: |2| client state: 4
ssl_tls.c:2755: |2| => flush output
ssl_tls.c:2767: |2| <= flush output
ssl_cli.c:2478: |2| => parse server key exchange
ssl_tls.c:4311: |2| => read record
ssl_tls.c:2536: |2| => fetch input
ssl_tls.c:2697: |2| in_left: 0, nb_want: 5
ssl_tls.c:2721: |2| in_left: 0, nb_want: 5
ssl_tls.c:2722: |2| ssl->f_recv(_timeout)() returned 5 (-0xfffffffb)
ssl_tls.c:2742: |2| <= fetch input
ssl_tls.c:4047: |4| dumping 'input record header' (5 bytes)
ssl_tls.c:4047: |4| 0000:  16 03 03 00 d9                                   .....
ssl_tls.c:4056: |3| input record: msgtype = 22, version = [3:3], msglen = 217
ssl_tls.c:2536: |2| => fetch input
ssl_tls.c:2697: |2| in_left: 5, nb_want: 222
ssl_tls.c:2721: |2| in_left: 5, nb_want: 222
ssl_tls.c:2722: |2| ssl->f_recv(_timeout)() returned 217 (-0xffffff27)
ssl_tls.c:2742: |2| <= fetch input
ssl_tls.c:4233: |4| dumping 'input record from network' (222 bytes)
```

Figure 5.6: The TLS client successfully authenticates the server based on the SHA-256 digest of the raw public key presented by the server.



```
 1 0.000000000   127.0.0.1   127.0.0.1   TCP      74 33820 → 4433 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=3829497802 TSecr=0 WS=
 2 0.000012648   127.0.0.1   127.0.0.1   TCP      74 4433 → 33820 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=3829497802
 3 0.000021721   127.0.0.1   127.0.0.1   TCP      66 33820 → 4433 [ACK] Seq=1 Ack=1 Win=43776 Len=0 TSval=3829497802 TSecr=3829497802
 4 0.000227691   127.0.0.1   127.0.0.1   TLSv1.2 215 Client Hello
 5 0.000230802   127.0.0.1   127.0.0.1   TCP      66 4433 → 33820 [ACK] Seq=1 Ack=150 Win=44800 Len=0 TSval=3829497802 TSecr=3829497802
 6 0.000761808   127.0.0.1   127.0.0.1   TLSv1.2 144 Server Hello
 7 0.000764816   127.0.0.1   127.0.0.1   TCP      66 33820 → 4433 [ACK] Seq=150 Ack=79 Win=43776 Len=0 TSval=3829497802 TSecr=3829497802
 8 0.000878034   127.0.0.1   127.0.0.1   TLSv1.2 169 Certificate
 9 0.000881405   127.0.0.1   127.0.0.1   TCP      66 33820 → 4433 [ACK] Seq=150 Ack=182 Win=43776 Len=0 TSval=3829497803 TSecr=3829497803
10 0.008584896   127.0.0.1   127.0.0.1   TLSv1.2  73 Alert (Level: Fatal, Description: Certificate Unknown)
11 0.008615423   127.0.0.1   127.0.0.1   TCP      66 33820 → 4433 [FIN, ACK] Seq=157 Ack=182 Win=43776 Len=0 TSval=3829497810 TSecr=3829497803
12 0.011791384   127.0.0.1   127.0.0.1   TLSv1.2 287 Server Key Exchange
13 0.011804246   127.0.0.1   127.0.0.1   TCP      54 33820 → 4433 [RST] Seq=158 Win=0 Len=0
14 158.017075020 127.0.1.1   127.0.0.1   NBSS     70 Session keep-alive
15 158.018067552 127.0.0.1   127.0.1.1   TCP      66 50450 → 139 [ACK] Seq=1 Ack=5 Win=275 Len=0 TSval=688012778 TSecr=2616040741
Frame 10: 73 bytes on wire (584 bits), 73 bytes captured (584 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
Transmission Control Protocol, Src Port: 33820, Dst Port: 4433, Seq: 150, Ack: 182, Len: 7
Secure Sockets Layer
▾ TLSv1.2 Record Layer: Alert (Level: Fatal, Description: Certificate Unknown)
     Content Type: Alert (21)
     Version: TLS 1.2 (0x0303)
     Length: 2
   ▾ Alert Message
        Level: Fatal (2)
        Description: Certificate Unknown (46)
```

Figure 5.7: The prototype client aborts TLS Handshake when the raw public key of the server does not belong to the server trust tree.

5.5 Summary

A limited prototype to demonstrate practical use of VTT with TLS is developed in C on Linux environment. The prototype is portable to any embedded platform. The prototype client authenticates a server connection based on the trust tree hard-coded in the client. Connections to adversary not belonging to the server entities in the hard-coded tree are rejected as visible in Figure 5.7.

Evaluation of VTT with PKI is described in Chapter 6. However, already notice that the raw public key provided by the server in Figure 5.7 is much smaller than the certificate chain in Figure 5.4.

The revocation schemes proposed for VTT, Section 4.8 have not been implemented due to lack of time. As compared to Section 4.7, the prototype assumes only two access roles, server, and client. While this is sufficient for evaluation, it is far from real deployments. The prototype, however, as planned, serves the purpose of proof of concept.

Chapter 6 Evaluation

In this chapter, we evaluate PKI with VTT based on the performance of ECDSA verify and SHA-256 digest operations on an embedded platform. Dynamic memory and persistent storage requirements are also compared. For energy consumption of radio, the data exchanged on air for authentication during the handshake is compared. For this, a model based output for the target platform is used.

Notice that for the performance comparison, we measure the fundamental cryptographic operation, ECDSA verify and SHA-256. We do not measure the energy consumption for the entire handshake. This is to eliminate the use of other resource hungry components and conditions that can impact timing on target. Network conditions, radio power, other peripherals, and resources needed to run a protocol stack are examples of these.

6.1 Methodology

For the prototype, **ECDSA_SCEP256r1_SHA256** signature scheme verifies the PKI based X.509 certificate chain. [1] The length of the certificate chain is proportional to the number of **ECDSA_SCEP256r1_SHA256** based verification needed. The prototype setup, Section 5.4.1, defines a short chain, Figure 5.2, of length two, and a long certificate chain, Figure 5.1, of length eight. The number of ECDSA verify operations needed is two and eight, respectively.

VTT employs a Merkle tree with SHA-256 for the construction of the tree and tree membership verification. The depth of the tree is proportional to the number of hash operations needed to establish membership in the tree and the membership algorithm chosen. This is described in Section 4.5. The (4.6), (4.2) and (4.4) provide

[1] This signature algorithm is indicated in the `signature_algorithm` extension of the `ClientHello message`. TLS 1.3 [Res18] includes an explicit `signature_algorithm_cert` for signatures algorithm used for certificate verification.

the number of hashes needed for a tree of depth $\boldsymbol{n}$ with SLCO, SACL and the CASL methods respectively.

The method chosen for the prototype is SLCO. The depth of the trust tree, as described in Section 5.4.1, is three with eight members, as depicted in Figure 5.3.

Comparison of **ECDSA_SCEP256r1_SHA256** verification and **SHA-256** digest operations are good indicators of performance indicators for the PKI and VTT. Knowledge of cost of each operation and system setup provides extrapolated performance numbers for both the methods in any system.

6.1.1 Why ECDSA?

ECC based cryptography is chosen for the prototype as it provides better security strength with smaller key sizes as compared to RSA. 256-bit prime groups are adequate to achieve 128-bit security. Therefore, the key sizes and the signature size for ECC based ECDSA are shorter. With RSA, 3072-bit key length is needed for the same level of security. ECC is generally faster than RSA for key and signature generation. Verification, on the other hand, is slower. A white paper on public key cryptography on embedded applications [Mal15] reports RSA signature generation to be fifty to a hundred times slower than ECDSA.

For mutual authentication in TLS, signature generation is required. A downside of ECDSA is that verification is slower than RSA. Since ECDSA is used for the TLS cipher, ECDSA is also used for the signature verification on the X.509 certificates. Use of different schemes for the cipher and certificate signatures is possible. However, to keep the code base and toolbox limited on the embedded on the device, only ECC based cryptography is used.

6.1.2 Why SHA-256?

According to Wikipedia [Wik19g], SHA-1 is no longer considered collision resistant. In 2015, NIST issued an advisory to discontinue use of SHA-1. According to the advisory, any new application must support SHA-256 for interoperability. SHA-256 is one of the four hash functions offered in the SHA-2 family of hash functions. For 32-bit processors, SHA-256 provides a good balance between security and performance.

In the thesis, SHA-256 is used both for the construction of the trust trees and the verification of the X.509 certificates. The signatures are generated and verified for the hash of the certificate.

6.1.3 Platform

The device chosen for benchmarking is nRF52840 development kit by Nordic Semiconductor ASA. The development kit, Figure 6.1, is referred to as the nRF52840 DK or the PCA10056 and is the development platform for the nRF52840 SoC. The user guide for the development is available at [ASA19c].

The nRF52840 SoC [ASA19d] has a 32-bit ARM Cortex-M4 running at 64 MHz clock. The platform has 1MB flash and 256 kB RAM. The platform supports floating point unit and hardware accelerators for cryptographic operations. Originally, an evaluation for performance with and without hardware accelerators was planned. However, due to lack of time, the current work only results without the use of hardware accelerators.

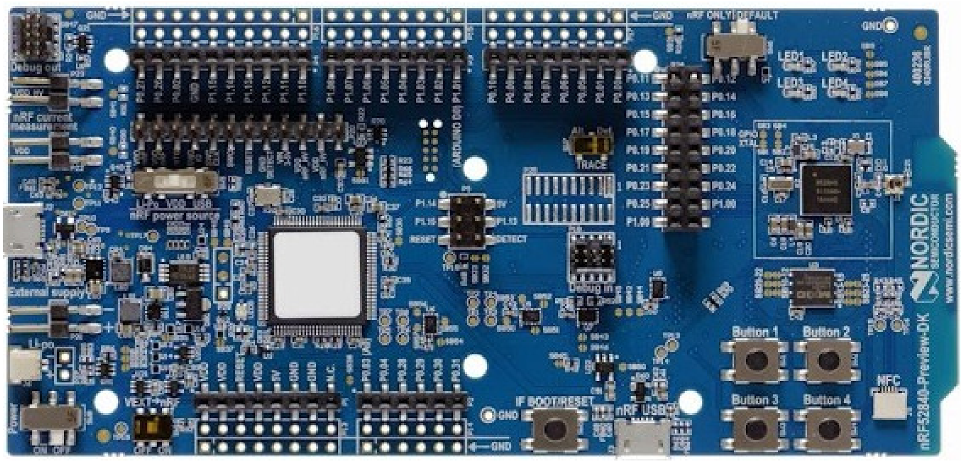


Figure 6.1: Nordic's nRF52840 is used to benchmark performance numbers for PKI and VTT.

6.1.4 Test parameters

The verify operation in ECDSA or any signature scheme, verifies the signature of the message digest and not on the message itself. **SHA-256** is the digest function used with ECDSA for the evaluation. The message size chosen is to be **512** bytes. The raw public keys used with VTT are never more than 128 bytes for ECC based ciphers. The X.509 certificates are typically at least 512 bytes, if not more. Therefore, 512 bytes is seen as a good compromise. Since ECDSA verifies the signature of a hashed message. Therefore, ECDSA performance measures do not change very much with the message size[2].

[2]This is true for reasonably sized messages. In theory, if the message is too large, the hash operation may overshadows the ECDSA verification process.

Test vector generation

Python is used for key and signature generation. The signature and the public key form the test vectors for the measurement application run on nRF52840. Python ECDSA module [Dev17a] is used to generate the test ECDSA signature. The test signature is verified with Python before use as a test vector on target. The Python script used for sign and verify operation is the Section A.2 of Appendix A. Python hashlib module [Dev17b] is used for SHA-256 message digest to confirm the correctness of the output of the embedded application. The script used is included in Section A.1.

6.1.5 Measurement tool

Nordic's Power Profiler Kit by Nordic Semiconductor ASA depicted in Figure 6.2, is the hardware used for performance measurement on the nRF52840. The PPK is quick and easy to use power measurement tool for Nordic's nRF5 series of SoCs.

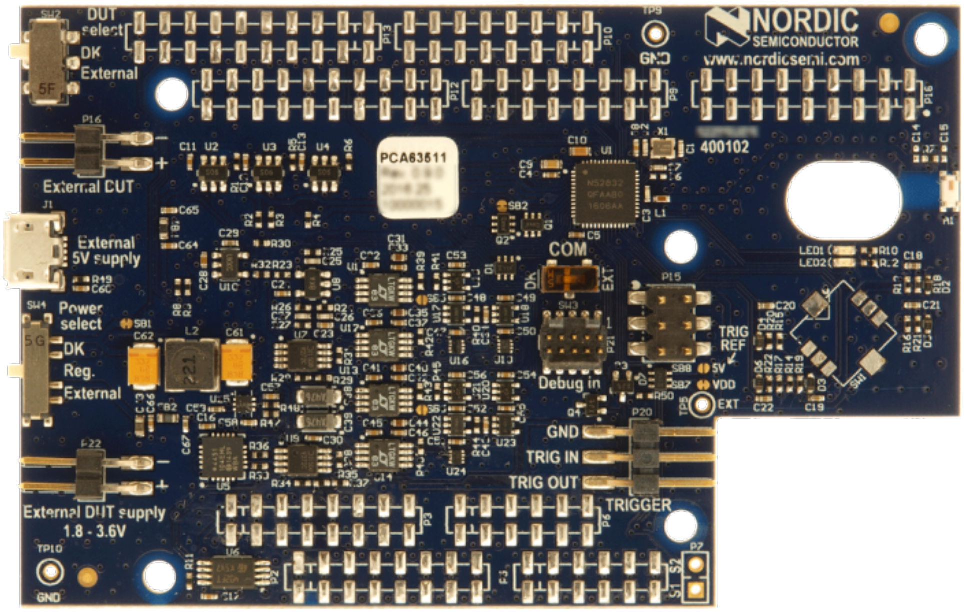
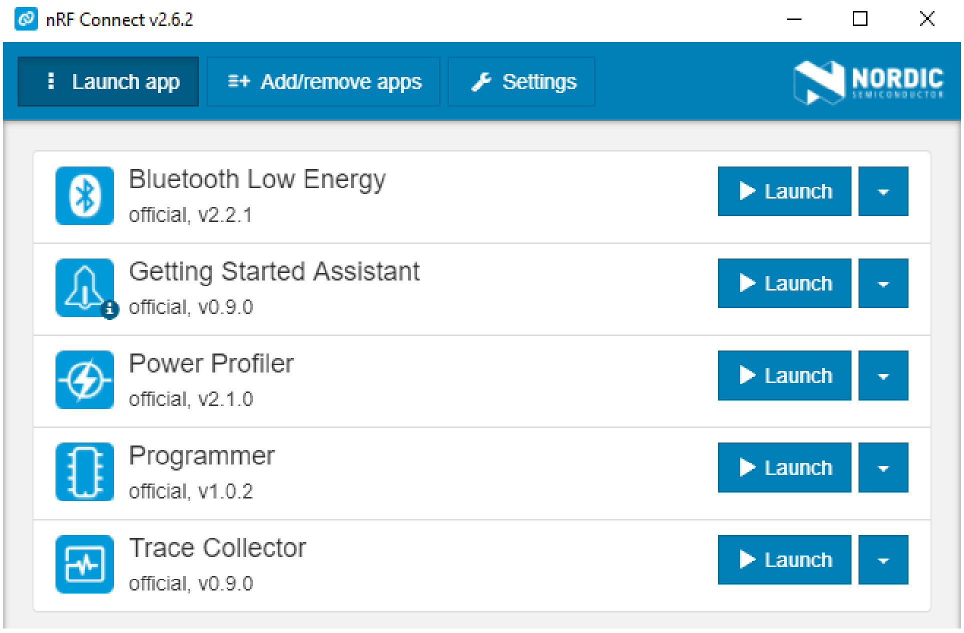


Figure 6.2: A Nordic Power Profiler Kit can be mounted on any variant of Nordic nRF52 DK to measure energy consumption of the application.

The nRF Connect for Desktop PC tool provides the needed software package (power profiler in Figure 6.3) to connect to the PPK. The nRFConnect Power Profiler software can update firmware on the PPK, start, stop measurements. Other features like setting the markers, setting the supply voltage, trigger events, etc. are offered as well. The complete user guide for the PPK is available at [ASA19b]. The nRF Connect for Desktop can be downloaded at [ASA19a].

The nRFConnect for Desktop is open source, [ASA19f], Javascript and HTML5 based ensemble of all PC tools by Nordic Semiconductor ASA. Each tool offered has its own scope purpose and revision history, as evident from Figure 6.3. Of the many tools, Power profiler is one of the tools offered and used here to measure the time of execution and current consumption during the measurement. Windows 10 operating system is used to run the PC software.



Bluetooth Low Energy
official, v2.2.1

Getting Started Assistant
official, v0.9.0

Power Profiler
official, v2.1.0

Programmer
official, v1.0.2

Trace Collector
official, v0.9.0

Figure 6.3: Nordic's nRFConnect for Desktop offers many PC tools to aid development on Nordic Development Kits. The Power Profiler PC software is one such tool for use with the PPK.

6.1.6 Application

The application(s), used is based on nRF SDK version 15.3.0, [ASA19g], from Nordic Semiconductor ASA. An RTC timer, a low power, high accuracy hardware timer is used to trigger the operation to be measured. The operation to be measured is either ECDSA or SHA-256. Once the measurement operations are complete, the CPU is put a sleep mode until woken up again by the RTC time interrupt. Many measurements are taken to ensure the measurements are consistent and hence reliable. The application used for measuring the ECDSA verification is included at Section A.5. The application used for SHA-256 is included at Section A.3.

Compiler

GNU Embedded Toolchain for Arm, [Hol19], version **7 2018-q2-update** is used to compiling and debugging the embedded applications included at Section A.5 and Section A.3. The Makefile used for the compilation is based on the examples included in nRF SDK release 15.3.0 [ASA19g].

6.2 Energy Consumption - Theory

Average current, I_{average} in ampere $(A)$is measured by the power profiler at a constant supply voltage, V_{supply} in volt (V). The average current is measured over time t_{m} in second (s). The average power, P_{average} in watt(W), is computed using (6.1) and the energy consumption for the measurement, E_{m}, in joule(J) is computed using (6.2).

$$P_{\text{average}} = I_{\text{average}} \times V_{\text{supply}} \tag{6.1}$$

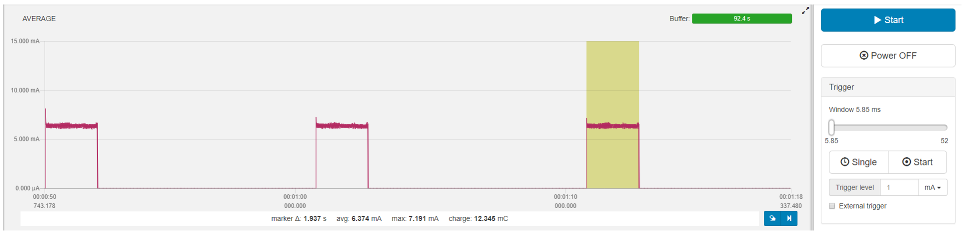
$$E_{\text{m}} = P_{\text{average}} \times t_{\text{m}} \tag{6.2}$$

6.3 Energy Consumption - Crypto

Time and current consumption on nRF52840 for ECDSA and SHA-256 is measured and compared here.

6.3.1 ECDSA Verify

Figure 6.4 depicts the duration and the average current consumption for the ECDSA verification of the test signature on nRF52840. Figure 6.5 depicts the interval between each measurement. Notice that in Figure 6.4 and Figure 6.5, the multiple measurements appear consistent in form.



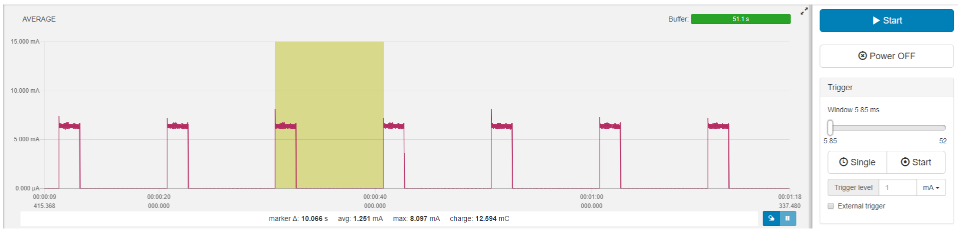
marker Δ: **1.937** s avg: **6.374** mA max: **7.191** mA charge: **12.345** mC

Figure 6.4: Measurements of the duration of the ECDSA Verify and the average current in the duration on nRF52840. At a constant voltage of 3.3 V, these measurements provide the energy consumption for the operation.

On nRF52840, each successful ECDSA verify operation requires **1937** ms, and the average current during this duration is **6.374** mA at **3.3** V. Therefore, from

(6.1), the average power consumption for ECDSA verify operation is **21.0342** mW. The energy consumption, by (6.2) is **40.743** mJ.

A side observation is that the power and energy consumption for failure in verification is the same as a successful verification. Additionally, the current consumption of 6.3 mA for ECDSA suggests that the load on the CPU is 100%. Refer the electrical specification of the nRF52840 SoC [ASA19e] for details.



Window 5.85 ms

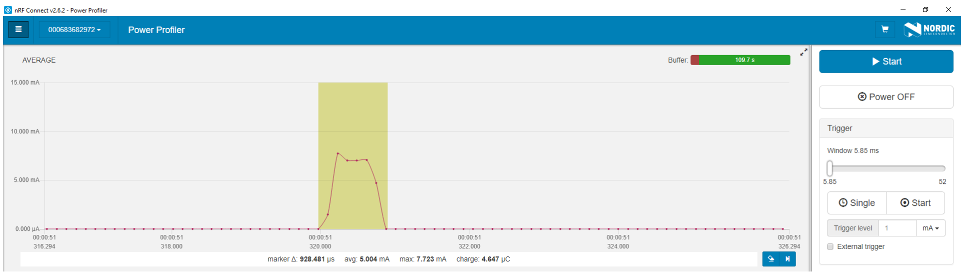
Trigger level 1 mA

marker Δ: **10.066** s avg: **1.251** mA max: **8.097** mA charge: **12.594** mC

Figure 6.5: The measurement of ECDSA verify is triggered at a regular interval of 10 seconds using the RTC peripheral. Multiple measurements help identify any inconsistencies in the measurements. Notice that the current consumption is consistent across three measurements.

6.3.2 SHA-256 Digest

Figure 6.6 and Figure 6.7 depict the measurements captured for SHA-256 digest operation for a 512-byte message. The execution time for SHA-256 is very short (< 1 ms) and may appear to be just a pulse, as seen in Figure 6.7. Each SHA-256 digest operation requires **0.928** ms, and the average current during this period is **5.004** mA at **3.3** V. Therefore, for SHA-256 operation on nRF52840, the average power, and the energy consumption, by (6.1), and (6.2), are **16.513** mW and **15.324** nJ, respectively.



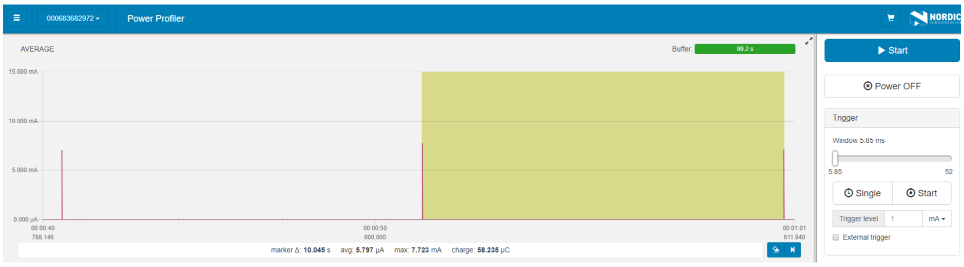
Window 5.85 ms

Trigger level 1 mA

marker Δ: **928.481** µs avg: **5.004** mA max: **7.723** mA charge: **4.647** µC

Figure 6.6: The measurement of the duration of the SHA-256 Digest and the average current in the duration on nRF52840. The digest is computed on a 512 byte message.

The cause of the initial spike at the start of measurement in Figure 6.4 and Figure 6.6 is likely due to switching from low-frequency clock to high-frequency clock on CPU wake up. This is the hypothesis provided by the engineers at Nordic. They share that this is the observed and hence the expected behavior. However, this is neither documented nor further analyzed.



marker Δ: **10.045** s avg: **5.797** µA max: **7.723** mA charge: **58.235** µC

Figure 6.7: The application on target triggers a SHA-256 digest operation every 10 seconds using the RTC. Therefore, we observe small sleep currents when the CPU is in the power saving mode. Notice that the duration of each SHA-256 digest is very small, and therefore, has the appearance of a pulse.

6.3.3 ECDSA Verify vs. SHA-256 Digest

Table 6.1 summarizes the performance of ECDSA verify and SHA-256 digest on nRF52840. For the measurements, the application is executed from flash. This is the most typical configuration for embedded firmware. Execution from RAM, though faster, is not considered as most embedded software have limited RAM and is not intended for code execution.

Measurement/ Function	**SHA-256**	**ECDSA Verify**
$\mathbf{t_m}$ (s)	928×10^{-6}	1.937
$\mathbf{I_{average}}$ (A)	5.004×10^{-3}	6.374×10^{-3}
$\mathbf{P_{average}}$ (W)	16.513×10^{-3}	21.0324×10^{-3}
$\mathbf{E_m}$ (J)	15.324×10^{-9}	40.743×10^{-3}

Table 6.1: ECDSA Verify vs. SHA-256: Performance comparison on nRF52840 DK at **3.3** V. As expected, SHA-256 performs better than ECDSA verify.

From 6.1, the execution time for one ECDSA verify operation is equivalent to **2000** times execution time that of SHA-256 digest operation. Moreover, the energy consumption of 2000 SHA-256 operations would be at least a **1000** times lower than one SHA-256 verify operation.

6.4 PKI vs. VTT: Energy consumption - Verify

The energy consumption comparison of an ECDSA verify operation with a SHA-256 digest in Section 6.3.3 provides the basis for the comparison of PKI with VTT. The number of ECDSA verify operations needed for certificate chain verification depends on the depth of the certificate chain. For VTT, as described in Section 4.5, the number of digest operations depends on the depth of the tree, and the choice of membership algorithm. Table 6.2 summarizes the count of operations needed for the setup described in Section 5.4.1.

Method/ Operation	**PKI**		**VTT**		
	Long	**Short**	**SLCO**	**SACL**	**CASL**
SHA-256[3]	N/A	N/A	1	4	15
ECDSA Verify	9	2	N/A	N/A	N/A

Table 6.2: PKI vs. VTT: Count of cryptographic operations per authentication for evaluation setup. For PKI, depth of certificate chain determines the count. For VTT, a combination of depth of trust tree and the membership algorithm.

Table 6.3 summarizes the performance parameters of PKI and VTT based on parameters for ECDSA and SHA-256 measurements in Table 6.1 and the number of operations needed in Table 6.2. As is evident, the smallest configuration for the PKI is very expensive for a system that has clients communicating to eight servers. These results are revisited in Chapter 7 to provide a better comparison based on the system as a whole.

Performance/ Authentication	**PKI**		**VTT**		
	Long	**Short**	**SLCO**	**SACL**	**CASL**
$\mathbf{t_{verify}}$ (s)	17.433	3.874	928×10^{-6}	3.712×10^{-3}	13.93×10^{-3}
$\mathbf{E_{verify}}$ (J)	0.366	0.081	15.324×10^{-9}	61.296×10^{-9}	229.863×10^{-9}

Table 6.3: PKI vs. VTT: Energy consumption for authentication. This table is the prodcut of the measured energy consumption for ECDSA verify and SHA-256 Digest in Table 6.1 and the count of these operations in Table 6.2.

6.5 Dynamic memory

The dynamic memory, that is, the heap-allocated memory at run time, for authentication using PKI and VTT, is compared here. The memory, $\mathbf{Heap_{parse}}$, needed for parsing an X.509 certificate or a raw public key is evaluated separately from the memory needed for computing the ECDSA signature or the SHA-256 digest, $\mathbf{Heap_{verify}}$. This $\mathbf{Heap_{verify}}$ depends only on the choice of the cryptographic algorithm, while $\mathbf{Heap_{parse}}$ depends on the length of the certificate chain in case of PKI.

The mbed TLS library allocates 1400 bytes per X.509 certificate and ECDSA public key context. **1400** bytes are observed to be allocated for each certificate in

[3]ECDSA verify and hence PKI requires a digest operation, here, SHA-256. These are excluded as the measured of time and performance already account for this.

the chain. For the raw public key, **500** bytes are allocated for public key context and additional memory to parse ASN.1 encoded signature algorithm id.

For ECDSA verify operation, **7200** bytes are allocated on the heap for computing the group operations needed for signature verification. SHA-256 digest requires only **500** bytes. Based on these measurements, a comparison of PKI and VTT is provided in Table 6.4.

Table 6.4 summarizes the dynamic memory requirements for PKI and VTT. The heap size for VTT is considerably smaller than PKI. Additionally, the larger heaps are not required for bigger trust trees.

Dynamic Memory/ Authentication	**PKI**		**VTT**		
	Long	**Short**	**SLCO**	**SACL**	**CASL**
Heap$_{\mathbf{parse}}$ (B)	12600	2800	400	400	400
Heap$_{\mathbf{verify}}$ (B)	7200	7200	500	500	500

Table 6.4: PKI vs. VTT: Dynamic memory(heap) requirements. Memory requirements are determined only by the choice of signature verification and digest functions.

6.6 Persistent Storage memory

PKI and VTT, both require an initial seeding of trust in each entity. The trust seed in case of PKI is the X.509 certificate of the root CAs. In the case of VTT, the provisioning of the trust tree is a prerequisite for authentication. There may be more than one trust anchors in the system; however, here, we assume only one is used.

The trust seed must be stored persistently in the embedded device, and should preferably be guarded against any tamper. Table 6.5 compares persistent memory requirements for trust seed with PKI and VTT.

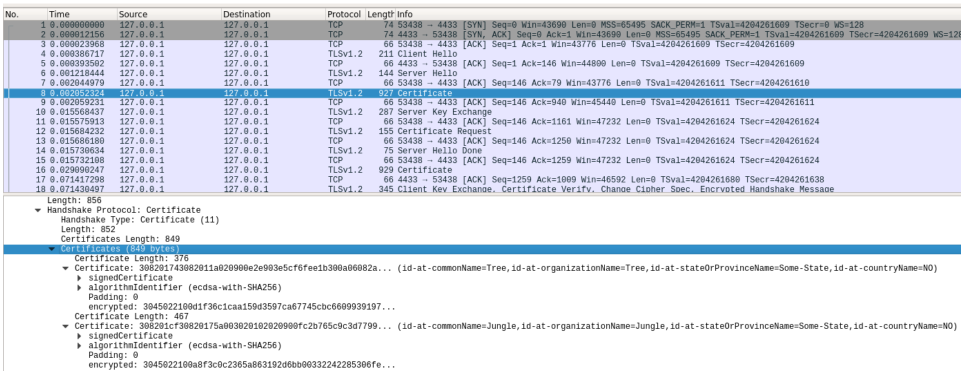
Persistent Memory/ Authentication	**PKI**		**VTT**		
	Long	**Short**	**SLCO**	**SACL**	**CASL**
Storage$_{\mathbf{trust}}$ (B)	476	476	288	480	288

Table 6.5: PKI vs. VTT: Persistent memory requirements for evaluation setup on nRF52840. For PKI, the count of root CA and the size of each root CA certificate determines the needed non-volatile memory. For VTT, the count and size of each trust determines the needed memory.

The size of the root certificate is determined by choice of the signature algorithm and the additional fields included in the certificate. The additional fields describe the entity and the use and the validity of the certificate. The public key and signature for ECDSA are 64 bytes each. Therefore, here, of 476, adds only 128 bytes to the certificate. With RSA of minimum **2048**-bit key size, the key and signature are of 256 bytes each. We discuss this further in Chapter 7.

6.7 Energy Consumption - Radio

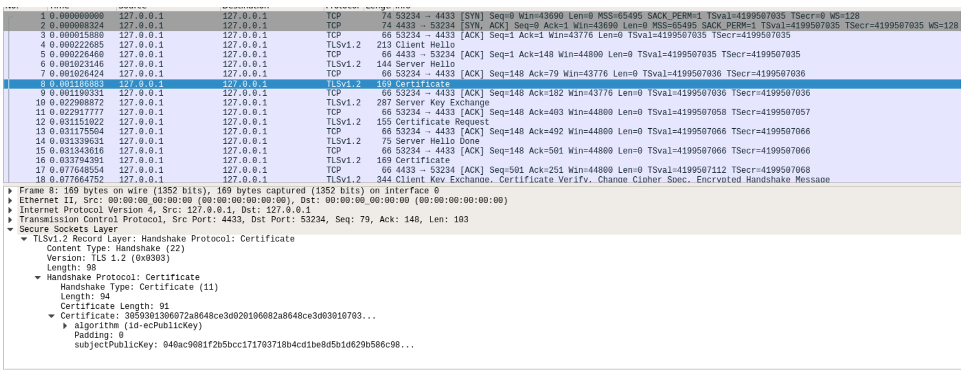
Radio is the primary contributor to current consumption in an embedded system apart from the CPU. Each byte exchanged on air, therefore, contributes to the energy consumption on an embedded device. The actual current consumption depends on the access technology, modulation, radio conditions, and several other factors. Given the energy consumed to exchange one byte on air, the amount of data needed per TLS handshake per entity authentication can be used to compare VTT with PKI.



```
No.    Time         Source      Destination  Protocol Length Info
     1 0.000000000  127.0.0.1   127.0.0.1    TCP        74 53438 → 4433 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=4204261609 TSecr=0 WS=128
     2 0.000012156  127.0.0.1   127.0.0.1    TCP        74 4433 → 53438 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=4204261609 TSecr=4204261609 WS=128
     3 0.000023968  127.0.0.1   127.0.0.1    TCP        66 53438 → 4433 [ACK] Seq=1 Ack=1 Win=43776 Len=0 TSval=4204261609 TSecr=4204261609
     4 0.000386717  127.0.0.1   127.0.0.1    TLSv1.2   211 Client Hello
     5 0.000393502  127.0.0.1   127.0.0.1    TCP        66 4433 → 53438 [ACK] Seq=1 Ack=146 Win=44800 Len=0 TSval=4204261609 TSecr=4204261609
     6 0.001218444  127.0.0.1   127.0.0.1    TLSv1.2   144 Server Hello
     7 0.002044979  127.0.0.1   127.0.0.1    TCP        66 53438 → 4433 [ACK] Seq=146 Ack=79 Win=43776 Len=0 TSval=4204261611 TSecr=4204261610
     8 0.002052324  127.0.0.1   127.0.0.1    TLSv1.2   927 Certificate
     9 0.002059231  127.0.0.1   127.0.0.1    TCP        66 53438 → 4433 [ACK] Seq=146 Ack=940 Win=45440 Len=0 TSval=4204261611 TSecr=4204261611
    10 0.015568437  127.0.0.1   127.0.0.1    TLSv1.2   287 Server Key Exchange
    11 0.015575913  127.0.0.1   127.0.0.1    TCP        66 53438 → 4433 [ACK] Seq=146 Ack=1161 Win=47232 Len=0 TSval=4204261624 TSecr=4204261624
    12 0.015684232  127.0.0.1   127.0.0.1    TLSv1.2   155 Certificate Request
    13 0.015686180  127.0.0.1   127.0.0.1    TCP        66 53438 → 4433 [ACK] Seq=146 Ack=1250 Win=47232 Len=0 TSval=4204261624 TSecr=4204261624
    14 0.015730634  127.0.0.1   127.0.0.1    TLSv1.2    75 Server Hello Done
    15 0.015732108  127.0.0.1   127.0.0.1    TCP        66 53438 → 4433 [ACK] Seq=146 Ack=1259 Win=47232 Len=0 TSval=4204261624 TSecr=4204261624
    16 0.029090247  127.0.0.1   127.0.0.1    TLSv1.2   929 Certificate
    17 0.071417298  127.0.0.1   127.0.0.1    TCP        66 4433 → 53438 [ACK] Seq=1259 Ack=1009 Win=46592 Len=0 TSval=4204261680 TSecr=4204261638
    18 0.071430497  127.0.0.1   127.0.0.1    TLSv1.2   345 Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message

      Length: 856
    ▼ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 852
        Certificates Length: 849
      ▼ Certificates (849 bytes)
          Certificate Length: 376
        ▼ Certificate: 30820174308201 1a020900e2e903e5cf6fee1b300a06082a... (id-at-commonName=Tree,id-at-organizationName=Tree,id-at-stateOrProvinceName=Some-State,id-at-countryName=NO)
          ▶ signedCertificate
          ▶ algorithmIdentifier (ecdsa-with-SHA256)
            Padding: 0
            encrypted: 3045022100d1f36c1caa159d3597ca67745cbc6609939197...
          Certificate Length: 467
        ▼ Certificate: 308201cf30820175a003020102020900fc2b765c9c3d7799... (id-at-commonName=Jungle,id-at-organizationName=Jungle,id-at-stateOrProvinceName=Some-State,id-at-countryName=NO)
          ▶ signedCertificate
          ▶ algorithmIdentifier (ecdsa-with-SHA256)
            Padding: 0
            encrypted: 3045022100a8f3c0c2365a863192d6bb00332242285306fe...
```

Figure 6.8: Size of Short Certificate Chain. The depth of the short certificate chain is determined by the setup in Figure 5.2.

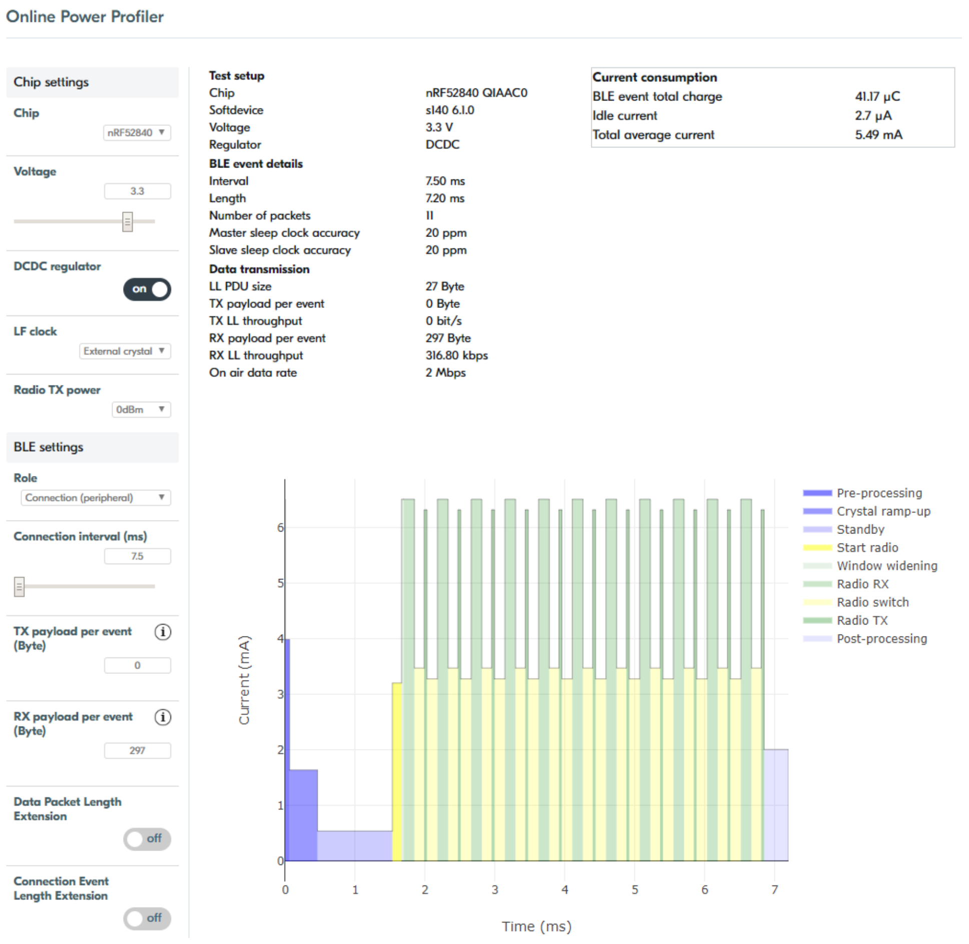


```
     1 0.000000000  127.0.0.1   127.0.0.1    TCP        74 53234 → 4433 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=4199507035 TSecr=0 WS=128
     2 0.000008324  127.0.0.1   127.0.0.1    TCP        74 4433 → 53234 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=4199507035 TSecr=4199507035 WS=128
     3 0.000015880  127.0.0.1   127.0.0.1    TCP        66 53234 → 4433 [ACK] Seq=1 Ack=1 Win=43776 Len=0 TSval=4199507035 TSecr=4199507035
     4 0.000222685  127.0.0.1   127.0.0.1    TLSv1.2   213 Client Hello
     5 0.000226460  127.0.0.1   127.0.0.1    TCP        66 4433 → 53234 [ACK] Seq=1 Ack=148 Win=44800 Len=0 TSval=4199507035 TSecr=4199507035
     6 0.001023146  127.0.0.1   127.0.0.1    TLSv1.2   144 Server Hello
     7 0.001026424  127.0.0.1   127.0.0.1    TCP        66 53234 → 4433 [ACK] Seq=148 Ack=79 Win=43776 Len=0 TSval=4199507036 TSecr=4199507036
     8 0.001186883  127.0.0.1   127.0.0.1    TLSv1.2   169 Certificate
     9 0.001190331  127.0.0.1   127.0.0.1    TCP        66 53234 → 4433 [ACK] Seq=148 Ack=182 Win=43776 Len=0 TSval=4199507036 TSecr=4199507036
    10 0.022908872  127.0.0.1   127.0.0.1    TLSv1.2   287 Server Key Exchange
    11 0.022917777  127.0.0.1   127.0.0.1    TCP        66 53234 → 4433 [ACK] Seq=148 Ack=403 Win=44800 Len=0 TSval=4199507058 TSecr=4199507057
    12 0.031151022  127.0.0.1   127.0.0.1    TLSv1.2   155 Certificate Request
    13 0.031175504  127.0.0.1   127.0.0.1    TCP        66 53234 → 4433 [ACK] Seq=148 Ack=492 Win=44800 Len=0 TSval=4199507066 TSecr=4199507066
    14 0.031339631  127.0.0.1   127.0.0.1    TLSv1.2    75 Server Hello Done
    15 0.031343616  127.0.0.1   127.0.0.1    TCP        66 53234 → 4433 [ACK] Seq=148 Ack=501 Win=44800 Len=0 TSval=4199507066 TSecr=4199507066
    16 0.033794391  127.0.0.1   127.0.0.1    TLSv1.2   169 Certificate
    17 0.077648554  127.0.0.1   127.0.0.1    TCP        66 4433 → 53234 [ACK] Seq=501 Ack=251 Win=44800 Len=0 TSval=4199507112 TSecr=4199507068
    18 0.077664752  127.0.0.1   127.0.0.1    TLSv1.2   344 Client Key Exchange, Certificate Verify, Change Cipher Spec, Encrypted Handshake Message

▶ Frame 8: 169 bytes on wire (1352 bits), 169 bytes captured (1352 bits) on interface 0
▶ Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 4433, Dst Port: 53234, Seq: 79, Ack: 148, Len: 103
▼ Secure Sockets Layer
  ▼ TLSv1.2 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 98
    ▼ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 94
        Certificate Length: 91
      ▼ Certificate: 3059301306072a8648ce3d020106082a8648ce3d03010703...
        ▶ algorithm (id-ecPublicKey)
          Padding: 0
          subjectPublicKey: 040ac9081f2b5bcc171703718b4cd1be8d5b1d629b586c98...
```

Figure 6.9: The size of the raw public key for any ECC 256-bit curve exchanged during the TLS handshake. With VTT, the client verifies the server's raw public key to be a member of the server trust tree.

In the case of PKI, the entire certificate chain is exchanged during the TLS handshake. As depicted in Figure 5.4, the long certificate chain is of size **3727** bytes and depth **8**. For the short certificate chain in Figure 6.8, an embedded client must receive **927** bytes to authenticate the server. The depth of the certificate chain and the size of each certificate contribute to the use of the radio resources. Similarly, the client must transmit its certificate chain to authenticate itself. In comparison, for VTT, as we see in Figure 6.9, an exchange of a raw public key of **91** bytes in each direction is adequate for entity authentication.


Online Power Profiler

Chip settings

Chip: nRF52840

Voltage: 3.3

DCDC regulator: on

LF clock: External crystal

Radio TX power: 0dBm

BLE settings

Role: Connection (peripheral)

Connection interval (ms): 7.5

TX payload per event (Byte): 0

RX payload per event (Byte): 297

Data Packet Length Extension: off

Connection Event Length Extension: off

Test setup	
Chip	nRF52840 QIAAC0
Softdevice	s140 6.1.0
Voltage	3.3 V
Regulator	DCDC
BLE event details	
Interval	7.50 ms
Length	7.20 ms
Number of packets	11
Master sleep clock accuracy	20 ppm
Slave sleep clock accuracy	20 ppm
Data transmission	
LL PDU size	27 Byte
TX payload per event	0 Byte
TX LL throughput	0 bit/s
RX payload per event	297 Byte
RX LL throughput	316.80 kbps
On air data rate	2 Mbps

Current consumption	
BLE event total charge	41.17 μC
Idle current	2.7 μA
Total average current	5.49 mA

Figure 6.10: Model-based estimate of the energy consumption for each BLE **7.5** *ms* connection interval on nRF52840.

We use Nordic's online power profiler tool [ASA16] to estimate the energy consumption on the radio on nRF52840. BLE, a low power access technology supported, [Wik19a], supported on nRF52840, is used for the estimates. As seen in Figure 6.10, at **3.3** *V*, with the connection interval of **7.5** *ms* for **2.0** *Mbps* data rate, each interval receives a maximum of **297** *bytes*. From this model-based estimate, the average current during the interval of **7.5** *ms* is **5.49** *mA*. Therefore, from (6.1) and (6.2), the energy consumed to receive **297 bytes** of data is **135.8775** μJ. Therefore, energy consumed by the radio to receive one byte is **457.5** *nJ/byte*.

One BLE connection event is adequate to receive a raw public key of **91** *bytes*. Therefore, VTT requires one BLE event. However, for PKI, multiple BLE events are necessary to receive the large certificates chain. A minimum of 4 and 13 connection events are used to receive the short and long certificate chains, respectively. Based on the bytes exchanged for authentication, and the energy consumed to receive one byte, Table 6.6 summarizes the average energy consumed for the radio with PKI and VTT.

Performance/ Authentication	**PKI**		**VTT**		
	Long	**Short**	**SLCO**	**SACL**	**CASL**
$\mathbf{Data_{rx}}$ (B)	3727	927	91	91	91
$\mathbf{t_{radio}}$ s	97.5×10^{-3}	30×10^{-3}	7.5×10^{-3}	7.5×10^{-3}	7.5×10^{-3}
$\mathbf{E_{radio}}$ J	1.705×10^{-3}	0.428×10^{-3}	41.632×10^{-6}	41.632×10^{-6}	41.632×10^{-6}

Table 6.6: PKI vs. VTT: Energy consumption for the radio on nRF52840. The comparison PKI and VTT based on the amount data exchange on air to exchange for authentication using BLE @ 2.0 *Mbps*.[4]

Notice in Table 6.6, the energy consumption for the radio for VTT based authentication is at least 10 times less than the smallest possible with PKI. The energy consumption for VTT only depends on the size of the raw public key, and hence, the identity algorithm. The depth of the trust tree or the choice of membership algorithm do not influence the radio resources. However, for PKI, apart from the size of the X.509 certificate, the depth of the certificate chain is directly proportional to the radio resources needed.

6.8 Consolidated results and analysis

In Section 4 of Chapter 4, when describing the alternative to the Public Key Infrastructure (PKI), two goals are defined. Here, we evaluate if the two goals are met. The various individual results are consolidated to investigate any dependencies or correlation between memory, storage, and energy consumption for the two schemes. Table 6.8 combines performance and resource usage numbers from Table 4.1, Table 6.4, Table 6.5, and Table 6.6.

[4]The time needed on the radio is computed based on the number of connection events needed to exchange data, and the duration of the interval. As there is a minimum time to wait before the next connection interval is scheduled.

Performance/ Authentication	**PKI**		**VTT**		
	Long	**Short**	**SLCO**	**SACL**	**CASL**
$\mathbf{t_{verify}}$ (s)	17.433	3.874	928×10^{-6}	3.712×10^{-3}	13.93×10^{-3}
$\mathbf{E_{verify}}$ (J)	0.366	0.081	15.324×10^{-9}	61.296×10^{-9}	229.863×10^{-9}
$\mathbf{Heap_{parse}}$ (B)	12600	2800	400	400	400
$\mathbf{Heap_{verify}}$ (B)	7200	7200	500	500	500
$\mathbf{Storage_{trust}}$ (B)	476	476	480	480	288
$\mathbf{Data_{rx}}$ (B)	3727	927	91	91	91
$\mathbf{t_{radio}}$ s	97.5×10^{-3}	30×10^{-3}	7.5×10^{-3}	7.5×10^{-3}	7.5×10^{-3}
$\mathbf{E_{radio}}$ J	1.705×10^{-3}	0.428×10^{-3}	41.632×10^{-6}	41.632×10^{-6}	41.632×10^{-6}

Table 6.7: PKI vs. VTT: Evaluation Summary. All membership verification for VTT fairs better than the most optimistic setup for PKI. The results are consistent with the expected as VTT uses SHA-256 while PKI uses ECDSA Verify for authentication.

We find that the energy required for ECDSA verify is equivalent to the energy needed to receive 87 kB on BLE radio at 2.0 *Mbps*. Further, it comparable to the energy needed to compute 2.5 million SHA-256 digests on 512 *B* messages. The results in Table 6.8 are consistent with the findings in [WGE+05] on an 8-bit processor. The exact values differ, as the test parameters and the target processors are not identical. However, energy consumption wise, ECC being most predominant, even marginalizing the energy needed for the radio is consistent with the findings in this thesis. Reference [WGE+05] confirms that the SHA operations adding negligibly to energy consumption in comparison to radio. The hash-based VTT is more efficient than PKI.

Many of the findings are specific to the choice of the evaluation parameters used. For example, the persistent memory requirement for VTT increases with the depth of the tree. The choice of the signature verification function, the certificate chain depth and size, the depth of the trust tree, and the choice of radio are all factors that influence the performance and the resource usage. However, based on the findings, some generalizations may be derived. These generalizations are useful for later in Section 7.3 to make deployment recommendations for PKI.

For PKI, the following generalizations apply.

$$\begin{aligned} \mathrm{E}_{\mathrm{verify}} \quad & \perp\!\!\!\perp \mathrm{Algorithm}_{\mathrm{Hash}} \\ & \perp\!\!\!\perp \mathrm{Depth}_{\mathrm{CertChain}} \end{aligned} \tag{6.3}$$

$$Heap_{\text{verify}} \perp\!\!\!\perp Algorithm_{\text{Signature}} \tag{6.4}$$

$$Storage_{\text{Trust}} \perp\!\!\!\perp Algorithm_{\text{Signature}} \tag{6.5}$$

$$E_{\text{radio}} \propto Depth_{\text{CertChain}} \times KeySize_{\text{Algorithm}_{\text{Signature}}} \tag{6.6}$$

$$Heap_{\text{radio}} \propto Depth_{\text{CertChain}} \times KeySize_{\text{Algorithm}_{\text{Signature}}} \tag{6.7}$$

$$E_{\text{verify}} >>> E_{\text{radio}} \tag{6.8}$$

For VTT, the following generalizations apply.

$$\begin{aligned} \mathrm{E}_{\mathrm{verify}} \; & \perp\!\!\!\perp \mathrm{Algorithm}_{\mathrm{Hash}} \\ & \perp\!\!\!\perp \mathrm{Algorithm}_{\mathrm{TreeMembership}} \\ & \perp\!\!\!\perp \mathrm{Depth}_{\mathrm{TrustTree}} \end{aligned} \tag{6.9}$$

$$\begin{aligned} \mathrm{Heap}_{\mathrm{verify}} \; & \perp\!\!\!\perp \mathrm{Algorithm}_{\mathrm{Hash}} \\ & \perp\!\!\!\perp \mathrm{Algorithm}_{\mathrm{TreeMembership}} \\ & \perp\!\!\!\perp \mathrm{Depth}_{\mathrm{TrustTree}} \end{aligned} \tag{6.10}$$

$$\begin{aligned} \mathrm{Storage}_{\mathrm{Trust}} \; & \perp\!\!\!\perp \mathrm{Algorithm}_{\mathrm{Hash}} \\ & \perp\!\!\!\perp \mathrm{Algorithm}_{\mathrm{TreeMembership}} \\ & \perp\!\!\!\perp \mathrm{Depth}_{\mathrm{TrustTree}} \end{aligned} \tag{6.11}$$

$$E_{\mathrm{radio}} \propto KeySize_{\mathrm{SignatureAlgorithm}} \tag{6.12}$$

$$Heap_{\mathrm{radio}} \propto KeySize_{\mathrm{SignatureAlgorithm}} \tag{6.13}$$

$$E_{\mathrm{radio}} >> E_{\mathrm{verify}}, \, \mathit{if} \; n < 10 \tag{6.14}$$

6.8.1 Embedded device-friendly

With the evaluation set up for VTT, the energy consumption for radio is the more predominant than the membership verification in a trust tree. From equation 6.12, the radio requirements can be kept in check by the apt choice of key agreement algorithm. From equation 6.9, the energy consumption for membership verification in a tree depends on the depth of the tree and membership algorithm. Notice in Figure 6.11, the energy consumption for membership verification with the CASL method surpasses the energy consumed by the radio for large trees (depth $n \geq 10$) with 1024 members or more. The order of energy consumption, however, remains marginal in comparison to PKI.

The VTT heap requirements are trivial in comparison to PKI, and constant across various membership methods - the computation of digest operation is not heap intensive, and the message digested is the ECC raw public key of size 91 bytes. In comparison, PKI requires heap intensive computation of ECC group operations, and publick key context per certificate.

The storage requirements, however, increase with the depth of the tree as depicted in Figure 6.12. In light of persistent memory requirements, the energy efficient SACL method may be less lucrative in comparison to CASL. For the very constrained devices in a system with large trust trees, the insecure SACL may be the only viable option to curb the cost of energy and persistent memory.

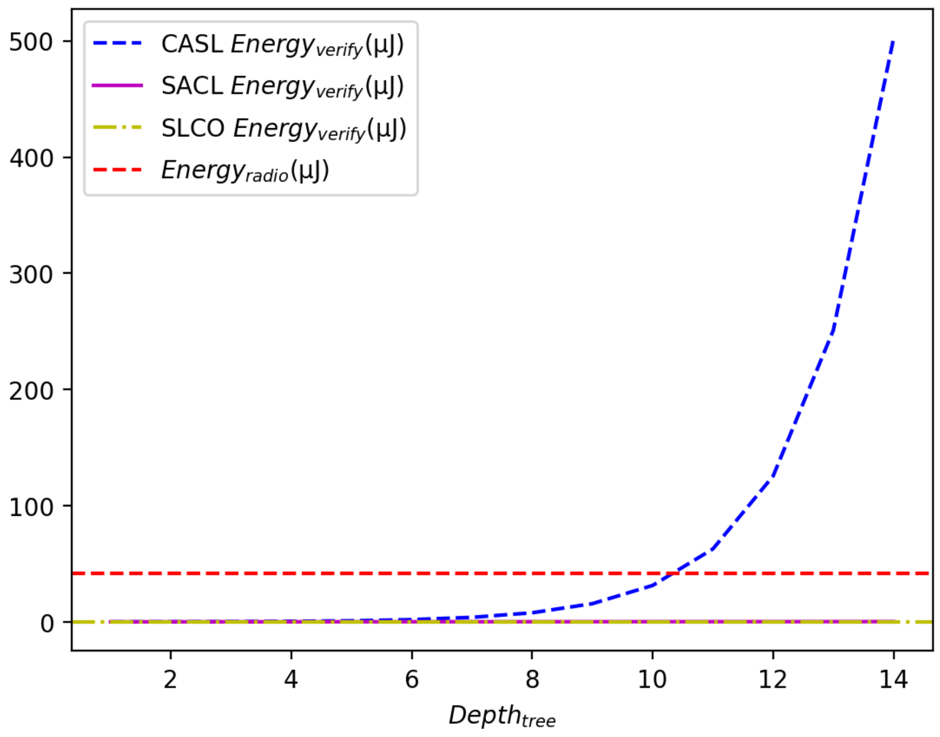


Figure 6.11: The plot of energy consumption as a function of tree depth for VTT shows that the energy consumption for CASL method surpasses energy consumption for radio for very large trees.

6.8.2 The cost of limited trust

VTT, to limit the vulnerability stemming from trust, tightens the trust periphery. Only the entities that interact with each in the system are trusted. Such trust boundaries, as evident in Figure 6.12, come at the cost of persistent memory. With VTT, a constrained device persistently stores one trust tree per service.

To explain with an example, consider again Figure 4.6. The constrained device in access role **'Thing'**, interfaces with three other access roles in the system - **'Data Server', 'Firmware Upgrade server'** and **'User device'**. Therefore, must be provisioned with, trust trees of depths 2, 1, and 1. For these sizes, the persistent memory requirement is less than 1k and hence, nominal. However, if each of these trees were of depth 5, with 32 entities in each trust tree, then, 6kB of persistent memory would be needed.

PKI, by design, allows trust propagation to many entities with trust anchored in

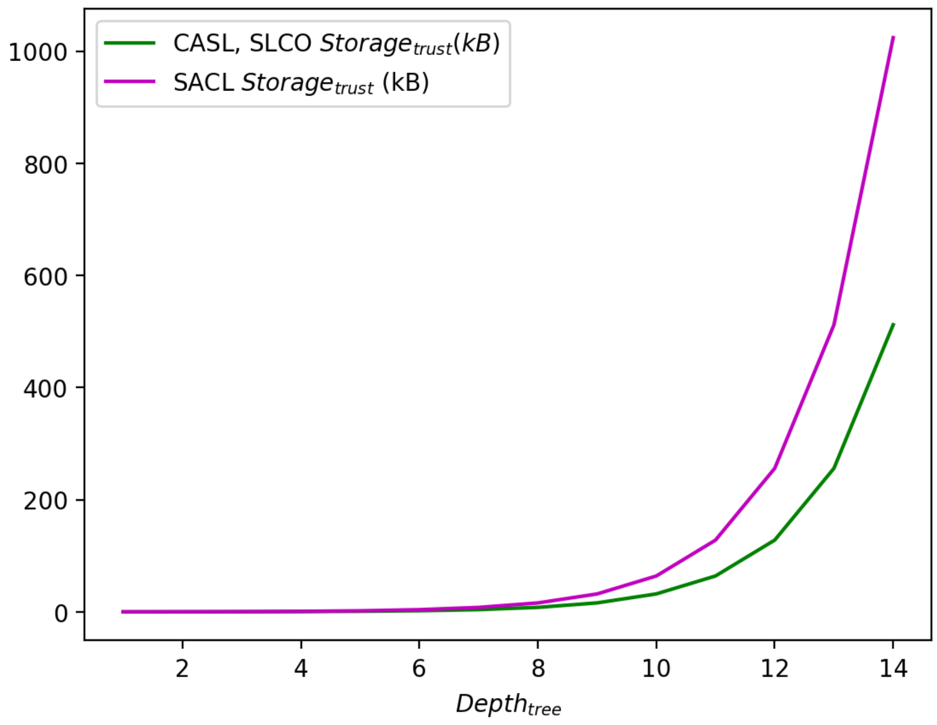


Figure 6.12: The plot of persistent memory requirement as a function of tree depth for VTT shows that SACL may not be a viable method for large trust trees.

a single CA. Therefore, a single CA could be set up for all the services in the system. This way, the persistent memory requirements on a device are minimized. However, recall the problems discussed in Section 2.2 with such a design.

Further, most systems hope to expand their system based on the success of their deployment. For PKI, this is no problem. With VTT, new entities require more persistent memory. Lack of necessary memory provisions in embedded devices may limit the aspirations of systems and hence, businesses.

Chapter 7 Discussion

Until now, the constraints of an embedded device have defined all the narrative. With Section 7.1, by including alternative narratives and otherwise ignored considerations, we demonstrate that one-sided narratives can never lead to a secure, and usable solution. In Section 7.2, we discuss the open addressed topics. Also, based on the findings in Section 6.8, concrete deployment recommendations based on results are made to make PKI more constrained device-friendly.

In this chapter, we see that VTT leaves many aspects open and hence not considered ready for deployment. Hence, we offer no deployment recommendations for VTT.

7.1 Towards a holistic view

If PKI based IoT solutions are guilty of ignoring the weakest links in the system, VTT, and this thesis should be charged with a limited and narrow one-sided perspective to evaluate solutions. We rectify this with the discussion on some aspects that have intentionally been left out to keep the focus on the constraints of an embedded device is discussed here. With this discussion, we advocate the need to take a holistic view is needed to make the right choice for current and future needs of functioning and secure system.

7.1.1 The Service Perspective

The perspective of IoT service has been dismissed throughout the thesis as the servers are considered very resourceful. While this is true, ignoring the server efficiency gain with authentication resourceful may not be and the efficiency gain offered VTT may be a neutralized if

VTT assumes prior knowledge of all services, the number of servers in each services, and identity of each server in the system. In closed, private IoT systems, this may not be a challenge. However, most cloud service providers today provide elastic, load based auto-scaling of the servers used for a service. Existing trusted CA signs certificates for any additional certificates. Therefore, the scaling of servers may not be visible to other components on the system.

VTT, with its tightening of trust, imposes rigidity on services. Such auto-scaling may be a challenge for VTT. Trust trees to group all the servers in the pool to scale to the load requirements can be large. Large trees impose a storage cost on a constrained device.

An alternate practice visible with PKI is the installation of certificates on the load balancer instead of servers. In these cases, all server would appear to be using a common certificate. Here, the certificates do not identify physical servers in the system, rather only a service. Such a model is possible with VTT as well. Not tying server identities in the trust tree to physical servers is an option. This way, many servers could share the common individual identity and therefore, the group identity. Circulating private key across many servers, however, is not recommended. Time-limited security token to gain access to centralized key management services are worth consideration. Such a service would allow key usage without revealing the key to any of the servers.

7.1.2 Key Expiration

The X.509 certificates and the PKI determine the expiry of the key already at validity. Such predetermined expiry can limit the damage caused by key compromise or weakening of cryptographic systems with hitherto unknown vulnerabilities. As discussed in Section 2.3, the use of time for managing the life of each key is cumbersome and susceptible to exploits. Despite this, PKI, by attaching the terms of use for the key, provides a security feature that is not considered or intentionally ignored by VTT.

VTT in Section 4.8, specifies that the members of any tree may update the tree. The update may evict members from the tree. The eviction may be a result of key expiry. However, in case of a weakening of cryptographic system, the entire trust tree may need to be revoked. VTT does not offer any Seppuku[1], nor any means for resurrection with a stronger replacement cryptographic primitive.

Systems that chose PKI for the ability to limit key usage must plan mechanisms to update the root certificates across all entities devices as these too expire. For VTT, multiple trees could collaborate and authorize eviction and replacement of other trees. Management protocols and interfaces for the entities in trust tress would

[1]Japanese voluntary suicide ritual to die with honor, source [Wik19f]

then become necessary. The recommendation in Section 4.7, however, is to limit interaction between the trust trees. Recall in Figure 4.6, the Data Server and the Firmware Upgrade Server interact only with two other access roles and trust trees. Expecting sleepy devices to collaborate to evict servers may be ill-advised. In such cases, master trust tree to manage other trees may be designed. Keep in mind, however, that additional trees come at the cost persistent memory in the system.

In practice, regardless of PKI or VTT, to replace weak security schemes, distribution of new keys, and firmware updates to support the new schemes is necessary. Firmware updates are typically managed remotely; any remote operation would need entity authentication. Diligent planning and foresight are needed to update weakened security primitives. Fixing a key expiration date is one step towards it and not a complete solution.

7.1.3 System Setup and Management

Internet Engineering Task Force (IETF) has taken the initiative to draft the security challenges for IoT devices [GMKS18]. Figure 7.1, borrowed from this draft, describes the life cycle of the device. We do not detail the life cycle states. Rather, discuss some of the challenges that manifest for installation and maintenance of devices.

```
 _Manufactured           _SW update          _Decommissioned
/                       /                   /
|   _Installed          |   _ Application   |   _Removed &
|  /                    |  / reconfigured   |  /  replaced
|  |   _Commissioned    |  |                |  |
|  |  /                 |  |                |  |   _Reownership &
|  |  |    _Application |  |   _Application |  |  / recommissioned
|  |  |   /   running   |  |  / running     |  |  |
|  |  |   |             |  |  |             |  |  |             \\
+##+##+###+#############+##+##+#############+##+##+##############>>>
    \/  \______________/ \/  \_____________/ \___/         time //
    /           /         \          \          \
Bootstrapping  /      Maintenance &   \      Maintenance &
              /      re-bootstrapping  \   re-bootstrapping
        Operational                     Operational
```

Figure 7.1: IoT device life cycle. Source [GMKS18].

Key Generation

Most deployments make an assumption of provisioning credentials into the system at installation. Key generation on the device at first boot, and exporting the public key as the device identity is not a popular norm. Though, this may be more secure as the private key of the device is known only to device. The knowledge of the private key can break the strongest of identity schemes. Despite this, the private key or other entities are generated one entity and provisioned into the constrained device by another.

The reasons for not using such a scheme could be many. Devices may be assumed to be incapable of key generation. Or, the target eco-system determines the identity scheme. The heterogeneous nature of IoT components and devices is a huge challenge to identify the right policies and mechanisms to enforce them.

Root of trust

PKI and VTT, each identify the prerequisites for an entity to participate in the system. The setup in case of PKI includes the installation of root certificates for target services and device credentials. For VTT, peer trust tress must be installed before a device becomes operational. This installation assumes a secure channel.

Rogue root certificates or trust trees may make their way into the device during the commissioning phase. A compromised commissioning phase has been identified as a major challenge for IoT. Any requirements for remote commissioning only make the problem more acute. Intel's Secure Device Onboarding (SDO) [Int18] offers secure remote installation of credentials. The solution comes with a mandate of Enhanced Privacy ID (EPID) based Root of Trust (RoT). Such mandates need a security and efficiency evaluation of their own.

The many actors

Many actors - device manufacturers, cloud service providers, integrators, and end users play a role in the installation and maintenance of any IoT system. There may be many more. Each of these actors contributes to the configuration of the system at different life cycle states on their components, and play a role in keeping the system functional.

Device manufacturers build and manage firmware and, any needed updates for released firmware. The cloud service provider defines the IAM scheme; the integrator installs credentials, and the end users personalize the system and hence the individual devices.

A particular challenge here is that the device manufacturers and cloud service providers want to make their products and services generic and flexible for any IoT use case. As we already stated in Chapter 1, the use cases vary a lot. Therefore, generic solutions imply implementations with many options and features that must be fine-tuned at integration.

The integrators and the end users, on the other hand, expect fine-tuned components that can be quickly pieced together to realize their use case. Many choices and decisions are needed to lock down the needed features from the very many offered by the vendors. The target IoT system integrates many devices from many vendors. This only compounds the decision stress for integrators and end-users. Making security choices is not easy and requires special attention and competence. Appropriate choices may not be clear under these circumstances.

Firmware upgrade

Securities vulnerabilities in the system may arise due to drawbacks in individual implementations, the security model of the eco-system, and, or a combination of both. It may not be possible to address all vulnerabilities with firmware updates; hardware limitations may be one reason for this. Therefore it is important to set clear scope and requirements of the firmware update feature before deployment.

It is critical to isolate any patch update from updates that add new features in the system. It may be undesirable that a patch update wipes or tamper the provisioned credentials and end-user configuration. This would trigger a reconfiguration of the system. On the other hand, updates that require additional features to secure the system may require additional configuration and new credentials too. In this case, it may be acceptable to require a reconfiguration of the device. Remember that seems practical in the scope of one device, but may be quite tedious in context many devices. Cost of replacements may become more practical and affordable.

Firmware upgrades are critical to patch device firmware. In most cases, the upgrades are managed remotely for efficiency. It is unpractical to update devices manually with physical access. Most deployments would disable access ports on the device for security reasons. This leaves remote updates as the only viable option. In such cases, credentials needed to verify firmware before installation need to be provisioned and guarded against tamper. This is an important element for the trusted execution environment. Updating the cryptographic primitive used firmware verification may not be possible. Therefore, some critical and far-sighted deployment choices are needed.

End of life cycle

Awareness of misuse of keys extracted from a discarded product at the end of its life cycle can lead to good security practices like wiping keys and any personal information and configuration from the device. These security practices, however, need planning and support. Identification of the exploitable and critical assets in the device and methods to treat them according to life cycle state are necessary. This may require collaboration between various actors.

7.2 The elephants in the room

Open issues raised but addressed are collected in this section. This to emphasize there is much left to be done to secure IoT, it's a big puzzle.

7.2.1 Detection and Report of Security Exceptions

The problem Section 2.4 points out that humans manually detect and report the misuse of certificates. VTT provides no automated methods to address this issue. Key expiration policies may address one part of the problem.

Key expiration need not be detected in the constrained device. Components, like routers, and cellular networks, that already peek into the network packets for the purpose of fire-walling and value-added services take on an additional role of the certificate-police. These components could curb routing packets on TCP stream that use expired and revoked certificates. With TLS 1.2, the X.509 certificates are visible in plain text. Therefore, such measures may be possible to detect, report, and even act against by observers of the traffic. Such measures, however, may remain limited to cellular network based IoT deployments.

7.2.2 Trust and Access Policy propagation

VTT, in Chapter 4, claims to weave together identity, trust, and access management together. Organization of identities in a trust tree attached to an access role is not adequate of access management. In the absence of mechanisms to formally describe and verify an access policy, and to attach the policy to the right trust tress, the VTT falls short on its claim.

With PKI, the hostname of a service is bound to its certificate, and usage descriptions can be included. The embedded devices must have prior knowledge of service host-names. This knowledge must be provisioned in the device before the operational phase. Moreover, revising service information may be trigger firmware upgrades and reconfiguration.

7.2.3 Security Analysis

Merkle trees are fundamental VTT design. A Merkle tree is known to be as secure as the hash function used to construct it. However, VTT extends the functionality of a Merkle tree. VTT defines autonomous trees that can self-update to expand and shrink themselves. Such extensions need thorough analysis, but none is provided in the current work.

The limited evaluation looks promising for VTT makes it worthy of further scrutiny and formal assessment. PKI is a mature architecture with decades of feedback and many iterations to refine the solution. Any solution proposed as an alternative will need to be thorough and time tested to be considered seriously.

7.3 Deployment recommendations for PKI

PKI is in use despite the challenges. Findings from this work could provide recommendations to strengthen PKI-based deployments. We provide these recommendations here.

From equations 6.3, 6.4, 6.6, and 6.7, the certificate chain depth and the choice of the certificate verification algorithm determine the energy and heap requirements on a constrained device. Any deployment that must use PKI should, therefore, minimize certificate chain depth.

From equation 6.8, the energy needed for signature verification overshadows the energy needed to receive the certificates. Therefore, despite larger key and signature sizes, the RSA signature on certificates may be more energy efficient than ECDSA. Results from [PRRJ03] and [Ime15] show that verify operation for RSA is cheaper than verify operation for ECDSA.

For signature generation, however, ECDSA is more efficient. Therefore, use of different algorithms for certificate verification and entity authentication is recommended. RSA for certificate verification and ECDSA for authentication and key agreement. TLS version 1.3 [Res18] defines the `signature_algorithms_cert` extension in addition to the `signature_algorithms`. This enables any client to specify distinct preferences for certificate verification and key agreement. This recommendation, however, implies that an embedded client shall support both RSA and ECC based cryptography.

Further, limiting the number of trust anchors to a handful of root CAs implies smaller persistent memory requirements on the constrained device. A common CA for the various services is recommended. Even though this may not be practical if the service providers are third parties that do not collaborate.

Finally, the problems with lack of time information, revocation management, and CA compromise do not go away. If PKI must be used, an embedded device capable enough to implement the needed services for time synchronization, and revocation management should be used. All current countermeasures implemented in a browser should be considered for the constrained device as well. Any device not capable of implementing these features should be avoided in the system.

Application layer measures like the OCSP stapling and HTTP pinning serve as an additional layer of security for existing web clients. HTTP pinning is unavailable for IoT devices as the application layer protocol is no longer HTTP. Attempts must be made to ensure that countermeasures are available to all clients regardless of the application layer protocol. Making these available with TLS instead of application protocol-specific may be a reusable and consistent solution for all services on the Internet.

Chapter 8 Conclusion and Future work

8.1 Conclusion

We established the disparity between the terms of use of PKI and IoT requirements and constraints. Such disparity can compromise the security offered with PKI. There is no effort to adapt PKI for IoT. Therefore, IoT must adapt to PKI. IoT devices must implement time synchronization and revocation management services. These services are not optional for PKI.

Our novel approach, VTT tailored for IoT, is efficient in comparison to on an embedded platform. However, we lack security analysis and leave the proposed revocation methods unverified. Such critical yet missing pieces imply that the model needs more effort and analysis. Besides, PKI, with its use in web services, has enjoyed the privilege of feedback. Feedback through use, attacks, analysis, and the many implementation mistakes; this has strengthened PKI that now has a compilation of a comprehensive list of known vulnerabilities and their countermeasures. Therefore, as an alternative to PKI, VTT needs more effort and scrutiny, but with its novel approach shows that practical new designs tailored for emerging use cases are possible.

Thorough use case and security analysis, requirement planning, and foresight are essential to identify the right security model and features for IoT, even for PKI. Remotely distributed firmware update feature may not be capable of retrofitting these elements into the system. Also, as we see in this work, system choices impact the constrained device greatly IoT, short certificate chains, choice of right algorithms may improve the PKI experience for the constrained devices.

8.2 Future Work

As mentioned multiple times earlier, to complement the assessment of the practical application of VTT in IoT, VTT much work and scrutiny. The implementation and analysis of the proposed revocation management for autonomous trees is a starting point for further work on VTT.

For PKI, threat analysis and risk assessment that includes third-party actors like the CA and the limited capability of IoT devices can guide us towards sound design and implementation. Such analysis is missing today.

Work to provide mechanisms for automated detection and report of security exceptions in IoT devices is critical to avoid large-scale exploits. Use of remote services to report such exceptions may be the only option for devices lacking a user interface. However, this brings us back to the problem of entity authentication. Out-of-the-box thinking and innovations are needed to solve these issues. In the meanwhile, observers of network traffic may provide interim relief if ideas in Chapter 7.2.1 are converted to design specification and implementations.

References

[AB17] Mustafa Al-Bassam. SCPKI: A Smart Contract-based PKI and Identity System. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, BCC '17, pages 35–40, New York, NY, USA, 2017. ACM.

[ARM] ARMmbed. mbed TLS – An Open Source, Portable, Easy to Use, Readable and Flexible SSL Library. https://tls.mbed.org/.

[ARP03] SS Al-Riyami and Kg Paterson. Certificateless public key cryptography. *Advances In Cryptology - Asiacrypt 2003*, 2894:452–473, 2003.

[ASA16] Nordic Semiconductor ASA. Nordic Online Power Profiler. https://devzone.nordicsemi.com/, 2016.

[ASA19a] Nordic Semiconductor ASA. nRF Connect for Desktop. https://www.nordicsemi.com, 2016-2019. Rev. 2.6.2.

[ASA19b] Nordic Semiconductor ASA. nRF Power Profiler Kit: User Guide. https://infocenter.nordicsemi.com, 2016-2019. Rev. 2.3.

[ASA19c] Nordic Semiconductor ASA. nRF52840 DK: User Guide. https://infocenter.nordicsemi.com, 2017-2019. Rev. 1.2.

[ASA19d] Nordic Semiconductor ASA. nrf52840 product specification. https://infocenter.nordicsemi.com, 2017-2019.

[ASA19e] Nordic Semiconductor ASA. nRF52840 Product Specification: Current consumption. https://infocenter.nordicsemi.com, 2017-2019.

[ASA19f] Nordic Semiconductor ASA. nrfConnect for Desktop: Source Code. Technical report, GitHub, 2017-2019. https://github.com/NordicSemiconductor/pc-nrfconnect-core.

[ASA19g] Nordic Semiconductor ASA. nRF SDK v15.3.0. https://infocenter.nordicsemi.com, Feb 2019.

[BEK14] 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.

[Ber16a] Olaf Bergmann. TinyDTLS eclipse IoT project proposal. Eclipse.org, 2016.

[Ber16b] Olaf Bergmann. TinyDTLS official page. SourceForge.net, 2016.

[BG14] Andrew Banks and Rahul Gupta. MQTT 3.1.1 Specification, 2014. [Online; accessed 22-May-2019].

[BL12] E Brickell and Jiangtao Li. Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities. *IEEE Transactions on Dependable and Secure Computing*, 9(3):345–360, 2012.

[Blu10] Bluetooth SIG. Bluetooth Core Specification. url="https://www.bluetooth.com/specifications/bluetooth-core-specification/, 2010. [Online; accessed 03-June-2019].

[BR94] M. Bellare and P. Rogaway. Entity authentication and key distribution. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 773:232–249, 1994.

[BSS02] Emmanuel Bresson, Jacques Stern, and Michael Szydlo. Threshold ring signatures and applications to ad-hoc groups. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 465–480, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[Bud16] Budroni, Alessandro and McCusker, Kealan. Milagro TLS Library - White Paper. http://docs.milagro.io/en/tls/milagro-tls-library-white-paper.html, 2016. [Online; accessed 01-June-2019].

[Cer18] Vinton Cerf. Self-authenticating identifiers. *Communications of the ACM*, 61(12):5–5, 2018.

[con15] Wikipedia contributors. Iterative and incremental development, 2015. https://en.wikipedia.org/wiki/Iterative_and_incremental_development.

[Con19a] Bitcoin Contributors. Bitcoin: Multisignature. https://en.bitcoin.it/, 2015-2019.

[con19b] Wikipedia contributors. Certificate Authority, 2019. https://en.wikipedia.org/wiki/Certificate_authority.

[Cor19] Google Cloud IoT Core. Google Cloud: IoT: Using JSON Web Tokens (JWTs), 2018-2019.

[CSF$^+$08] 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.

[Dev17a] "Python Developers". Python ECDSA cryptographic signature library. https://pypi.org/project/ecdsa/, 2001-2017. [Online; accessed 03-June-2019].

[Dev17b] "Python Developers". Python hashlib module - a common interface to many hash functions. https://pypi.org/project/hashlib/, 2001-2017. [Online; accessed 03-June-2019].

[DPP16] Rasmus Dahlberg, Tobias Pulls, and Roel Peeters. Efficient sparse merkle trees: Caching strategies and secure (non-)membership proofs. In *Secure It Systems*, volume 10014, pages 199–215, 2016.

[DZH18] Ashok Kumar Das, Sherali Zeadally, and Debiao He. Taxonomy and analysis of security protocols for Internet of Things. *Future Generation Computer Systems*, 89:110–125, 2018.

[Eas11] D. Eastlake. Transport Layer Security (TLS) Extensions: Extension Definitions. RFC 6066, RFC Editor, January 2011. http://www.rfc-editor.org/rfc/rfc6066.txt.

[EFL$^+$99] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI Certificate Theory. RFC 2693, RFC Editor, September 1999. http://www.rfc-editor.org/rfc/rfc2693.txt.

[Ell99] C. Ellison. SPKI Requirements. RFC 2692, RFC Editor, September 1999.

[EPS15] C. Evans, C. Palmer, and R. Sleevi. Public key pinning extension for http. RFC 7469, RFC Editor, April 2015. http://www.rfc-editor.org/rfc/rfc7469.txt.

[Ern04] Ernie Brickell and Jan Camenisch and Liqun Chen. Direct Anonymous Attestation. https://eprint.iacr.org/2004/205.pdf, 2004. [Online; accessed 4-June-2019].

[Eth15] Ethereum. Ethereum Homepage. url="https://www.ethereum.org/, 2015. [Online; accessed 03-June-2019].

[FA18] Paul Fremantle and Benjamin Aziz. Cloud-based federated identity for the Internet of Things. *Annals of Telecommunications*, 73(7):415–427, 2018.

[FGM$^+$99] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2616, RFC Editor, June 1999. http://www.rfc-editor.org/rfc/rfc2616.txt.

[FS87] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 263, pages 186–194. Springer Verlag, 1987.

[GMKS18] Oscar Garcia-Morchon, Sandeep Kumar, and Mohit Sethi. State-of-the-art and challenges for the internet of things security. Internet-Draft draft-irtf-t2trg-iot-seccons-16, IETF Secretariat, December 2018. http://www.ietf.org/Internet-drafts/draft-irtf-t2trg-iot-seccons-16.txt.

[Gro04] IEEE 802.11 Working Group. IEEE Standard for information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements-Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications: Amendment 6: Medium Access Control (MAC) Security Enhancements. *IEEE Std 802.11i-2004*, pages 1–190, July 2004.

[Gro17] Thread Group. *Thread Specification.* Thread Group, Feb 2017. Rev. 1.1.1.

[Guo12] Vector Guo. C implementation for merkle-tree use md5 as hash function, 2012. https://github.com/weichaoguo/Merkle-tree.

[Har12] D. Hardt. The oauth 2.0 authorization framework. RFC 6749, RFC Editor, October 2012. http://www.rfc-editor.org/rfc/rfc6749.txt.

[Har15] D. Harkins. Dragonfly Key Exchange. RFC 7664, RFC Editor, November 2015.

[Hol19] Arm Holdings. GNU Embedded Toolchain for Arm. https://developer.arm.com, 2012-2019.

[HW15] Daniel Hein and Johannes Winter. A c implementation of a dynamically resizeable binary sha-256 hash tree (merkle tree), 2015. http://www.iaik.tugraz.at/content/research/opensource/merkle_tree/.

[HWF15] Daniel Hein, Johannes Winter, and Andreas Fitzek. Secure device block: A software library that applies cryptographic confidentiality and integrity protection, including data freshness, to arbitrary block device like storage mechanisms, 2015. http://www.iaik.tugraz.at/content/research/opensource/secure_block_device/.

[Hyp19a] Hyperledeger. Hyperledeger Fabric: A Blockchain Platform for the Enterprise. url="https://hyperledger-fabric.readthedocs.io, 2017-2019. [Online; accessed 03-June-2019].

[Hyp19b] Hyperledeger. Hyperledeger Indy. url="https://hyperledger-indy.readthedocs.io/en/latest/index.html, 2017-2019. [Online; accessed 03-June-2019].

[Hyp19c] Hyperledeger Internship Program. Microcontroller (ESP32/ESP8266) IOT Indy Agent. url="https://wiki.hyperledger.org/pages/viewpage.action?pageId=6425293, 2019. [Online; accessed 03-June-2019].

[Ime15] Ali Al Imem. Comparison and evaluation of digital signature schemes employed in ndn network. *CoRR*, abs/1508.00184, 2015.

[Int18] Intel. Intel Secure Device Onboarding. url="https://www.intel.com/content/www/us/en/Internet-of-things/secure-device-onboard.html, 2018. [Online; accessed 28-May-2019].

[JBS15] M. Jones, J. Bradley, and N. Sakimura. Json web token (jwt). RFC 7519, RFC Editor, May 2015. http://www.rfc-editor.org/rfc/rfc7519.txt.

[JCM15] M. Jones, B. Campbell, and C. Mortimore. Json web token (jwt) profile for oauth 2.0 client authentication and authorization grants. RFC 7523, RFC Editor, May 2015.

[Jef16] Jeffrey Voas. NIST Special Publication 800-183: Networks of 'Things'. url= "https://wiki.hyperledger.org/pages/viewpage.action?pageId=6425293, July 2016. [Online; accessed 03-June-2019].

[Jos06] S. Josefsson. The base16, base32, and base64 data encodings. RFC 4648, RFC Editor, October 2006. http://www.rfc-editor.org/rfc/rfc4648.txt.

[Jou00] A Joux. A one round protocol for tripartite diffie-hellman. *Algorithmic Number Theory*, 1838:385–393, 2000.

[JWET18] M. Jones, E. Wahlstroem, S. Erdtman, and H. Tschofenig. Cbor web token (cwt). RFC 8392, RFC Editor, May 2018.

[Kah15] Bryce Kahle. mbed TLS: Support Raw Public Key mode (RFC7250) pull request. Technical report, GitHub, Oct 2015. https://github.com/ARMmbed/mbedtls/pull/336.

[KL07] J.G. Kim and D.-H. Lee. An access control using SPKI certificate in peer-to-peer environment. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4706(2):148–156, 2007.

[Lau14] Ben Laurie. Certificate Transparency. *Commun. ACM*, 57(10):40–46, September 2014.

[Mal15] Kerry Maletsky. RSA vs ECC Comparison for Embedded Systems. http://ww1.microchip.com/, 2015.

[McC16] McCusker, Kealan. Apache Milagro MFA TLS Incubator Repository. https://github.com/apache/incubator-milagro-tls, 2016. [Online; accessed 01-June-2019].

[Mer79] Ralph Merkle. Secrecy, authentication and public key systems/ a certified digital signature. *Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University*, 6 1979.

[Mil06] D. Mills. Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi. RFC 4330, RFC Editor, January 2006. http://www.rfc-editor.org/rfc/rfc4330.txt.

[Miz14] T. Mizrahi. Security requirements of time protocols in packet switched networks. RFC 7384, RFC Editor, October 2014. http://www.rfc-editor.org/rfc/rfc7384.txt.

[MMBK10] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network time protocol version 4: Protocol and algorithms specification. RFC 5905, RFC Editor, June 2010. http://www.rfc-editor.org/rfc/rfc5905.txt.

[MS11] Moez Ben MBarka1 and Julien P. Stern. Observations on certification authority key compromise. In *Public Key Infrastructures, Service and Applications*, volume 6711, pages 178–192, 2011.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 10 2008.

[NK00] M. Nystrom and B. Kaliski. PKCS #10: Certification Request Syntax Specification Version 1.7. RFC 2986, RFC Editor, November 2000.

[oI07] Telecommunication Standardization Sector of ITU. *ITU-T Recommendation X.1035: Password-authenticated key exchange (PAK) protocol.* Telecommunication Standardization Sector of ITU, Feb 2007.

[Ope05] OpenID Foundation. OpenID Connect Core 1.0, 2005.

[OTYO18] Takeshi Okamoto, Raylin Tso, Michitomo Yamaguchi, and Eiji Okamoto. A k-out-of-n Ring Signature with Flexible Participation for Signers. *IACR Cryptology ePrint Archive*, 2018:728, 2018.

[Pet13] Y. Pettersen. The Transport Layer Security (TLS) Multiple Certificate Status Request Extension. RFC 6961, RFC Editor, June 2013. http://www.rfc-editor.org/rfc/rfc6961.txt.

[PNV11] Max Pritikin, Andrew Nourse, and J Vilhuber. Simple certificate enrollment protocol. Internet-Draft draft-nourse-scep-23, IETF Secretariat, September 2011. http://www.ietf.org/Internet-drafts/draft-nourse-scep-23.txt.

[PRRJ03] Nachiketh R. Potlapally, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. Analyzing the Energy Consumption of Security Protocols. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, ISLPED '03, pages 30–35, New York, NY, USA, 2003. ACM.

[PTM$^+$18] A Panarello, N Tapas, G Merlino, F Longo, and A Puliafito. Blockchain and IoT Integration: A Systematic Survey. *Sensors*, 18(8), 2018.

[PYH13] M. Pritikin, P. Yee, and D. Harkins. Enrollment over secure transport. RFC 7030, RFC Editor, October 2013.

[Res18] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.

[Riv99] Ron Rivest. Simple Distributed Security Infrastructure. https://people.csail.mit.edu/rivest/pubs/RL96.slides-rsalabs96.pdf, September 1999.

[SE17] Ludwig Seitz and Samuel Erdtman. Raw-Public-Key and Pre-Shared-Key as OAuth client credentials. Internet-Draft draft-erdtman-ace-rpcc-02, IETF Secretariat, October 2017. http://www.ietf.org/Internet-drafts/draft-erdtman-ace-rpcc-02.txt.

[SGKR06] H. Schulzrinne, V. Gurbani, P. Kyzivat, and J. Rosenberg. Rpid: Rich presence extensions to the presence information data format (pidf). RFC 4480, RFC Editor, July 2006. http://www.rfc-editor.org/rfc/rfc4480.txt.

[SHB14] 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.

[Shi19a] Krishna Shingala. JSON Web Token (JWT) based client authentication in Message Queuing Telemetry Transport (MQTT), 2019.

[Shi19b] Krishna Shingala. mbed TLS: VTT based verification of the raw public key. Technical report, GitHub, Feb 2019. https://github.com/KShingala/mbedtls/tree/raw_public_key_with_merkle_tree_verification.

[Shi19c] Krishna Shingala. particleiot: mbed TLS: RF7250 rebase. Technical report, GitHub, Jan 2019. https://github.com/particle-iot/mbedtls/pull/2.

[Shi19d] Krishna Shingala. Python-based automation scripts for the VTT prototype. Technical report, GitHub, Feb 2019. https://github.com/KShingala/certs/tree/master/scripts.

[Shi19e] Krishna Shingala. VTT and PKI: Evaluation keys, certificates and scripts. Technical report, GitHub, Feb 2019. https://github.com/KShingala/certs.

[Shi19f] Krishna Shingala. VTT prototype: SHA-256 digest based authentication. Technical report, GitHub, Feb 2019. https://github.com/KShingala/mbedtls/tree/raw_key_with_sha256_verification.

[SMA$^+$13] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 internet public key infrastructure online certificate status protocol - ocsp. RFC 6960, RFC Editor, June 2013. http://www.rfc-editor.org/rfc/rfc6960.txt.

[SmTc19] Krishna Shingala and mbed TLS contributors. mbed TLS configuration for the VTT prototype. Technical report, GitHub, Feb 2019. https://github.com/KShingala/mbedtls/blob/ee44afc3d406418de76e088bca2af8bea3d4432e/include/mbedtls/config.h.

[Sov19] Sovrin Alliance. Sovrin. url="https://sovrin.org/, 2017-2019. [Online; accessed 03-June-2019].

[SSW$^+$19] Ludwig Seitz, Goeran Selander, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig. Authentication and authorization for constrained environments (ace) using the oauth 2.0 framework (ace-oauth). Internet-Draft draft-ietf-ace-oauth-authz-24, IETF Secretariat, March 2019. http://www.ietf.org/Internet-drafts/draft-ietf-ace-oauth-authz-24.txt.

[Sta02] Frank Stajano. *Security for Ubiquitous Computing.* John Wiley and Sons, February 2002.

[Tae16] Taejoong Chung and Yabing Liu and David Choffnes and Dave Levin and Bruce M. Maggs and Alan Mislove and Christo Wilson. Measuring and Applying Invalid SSL Certificates: The Silent Majority. http://www.ccs.neu.edu/home/amislove/publications/Invalid-IMC.pdf, 2016. [Online; accessed 05-June-2019].

[Tea05] Python Core Team. Python: A dynamic, open source programming language, 2005.

[The] The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. www.openssl.org.

[vDM13] Nicole S. van Der Meulen. Diginotar: Dissecting the first dutch digital disaster. *Journal of Strategic Security*, 6(2):46–58, 2013.

[Vid05] Alexander G Vidergar. Simple public key infrastructure protocol analysis and design, 2005.

[WGE+05] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz. Energy analysis of public-key cryptography for wireless sensor networks. In *Third IEEE International Conference on Pervasive Computing and Communications*, pages 324–328, March 2005.

[Wik19a] Wikipedia contributors. Bluetooth Low Energy — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Bluetooth_Low_Energy&oldid=895821514, 2019. [Online; accessed 28-May-2019].

[Wik19b] Wikipedia contributors. Comparison of TLS implementations — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 3-June-2019].

[Wik19c] Wikipedia contributors. DigiNotar — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=DigiNotar&oldid=894653086, 2019. [Online; accessed 22-May-2019].

[Wik19d] Wikipedia contributors. Kerckhoffs's principle — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Kerckhoffs%27s_principle&oldid=891035763, 2019. [Online; accessed 6-June-2019].

[Wik19e] Wikipedia contributors. Message authentication code — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Message_authentication_code&oldid=896310957, 2019. [Online; accessed 4-June-2019].

[Wik19f] Wikipedia contributors. Seppuku — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 30-May-2019].

[Wik19g] Wikipedia contributors. SHA-1 — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 23-May-2019].

[Wir06] Wireshark.org. Wireshark official website, 2006.

[WK16] Neal H. Walfield and Werner Koch. TOFU for OpenPGP. In *Proceedings of the 9th European Workshop on System Security*, EuroSec '16, pages 2:1–2:6, New York, NY, USA, 2016. ACM.

[WTG+14] P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, and T. Kivinen. Using raw public keys in transport layer security (tls) and datagram transport layer security (dtls). RFC 7250, RFC Editor, June 2014.

[Yee13] P. Yee. Updates to the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 6818, RFC Editor, January 2013. http://www.rfc-editor.org/rfc/rfc6818.txt.

[YL06] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Authentication Protocol. RFC 4252, RFC Editor, January 2006. http://www.rfc-editor.org/rfc/rfc4252.txt.

[YWN+18] B. Yu, J. Wright, S. Nepal, L. Zhu, J. Liu, and R. Ranjan. IoTChain: Establishing Trust in the Internet of Things Ecosystem Using Blockchain. *IEEE Cloud Computing*, 5(4):12–23, Jul 2018.

Appendix A
Evaluation - Code and Scripts

Script A.1 Python SHA256 Digest

```
 # Prograam to generate SHA256 of test data.
from hashlib import sha256

# The message here could match the m_message in sha256_digest.c
message =  b'\x89' * 512

# Instantiate SHA256 object.
sha256_object = sha256()

# Print the digest in hex format to compare the output from sha256_digest.c
print(sha256_object.digest(message).hex())
```

Script A.2 Python ECDSA Sign and Verify

```
# Generate ECDSA keys and signature on test data.
import ecdsa
from hashlib import sha256

# The message here could match the m_message in ecdsa_verify.c
message =  b'\x89' * 512

# Key Generation.
#   NIST256p = SCEPr1, the curve used for evaluation.
#   SHA256 is used for message hash before signing.
sk = ecdsa.SigningKey.generate(curve=ecdsa.NIST256p, hashfunc=sha256)

# Create verification key.
vk = sk.get_verifying_key()

#Print verification key in hex format.
#   The output is appended and used as test vector for the target.
#   m_raw_public_key in ecdsa_verify.c is set to 0x04 followed by this output.
print(vk.to_string().hex())

# Generate signature for message in DER format.
signature = sk.sign(message, sigencode=ecdsa.util.sigencode_der)

# Print Signature.
#   The output is the test vector for the test application on target.
#   m_signature in ecdsa_verify.c is set to this output.
print(signature.hex())

#Verify signature to gain confidence in the test. True is the expected outcome on
     the screen
print( vk.verify(signature, message, hashfunc=sha256, sigdecode=ecdsa.util.
     sigdecode_der))
```

Source code A.3 SHA256 Digest measurement application - Part 1.

```
/**
 * Copyright (c) 2018 - 2019, Nordic Semiconductor ASA
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification,
 * are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice, this
 *    list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form, except as embedded into a Nordic
 *    Semiconductor ASA integrated circuit in a product or a software update for
 *    such product, must reproduce the above copyright notice, this list of
 *    conditions and the following disclaimer in the documentation and/or other
 *    materials provided with the distribution.
 *
 * 3. Neither the name of Nordic Semiconductor ASA nor the names of its
 *    contributors may be used to endorse or promote products derived from this
 *    software without specific prior written permission.
 *
 * 4. This software, with or without modification, must only be used with a
 *    Nordic Semiconductor ASA integrated circuit.
 *
 * 5. Any software provided in binary form under this license must not be reverse
 *    engineered, decompiled, modified and/or disassembled.
 *
 * THIS SOFTWARE IS PROVIDED BY NORDIC SEMICONDUCTOR ASA "AS IS" AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL NORDIC SEMICONDUCTOR ASA OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include "sdk_common.h"
#include "nrf_assert.h"
#include "app_error.h"
#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"
#include "mem_manager.h"
#include "mbedtls/platform.h"
#include "mbedtls/sha256.h"
#include "boards.h"
#include "nrf.h"
#include "nrf_drv_rtc.h"
#include "nrf_drv_clock.h"

/**< Measurement interval in seconds. */
#define MEASUREMENT_INTERVAL  (10UL)

/**< Declaring an instance of nrf_drv_rtc for RTC0. */
const nrf_drv_rtc_t rtc = NRF_DRV_RTC_INSTANCE(0);

/**< Triggers measurement in thread context. */
static volatile bool m_measure = false;

/** @brief Create SHA256 digest of message. */
void sha256_digest(void)
{
    NRF_LOG_INFO("SHA256␣Message␣Digest");

    uint8_t m_message[512];

    memset (m_message, 0x89, 512);
}

/** @brief Initialize the log module, used for debugging */
static void log_init(void)
{
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_DEFAULT_BACKENDS_INIT();
}
```

Source code A.4 SHA256 Digest measurement application - Part 2.

```
/** @brief RTC Callback triggered every MEASUREMENT_INTERVAL. */
static void rtc_handler(nrf_drv_rtc_int_type_t int_type)
{
    if (int_type == NRF_DRV_RTC_INT_COMPARE0)
    {
        m_measure = true;
    }
}

/** @brief Initialize LFCLK XTAL oscillator. */
static void lfclk_config(void)
{
    ret_code_t err_code = nrf_drv_clock_init();
    APP_ERROR_CHECK(err_code);

    nrf_drv_clock_lfclk_request(NULL);
}

/** @brief Initialize and configure RTC instance. */
static void rtc_config(void)
{
    uint32_t err_code;

    //Initialize RTC instance
    nrf_drv_rtc_config_t config = NRF_DRV_RTC_DEFAULT_CONFIG;
    config.prescaler = 4095;
    err_code = nrf_drv_rtc_init(&rtc, &config, rtc_handler);
    APP_ERROR_CHECK(err_code);

    //Enable tick event & interrupt
    nrf_drv_rtc_tick_enable(&rtc,true);

    //Set compare channel to trigger interrupt after MEASUREMENT_INTERVAL seconds
    err_code = nrf_drv_rtc_cc_set(&rtc,0 , MEASUREMENT_INTERVAL * 8,true);
    APP_ERROR_CHECK(err_code);

    //Power on RTC instance
    nrf_drv_rtc_enable(&rtc);
}

/** @brief Wrapper for calloc to measure dynamic memory usage. */
static void * app_calloc(size_t count, size_t size)
{
    return nrf_calloc(count, size);
}

int main(void)
{
    log_init();
    NRF_LOG_INFO("App␣start!\n");
    lfclk_config();

    rtc_config();

    uint32_t err_code = nrf_mem_init();
    APP_ERROR_CHECK(err_code);

    (void)mbedtls_platform_set_calloc_free(app_calloc, nrf_free);

    while (true)
    {
        if (m_measure)
        {
            m_measure = false;
            sha256_digest();
            err_code = nrf_drv_rtc_cc_set(
                        &rtc,
                        0,
                        nrf_rtc_cc_get(rtc.p_reg, 0) + MEASUREMENT_INTERVAL * 8,
                        true);
            APP_ERROR_CHECK(err_code);
        }
        // Save power until triggered again.
        __SEV();
        __WFE();
        __WFE();
    }
}
```

Source code A.5 ECDSA Verify measurement application - Part 1.

```
/**
 * Copyright (c) 2018 - 2019, Nordic Semiconductor ASA
 *
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without modification,
 * are permitted provided that the following conditions are met:
 *
 * 1. Redistributions of source code must retain the above copyright notice, this
 *    list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form, except as embedded into a Nordic
 *    Semiconductor ASA integrated circuit in a product or a software update for
 *    such product, must reproduce the above copyright notice, this list of
 *    conditions and the following disclaimer in the documentation and/or other
 *    materials provided with the distribution.
 *
 * 3. Neither the name of Nordic Semiconductor ASA nor the names of its
 *    contributors may be used to endorse or promote products derived from this
 *    software without specific prior written permission.
 *
 * 4. This software, with or without modification, must only be used with a
 *    Nordic Semiconductor ASA integrated circuit.
 *
 * 5. Any software provided in binary form under this license must not be reverse
 *    engineered, decompiled, modified and/or disassembled.
 *
 * THIS SOFTWARE IS PROVIDED BY NORDIC SEMICONDUCTOR ASA "AS IS" AND ANY EXPRESS
 * OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
 * OF MERCHANTABILITY, NONINFRINGEMENT, AND FITNESS FOR A PARTICULAR PURPOSE ARE
 * DISCLAIMED. IN NO EVENT SHALL NORDIC SEMICONDUCTOR ASA OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE
 * GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT
 * OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */
#include <stdbool.h>
#include <stdint.h>
#include <string.h>
#include "sdk_common.h"
#include "nrf_assert.h"
#include "app_error.h"
#include "nrf_log.h"
#include "nrf_log_ctrl.h"
#include "nrf_log_default_backends.h"
#include "mem_manager.h"
#include "mbedtls/platform.h"
#include "mbedtls/sha256.h"
#include "mbedtls/pk.h"
#include "mbedtls/ecdsa.h"
#include "boards.h"
#include "nrf.h"
#include "nrf_drv_rtc.h"
#include "nrf_drv_clock.h"

/**< Measurement interval in seconds. */
#define MEASUREMENT_INTERVAL   (10UL)

/**< Declaring an instance of nrf_drv_rtc for RTC0. */
const nrf_drv_rtc_t rtc = NRF_DRV_RTC_INSTANCE(0);

/**< Triggers measurement in thread context. */
static volatile bool m_measure = false;

/** @brief Public key generated using pyecdsa */
static const uint8_t m_raw_public_key[] =
{
    0x04, // Point format.
    0x82, 0xc0, 0x16, 0xb1, 0xd9, 0x4c, 0x3f, 0x67,
    0x14, 0xd5, 0x9c, 0x36, 0x43, 0x66, 0x1c, 0xf4,
    0xee, 0x35, 0xaf, 0xde, 0x73, 0x5c, 0xe6, 0xa6,
    0x18, 0x67, 0x21, 0x94, 0x70, 0xcb, 0x34, 0xf4,
    0x7a, 0xe1, 0xce, 0xa5, 0x61, 0xc0, 0xf0, 0x2d,
    0xb7, 0xfa, 0x7c, 0x98, 0xd7, 0xf5, 0x3e, 0xa1,
    0xb8, 0xaf, 0x44, 0x7a, 0xaf, 0xc4, 0xb1, 0xb3,
    0x71, 0x89, 0x74, 0x0c, 0x37, 0xd6, 0x6d, 0x43

};
```

Source code A.6 ECDSA Verify measurement application - Part 2.

```
/** @brief ECDSA signature in DER format generated using pyecdsa for
 *         512 byte array of 0x89. */
static unsigned char m_signature[] =
{
    0x30, 0x45, 0x02, 0x21, 0x00, 0xf3, 0xec, 0x9a,
    0x5c, 0xf3, 0xdc, 0x31, 0x99, 0xb8, 0x9a, 0x03,
    0x2b, 0xdd, 0xa9, 0x09, 0xdf, 0xc1, 0xd8, 0xc8,
    0x44, 0x39, 0xe5, 0xfc, 0xa6, 0x48, 0xad, 0xe0,
    0x1c, 0xd7, 0xf2, 0x93, 0x6a, 0x02, 0x20, 0x1a,
    0x03, 0xbe, 0x1d, 0x6e, 0xcd, 0x49, 0x56, 0xfb,
    0xd8, 0xf4, 0xd4, 0x69, 0xe0, 0xb6, 0x69, 0x94,
    0x98, 0x19, 0x42, 0x9b, 0xb8, 0x53, 0x77, 0xa4,
    0xb0, 0xb3, 0x2d, 0x5c, 0x60, 0x5a, 0x51
};

/** @brief Verify ECDSA signature. */
void ecdsa_verify(void)
{
    NRF_LOG_INFO("ECDSA␣message␣verification");

    uint8_t m_message[512];

    memset (m_message, 0x89, 512);

    unsigned char hash[32];
    mbedtls_sha256(m_message, sizeof(m_message), hash, 0);

    mbedtls_pk_context context;

    mbedtls_pk_init(&context);

    int retval = mbedtls_pk_setup(&context,
                                  mbedtls_pk_info_from_type(MBEDTLS_PK_ECKEY));
    NRF_LOG_INFO( "␣mbedtls_pk_setup␣returned␣%x\n", retval );

    if (retval == 0)
    {
        retval = mbedtls_ecp_group_load(&mbedtls_pk_ec(context)->grp,
                                        MBEDTLS_ECP_DP_SECP256R1);
        NRF_LOG_INFO( "␣mbedtls_ecp_group_load␣returned␣%x\n", retval );

        if (retval == 0)
        {
            retval = mbedtls_ecp_point_read_binary(&mbedtls_pk_ec(context)->grp,
                                                   &mbedtls_pk_ec(context)->Q,
                                                   m_raw_public_key, 65);
            NRF_LOG_INFO( "␣mbedtls_ecp_point_read_binary␣returned␣%x\n", retval );
            if (retval == 0)
            {
                retval = mbedtls_pk_verify( &context, MBEDTLS_MD_SHA256,
                                            hash, 32,
                                            m_signature, 71 );

                if (retval != 0)
                {
                    NRF_LOG_INFO( "mbedtls_ecdsa_read_signature␣returned␣%x\n",
                                  retval );
                }
                else
                {
                    NRF_LOG_INFO( "Signature␣valid!\n");
                }
                mbedtls_pk_free(&context);
            }
        }
    }
}

/** @brief Initialize the log module, used for debugging. */
static void log_init(void)
{
    ret_code_t err_code = NRF_LOG_INIT(NULL);
    APP_ERROR_CHECK(err_code);

    NRF_LOG_DEFAULT_BACKENDS_INIT();
}
```

Source code A.7 ECDSA Verify measurement application - Part 3.

```c
/** @brief RTC Callback triggered every MEASUREMENT_INTERVAL. */
static void rtc_handler(nrf_drv_rtc_int_type_t int_type)
{
    if (int_type == NRF_DRV_RTC_INT_COMPARE0)
    {
        m_measure = true;
    }
}

/** @brief Initialize LFCLK XTAL oscillator. */
static void lfclk_config(void)
{
    ret_code_t err_code = nrf_drv_clock_init();
    APP_ERROR_CHECK(err_code);

    nrf_drv_clock_lfclk_request(NULL);
}

/** @brief Initialize and configure RTC instance. */
static void rtc_config(void)
{
    uint32_t err_code;

    //Initialize RTC instance
    nrf_drv_rtc_config_t config = NRF_DRV_RTC_DEFAULT_CONFIG;
    config.prescaler = 4095;
    err_code = nrf_drv_rtc_init(&rtc, &config, rtc_handler);
    APP_ERROR_CHECK(err_code);

    //Enable tick event & interrupt
    nrf_drv_rtc_tick_enable(&rtc,true);

    //Set compare channel to trigger interrupt after MEASUREMENT_INTERVAL seconds
    err_code = nrf_drv_rtc_cc_set(&rtc,0 , MEASUREMENT_INTERVAL * 8,true);
    APP_ERROR_CHECK(err_code);

    //Power on RTC instance
    nrf_drv_rtc_enable(&rtc);
}

/** @brief Wrapper for calloc to measure dynamic memory usage. */
static void * app_calloc(size_t count, size_t size)
{
    return nrf_calloc(count, size);
}

int main(void)
{
    log_init();
    NRF_LOG_INFO( "App␣start!\n");
    lfclk_config();

    rtc_config();

    uint32_t err_code = nrf_mem_init();
    APP_ERROR_CHECK(err_code);

    (void)mbedtls_platform_set_calloc_free(app_calloc, nrf_free);

    while (true)
    {
        if (m_measure)
        {
            m_measure = false;
            ecdsa_verify();
            err_code = nrf_drv_rtc_cc_set(
                        &rtc,
                        0,
                        nrf_rtc_cc_get(rtc.p_reg, 0) + MEASUREMENT_INTERVAL * 8,
                        true);
            APP_ERROR_CHECK(err_code);
        }
        // Save power until triggered again.
        __SEV();
        __WFE();
        __WFE();
    }
}
```

Appendix B

Conference Paper - Draft

Vriksh trust model for IoT

Krishna Shingala* and Danilo Gligoroski† and Katina Kralevska† and Torstein Heggebø*
*Nordic Semiconductor ASA, Trondheim, Norway
†Department of Information Security and Communication Technologies, NTNU, Norway,
Email: {krishna.shingala, torstein.heggeboe}@nordicsemi.no, {danilog, katinak}@ntnu.no

***Abstract*—Today, an IoT service, like any web service, uses PKI for service authentication. However, the clients are no longer user-operated browsers. Rather, constrained devices with limited resources. IoT use cases are not identical to that of web services either. IoT has changed the landscape of requirements and constraints; and warrants a fresh treatment. We present a novel approach called the Vriksh: The Tree of Trust for use in IoT. This model aims to provide an embedded device-friendly entity authentication and limit the trust peripheries. With VTT, trust trees group the system identities with equal access rights in the system using Merkle trees. We verify the energy and resource of efficiency VTT on an embedded platform, as expected, SHA-256 based ECDSA based VTT is more efficient than PKI.**

***Index Terms*—IoT, Entity Authentication, IAM, PKI**

I. Introduction

The exponential growth of the number of IoT devices that lack a user interface for management of the user name and password, makes the need for their unique identification a pressing and challenging problem [1]. An industry trend is to invent new authentication schemes for authentication of clients without user input like the use of JWT. However, the authentication of IoT services is identical to the web services - the client must verify the X.509 certificate chain of the server during a TLS handshake. Such an architecture choice overlooks the various assumptions and prerequisites to use PKI for entity authentication. Time information to validate certificates, and revocation management are not optional to implement with PKI. The resources needed to implement such additional services is a burden on embedded devices that already find verification of the X.509 certificate chain is resource intensive. Moreover, we seem to forget that human inspection and action are critical to handling of all X.509 related security exceptions and detection of compromises. Many remotely operated constrained devices in IoT exist to save costs arising from manual inspection and maintenance.

Regardless of the computational powers of the devices in the network, an appropriate Identity and Access Management (IAM) policy must be defined. IAM involves defining and managing the access policies for each resource in the system. Access policies are attached to each entity in the system to determine it's access in the system.

A problem for IoT is the fact that PKI allows spreading a wide umbrella of trust at the expense of poor visibility. The trust anchor does not know the entities endorsed by it. Such an architecture already adversely affects the users of existing web services. The constraints of an IoT device only worsen the situation. A server compromise that goes undetected has large-scale impact. A compromised CA has a larger impact, even if detected. The impact spans across services and systems. Recovering from the effects of such compromise is tedious even with human-aided browsers and operating systems. However, the clients accessing the services have changed, these are no longer user operated browsers. Rather, constrained devices with limited resources. The IoT use case is not identical to that of the web services either.

IoT changes the landscape of assumptions, requirements, and constraints. The changed landscape demands a fresh treatment tailored to effective and efficient.

A. *Objective, Methodology and Our Contribution*

The objective of this paper is to find an entity authentication scheme suitable for the new challenges and requirements of IoT. PKI provides the trust model for all existing IAM in use. Therefore, we first study PKI and any alternatives in light of the constraints of an embedded IoT device. The study aims to establish their effectiveness and efficiency for use in IoT. This study is biased towards constrained devices, as these are the weakest and most vulnerable links in the security chain.

Our contribution is a novel proposal of an alternative trust model named Vriksh. It is designed to make public key cryptography usable in remotely operated IoT devices with no user interface. Additionally, we report on the first prototype implementation of Vriksh and its energy consumption in comparison with the existing PKI IoT authentication solutions.

B. *Structure of the Paper*

In section II, we study existing alternatives to PKI for entity authentication in IoT. The novel scheme, VTT is presented in section III. A proof-of-concept and a limited prototype of VTT is built and presented in section IV. Section V contains some noteworthy observations, conclusions, and possible future work.

II. Existing Alternatives to PKI

Here we survey existing alternate approaches to PKI in light of the IoT use cases. From this survey we seek a way to provide entity authentication with simple key management, and limited trust periphery.

1) Simple Public Key Infrastructure: Simple Public Key Infrastructure (SPKI) [2] advocates the use of local domains and binding an identity, with its authorization in a SPKI certificate. SPKI certificates expressed in LISP format, use Simple Distributed Security Infrastructure (SDSI) [3] names, defined to local scope instead of globally unique identifiers. The SPKI certificates are not therefore public information.

SPKI is promising, however, requires additional services to implement time and revocation lists.

2) OAuth and Federated Identity: The paper [4] presents identity and access management schemes for smart homes based on Personal OAuth servers. The IETF has started an initiative to enable the use of OAuth called the Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth) (OAuth-ACE). The IETF draft [5] defines a profile over the ACE-OAuth framework. As for OAuth, OAuth-ACE relies on TLS or DTLS for securing the communication between servers and clients. All schemes rely on server authentication using on server certificate provided during the TLS handshake. The authentication requires trusted root CA. Therefore, federated identifies with OAuth, despite a new approach is not independent of PKI.

3) Web Of Trust WoT, used in OpenPGP, allows a trusted user to introduce a stranger. Trust from know entity to the unknown is propagated using digital signatures. Physical, face to face verification of key, and thorough inspection of associated attributes is required before signing the certificate of the new user. OpenPGP and WOT are designed to provide equal rights to all participants. In IoT, not all participants are equally capable, and hence, functionality and access rights cannot be the same. IoT benefits from a layered architecture with clear access policies. A device-to-device wireless network automatically extending trust may compromise the security of the entire system.

4) Pairing Based Cryptography PBC pairs elements between two groups to a third to construct a cryptographic system. Three-party one-round DiffieHellman key agreement [6] secure is one of the first use of PBC. IBC, rooted in PBC, has become a research topic of its own. PBC maps are in general expensive, with not many efficient implementations available.

4) Enhanced Privacy ID EPID [7] proposes an identity scheme using Direct Anonymous Attestation (DAA) [8]. Under the strong RSA assumption, EPID and DAA offer remote and anonymous attestation. EPID is deployed with recent Intel-based laptops to identify legitimate products in the field.

Intel's Secure Device Onboarding (SDO) enables provisioning of legitimate IoT devices. EPID embedded in a TPM identifies a device. The solution focuses on authentication of devices to an attestation service. The attestation service can provision cloud-service and user-specific credentials into the device. Authentication of cloud services by the devices is left to the cloud service.

5) Trust On First Use TOFU binds the entity with the public key it presents on the first contact. On subsequent connections, the entity is verified to present the same public key. Secure Shell (SSH) and HPKP uses TOFU. The author of [9], shares the idea of applying TOFU in IoT. The intention is to allow an IoT device to generate its own private and public keys, and register the public key on the first use. The registration of the public key is expected to occur in a secure environment. On registration, the public key is mapped to an IP address, or, similar identifier. The mapping enables a verifier to ensure the device uses its registered public key.

TOFU is vulnerable to many attacks such as: MITM attacks, impersonation attack, imprinting false identities vis DNS poisoning. Since there are no methods for key expiry and revocation, it exposes the system to long lasting exploits.

6) Blockchain Blockchain is the trendy new solution for all problems, including IoT. Authors in [10] propose the use of blockchain to create a *Trustchain* between various actors in the system. The trust, here, means reliability and integrity of data and device ownership. Secure device life cycle management is emphasized to be critical to achieving this goal of trust. The implementation in [10] is based on Hyperledger Fabric. Supply chain and device ownership management are prominent use cases of Hyperledger Fabric. PKI and X.509 certificate provide entity authentication. TLS is optional for inter-node communication.

7) TLS: Raw Public Keys IETF defines the use of raw public keys with TLS [11] with the intent to make public key cryptography constrained device-friendly. CoAP mandates the use of raw public keys to secure communication between a server and a client. In [11] the authors suggest out-of-band mechanisms for trust and access management. Therefore today, there exists no trust model proposed to know if a public key identifies a legitimate authorized entity with certain access rights. No model exists to manage key expiry and revocation.

III. Vriksh: Tree of Trust

We see the existing alternatives to PKI are either not independent of PKI, or are not suitable for IoT. We therefore, propose a new approach, Vriksh: The Tree of Trust (VTT) [1] with the goal of providing an embedded device-friendly use of public key cryptography for identity and access management and to limit the periphery of trust to only the relevant entities in the system. As in SPKI, VTT confines IAM to local domains and identical to Blockchain, uses Merkle Tree as a building block.

A. Setup

All participants in the system, have unique key pair ***(sk, pk)*** of a cryptographic identity algorithm ***C***. Here, ***sk***, is the secret key and the ***pk*** is the public key.

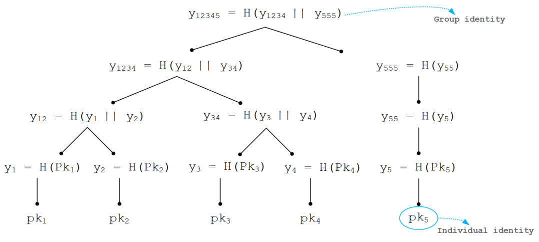


Fig. 1: A sample trust tree with identity y_{12345}.

A Merkle tree, using a cryptographic hash function ***H*** groups together the member public keys ***pk*** as seen in Figure 1. The grouping of identities is based on access roles and rights as explained in section III-E.

[1]In Sanskrit, Vriksh means a tree.

B. Tree size and depth

The size of the tree determines it's depth. The size, here, refers to the number of end or leaf nodes in the tree. For example, the size of the tree in figure 1 is 5. In 1, the depth of the tree is 3. In general, if $\boldsymbol{m}$ is the number of end nodes in the tree, and then the depth of the tree, $\boldsymbol{n}$, is $\lceil log_2 m \rceil$.

C. Tree membership

The membership of an entity requires ensuring that the computed Merkle root or the group identity matches the stored root for the Merkle Tree. Three possible are possible for establishing membership in a Merkle Tree based on the trade-off between computation and storage are described here.

The choice of algorithm for tree membership is local to the verifier and not visible to the prover of membership.

1) Store all, compute less: With Store all, compute less (SACL), complete Merkle tree is stored. Here, complete implies storing hash values of the all end nodes, the intermediate nodes, and the root of the tree.This method is analogous to the authentication path described in [12] for signature scheme. Here, the number of transmissions needed to verify the signature establishes the efficiency of the scheme. With this method, for a tree of depth $\boldsymbol{n}$, the maximum number of hashes $\boldsymbol{c_{max}}$, needed for authentication is given by equation (1) and the maximum storage size, s_{max}, needed in bytes, given by (2).

$$c_{\text{max}} = n + 1 \tag{1}$$

$$s_{\text{max}} = h_s * (2^{(n+1)} - 1) \tag{2}$$

2) Compute all, store less: With CASL, only the leaf nodes and the Merkle Root. Here, only the hashes of the end nodes and the Merkle root hash is stored. Here, we trade more computation power and hence energy for storage. For this method, for a tree of depth $\boldsymbol{n}$, the maximum number of hashes and the maximum storage size are (3) and (4), respectively.

$$c_{\text{max}} = (2^{(n+1)} - 1) \tag{3}$$

$$s_{\text{max}} = h_s * (2^n + 1) \tag{4}$$

3) Store less, compute one: For very constrained devices, further optimization is possible. Here, the verifier can store only the hashes of the end entities and the root of the Merkle tree. During verification, the identity of the entity to be authenticated is hashed. The membership algorithm is reduced to search in look up in the end entities entry of the look-up table, that is, the maximum number of hashes in (5) is always one and the maximum storage size, s_{max}, needed in bytes, given by (6).

$$c_{\text{max}} = 1 \tag{5}$$

$$s_{\text{max}} = h_s * (2^n + 1) \tag{6}$$

This method has limited use for client devices that, before requesting a connection to a server, know which group of servers it wishes to connect to. Therefore, it can limit the search to the database identified the Merkle Tree root. In addition, since the intermediate hash values are not stored, or computed, authentication of subtrees is not possible for such clients.

4) The trade-off: The space-time trade-off may not be significant enough for small trees. However, with larger trees, the cost of storage may be significant to consider the trade-off. The storage requirement doubles with every increment of the depth of the tree regardless of the algorithm. The cost of computing each hash on target can aid the choice of method used for tree membership.

SACL vs. SLCO SACL verifies data-at-rest tree. Hence, may be important for systems that do not have secure storage in hardware. Also, the SLCO can authenticate in subtrees. SLCO, therefore, has limited use and fine-tuned for very constrained devices.

D. Identity and Authentication

VTT requires every participant in the system to a part of a trust tree. Authentication involves verification of both the group identity and the individual identity. Preferably, in that order. Verification of the two identities is detailed below.

1) Group identity. The group identity depicted in Figure 1 is verified by membership proof of the individual identity in the Merkle tree. Section III-C describes methods for membership proof.
2) Individual identity. The identity verification procedure is determined by the Identity Scheme. Any Identity Scheme of choice may be used with VTT. VTT recommends the use of a common scheme for all the members in the tree. The group identity should be verified before the individual identity. As it is typically cheaper to verify the group identity as compared to the individual identity.

Let us assume represents the trust tree in Figure 1 represent a remote service S with five load balanced servers $\boldsymbol{s_1}$, $\boldsymbol{s_2}$, $\boldsymbol{s_3}$, $\boldsymbol{s_4}$, $\boldsymbol{s_5}$. A client, to authenticate the service, must be provisioned with the server's trust tree in 1. At any given time, only one of the servers will communicates with a given client, say $\boldsymbol{s_5}$. The client to authenticate the server must verify:

1) The public key, $\boldsymbol{pk_5}$, is a member of the Merkle tree $\boldsymbol{y_{12345}}$.
2) The server has the secret key corresponding to the public key $\boldsymbol{pk_5}$. The cryptographic algorithm C determines the exact verification algorithm.

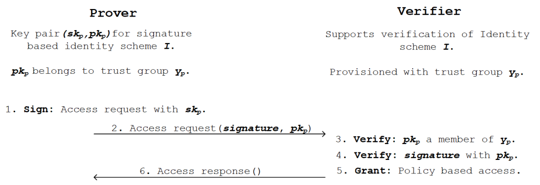

Prover
Verifier
Key pair (sk_p,pk_p) for signature based identity scheme I.
Supports verification of Identity scheme I.
pk_p belongs to trust group y_p.
Provisioned with trust group y_p.
1. Sign: Access request with sk_p.
2. Access request(signature, pk_p)
3. Verify: pk_p a member of y_p.
4. Verify: signature with pk_p.
5. Grant: Policy based access.
6. Access response()

Fig. 2: VTT based normative entity authentication using a signature scheme for identity. The verifier verifies the group identity $\boldsymbol{y_p}$ before verification of the signature to verify $\boldsymbol{pk_p}$.

Figure 2 depicts VTT based entity authentication using a signature-based Identity Scheme. The prover identified by $\boldsymbol{pk_p}$, in trust tree $\boldsymbol{y_p}$ is authenticated on the access request. The membership verification in step 3 is a function of the depth of the Merkle tree, as described in Section III-C. The protocol depicts a successful run of the protocol. The protocol could fail at any stage, and access request will fail.

Most use cases require mutual authentication, i.e. the prover must be sure that it is talking to the right verifier. Mutual authentication is not depicted in Figure 2 for simplicity. In any system, servers and clients have different access rights and therefore, belong to different trust trees. Regardless of the Identity Scheme for the individual identity, the group identity is always verified by the membership of an individual's identity in the trust tree corresponding to the access role.

E. *The garden of trust*

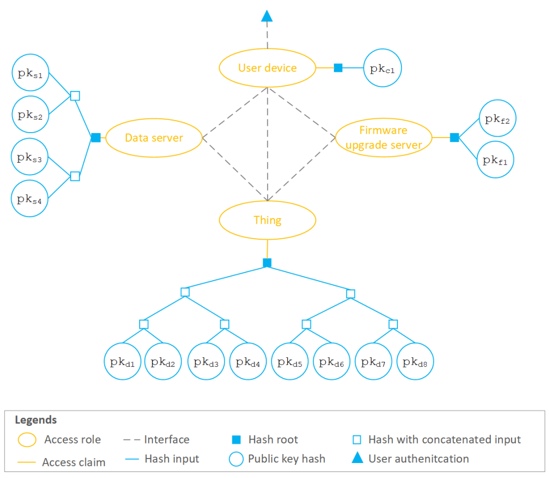


Fig. 3: Trust trees attached to access role to identify entities with certain access right in a hypothetical system.

VTT proposes an access role-based approach for IAM. It is typical for IAM to define many roles for granular access management. Notice that in Figure 3, two server roles are defined. Only the firmware upgrade servers can push new firmware to the devices grouped under the access role 'Things'. Data servers do not enjoy this right. Since the data servers and firmware upgrade servers have different access rights in the system, these are a part of separate trust trees.

With VTT, every entity must be a member of a trust tree. An entity is added to a trust tree corresponding to it's access rights in the system. Each trust tree is attached to an access role. Each member of the trust tree has equal access rights in the system. There can be more than one trust tree attached to an access role; however, only one access role per trust tree.

The distinction between trust tree and access roles enables the system to separate the entities with certain access rights from the access rights. Trust trees determine and manage entities that exercise the access rights by the access role.

In Figure 3, for simplicity, each access role has only one trust tree attached to it. There could be more. In the figure, the interface defines the interaction methods, if any, between access roles. The entities in the trees execute the interaction methods. The sample access management in Figure 3 depicts trust trees of different sizes, age, and each with a role. Hence, IAM with the model implies gardening of these trust trees through their life cycle. As detailed in Section III-G, the trust trees are born, live, and die. The death of a tree may be due to age (expiry) or sickness (revocation).

F. *The trust periphery*

With trust comes vulnerability. VTT aims to limit the vulnerability by limiting the trust expansion to all entities within the system. The interactions and interfaces between access roles determine the trust periphery. Any entity trusts only trust trees attached to other access roles that it interacts with. To clarify with an example, consider the access roles and interfaces in Figure 3. Notice that in the figure, data server and the firmware upgrade server do not have any interfaces defined. This implies no access claims are made by any entities in these access roles to resources in each other's realm.

G. *The living tree*

VTT assumes that at the initial setup of the system, all entities in the system, their identities, access rights, and hence, access roles are known. Therefore, at setup, initial trust trees can be formed. In VTT, each trust tree in the system is of different size and function with a life span, and can be independently updated. The update may be to retire existing members or add new members. We offer two different methods to update a tree based on the capabilities of its members.

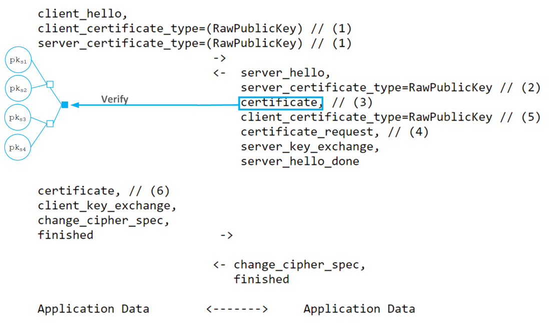
1) *Autonomous trees:* For trees that group resourceful and always available members of the system, like the servers, the tree could be self-managed and hence autonomous. VTT proposes the use of threshold ring signature to update a trust tree. The ring signatures must be generated by the entities of the trust tree. Therefore, a tree can be updated and maintained by the entities within the tree. Much research exists on ring signature and threshold signature. Authors in [13] have proposed the use of a ring signature for the update problem.

2) *Non-autonomous trees:* The threshold based scheme is not practical for sleepy constrained devices. Getting all devices to collaborate within a time window may be a challenge. User-initiated updates either via user web portals, and/or other dedicated management devices, may be a suitable approach to update trust trees of constrained devices. In this case, the tree is updated by other entities, and hence, such trees are considered to be non-autonomous. Any updates to the tree, however, should be verifiable. Here too, threshold signatures may be used.

H. *Use of VTT with TLS*

To demonstrate practical use of VTT, we integrate VTT with TLS. We use of raw public key as individual identity of all entities in the system grouped based on access roles. The group identity would then be the Merkle root of the tree built with the raw public keys of the members of the tree.

In Figure 4, the client first verifies the raw public key of the server to be a member of the server trust tree. On successful verification, it proceeds with the handshake according to the



```
client_hello,
client_certificate_type=(RawPublicKey) // (1)
server_certificate_type=(RawPublicKey) // (1)
                                 ->
                                 <-   server_hello,
                                      server_certificate_type=RawPublicKey // (2)
                                      certificate, // (3)
                                      client_certificate_type=RawPublicKey // (5)
                                      certificate_request, // (4)
                                      server_key_exchange,
                                      server_hello_done

certificate, // (6)
client_key_exchange,
change_cipher_spec,
finished                          ->

                                 <- change_cipher_spec,
                                    finished

Application Data          <------->     Application Data
```

Fig. 4: VTT based server authentication in TLS using raw public key. The handshake exchanged is copied from [11] with VTT specific depictions added.

methods already defined for the cipher with TLS. Therefore, to use VTT with TLS, no new methods or alterations are needed. The server can similarly authenticate the client using the VTT and raw public keys. Alternatively, JWT based or other eco-system specific schemes may be used.

IV. Prototype and Results

We compare PKI operations that use ECDSA signatures and SHA-256 digest operations with VTT. Both models are implemented on nRF52840 SoC with a port of mbed TLS library. Dynamic memory and persistent storage requirements are also compared. For energy consumption of radio, we combine the amount of data exchanged on air with a model based estimate for energy consumption for BLE on for nRF52840 SoC.

A. *Set-Up*

For PKI, we create two certificate chains. A **Long Certificate Chain** of X.509 certificate chain of depth eight, and, a **Short Certificate Chain** of depth 2. The choice of certificate chains is based on observation of industry practices. In particular, the choice is influenced by two major cloud service providers for IoT, Google, and Amazon. Google uses a long certificate chain while Amazon uses a shorter one.

For VTT, we construct a trust tree with the raw public keys of eight evaluation servers. The client is pre-provisioned with this trust tree and allows TLS connections only from the members of the tree.

B. *VTT Evaluation*

We evaluate VTT against the two goals defines for in section III. We compare energy consumption on radio, and the energy consumption for authentication along with dynamic and persistent memory requirements on the target embedded device.

We find that the energy required for ECDSA verify is equivalent to the energy needed to receive 87 kB on BLE radio at 2.0 $Mbps$. Further, it comparable to the energy needed to compute 2.5 million SHA-256 digests on 512 B messages. The results in Table I are consistent with the findings in [14] on an 8-bit processor - energy consumption wise, ECC is predominant and marginalizes the energy consumption of the radio. Reference [14] confirms that the SHA operations adding negligibly to energy consumption in comparison to radio. The hash-based VTT is more efficient than PKI.

Many of the findings are specific to the choice of the evaluation parameters used. For example, the persistent memory requirement for VTT increases with the depth of the tree. The choice of the signature verification function, the certificate chain depth and size, the depth of the trust tree, and the choice of radio are all factors that influence the performance and the resource usage. However, based on the findings, some generalizations may be derived. For VTT, the following generalizations apply.

$$\begin{aligned}\mathrm{E}_{\text{verify}} \quad & \perp\!\!\!\perp \mathrm{Algorithm}_{\text{Hash}} \\ & \perp\!\!\!\perp \mathrm{Algorithm}_{\text{TreeMembership}} \\ & \perp\!\!\!\perp \mathrm{Depth}_{\text{TrustTree}}\end{aligned} \tag{7}$$

$$\begin{aligned}\mathrm{Heap}_{\text{verify}} \quad & \perp\!\!\!\perp \mathrm{Algorithm}_{\text{Hash}} \\ & \perp\!\!\!\perp \mathrm{Algorithm}_{\text{TreeMembership}} \\ & \perp\!\!\!\perp \mathrm{Depth}_{\text{TrustTree}}\end{aligned} \tag{8}$$

$$\begin{aligned}\mathrm{Storage}_{\text{Trust}} \quad & \perp\!\!\!\perp \mathrm{Algorithm}_{\text{Hash}} \\ & \perp\!\!\!\perp \mathrm{Algorithm}_{\text{TreeMembership}} \\ & \perp\!\!\!\perp \mathrm{Depth}_{\text{TrustTree}}\end{aligned} \tag{9}$$

$$E_{\text{radio}} \propto KeySize_{\text{SignatureAlgorithm}} \tag{10}$$

$$Heap_{\text{radio}} \propto KeySize_{\text{SignatureAlgorithm}} \tag{11}$$

$$E_{\text{radio}} >> E_{\text{verify}} \tag{12}$$

Note however that for trees of depth larger than ten, the energy needed for verification surpasses the energy needed to receive the raw public key.

C. *Embedded device-friendly*

For test values of VTT, energy consumption for radio is the most predominant and not the verification itself. From equation 6.12, the radio requirements can be kept in check by the apt choice of key agreement algorithm. From equation 6.9, the energy consumption for verification depends on the depth of the trust tree and membership algorithm. In Figure 5, the energy requirements for CASL method surpasses the energy consumed by the radio for large trees. The order of energy consumption, however, remains marginal in comparison to PKI.

The VTT heap requirements are trivial in comparison to PKI, and constant across various membership methods. The storage requirements, however, increase with the depth of the tree as depicted in Figure 6.

D. *The cost of limited trust*

VTT, to limit the vulnerability, tightens the trust periphery. Trust is extended only between entities that interact with each other in the system. Such trust boundaries, as evident in Figure 6, come at the cost of persistent memory. For example, a device that communicates to three services would need 6kB to store one tree of depth five each per service.

V. Conclusions

The proposed Vriksh: The Tree of Trust (VTT) tailors an access based solution for IoT using existing cryptographic primitives. In comparison to the traditional PKI for IoT, the energy savings offered by VTT is considerable on an embedded platform. However, the proposed revocation methods unverified and no security analysis is included. Also, as a newly proposed model, independent security analysis in unavailable. Moreover, PKI with its use in web services, has enjoyed the privilege of the users feedback for many years. Feedback through use, attacks, analysis, and the many implementation

Performance/ Authentication	**PKI**		**VTT**		
	Long	**Short**	**SLCO**	**SACL**	**CASL**
$\mathbf{E_{verify}}$ (J)	0.366	0.081	15.324×10^{-9}	61.296×10^{-9}	229.863×10^{-9}
$\mathbf{Heap_{parse}}$ (B)	12600	2800	400	400	400
$\mathbf{Heap_{verify}}$ (B)	7200	7200	500	500	500
$\mathbf{Storage_{trust}}$ (B)	476	476	480	480	288
$\mathbf{Data_{rx}}$ (B)	3727	927	91	91	91
$\mathbf{E_{radio}}$ J	1.705×10^{-3}	0.428×10^{-3}	41.632×10^{-6}	41.632×10^{-6}	41.632×10^{-6}

TABLE I: PKI vs. VTT: Evaluation Summary. VTT fairs better than PKI for the chosen evaluation parameters on most counts.

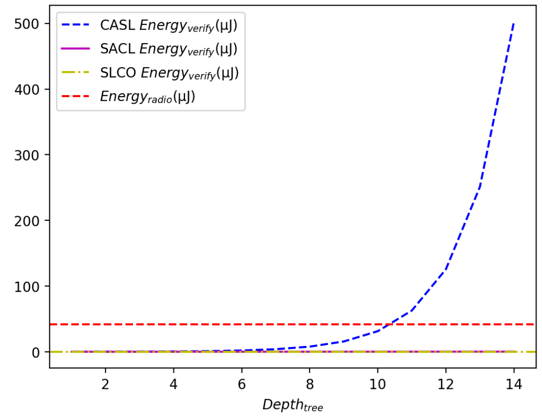


Fig. 5: Energy consumption as a function of tree depth for VTT. The energy consumption for CASL method surpasses energy consumption for radio for very large trees.

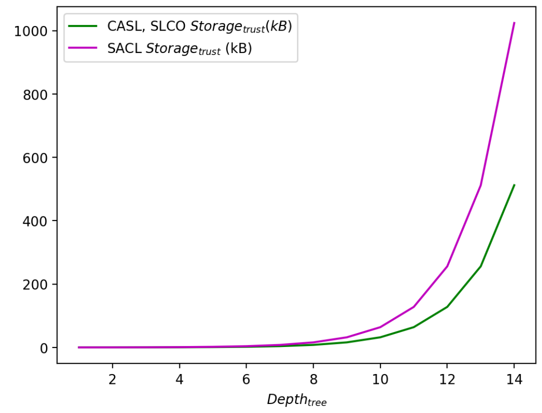


Fig. 6: Persistent memory as a function of tree depth.

mistakes; this has strengthened PKI that now has a compilation of a comprehensive list of known vulnerabilities and their countermeasures. Therefore, as an alternative to PKI, VTT needs more effort and scrutiny, but with its novel approach shows that practical new designs tailored for emerging use cases are possible.

References

[1] G. M. Lee, J. Park, N. Kong, and N. Crespi, "The internet of things : concept and problem statement : 01," Dépt. Réseaux et Service Multimédia Mobiles ... , Research Report, Mar. 2011. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00634616
[2] C. Ellison, "SPKI Requirements," Internet Requests for Comments, RFC Editor, RFC 2692, September 1999.
[3] R. Rivest, "Simple Distributed Security Infrastructure," https://people.csail.mit.edu/rivest/pubs/RL96.slides-rsalabs96.pdf, September 1999.
[4] S. W. Jung and S. Jung, "Personal OAuth authorization server and push OAuth for Internet of Things," *International Journal of Distributed Sensor Networks*, vol. 13, no. 6, p. 1550147717712627, 2017. [Online]. Available: https://doi.org/10.1177/1550147717712627
[5] L. Seitz, G. Selander, E. Wahlstroem, S. Erdtman, and H. Tschofenig, "Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-ace-oauth-authz-24, March 2019. [Online]. Available: http://www.ietf.org/Internet-drafts/draft-ietf-ace-oauth-authz-24.txt
[6] A. Joux, "A one round protocol for tripartite diffie-hellman," *Algorithmic Number Theory*, vol. 1838, pp. 385–393, 2000.
[7] E. Brickell and J. Li, "Enhanced Privacy ID: A Direct Anonymous Attestation Scheme with Enhanced Revocation Capabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 3, pp. 345–360, 2012.
[8] Ernie Brickell and Jan Camenisch and Liqun Chen, "Direct Anonymous Attestation," https://eprint.iacr.org/2004/205.pdf, 2004, [Online; accessed 4-June-2019].
[9] V. Cerf, "Self-authenticating identifiers," *Communications of the ACM*, vol. 61, no. 12, pp. 5–5, 2018.
[10] B. Yu, J. Wright, S. Nepal, L. Zhu, J. Liu, and R. Ranjan, "IoTChain: Establishing Trust in the Internet of Things Ecosystem Using Blockchain," *IEEE Cloud Computing*, vol. 5, no. 4, pp. 12–23, Jul 2018.
[11] P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," Internet Requests for Comments, RFC Editor, RFC 7250, June 2014.
[12] R. Merkle, "Secrecy, authentication and public key systems/ A certified digital signature," *Ph.D. dissertation, Dept. of Electrical Engineering, Stanford University*, 6 1979. [Online]. Available: http://www.Merkle.com/papers/Thesis1979.pdf
[13] T. Okamoto, R. Tso, M. Yamaguchi, and E. Okamoto, "A k-out-of-n Ring Signature with Flexible Participation for Signers," *IACR Cryptology ePrint Archive*, vol. 2018, p. 728, 2018.
[14] A. S. Wander, N. Gura, H. Eberle, V. Gupta, and S. C. Shantz, "Energy analysis of public-key cryptography for wireless sensor networks," in *Third IEEE International Conference on Pervasive Computing and Communications*, March 2005, pp. 324–328.

Krishna Shingala

An alternative to PKI for IoT

NTNU
Norwegian University of
Science and Technology

NORDIC®
SEMICONDUCTOR