POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Decentralized PKI based on blockchain

Supervisor:

Prof. Cataldo Basile

Candidate:

Dario Lanfranco

Company supervisor:

Emiliano Orrù

Academic Year 2021/2022
Torino

To my family

List of Tables

List of Figures

Acronyms

API
Application Programming Interface

AU
Auditor

CA
Certificate Authority

CO
Crypto Officer

CRL
Certificate Revocation List

CU
Crypto User

DBPKI
Decentralized Blockchain-based PKI

DCS
Decentralized Consensus Scale

DLL
Dynamic Link Library

DPBFT

Diffused Practical Byzantine Fault Tolerance

GDPR

General Data Protection Regulation

HSM

Hardware Security Module

IoT

Internet of Things

IoV

Internet of Vehicles

IT

Information Technology

ITU

International Telecommunication Union

MQTT

Message Queue Telemetry Transport

OCSP

Online Certificate Status Protocol

OS

Operating System

PBFT

Practical Byzantine Fault Tolerance

PGP

Pretty Good Privacy

PIV

Public key and Identity Validation

PKCS

Public Key Cryptography Standards

PKI

Public Key Infrastructure

PO

Partition Security Officer

PoC

Proof of Concept

PoS

Proof of Stake

PoW

Proof of Work

QoS

Quality of Service

RNG

Random Number Generator

RoT

Root of Trust

SHA

Secure Hash Algorithm

SO (HSM)

Security Officer

SO (Linux)

Shared Object

TCP

Transmission Control Protocol

TRNG

True Random Number Generator

UDP

User Datagram Protocol

UI

User Interface

URI

Uniform Resource Identifier

URL

Uniform Resource Locator

V2X

Vehicle to Everything

WoT

Web of Trust

Abstract

In the last years, role of technology is becoming more and more important, and an increasing number of IoT (Internet of Things) devices are spread all over the world. The presence of this large number of devices creates an IT security problem, thus communication between them traditionally occurs through the use of asymmetric cryptography, whose keys are distributed by means of a Public Key Infrastructure (PKI). However, traditional PKIs have some downsides as they are defined by a centralized structure, which intrinsically leads to single-point-of-failures and complex revocation mechanisms.

The purpose of this thesis is therefore to create a Proof of Concept of a Public Key Infrastructure that is no longer centralized, but distributed by means of the innovative blockchain technology. The project was carried out with the collaboration of a cybersecurity team of Security Reply S.r.l., by starting from scouting the state-of-the-art of PKI and blockchain and, in particular, from the research carried out by M. Toorani and C. Gehrmann at the Swedish Lund University, who proposed a general model to create a distributed PKI based on blockchain.

The developed and described framework has been designed for a set of nodes that could represent IoT devices, vehicles using V2X (Vehicle to Everything) technology or elements of a smart city. It demonstrates how a decentralized structure can offer advanced security, as it eliminates the weakness of single point of failure and avoids the issuance of fraudulent certificates by centralized Certificate Authorities (CAs). The proposed model has been built following Web of Trust concepts and integrating Hardware Secure Module devices as Roots of Trust.

Acknowledgements

I would like to thank my supervisor Cataldo for his availability and for following me in every phase of this thesis. Acknowledgements also go to Emiliano and his team at Security Reply S.r.l., who provided me with the tools and support necessary for the development of the project. In particular, I thank Andrea and Fabio. Finally, I would also like to thank all the awesome people I met at the Politecnico and who accompanied me on this journey.

Table of Contents

Chapter 1

Introduction

Systems connected through a network, such as IoT (Internet of Things) devices, are now part of everyday life [1]. All these devices require a huge number of interconnections and communication channels, which must be protected and must guarantee properties including authentication, authorization, integrity, privacy, etc. Nowadays, in most cases, all these cybersecurity properties are provided thanks to the use of asymmetric cryptography, which envisages the use of key pairs composed by a private key, that must be kept secret by its owner, and a public key, which is used to exchange messages with other entities. Those keys are commonly distributed by means of traditional PKIs, which are characterized by a hierarchical tree-structure where there are root Certificate Authorities (root CAs) on the top, and other CAs below them.

Each CA issues several digital certificates: signed documents which certify the validity of a certain public key and guarantee the identity of its owner. Therefore, PKIs have long played a central role in ensuring the security of communication channels and in protecting sensitive data within distributed systems. Their use is widespread all over the world and hierarchical systems of Certificate Authorities allow for remarkable performance and security, starting from root CAs.

However, these infrastructures have led to some problems, especially with regard to the issuance of fraudulent certificates by compromised authorities.

There are several causes for which a CA could be compromised and sign fraudulent certificates [2]: ineffective identity validation, as happened for FireFox browser in 2015 [3]; breaches in the system, as in DigiNotar case [4]; insecure cryptographic algorithms (e.g. MD5) [5]; governments' control over CAs [6]; generic malfunctions or insecurities that can be exploited by attackers [7]. For instance, the report of the DigiNotar breach (2011) indicated a total of 531 issued fraudulent certificates, even for famous domains (e.g., *.google.com) [4]. Moreover, CAs' main weakness lies in being a single point of failure and in requiring the use of complex mechanisms for disseminating information on certificates that have been revoked (i.e. CRLs and OCSP). Hence, since in recent years, with the increase of low-consumption devices such as those in IoT or IoV (Internet of Vehicles), which require multiple authentication in a short time, and with the creation of entire smart cities, it is possible to notice that it might be helpful to find alternative methods to manage these public key infrastructures.

The first attempts to try to improve PKIs have been to create log-based PKIs, in which certificates are not considered valid as long as they are not logged in public append-only logs, or blockchain-based ones, in which blockchain replaces the logs as they are used as append-only database of signed transactions [8]. Those attempts tried to improve security of PKIs, but indeed they did not removed the single points of failure, which were CAs. For this reason, between years 2020 and 2021, Mohsen Toorani and Christian Gehrmann, in their research at Swedish Lund University, proposed a conceptual model which aims to completely remove CAs from the infrastructure, by exploiting potential of blockchain technology as a Web of Trust, in which certificate issuance and revocation can take place only after a consensus between reciprocal-trusting entities (i.e. nodes) is successfully reached [8]. These two researchers called their model DBPKI, which stands for Decentralized Blockchain-based PKI.

Although DBPKI model, for simplicity, does not consider some privacy issues, it is a good starting point to be able to create an efficient infrastructure without centralized entities, especially since it plans to use an accumulator in order to avoid the scanning of the entire blockchain to perform required operations. Indeed, a Merkle-tree-based accumulator allows the devices depending on DBPKI to extrapolate useful data or check whether a public key has been revoked or not, by simply verifying the corresponding generated

witness. Each new block in the blockchain contains the last updated version of the accumulator and the witnesses corresponding to the valid keys. Thus, by looking at the accumulator contained in the last block of the blockchain only, a lot of computing time is saved, and all the other previous blocks in the chain are kept as history for auditing purposes only.

Nodes belonging to DBPKI infrastructure are able to perform some operations on the PKI as a leader node, according to their role type, and each operation needs to be approved by the consensus group. The latter is composed by all the other nodes that, working as validators, will validate the new proposal of the leader node. Selected consensus algorithm is PBFT (Practical Byzantine Fault Tolerance), which is carried on following few phases, in which specific messages are exchanged in multicast among validator nodes. If a sufficiently large number of nodes agree on the new proposal, then it will be added to the blockchain, otherwise it will be rejected.

Many other researches about alternatives to traditional PKIs have been published in recent times until today, like [9, 10, 11, 12, 13, 14, 15, 16], but most of them deviate from the idea of WoT that underlies the work of Toorani and Gehrmann. Some of these, however, are very interesting and provide valuable insights to further improve their DBPKI model. In particular, the work of Chai et al. [14], which, by implementing the CyberChain framework for V2X networks, proposes the use of DPBFT (Diffused Practical Byzantine Fault Tolerance) instead of the standard PBFT proposed by Toorani et al., or the research by Koa et al. [16], which implements reward-and-punishment mechanism for the nodes. The latter, in particular, has been used as inspiration for this thesis project, in order to improve the starting DBPKI model. That improvement allows carrying out operations on the basis of validations weighted on trust levels assigned to the individual nodes, according to a custom mechanism for incentivization and disincentivization.

1.1 Objectives

This Master's degree thesis project, carried out at Security Reply S.r.l. consulting company, whose focus is on cybersecurity and personal data protection, had the aim to realize a Proof of Concept of the PKI conceptual

model proposed by Toorani et al. and, after that, to analyze it and look for possible improvements. In more details, goal of this work was to achieve the following objectives:

- Research for state of the art and scouting of possible solutions for blockchain-based PKI;
- Study of main concepts related to HSM (Hardware Secure Module) and PKCS#11 standard;
- Realization of a PoC of a decentralized PKI based on WoT and blockchain, by using a HSM as Root of Trust;
- Analysis of the produced PoC and implementation of effective enhancements based on the use of dynamic trust weights and of a reward-and-punishment mechanism.

1.2 Outline

The remaining parts of this document are organized as follows:

- Chapter 2 - *Background*: within this chapter, a brief description of main concepts needed as basic knowledge background is provided
- Chapter 3 - *State of the art*: in this chapter, current state-of-the-art of alternative solutions to centralized PKI is given
- Chapter 4 - *Solution design*: this chapter introduces conceptual model and design of the proposed solution
- Chapter 5 - *PoC implementation*: this chapter describes the actual implementation of the Proof of Concept
- Chapter 6 - *Results and validation*: this chapter contains the achieved experimental results and the PoC validation, as a comparison with all requirements presented in previous chapters
- Chapter 7 - *Conclusion*: in the end, this last chapter summarizes the achieved results and draws the conclusions of the thesis

Lastly, at the end of this document, there are comparable appendices containing some of the used algorithms and procedures.

Chapter 2

Background

As PKI and asymmetric cryptography are two of the main topics for cybersecurity nowadays, it is very important to understand how they usually work, in order to be able to better compare traditional PKI with the proposed one. On the other hand, blockchain is an innovative technology that has spread quickly in recent years, especially regarding cryptocurrencies, so it is worth investigating to discover its potential.

This chapter introduces basic knowledge of the aforementioned relevant topics and of all other topics concerning this document. In particular, section 2.1 describes Public Key Infrastructures and their functionalities. Section 2.2 refers to the notion of Root of Trust and how it is related to the concept of HSM, the hardware module needed to handle keys distributed by means of a PKI. It also presents PKCS#11, the programming standard used to communicate to HSMs. Section 2.3 describe blockchain technology, while sections 2.3.1, 2.4 and 2.5 introduce the reader to other basic concepts, necessary in understanding how a consensus mechanism should work, what a Web of Trust is made of and what an accumulator is, respectively.

2.1 PKI

A Public Key Infrastructure, often abbreviated as PKI, is an "infrastructure whose services are implemented and delivered using public-key concepts and techniques" [17].

2.1.1 Symmetric and asymmetric cryptographies

Before discussing the infrastructure itself, it is needed to make a premise and understand what asymmetric cryptography is and what is meant by the term "public key". To better understand these significant concepts, a comparison between symmetric and asymmetric cryptographies is presented.

Symmetric cryptography

Imagine two friends, Alice (A) and Bob (B), would like to share some news among them, but they want no one else to understand their messages. They decide to choose a common passphrase which has significance only for them, and use it to modify their messages, in a way that only who knows the secret passphrase will be able to decipher and read them. In this very simple example, A and B, which could be two people, two devices or, more generally, two entities, are hiding the information contained in those messages by means of symmetric cryptography. The secret message that sender A is sending to recipient B, is usually known as *plaintext* when it is in clear, and as *ciphertext* when it is incomprehensible for others. The modification used to make a plaintext become a ciphertext is called *encryption*, while the opposite transformation is the *decryption*. The pre-shared common information, known only by A and B, is the *key* for that communication (sometimes known as *secret key* or *symmetric key*) [18, 19].

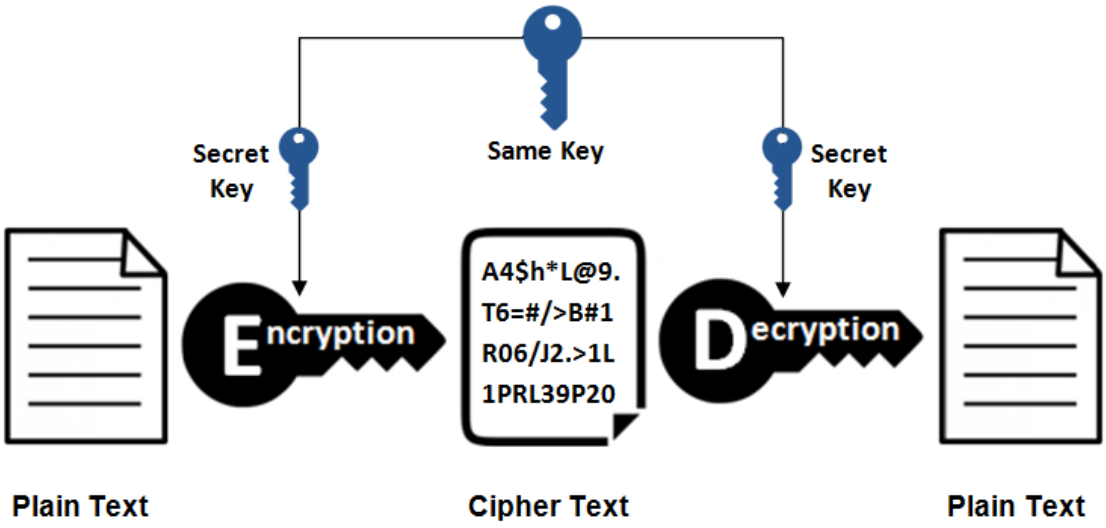


Figure 2.1: Symmetric cryptography model (*Source*)

Asymmetric cryptography

In a different way, instead, asymmetric cryptography allows secure communication between two entities that do not have any pre-shared secret. This is possible by assigning a pair of keys to each entity, instead of using a single key for both of them. A keypair is composed by a *secret key* (*sk*), that must be known only by its owner, and a *public key* (*pk*), which can be known by all external entities, as per definition. The two keys must be mathematically related and it should be infeasible to recover a secret key starting from the corresponding public key[1].

In this way, by means of asymmetric encryption, it is possible for A to send a secret message to B, by simply encrypting it using B's public key. Doing so, B, being the only one to know the corresponding secret key, will be the only one to be able to decrypt the generated ciphertext [18, 19].

Moreover, asymmetric cryptography provides another security functionality:

[1]Infeasible since the amount of time and computing resources necessary to derive a private key would be too large.

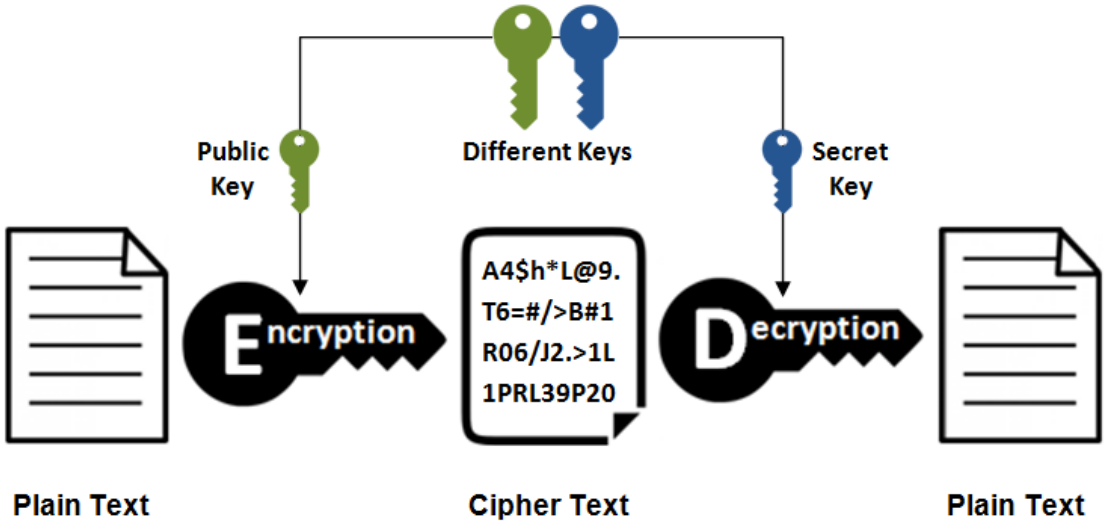


Figure 2.2: Asymmetric cryptography model (*Source*)

the *digital signature*.

Digital signature is the analogue of handwritten signature, since, when Alice signs some document or data, that new signature will bind Alice's identity with those data, and all other entities will be able to verify validity of that signature. Digital signature is even more secure than handwritten one, since it cannot be forged. As modeled in Figure 2.3, signing operation is made by encrypting some data (usually the result of a hash function[2]) with a private key, while verification of a signature, received along with some data, is made by decrypting it using the corresponding public key, and verifying that result obtained from decryption is equal to received data.

2.1.2 Certificate Authority (CA)

As explained above, associations between entities and public keys are needed, especially when these entities do not know each other, which is the most

[2]A hash function is a function mapping data of any size to a fixed-size value.

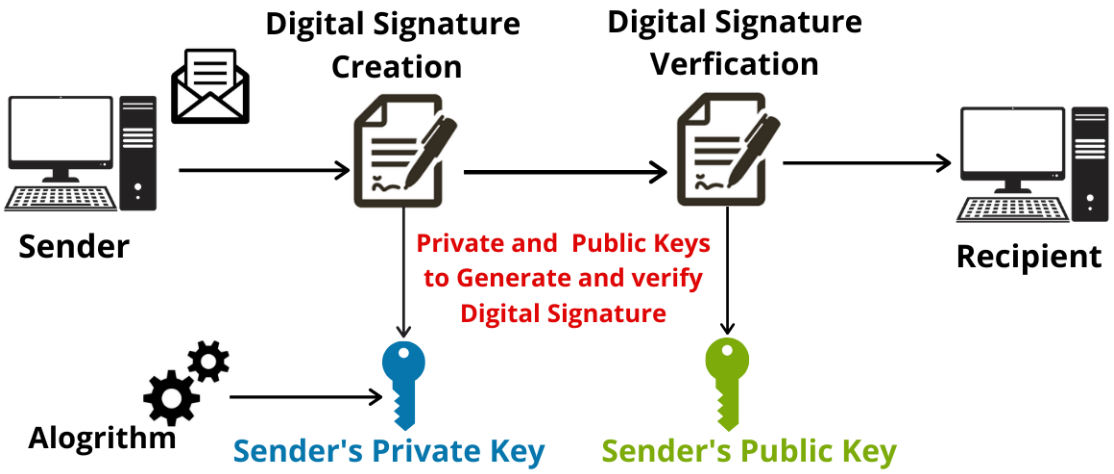


Figure 2.3: Digital signature model (*Source*)

common and generic situation. The way these associations are usually created, is by appointing few trusted authorities, in charge of assigning a public key to the requesting entity. These authorities are named *Certificate Authorities* (*CA*) and form the basis of a Public Key Infrastructure.

A CA assigns a public key to an entity, which could be a person or an organization, by digitally signing a data structure containing the public key and some data about entity's identity. This particular signed data structure is known as *Public Key Certificate*, since it certifies that a public key is related to a certain entity. Usually, these certificates are also called *X.509 certificates*, from the name of the ITU standard used to define their format.

But how can someone trust a certificate issued by some CA? Here comes into play the actual PKI infrastructure. A CA is an entity like any other, so it will have its own key pair with which it will sign certificates and that must be certified by another CA, too. In this way, a tree-structure-like infrastructure is generated by a chain of CAs, one certifying the other. That chain is often called *chain of trust*, and, as Figure 2.4 shows, it has on the top a *root Certificate Authority* (*Root CA*), which certifies itself by its own, so its certificate is self-signed.

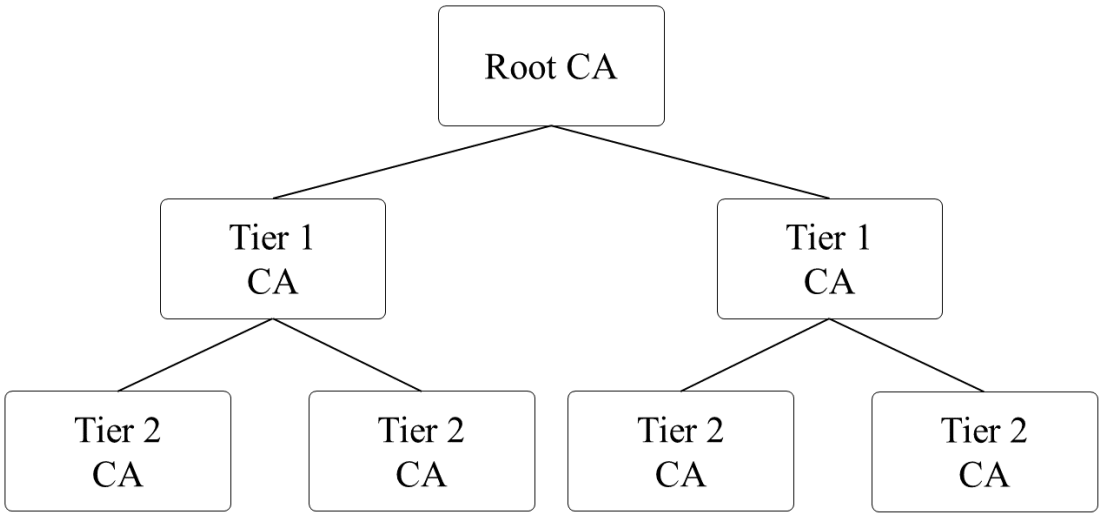


Figure 2.4: Chain of trust example

For instance, if Alice would like to send Bob some data, it can verify if Bob's public key is the right one by looking at which CA issued it. If that CA is trusted by Alice, or by another CA trusted by Alice, then it can assume that public key as valid and use it.

2.1.3 Registration Authority (RA)

Inside a Public Key Infrastructure there are also other significant roles. One of these is the *Registration Authority* (*RA*). It deals with incoming digital certificate requests and has to authenticate identity of the entity who made the request.

2.1.4 Validation Authority (VA) and certificate revocation

Sometimes, for several reasons, it may be necessary to revoke a digital certificate. Those reasons could be, for example, a change in usage of the

certificate is needed, or owner of certificate is concerned or sure that its private key has been compromised [20]. This can happen by sending a *Certificate Revocation Request*.

Thus, another important role in a PKI is played by the *Validation Authority* (*VA*), which is responsible for guaranteeing validity of certificates, by providing a mechanism to check if a certificate is still currently valid, or if it has been revoked before its expiration date. VA role is often held by the CA which issued the certificate. There are different ways to ensure certificate validity:

- *CRL* (*Certificate Revocation List*): CRL is "a list of digital certificates that have been revoked by the issuing CA before their scheduled expiration date and should no longer be trusted" [21]. CRLs are periodically published, so it is possible that a revoked certificate is accepted as valid even if corresponding CRL has not be updated yet.

  CRLs can be retrieved as a response from a CRL Issuer, by making browsers access one of the URLs contained within the certificate that has to be validated. In Figure 2.5, the process of validating a certificate by means of a CRL is shown[3].

  Main disadvantages of CRLs are [23, 24]:

  – Large amount of overhead, since client has to search through the CRL, which can be long (even on the order of a thousand lines);
  – Problems with network resources if CRL is huge;
  – Potentially open attack surface until next CRL update;
  – If CRL download fails, client will trust the certificate by default.

  Distribution Points and Delta-CRLs are two alternative methods, proposed in order to solve overhead caused by the growing of CRL: the former have the purpose to partition a CRL by pointing at different URIs, while the latter consists in incremental CRLs, which contains only the new revocations that have taken place since the last base CRL[4] was

[3]The term "SSL/TLS certificate" is another way of calling the digital certificate.

[4]A base CRL is a standard complete CRL, which is published less often than Delta-CRLs.

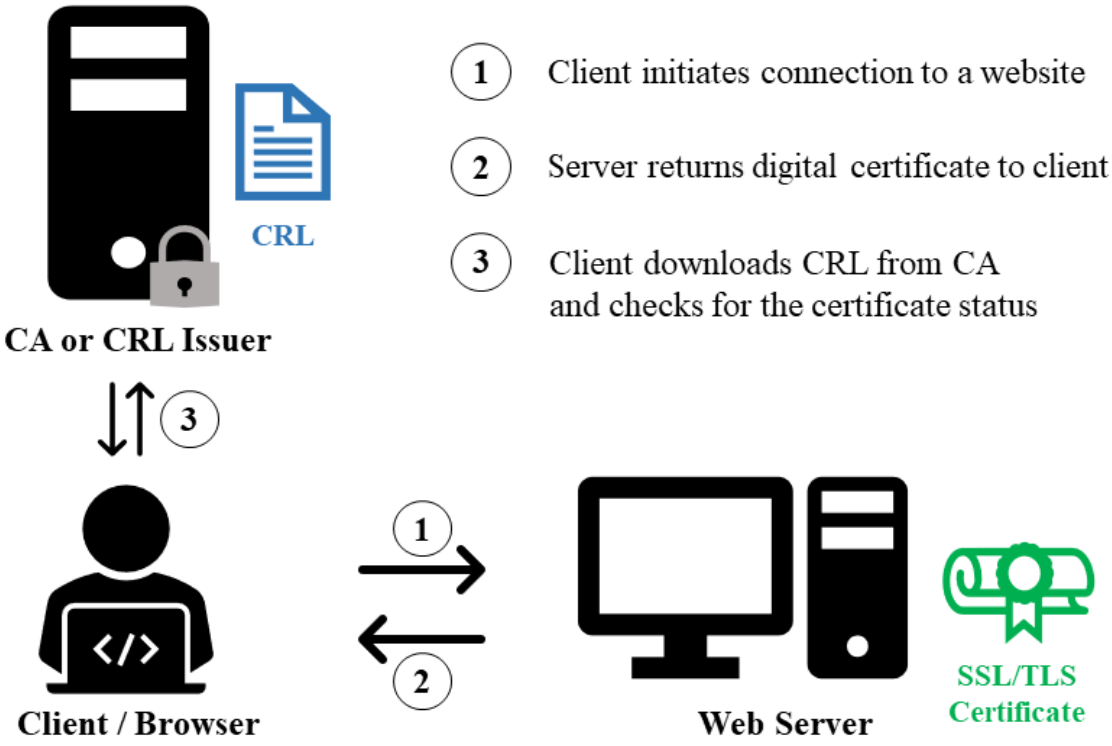


Figure 2.5: Certificate checking process by means of a CRL [22]

published [24]. These solutions are promising, but they do not solve all CRL issues.

- *OCSP* (*Online Certificate Status Protocol*): OCSP was created as an alternative to CRLs. This internet protocol allows an entity to send an OCSP Request to an OCSP Responder, asking if a specific certificate is valid in this particular moment. Usually, OCSP Request is sent by a CA, whose task is to allow a user to verify the validity of a certificate issued by that CA. Figure 2.6 shows an example of a client verifying a certificate by sending an OCSP Request.

  Main disadvantages of OCSP are [23, 25]:

  – Large overhead on OCSP Responder, since an OCSP Request is sent for each certificate;

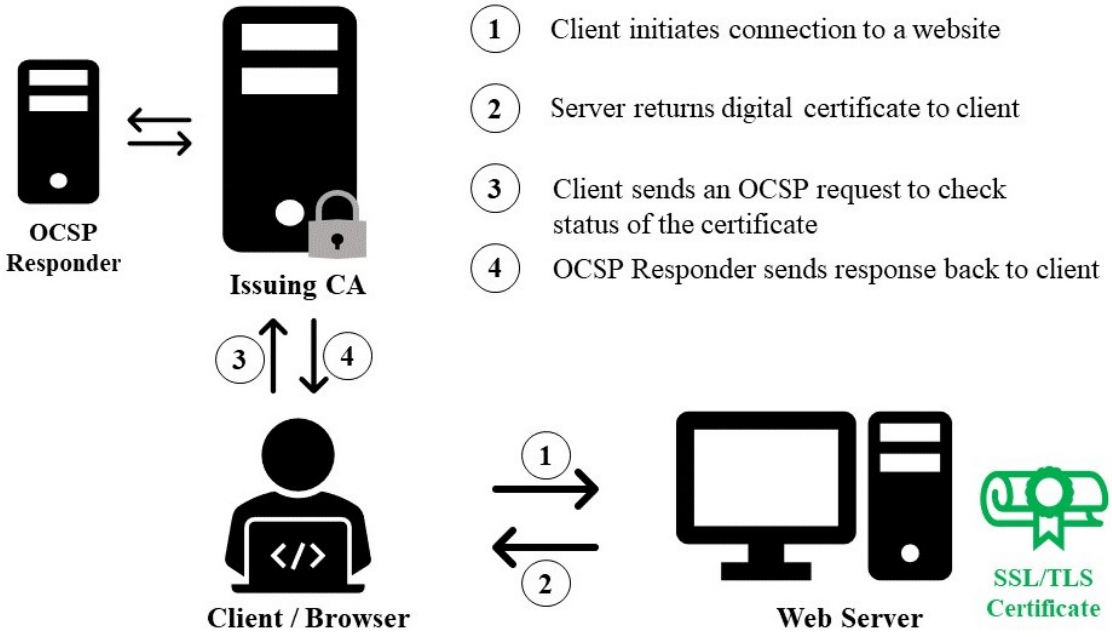


Figure 2.6: Certificate checking process by means of OCSP [22]

- A third party can track user's browsing history, since OCSP Requests for certificates of visited websites are sent out;
- Client could suffer of connection latency due to OCSP queries;
- Most browsers ignore OCSP if the protocol times out.

An enhancement to the standard protocol is the one known as OCSP Stapling. It reduces overhead and improves user privacy by making web servers periodically caching OCSP Responses. Thus, when user visits the corresponding website, it will receive the latter's certificate, along with a "stapled" OCSP Response [22]. However, OCSP Stapling is not widely used, because it is still not supported by many browsers [23].

2.2 Root of Trust

A *Root of Trust* (*RoT*) is the root of some framework that can always be regarded as truthful by all the components of the infrastructure, as Thales Group explains [26]:

> *Root of Trust (RoT) is a source that can always be trusted within a cryptographic system. Because cryptographic security is dependent on keys to encrypt and decrypt data and perform functions such as generating digital signatures and verifying signatures, RoT schemes generally include a hardened hardware module. A principal example is the hardware security module (HSM) which generates and protects keys and performs cryptographic functions within its secure environment.*
>
> *Because this module is for all intents and purposes inaccessible outside the computer ecosystem, that ecosystem can trust the keys and other cryptographic information it receives from the root of trust module to be authentic and authorized.*

Therefore, RoT has the primary role of maintaining a Public Key Infrastructure authentic and secure, by protecting its root keys.

2.2.1 Hardware Security Module (HSM)

Cryptographic devices constitute a family of machines able to handle and store cryptographic objects (e.g. asymmetric keys or digital certificates), and use those objects to perform some cryptographic operations. There are different types of cryptographic devices, like USB-based cryptographic tokens, smart cards and *Hardware Security Modules* (*HSMs*). While smart cards are designed for a single user (i.e. they usually store a single key) and can perform few cryptographic operations, HSMs are very advanced modules able to store thousands of keys and to implement a large amount of cryptographic algorithms, by using cryptographic accelerators [27, 28].

Hardware Secure Modules are tamper-resistant hardware devices able to

Figure 2.7: HSM (*Source*)

generate, manage and store cryptographic keys, encrypt or decrypt data, and perform or verify a digital signature. They are often used as RoTs by CAs in traditional PKIs.

HSMs can be in the form of standalone network-attached appliances, hardware cards that plug into existing network-attached systems, USB-connected backup HSMs, etc.

HSMs provide a layered encryption, made through encrypting multiple times the same object, and fully decrypting it in temporary (volatile) memory only when needed [29]. They also provide tamper protection, which means that when a physically security breach is detected, these intrusion-resistant modules lock or reset themselves, by erasing all stored objects. They have multiple hot-swappable power suppliers (that can be replaced without turning off the system) and suitable cooling systems, which make HSMs very reliable devices. One of the main features of HSMs is the use of a True Random Number Generator (TRNG), which is a nondeterministic Random Number Generator (RNG) based on a specific physical process [30]. TRNGs are used by HSMs in order to generate true random values, which lead to better security services rather than pseudo-random ones, since their guessability is minimal. Furthermore, depending on the network configuration, Backup HSMs can be also used to perform a backup of secure material.

HSMs' storage is often logically divided up in partitions, each one having its own data, policies and access control. The latter allows a set of users to perform operations on HSM, according to their role. Each role provides privileges to perform different actions, such as initialize the HSM, create other users, create or delete partitions, configure policies for the entire HSM, configure policies for a single partition, create or modify cryptographic

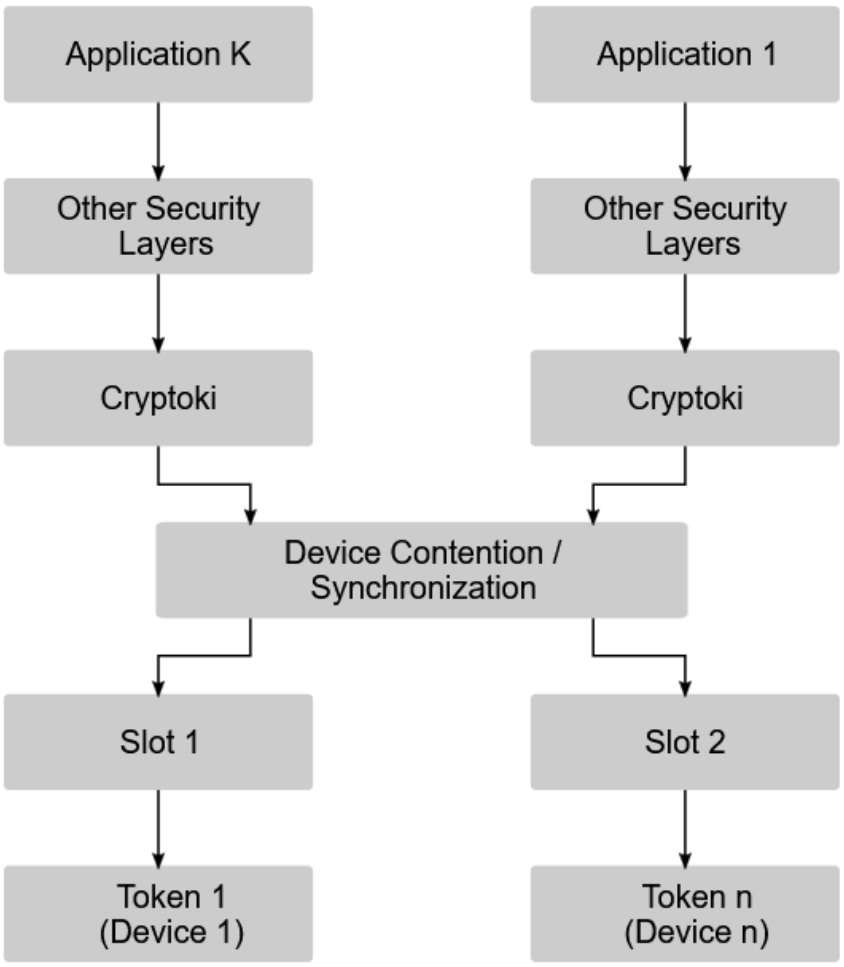


Figure 2.8: PKCS#11 general model [28]

objects, etc.

2.2.2 PKCS#11

PKCS (Public Key Cryptography Standards) is a family of standards, related to public key cryptography, published by RSA Security Inc. PKCS#11, also known as *Cryptoki* is one of these standards, and it provides a cryptographic token interface, so an API (Application Programming Interface) to handle cryptographic objects and use cryptographic algorithms [31].

There are a lot of cryptographic devices from different vendors. If all vendors provide their own proprietary way of accessing their products, the applications working with the latter would be strictly tied to a specific vendor's product. Hence, PKCS#11 provides a standard way of communicating with cryptographic devices, like HSMs, following a object-based approach [32]. This standard provides the API model as multiple C header files, that will be implemented by vendors, and then distributed as DLL (Dynamic Link Library) files for Windows OSes or as SO (Shared Object) files for Linux ones [27]. Indeed, low level programming of cryptographic devices is mainly based on C programming language.

PKCS#11 has its own terminology allowing standardized working on cryptographic devices [27]. As mentioned in Chapter 2.2.1, storage of HSMs can be divided up in logical partitions, that PKCS#11 refers to as *slots*. Inside a single HSM slot there can be stored hundreds of cryptographic objects and, for each slot, a related token is present. In other kinds of devices, like smart cards, only one slot is present. According to this standard, for a HSM, a token is a component which is the correspondent of the smart card reader for the smart card, so it must provide a safe environment for executed applications [33]. Token's purpose is to allow the establishing of a *session*: a logical connection between the slot and the running application, allowing the latter to perform cryptographic operations, by exploiting a certain *mechanism*, which is the way cryptographic algorithms are called in PKCS#11. Access control provided by HSMs expects existence of *users*, each one with its role and related privileges.

Objects stored inside a slot can be public or private. Public objects are inspectable by any user or application, while private ones requires that user is logged into the corresponding token, and that it has enough privileges. Usually, a HSM has a user with the role of Security Officer (SO), whose task is to initialize the HSM itself, and other users with minor roles too, depending on the HSM model. For example, Thales Luna HSMs, deployed inside a secure appliance, expect different types of user roles at different logical levels, as summarized in Table 2.1 [29].

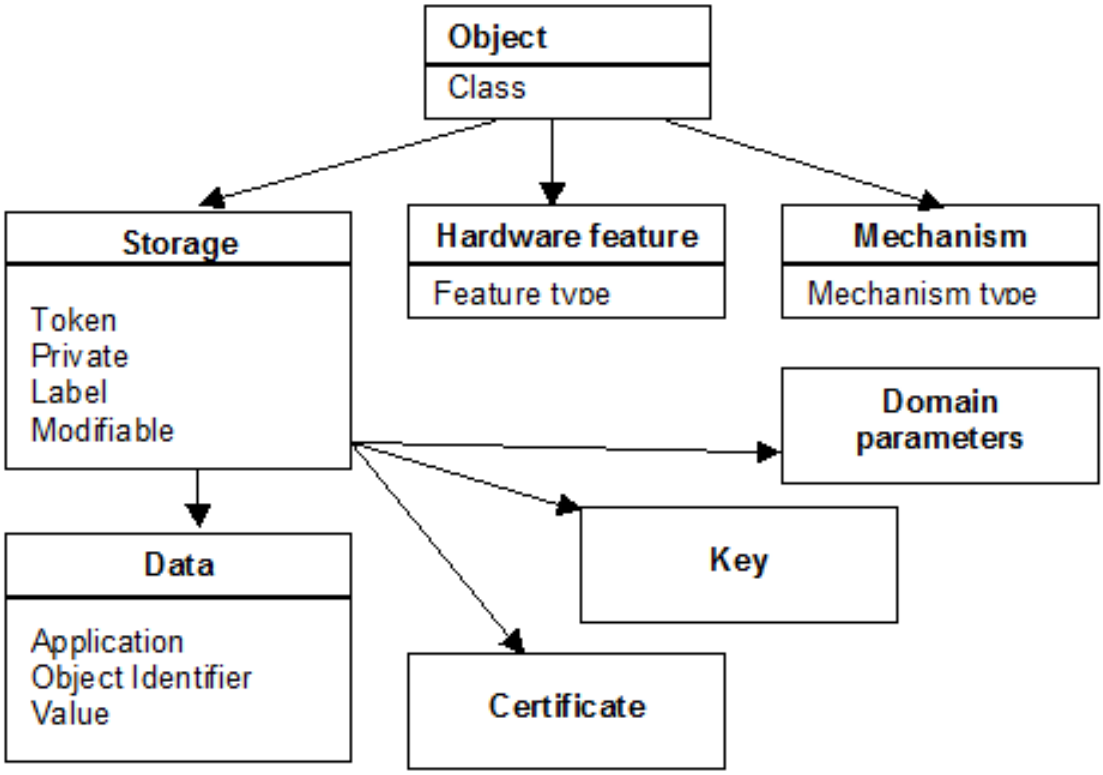


Figure 2.9: PKCS#11 classes of objects and their attributes [32]

Appliance-level roles

Admin has the rights to perform all administrative and configuration tasks on the appliance, and to create custom users and roles. Operator role can perform administrative tasks only. Monitor is able to provide information about the appliance and HSM. Audit role is used for managing HSM audit logging instead.

Appliance-level roles	HSM-level roles	Partition-level roles
• admin • operator • monitor • audit	• HSM Security Officer • Auditor	• Partition Security Officer • Crypto Officer • Crypto User

Table 2.1: Thales Luna HSMs users and roles [29]

HSM-level roles

HSM Security Officer (SO), who must have admin-access level to the appliance, initializes the HSM, manages its partitions and configures global HSM policies. Auditor (AU) manages HSM audit logging, by having audit-level access to the appliance.

Partition-level roles

Partition Security Officer (PO) initializes the partition and can configure its policies. Crypto Officer (CO) role instead, allows to create and modify cryptographic objects within the partition, and to use cryptographic functions, too. Crypto User (CU) can handle public objects only instead.

2.3 Blockchain

A blockchain is a secure digital ledger implemented in a distributed fashion without a central authority, and shared among users (i.e. nodes) of a community. This ledger aims to record users' transactions in a way that makes it not possible to modify them once published on blockchain. During the last decade, blockchain concept has been combined with other technologies

in order to develop cryptocurrencies (e.g. Bitcoin or Ethereum), made up by virtual electronic coins protected through cryptographic mechanisms instead of a central bank [34]. Blockchain technology is very promising, indeed it is spreading in many sectors such as Internet of Things, finance, data provenance, sharing economy and public sectors, too [35].

This technology is called "blockchain" since the main structure can be visualized as a chain of blocks, starting from a *genesis* block. Each of these contains a set of published and immutable transactions, a reference to the previous block (usually a hash) and some other metadata.

Blockchains can be classified into two high-level categories: *permissionless* and *permissioned*. The latter includes all blockchains where read and write operations are allowed only to authorized people or organizations. In permissionless (i.e. public) blockchains instead, anyone can have access without authorization [34].

In Bitcoin blockchain, taken as example since it was the first famous cryptocurrency, all electronic cash information are attached to an address: an alphanumeric string allowing to link balance of a user to its identity. Thus, in the cryptocurrency case, transactions describe exchange of cryptocurrencies between users of the blockchain.

Sometimes, blockchains are even used to store *smart contracts*, which consist of a set of trustworthy data and code, that can be run to perform some operations [34]. In a smart contract, agreement clauses written in the code are automatically executed when specific conditions are met, thus enabling the contractual terms of an agreement to be enforced without the intervention of a trusted third party. Automatic agreements available by means of smart contracts could be, for instance, contracts among buyers and a supplier, and related payment procedures by means of cryptocurrency [35].

2.3.1 Consensus models

A node able to publish a new block in the blockchain is usually called *publishing node*, and the publishing operation is made by reaching an agreement among all – or almost all – nodes, known as *consensus*.

Proof of Work

One of the most used consensus models in cryptocurrencies is the *Proof of Work* (*PoW*), which allows a node to publish a new block if it is the first one to solve a computationally intensive puzzle. The investing of computational power to solve the puzzle is generally known as *mining*. A common example of PoW puzzle requires to make the hash value of the block smaller than a target number. Usually, puzzle difficulty increases over time, in order to make more difficult to be able to publish new blocks [34].

Proof of Stake

Another consensus mechanism is the so called *Proof of Stake* (*PoS*). A stake is a sum of money bet on the outcome of a risky game or something similar. In the context of blockchain, the stake is represented by cryptocurrencies invested on the system. These kind of blockchain networks do not require great computing power, since they use the amount of stake a user owns as a decisive factor for publishing new blocks [34].

This model constitutes an energy-saving alternative to PoW, but the publishing node selection results to be quite unbalanced, since few very rich users would be dominant even in a huge network [36].

Practical Byzantine Fault Tolerance

L. Lamport, taking a cue from well-known problems such as that of philosophers or that of readers and writers, presented the "Byzantine generals problem". According to this problem, some Byzantine[5] generals are leading the Byzantine army to the siege of an enemy city attacking from different fronts, so they can communicate only by messengers. Solving the Byzantine generals problem consists in guaranteeing that those generals can agree unanimously on the same action plan, and that, if few generals are traitors, then their plan proposals cannot prevail on honest generals' ones [37, 38].

[5]The choice fell on Byzantine generals so as not to offend any nation existing today.

Byzantine faults are therefore the ones which happen when, in a distributed decision-making system, some of the system's nodes fail or have malicious intentions. Since software complexity is increasing year after year, the amount of possible software failures and malicious attacks is increasing, too. For these reasons, M. Castro and B. Liskov proposed an algorithm able to tolerate Byzantine faults: the *Practical Byzantine Fault Tolerance* (*PBFT*) consensus mechanism [39].

PBFT model expects a client instance C to interact with one of the nodes in the consensus group, trying to make all other nodes accepting a message or – in the context of a blockchain – a transaction. This mechanism is made up of several stages [39, 8]:

1. **Initialization request**: Consensus mechanism initialization is made by the creation of a request by client C, sent to one of the elements in the consensus group, which will be labeled as *primary* until the end of consensus mechanism.

2. **Pre-prepare**: In Pre-prepare phase, the primary node sends a signed pre-prepare message as a multicast[6] to all other nodes in consensus group. Pre-prepare message is used to allow all nodes to have the same view of client request. At the end of this phase, all nodes will be in the *pre-prepared* state.

3. **Prepare**: In Prepare phase, all nodes that have previously received a pre-prepare message, validate the latter and share a signed prepare message as multicast, saying if current request made by C is valid or not, according to them. At the end of this phase, all nodes will be in the *prepared* state.

4. **Commit**: In Commit phase, all nodes in consensus group (even the primary one), verify all received prepare messages. If the latter indicate that enough nodes have evaluated the request as valid, they multicast a positive commit message (a negative one otherwise). At the end of this phase, all nodes will be in the committed phase, and the result of current client request will be committed to their logs.

[6] A communication method allowing a source to send a message to many receivers, which belong to the same multicast group.

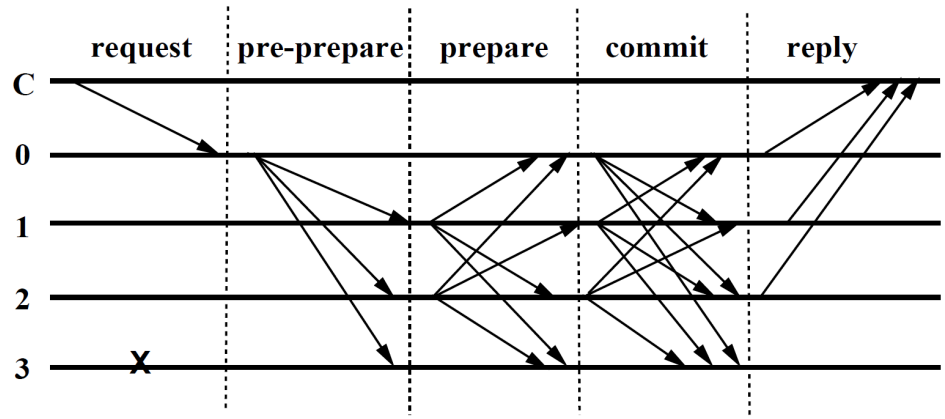


Figure 2.10: PBFT example case [39]

5. **Finalization reply**: In the end, consensus group will share the result of evaluated request back to the client.

Figure 2.10 shows an example situation in which, besides client C, there are four nodes composing the consensus group, one of which (i.e. node 3) is malicious. In this example, despite the presence of that malicious node, consensus has been achieved anyway. This because PBFT model can tolerate up to $\lfloor \frac{n-1}{3} \rfloor$ simultaneous Byzantine faults, where n is the currently total amount of nodes in the network (client instance is not included) [39]. In that example, n is equal to 4, therefore that network is able to tolerate up to 1 faulty node.

Given the previous formula, it is clear that the amount of tolerable Byzantine (i.e. faulty) nodes grows linearly as the number of nodes in the network increases.

2.4 WoT

A *Web of Trust* (*WoT*) is a kind of distributed PKI which, unlike traditional hierarchical structures, allows each of its nodes to create their own trust links with other nodes in the network, without the need for centralized authorities

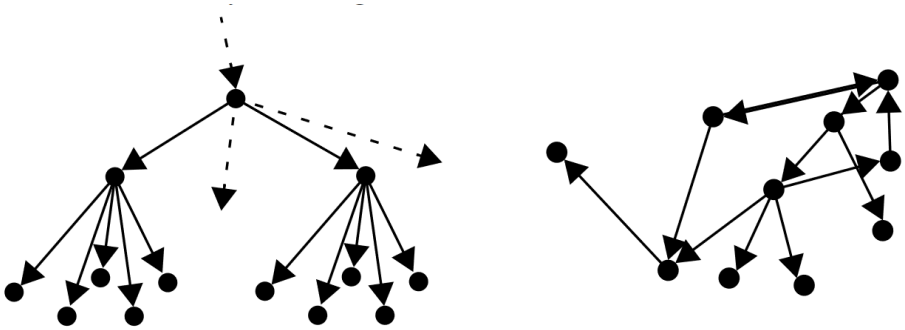


Figure 2.11: Hierarchical PKI vs. Web of Trust [40]

[40]. Figure 2.11 shows, in a simple way, the difference between a traditional hierarchical PKI (on the left) and a WoT (on the right).

WoT concept was firstly introduced in Zimmermann's PGP: an encryption framework developed to secure electronic mails. In PGP, a user decides which entities to trust, assigning them trust levels (e.g. untrusted, partially trusted, trusted), and put their public keys in a single file, called *key ring* [41]. In this way, infrastructure's security depends on which criteria the nodes choose while assigning their trust, and on the honesty of trusted elements.

2.5 Cryptographic accumulator

A cryptographic accumulator ACC is a set of heuristics and polynomial-time algorithms allowing to accumulate a finite set $X = \{x_1, ..., x_n\}$. For each of the accumulated values, it provides a witness ω^{x_i}, which is a proof of membership for value $x_i \in X$, meaning that it has been accumulated in ACC. Moreover, an accumulator should provide *soundness* property: it should be infeasible to find a valid witness for any non-accumulated value $y_i \notin X$ [42].

An accumulator can be useful in many scenarios, for instance when there is the need of a list of credentials that have been authorized for an access control. If those values are stored in a list, it would require a linear-time complexity ($O(n)$) to scan the list and find a match, where n represents the length of the list. That complexity can be lowered to sublinear ($O(log(n))$) by means of a binary search, but it will require some pre-computations

(e.g. sorting) on the elements of the list, which increases complexity to $O(n \cdot log(n))$. The use of cryptographic accumulators is a valid space-efficient alternative to achieve sublinear time complexity. Moreover, accumulators have another advantage: the checking for membership of a certain value can be done with just a portion of the tree, so it could be unnecessary to download the entire data structure [43].

One of the main possible classifications that can be made on these cryptographic objects, is the distinction between static and dynamic accumulators. Static accumulators basically consists of four algorithms [8]:

- $AccGen(1^{\lambda}) \rightarrow a_0$: algorithm for the initialization of an accumulator a_0 and all its parameters. A security parameter λ could be needed, too.
- $AccAdd(a_i, x) \rightarrow (a_{i+1}, \omega_{i+1}^x, updmsg_{i+1})$: adds to accumulator a_i the value x received as input, giving as output the updated accumulator a_{i+1} and the membership witness ω_{i+1}^x for x. Moreover, an update message $updmsg_{i+1}$ is provided in order to be used by witness holders for updating old set of witnesses.
- $AccWitAdd(\omega_i^x, y, updmsg_{i+1}) \rightarrow \omega_{i+1}^x$: given the update message $updmsg_{i+1}$ generated from the accumulation of a new element y, this algorithm updates the witness ω_i^x for element x, which was accumulated before y.
- $AccVer(a_i, x, \omega_i^x) \rightarrow 1/\ \perp$:[7] verifies membership of element x using related witness ω_i^x and accumulator a_i.

Dynamic accumulators are similar to static ones, but they additionally support deletion of elements from the accumulator itself. Hence, they provide these two following algorithms [8]:

- $AccDel(a_i, x) \rightarrow (a_{i+1}, updmsg_{i+1})$: deletes element x from accumulator a_i, and gives as output the updated accumulator a_{i+1} and the related update message used to update old set of witnesses.

[7] The $\perp$ ("falsum") symbol represents the logical value `False`.

- $MemWitUpdateDel(x, \omega_i^x, updmsg_{i+1}) \rightarrow \omega_{i+1}^x$: updates membership witness ω_i^x for element x, after the deletion of another element y, and it returns the updated proof ω_{i+1}^x for x.

2.5.1 Merkle tree

There are distinct types of accumulators providing different key properties, as classified in [43]. Sometimes, *Merkle Tree* data structure is used as basis to build *strong* accumulators, which main advantage is to not require any central authority for the signing of witnesses [44]. A Merkle Tree is a hash tree having a Merkle root on the top, whose value is "a pairwise accumulated hash of all the non-root nodes in the tree"[43]. Root value, which represents the value of the accumulator itself, has to be updated each time a new item has been added to or removed from the accumulator. These structures are very efficient since their time and size complexities are sublinear. However, their size continuously grows, even after deletions. As the root node, all the non-leaf nodes (i.e. inner nodes) of the tree contains an accumulated hash of the nodes below them in the tree structure. Leaf nodes contains the original values that have been added to the accumulator up to the current time [43].

Figure 2.12 shows how elements (usually hashes) in a Merkle binary tree are combined starting from the leaves up to the root node, where the + symbol stands for concatenation. In that example, values LA and LB are combined into their parent node, which is itself combined with another inner node into the root.

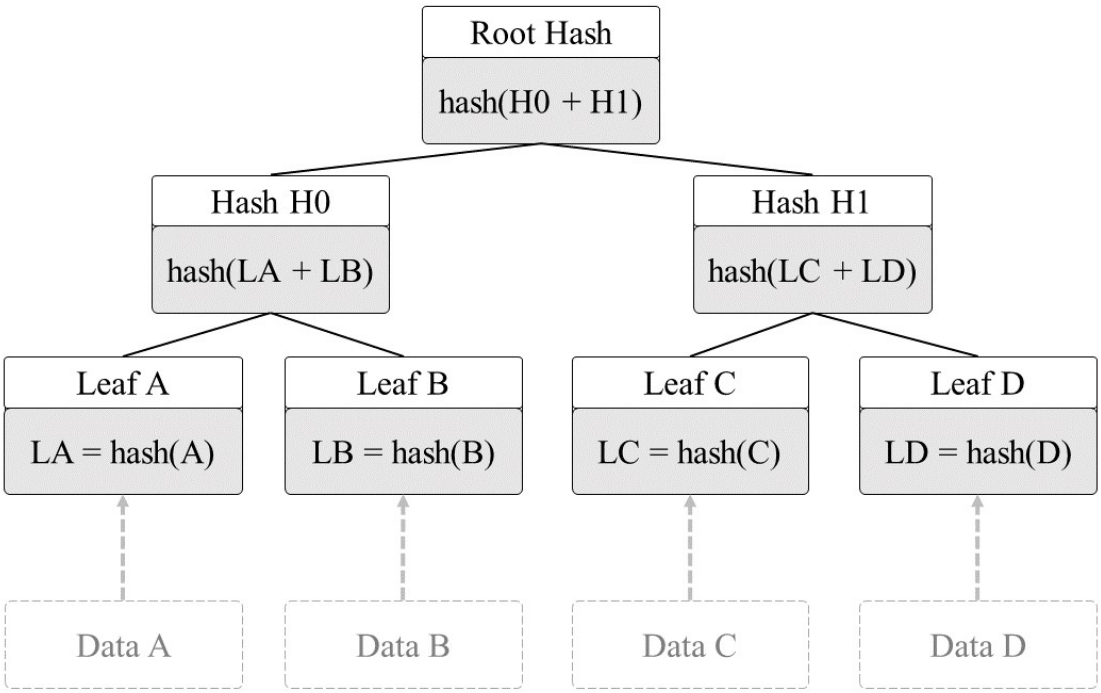


Figure 2.12: Merkle tree pairwise hashing example

Chapter 3

State of the art

Although standard design of PKI is widespread, as it guarantees great security properties, it suffers from weaknesses like the issuing of fraudulent certificates, or the use of complex revocation mechanisms with their related disadvantages, as already pointed out in the previous chapters. For this reason, for some time now researches have been looking for alternatives to that classic design solution. Some of these alternatives have been found thanks to the novel blockchain technology, used as a means of making network information secure, immutable and transparent, without the use of Certificate Authorities or other third parties. The transparency property, in particular, requires that all data in blockchain is publicly available to anyone.

During scouting phase of this thesis project, also carried out thanks to surveys like [45, 46], a lot of alternatives to standard PKI implementation have been found and analyzed. However, most of them did not removed the main problem of PKI centralization, which are CAs.

The first alternative PKI models proposed are mainly based on public logs. In log-based PKIs, all certificates issued by CAs must be logged on a public append-only log, before being considered as valid. In this way, any entity can audit CA activities and verify certificate validity. These kind of PKIs still have several drawbacks, since centralized CAs and sources of information are required [8]. An example of log-based PKI is given by *Certificate Transparency*, proposed by Google [47].

Another approach used to find alternatives to standard PKIs consists in using a blockchain as append-only ledger of signed transactions [8]. A first example of blockchain-based approach is the one presented by Kakei et al. [48], which proposed a PKI using some special meta-CAs communicating with classic CAs, using a distributed ledger technology. Zhao et al. [49] proposed a system in which certificates are stored inside smart contracts, while Wang et al. [50, 2] tried different approaches using Ethereum-based[1] blockchains, but all these solutions still use certificates generated by CAs. Other similar approaches exist, for example the work of Quin et al. [51], in which interactions among CA certificates are treated as a currency in a Bitcoin-based blockchain, or the proposals of Yakubov et al. [52] and Li et al. [53], which try to take advantage of smart contracts to emulate or to just support issuing authorities. There have been proposals for improving revocation system or PKI's transparency, without changes on traditional CA infrastructure [45]. Some other solutions instead, like [54], make CAs issue certificates through a system based on ACME (Automated Certificate Management Environment), a recent technology able to automate certificate issuance.

Unlike the just mentioned solutions, some of the most promising blockchain-based PKIs are constituted by a WoT-like infrastructure, which tries to overcome the issues related to single point of failures. Indeed, these recent solutions propose ways to fully decentralize Public Key Infrastructures using blockchain technology, and avoiding the use of CAs for identity management [45]. These proposals usually replace CAs with miners of public blockchains, and generation of certificates is made by mining after Proof of Work. For instance, Fromknecht et al. [55] proposed *Certcoin*, a PKI based on a cryptocurrency forked from the Bitcoin, that ensures identity retention[2], by exploiting PoW and mining concepts. The work of Axon et al. [56] and the one of Plessing et al. [57] lead to the realization of PB-PKI, a PKI based on Certcoin blockchain, where user's identity has been dissociated from the certificate and an accumulator has been used, ensuring anonymity. Al-Bassam [58] invented a PKI which uses smart contracts to publish attributes and revocations on Ethereum platform, and that performs identity management by means of a PGP-like WoT [8]. However, PGP itself is different from a PKI

[1]Ethereum is another blockchain cryptocurrency, like Bitcoin.

[2]Preventing registration of different public keys for a single entity.

as public keys cannot be retrieved from it [58]. Sermpinis et al. [59] proposed another solution using a blockchain storing the hash of each X.509 certificate, and where transactions has to be paid in Ethereum's gas currency. Toorani et al. [8] realized a WoT-like system based on blockchain named "DBPKI", which stands for Decentralized Blockchain-based PKI. DBPKI distributes trust among entities and allows certification and revocation after a consensus between nodes has taken place. Other similar WoT-like approaches have been proposed in [60, 61, 62], among which one is exploiting smart contracts' functionalities, another one is using graphs to relate keys and identities, while last one consists in a purely mathematical approach used for key validations and identity approvals. A different solution has been presented by Li et al. [10], who decided to use the cryptographic features of RSA[3] in order to protect users' privacy by storing pseudonyms on blockchain, instead of their actual identity.

Most of all the previous works researching for decentralized PKIs based on blockchain, fail to propose a decentralization of identity itself [45]. Among all, the few proposals able to allow individual users to create their identities by themselves are [45]: [59], [54], [8] and [10]. However, among them, the model proposed by Toorani et al. [8] is the one that most differs from traditional PKI. As a counterexample, the work of Kfoury et al. [54] is still based upon CA classic infrastructure design. Furthermore, Brunner et al. [46] stated that any blockchain-based PKI framework should follow at least these recommendations:

- **Permission type**: Both permissionless and permissioned blockchains can be used, even if sometimes the former is preferred due to better stability and end-user acceptance.

- **Revocation**: In a PKI, the possibility to revoke a certificate is a must, so it has to be supported.

- **Blockchain type**: A well-known and well-studied blockchain should be preferred to a custom one.

[3]RSA is a public-key cryptosystem relying on the hardness of factoring a number into its two large prime numbers [63].

- **Certificate format**: Use of a standardized certificate format is highly recommended.
- **PKI Type**: In order to create a decentralized PKI, the adoption of WoT structure is strongly recommended.
- **Storage Type**: The blockchain should store a minimum amount of data only, in order to reduce costs and to enhance performance.
- **Updatable Key**: Support for key updating is very important, especially for long-term usage.
- **Privacy**: Privacy features could be desirable, so they should be supported depending on the specific use case.
- **Incentives**: Incentives for participants are necessary to improve stability of blockchains, especially for custom ones.
- **Evaluation**: A blockchain should be evaluable and comparable to others, in terms of time and space complexity, as well as of monetary cost.

Conceptual DBPKI proposed by Toorani et al. follows almost all of the above suggestions. For this reason, and for the previously discussed ones, it has been chosen as starting point for the realization of the Proof of Concept described in this document. Moreover, a couple of other very promising researches have been taken into account to search for improvements for the starting DBPKI model, thus supporting even more of the above recommendations. For instance, the use of incentives for honest nodes that managed to successfully carry out some operations on the blockchain. Those works are: CyberChain, invented by Chai et al. [14], and ETHERST, created by Koa et al. [16]. Below, those researches that inspired the work of this thesis are briefly presented.

3.1 DBPKI

As already mentioned, DBPKI is the theoretical model proposed by M. Toorani and C. Gehrmann [8]. This model presents a fully decentralized PKI

based on a custom blockchain, and uses PBFT (Practical Byzantine Fault Tolerance) as consensus mechanism. DBPKI scheme includes the following entities:

- *Root units* (R_i): Root units are nodes existing in the beginning and assumed to be honest during the initialization of the PKI itself. Each root unit is identified by an identifier ID_{R_i} and has its own pair of asymmetric keys. DBPKI can be initialized by n root units $\{R_1, ..., R_n\}$.
- *Intermediate units* (I_i): Each intermediate unit is identified by an identifier ID_{I_i} and has its own key pair. Intermediate nodes could be, for example, organizations or institutions.
- *Ordinary units* (O_i): An ordinary unit, identified by an identifier ID_{O_i} and owning a key pair, is not directly part of the DBPKI itself since it can not perform any operation on the blockchain. Basically, they are users of the system, which can only read blockchain in order to validate public keys, without participating in the consensus mechanism. For instance, they might be resource-constrained units, like IoT devices or vehicles.

Any node can verify validity of a key, while only Root and Intermediate units can enroll, update or revoke a key by establishing the PBFT consensus mechanism as *leader node*. Blockchain structure and all DBPKI functionalities are explained in Chapter 4, since this model constitutes the basis of the design solution presented within this document.

3.2 CyberChain

Chai et al. proposed a blockchain for lightweight and privacy-preserving authentication in Internet of Vehicles based on CyberTwin (CT) technology [14]. The main issue of authenticating vehicles in V2X is constituted by their high mobility and frequent handover. CyberTwin is an emerging technology able to map physical entities into a cyberspace by constructing digital replicas, simulating communication behaviors of vehicles and giving real-time feedback to the physical world. Those researchers tried to exploit CT in order to

address authentication problems in IoV. Thus, they realized CyberChain (CC): a CyberTwin-empowered blockchain which uses blockchain to ensure security and CT to reduce consensus latency and storage consumption.

The framework is logically divided into a physical world and a virtual one. The former includes all physical entities such as vehicles and Edge Servers (ESs), which are servers allowing division of the entire physical world into multiple regions, according to their geographical location. Edge Servers has also the aim of maintaining multiple CTs and a CC in order to virtualize all vehicles present in their corresponding region. Thus, the virtual world refers to all the CyberTwin replicas, and it defines three virtual elements:

- *Sub-CyberSpace* (*SCS*): A SCS is the virtual space maintained by one Edge Server. It is constituted by the CTs of all vehicles in the region corresponding to that server, and the CT of the Edge Server itself.
- *CyberSpace* (*CS*): A CyberSpace is a set of SCSs, so it includes the virtualization of all the Edge Servers within one geographical region.
- *CyberChain* (*CC*): The CyberChain is the blockchain of a virtual space, and all the CT instances of the related CyberSpace constitute its nodes. A CC ledger only needs to record information about its region, unless a vehicle requests the identity handover process. In that case, the CC needs to communicate with the CCs in adjacent regions. Moreover, CC ledgers containing transactions, are cached by the Edge Servers of the related region only.

Edge servers constitute the bridge between physical and virtual world. Vehicle in a certain region will communicate with the nearest ES in order to make it construct a new CT for the vehicle itself. An Edge Server can simultaneously host several CTs, which virtually communicate to each other by means of interprocess communication. In this way, communication latency is greatly reduced.

There are two types of identity handover in this framework, depending if it happens between two SCSs or between two CSs. In the first case, the blockchain identity remains unchanged, so it just needs to authorize the new ES to reactivate its blockchain account, in order to rebuild the corresponding CyberTwin. On the other hand, when a cross-region vehicle requests the

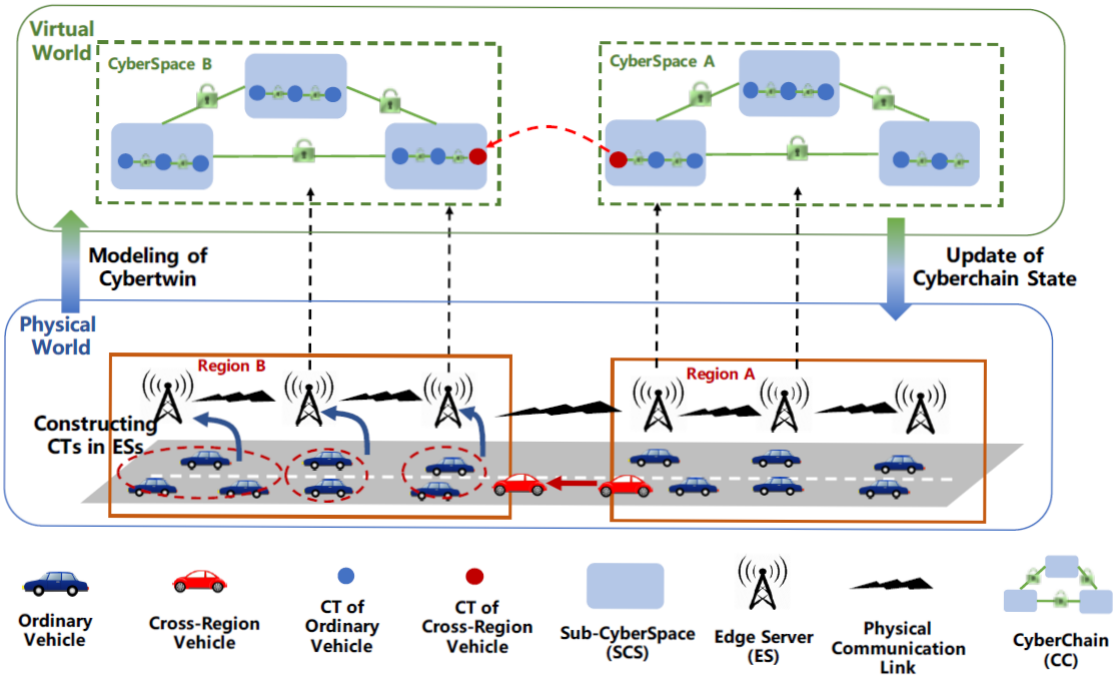


Figure 3.1: CyberChain framework model [14]

handover between two CSs, the consensus process requires every node of the network to reach the same view of the blockchain.

One of the main contributions of CyberChain is the use of DPBFT (Diffused Practical Byzantine Fault Tolerance), which is a variant of standard PBFT allowing to reach the consensus in a small part of the network first. Then, consensus will gradually reach all the other nodes of the entire network. Thus, DPBFT decomposes conventional PBFT consensus into multiple "sub-consensus areas", which in CyberChain coincide with SCS regions. In this way, assuming that each ES is maintaining at least ξ CTs, η of which malicious, the proposed DPBFT has a $\lfloor \eta / \lfloor \frac{\xi}{3} + 1 \rfloor \rfloor$ reduction in terms of single point of failure, if compared to traditional PBFT [14].

As shown in Figure 3.3, the use of the diffused version PBFT leads to less delay in reaching the consensus, especially for huge networks. Hence, as will be told in Chapter 7.1.3, this work could lead to a major improvement for the PoC developed during this thesis project.

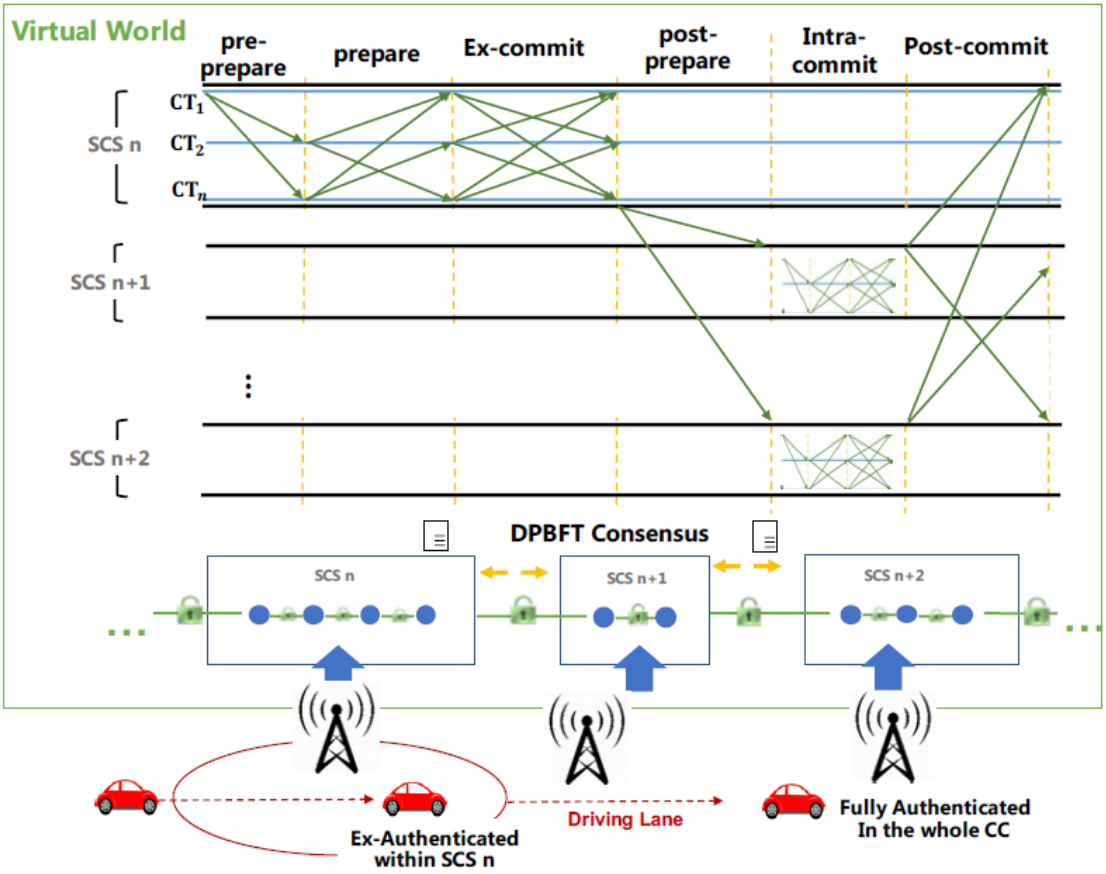


Figure 3.2: DPBFT consensus mechanism [14]

3.3 ETHERST

Some of the proposals for blockchain-based PKIs are based on the adding and running of program code into blockchain. Usually, for what has been previously discussed, well-known blockchains (e.g. Bitcoin) are used to develop these kind of PKIs. However, Bitcoin-based scripting language is not Turing-complete[4] [65], since nodes needs to process the script in order to verify validity of transactions [16]. Hence, the processing of a malicious script could cause nodes fall into infinite loops [16]. For this reason, usually

[4]Language able to simulate any Turing-machine, by providing ways for implementing loops and complex recursion [64].

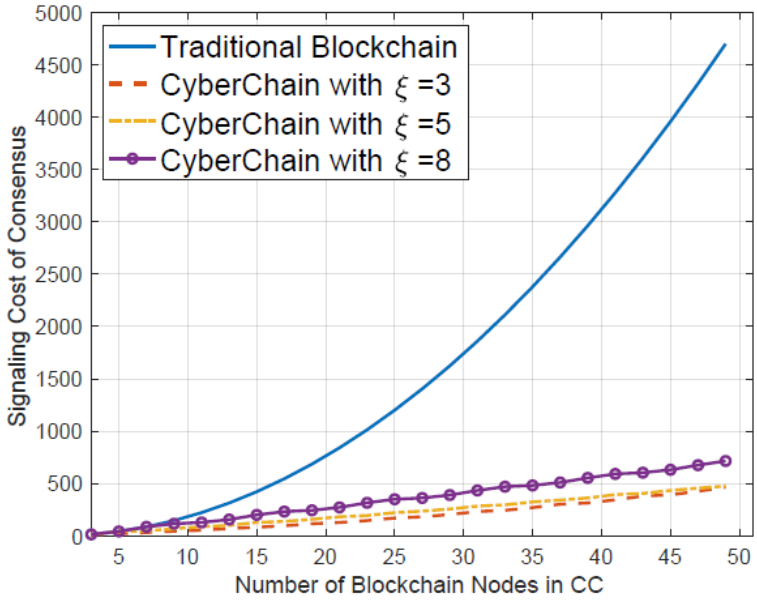


Figure 3.3: DPBFT communication overhead [14]

Ethereum-based blockchains are used instead, because Ethereum is a public blockchain platform supporting Turing-complete language, whose scripts are known as Smart Contracts. Similarly to Bitcoin, even Ethereum has its own cryptocurrency, called Ether. In addition, Ethereum allows to create custom cryptocurrencies [16].

One of the possible improvements of PKI based on Ethereum is the implementation of a reward-and-punishment mechanism, by exploiting its implicit currency property. Two examples of the applying of this kind of feature are the Instant Karma PKI proposed by Matsumoto et al. [66] and the Internet Web Trust System, invented by Li et al. [53]. The use of that currency leads to the problem of implementation costs that fluctuates together with the Ether market value [16]. For this reason, Koa et al. [16] introduced ETHERST, another blockchain-based PKI providing a reward-and-punishment mechanism, which uses custom ERC-20 token as cryptocurrency, thus solving the cost fluctuation issue. Additionally, the use of that new token, allows the owner of the distributed PKI to arbitrarily choose the values of reward and punishment.

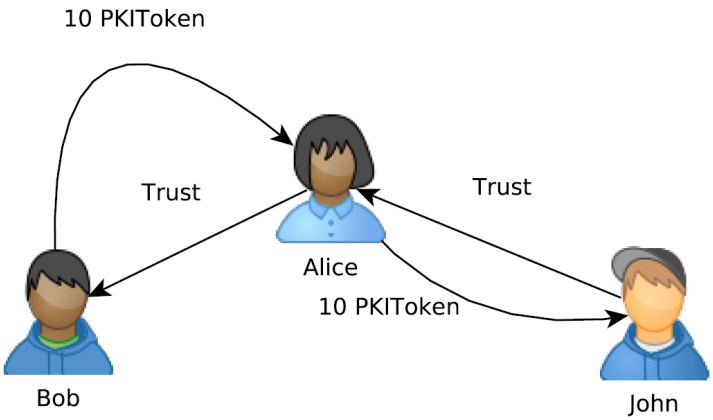


Figure 3.4: ETHERST trust and reward example [16]

ETHERST is the first blockchain-based system integrating Web of Trust with a reward-and-punishment mechanism. Its smart contract, written in Solidity programming language, defines some functions, like the creation or signing of an attribute, and the trusting or untrusting of a signature. These last two functionalities define also the implementation of the reward-and-punishment mechanism itself. When a user trusts another, the latter must send a fixed amount of PKIToken[5] currency to the first one. Only a certain amount of users (i.e. "TRUST-LEVEL") can trust the same signature. In this way, the request for trust gains a higher value, and the user giving its trust obtains a reward, thus encouraging the establishing of WoT.

On the other hand, during signature untrusting operation, the untrustor (i.e. the user which decides to untrust another) is rewarded with a fixed amount of PKIToken. The same amount of PKIToken is then deducted from the balance of the signature owner as a disincentive cost for not being trustworthy in the system. Indeed, a user's signature can only be untrusted by a maximum number of nodes (i.e. "UNTRUST-LEVEL"), and all rewards and the punishment are respectively given to untrustors and untrusted user, only when the UNTRUST-LEVEL-th untrust operation has been completed. Figure 3.5 shows an example scenario in which UNTRUST-LEVEL value is equal to 5 and the 5-th untrust operation has just triggered the punishment

[5]PKIToken is a custom token following ERC-20 specifications [16].

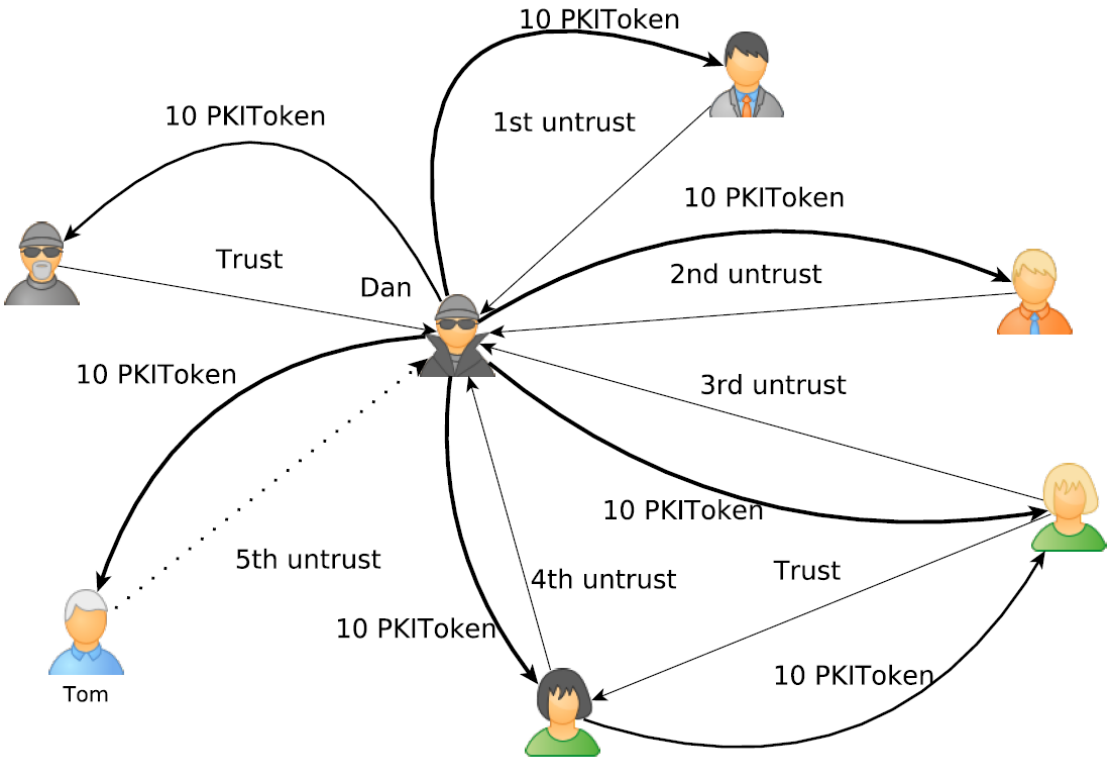


Figure 3.5: ETHERST untrust and punishment example [16]

appliance.

As said before, the functionalities of ETHERST have been taken as an inspiration for the realization of a custom reward-and-punishment mechanism in the PoC described starting from the next Chapter, where trust weights are used as incentive or disincentive for the various nodes.

Chapter 4

Solution design

The one presented within this document is a python project, in which a Proof of Concept of a decentralized blockchain-based PKI is implemented. The PoC has been built following the DBPKI model, proposed by Toorani and Gehrmann [8], which substitutes the traditional PKI with a WoT-based one. Additionally, this project proposes a management of trust weights weighted on time, and a new custom reward-and-punishment mechanism. Even if the proposed framework differs from DBPKI conceptual model, especially with regard to the use of trust weights, this document will refer to it as "DBPKI" anyway, since the basic concepts behind it are the same.

This Chapter introduces the design structure of the implemented solution, while its actual implementation is available in Chapter 5. Figure 5.5 will also display a summarizing model of design of developed Proof of Concept. Below, all the main elements of the PoC design are presented.

4.1 DBPKI User Interface

Solution has been designed in order to allow a single user to build its own blockchain-based PKI, and to try all its functionalities in a simulation environment, without the need for multiple applications or multiple users working together. Indeed, all nodes in blockchain are represented by processes

running on the same machine. To achieve that, a single DBPKI User Interface (DBPKI UI) has been created, and the PoC functioning has been divided into *rounds*. After the Setup phase, described later in section 4.6.1, the series of rounds begins. Each round is divided in three main phases:

1. **Node and operation choices**: At the beginning of the round, DBPKI User Interface asks the user to choose which node to impersonate in current round, and which operation that node should perform. After that, control is passed to that node, referred to as *leader node* for that specific round.

2. **Leader node operation**: Upon receiving the control over the UI interface, leader node will try to perform the requested operation and, if needed, it will start PBFT consensus mechanism. After operation success or failure, leader node will return control back to DBPKI User Interface.

3. **End of the round**: When requested operation has been performed and control has been returned to DBPKI User Interface, the latter communicates to all nodes that current round has finished, and then it will start another one, unless a termination command has been executed meanwhile.

4.2 DBPKI nodes

DBPKI nodes represent the actual elements of the PKI itself. Some of them could be, for instance, organizations or institutions, while others mirror entities like vehicles in V2X or low-consumption IoT devices. All of these constitute the nodes of a network, in which entities owning pairs of cryptographic keys, build reciprocal trust relations in order to avoid the need of centralized Certificate Authorities. That kind of network is known as Web of Trust.

DBPKI nodes can be divided in three classes, following the same division criteria used by Toorani et al. [8] in their work, mentioned in Chapter 3.1. Those classes are:

- *Root units* (R_i): Root units are nodes used to initialize the DBPKI itself. These units can be instantiated during Setup operation (4.6.1) only. They are assumed to be trusted a priori, and they are assigned an initial fixed trust weight of value `V_R`. Their role `R` allows them to participate in the consensus mechanism, and to perform all operations available on the blockchain.

- *Intermediate units* (I_i): Starting from the first round, Intermediate units can be enrolled by Root ones, or by other Intermediate nodes. After their enrolling operation has been successfully completed, they are assigned an initial fixed trust weight of value `V_I`. Their role `I` allows them to participate in the consensus mechanism, and to perform all operations available on the blockchain, just like Root units.

- *Ordinary units* (O_i): Ordinary units can be enrolled by any Root or Intermediate unit. They represent the actual users of the system, like IoT devices or vehicles. Indeed they are not directly part of the DBPKI, and according to their role `O`, they can access the blockchain in a read-only way, without either being able to participate in consensus mechanism. For this reason, their trust weight value `V_O` is set to zero by default.

All nodes can become the leader node of the round in progress, after user has chosen which one to impersonate. Root and Intermediate units, which could be associated to institutional entities or companies, are then able to: instantiate the consensus mechanism and wait for its final result, verify the validity of a public key of one of the nodes, or make the entire PoC terminate. Hence, their privileges allow them to modify the blockchain, by proposing new blocks that must be approved by the consensus group. Instead, an Ordinary unit that became the current leader node is only able to choose among public key verification or PoC termination commands. It is possible that some nodes need to be associated to several identities, with different roles. In that case they should create an account related to all their identities, however this feature has not been covered by proposed framework for the sake of simplicity: if a new identity is needed, a new unit has to be enrolled in the system.

4.3 Blockchain

In the presented work, blockchain is a custom digital ledger cached by the running nodes in DBPKI, including Root and Intermediate entities. Being a Proof of Concept, it is a simplified version of what would be a real DBPKI implementation. Hence, blockchain is cached also by Ordinary nodes, even if they are not directly part of the PKI. Moreover, when a new Intermediate or Ordinary node is enrolled, it obtains just the very last block in the chain (i.e. the one asserting its own enrolling), without compromising any PKI functionality. This is possible thanks to the use of a cryptographic accumulator, better described in section 4.4.

The very first block in the blockchain (i.e. the genesis block) includes all transactions regarding the enrolling of all the Root nodes, which issuance is made by the first of them, namely R_0. While all other blocks contain the transactions published by all the subsequent nodes that have become leader until current time. As it was structured the framework, all these other blocks, starting from the second one, refer to just one single transaction. This is due to the fact that operations performed after initialization are treated individually, alternating the work of the leader node with that of the DBPKI User Interface. Moreover, although the update operation of a key is substantially divided into two stages, it still generates only one transaction, as explained in section 4.6.3.

4.3.1 Block

A generic block in the chain includes the following items:

- **Timestamp**: Alphanumeric string indicating current date and time of the moment when leader node has built the block to be approved by consensus group.
- **Block identifier**: Numeric identifier of current block. First block ID is equal to 1.
- **Hash of the previous block**: Except for the genesis one, all the blocks

contain a secure one-way hash reference to the previous one in the chain, in order to achieve integrity of the links between them.

- **Accumulator**: Current cryptographic accumulator, updated considering the transactions of the block itself as the latest. This field implicitly includes also all the witnesses related to that accumulator.
- **Transactions**: All the transactions that leader node is trying to complete by placing current block under the judgment of all other nodes in DBPKI. As said before, while first block contains multiple transactions (i.e. one for each of the Root nodes), all other nodes will refer to just one transaction, due to a design choice. This does not mean that they could not include more than one transaction if necessary.
- **Trust weights**: Current value of the trust weights of nodes in DBPKI, since they vary as the PoC rounds go by. More on trust weights in section 4.7 and in Chapter 5.6.
- **Digital signatures**: All digital signatures of nodes in consensus group, including the one of the leader node, obviously. Except for the latter, digital signatures are stored together with the decision of the node on that proposed block (i.e. if it is approved or not).
- **Other minor data**: Additional data includes the identifier of the node who proposed the block (i.e. the leader node).

4.3.2 Transaction

Figure 4.1 shows the elements included in the genesis block and in a generic one. The blocks outlined in that figure do not include trust weights since in the work of Toorani et al. from which the image is taken [8], trust weights are designed to be fixed and so always immutable. From that picture, it is possible to notice the structure of transaction items. The latter include:

- **Node identifier**: Identifier of the node affected by current transaction.
- **Public Key**: This field contains the entire public key corresponding to the current transaction, if the latter is contained in the genesis block.

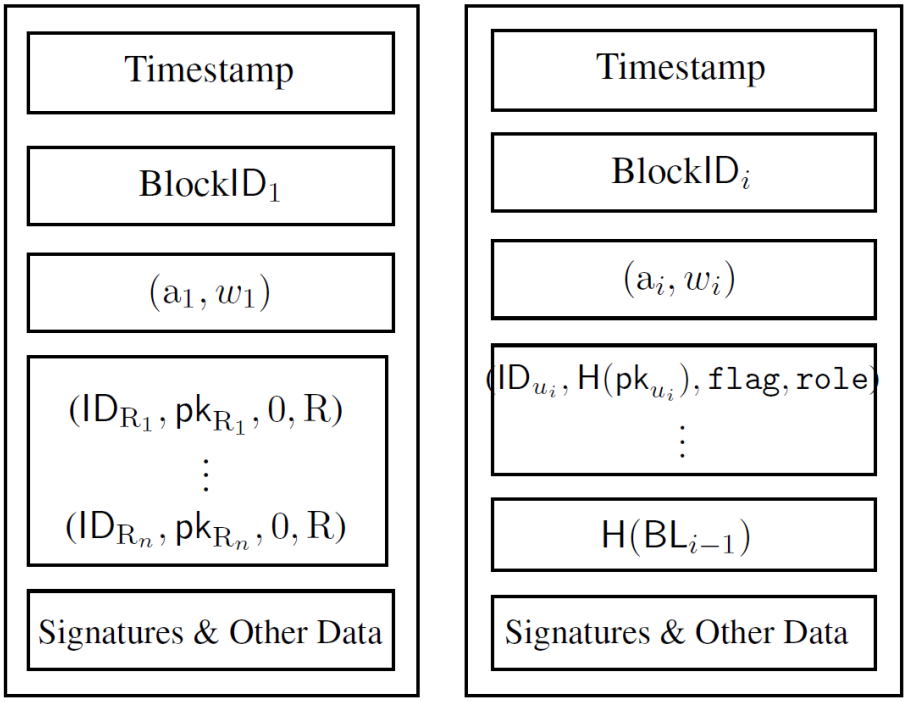


Figure 4.1: Contents of the first block (left) and i^{th} block (right) [8]

Otherwise, this field contains the hash of that public key only. Moreover, it refers to the new key, in case of enrolling or updating procedures, or to the revoked key if a revocation has taken place.

- **Flag**: `flag` $\in \{0, 1, 2\}$ indicates whether corresponding public key has been enrolled, updated or revoked, respectively.
- **Role**: `role` $\in \{\texttt{R}, \texttt{I}, \texttt{O}\}$ denotes the role of the affected node.

4.4 Accumulator

Within this framework, it is possible for users having access to blockchain to retrieve the history of a public key, from when it was issued, until its expiration or revocation, if already happened. This kind of operation would be possible by scanning the entire blockchain only. In order to know which

public keys are valid or not without examining the whole blockchain, an accumulator is needed. That cryptographic tool compacts all necessary data in a single object, and allows to obtain public key validity information by accessing the very last block in the chain only. According to DBPKI starting model, "any efficient and sound dynamic Merkle-tree based accumulator [...] can be used" [8]. As already pointed out in Chapter 2.5, an accumulator is sound if it is computationally infeasible to find a valid witness for a non-accumulated item, while it can be tagged as dynamic if it also supports the removal of previously accumulated objects. Many well-known accumulators are publicly available, as the ones described in [67, 43]. One of these is the asynchronous accumulator defined by L. Reyzin and S. Yakubov [44], which exploits a dynamic set of Merkle Trees in order to reduce the frequency of witness updates. In that case, accumulator value is computed by considering the roots of all its Merkle Trees.

Typically, the membership proof of an item added to a cryptographic accumulator has to be always synchronized with the accumulator value itself. This means that, every time accumulator is modified, all witnesses related to its accumulated items have to be updated accordingly. An example of synchronous accumulator is a simple Merkle Tree, in which proofs need to be constantly synchronized with the root of the tree. This kind of operation leads to a waste in computational time and consequently it also leads to communication overhead at the application level. This is an important issue to be considered in distributed systems, like the one of decentralized PKI. In that case, an asynchronous accumulator as the one proposed by Reyzin et al. [44] is useful, since it provides *low update frequency* for witnesses, and *old-accumulator compatibility* property, which makes it possible to verify a proof with outdated accumulator values, too.

Low update frequency

An accumulator has a *low update frequency* if verification of witness ω_x is successfully working even if that proof is only updated a number of times which is sub-linear in respect to the amount of elements accumulated after x. In this way, a user can not verify an arbitrary old proof with the last published accumulator value, but it can still verify it, if recent enough.

Old-accumulator compatibility

An accumulator is *old-accumulator compatible* if membership verification through an up-to-date witness ω^x is successful even with an outdated accumulator. This is possible even with an arbitrary old accumulator; the only constraint is that the element x has already been added to it.

The aforementioned properties make an accumulator asynchronous, allowing verification of a witness, even if the latter is non-synchronized with the current accumulator value (i.e. a bit older or newer). Assuming that t_x is the moment when x has been accumulated, and t_a represents the time of the last update of accumulator value, verification can work with a proof from any time t_ω after the moment given by the arithmetic average between t_x and t_a. Hence, there are two out of sync situations: accumulator is older than the witness to be verified, or vice versa [44]. Table 4.1 shows various complexity comparisons between a standard Merkle Tree construction, a sample synchronous accumulator and the proposed asynchronous one, where n is the number of accumulated elements.

Functionality	**Synch. accumulator**	**Standard Merkle Tree**	**Asynch. accumulator**
`AccAdd` algorithm runtime	$O(1)$	$O(log(n))$	$O(log(n))$
`AccAdd` algorithm storage	$O(1)$	$O(log(n))$	$O(log(n))$
`AccWitAdd` algorithm runtime	$O(1)$	$O(log(n))$	$O(log(n))$
`AccWitAdd` algorithm storage	$O(1)$	$O(log(n))$	$O(log(n))$
Accumulator size	$O(1)$	$O(1)$	$O(log(n))$
Witness size	$O(1)$	$O(log(n))$	$O(log(n))$
Update frequency	$O(n)$	$O(n)$	$O(log(n))$

Table 4.1: Accumulators complexity comparison [44]

However, as pointed out later in section 4.4.2 these properties are indeed not granted in the proposed framework since removal of accumulated elements should be provided. Nevertheless, that accumulator is suitable for distributed situations also because it does not require any knowledge on the accumulated items, like the number of already added elements, for new adding operations [44]. Moreover, as underlined in Table 4.1, space complexity of accumulator and its witnesses is logarithmic. Therefore, even if some properties of asynchronous accumulators are compromised by the need of deleting procedure, the proposed PoC deploys that cryptographic accumulator model and avoids those related issues by publishing an update both for accumulator and proofs in each new block in blockchain.

4.4.1 Construction

As already explained, developed accumulator is based on Reyzin and Yakubov's proposal [44], which is asynchronous and built on a set of Merkle Trees.

Accumulation of elements in the proposed accumulator consists in computing a hash of them, and inserting it in a leaf of one of the Merkle Trees. Then, starting from that leaf, hash of all the parents is iteratively recomputed up to the root, considering their children. Given a hash function h and a node in one of the Merkle Trees, computing the hash related to that node consists in hashing the values contained in its descendants. A node in a Merkle Tree can have one or two children, so hash function h can be applied to single elements or to pair of elements, depending on how many descendants the node has. When h is used to encode a single element, the latter is prefaced by a '0', while during an encoding of a pair of elements, first one is prefaced by a '1' and second one by a '2'. This is done in order to never confuse hashes of single elements with hashes of pairs.

The accumulator maintains a list of $D = \lceil log(n+1) \rceil$ Merkle-tree roots, where n represents the current amount of elements in the accumulator. Those elements are accumulated into the leaves of the trees. A sample root r_d is the root of a complete Merkle Tree with 2^d leaves if and only if the d-th least significant bit of the binary expansion of n is 1 [44].

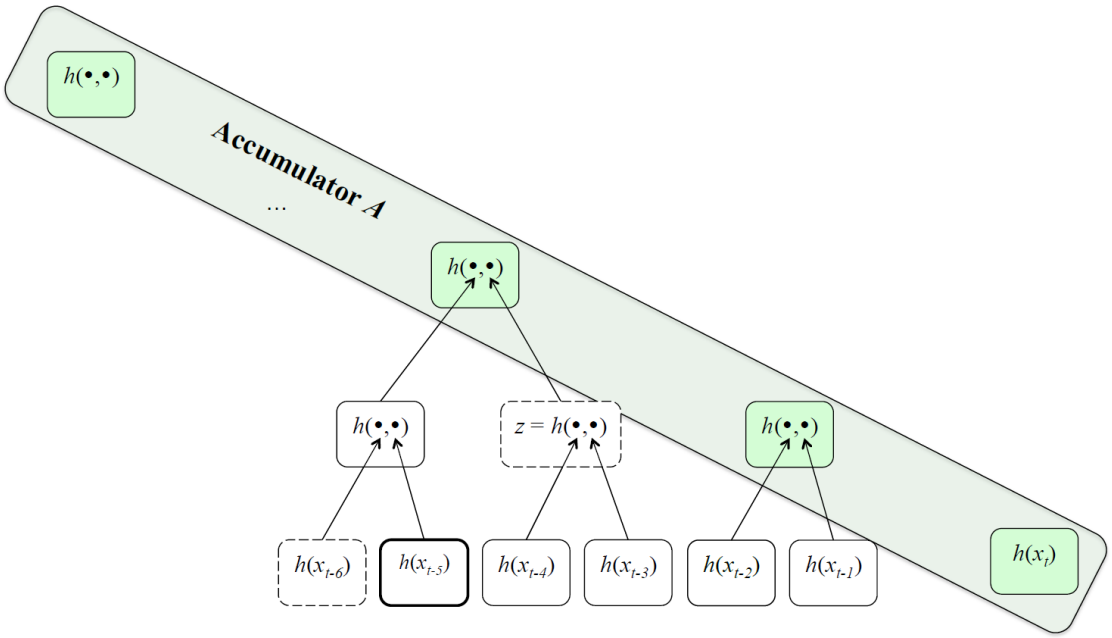


Figure 4.2: Developed accumulator example [44]

If element x is in the Merkle Tree with root r_d, then its corresponding witness is defined as $\omega^x = ((z_1, \texttt{dir}_1), ..., (z_{d-1}, \texttt{dir}_{d-1}))$, where z_i is a resulting hash, while `dir` indicate the direction of current branch in the authenticating path, that could be either `left` or `right`. Hence, a witness is an append-only list of items composing a rebuildable Merkle-tree path. Sibling elements along all those Merkle-tree branches, together with x, can be used to reconstruct the root of the Merkle Tree itself. An example illustration of the used accumulator is shown in Figure 4.2, where elements with dashed outlines belong to the authenticating path for x_{t-5} element, which means that its witness is computed as $\omega^{x_{t-5}} = ((h(x_{t-6}), \texttt{left}), (z, \texttt{right}))$ [44].

4.4.2 Operations

This section presents procedures provided by the accumulator deployed in the Proof of Concept. Additionally, Appendix A contains the specific algorithms that inspired accumulator implementation, as pseudo-code.

Element addition

New element addition operation (*AccAdd*) is performed by merging together existing Merkle Trees, so creating deeper ones. In particular, when element x is accumulated as n-th, if $r_0 = \perp$ (i.e. first Merkle Tree is empty), then $r_0 = h(x)$, otherwise a "carry" operation is needed. The latter consists in [44]:

1. Creating a new Merkle-tree root z equal to the pairwise hash $h(r_0, h(x))$.
2. Set r_0 to $\perp$, and then look to the next root (i.e. r_1).
3. If r_1 root's tree is empty, then $r_1 = z$, otherwise it must continue merging subsequent Merkle Trees and "carrying" up the chain.

In Figures 4.3 and 4.4 it is possible to observe an usage example of the used accumulator, in which x_{t+1} is being added and two "carries" are needed.

Witness update

Witness updating operation (*AccWitAdd*) occurs any time two Merkle Trees are merged or "carried", during an adding procedure. As better described in corresponding algorithm of Appendix A, membership proof updates are executed by means of an "update message" $\texttt{updmsg}_{t+1} = (y, \omega_{t+1}^{y})$, where y is the just added element and ω_{t+1}^{y} is its new corresponding generated witness [44]. After the completion of the updating procedure, all witnesses of the accumulated elements are up-to-date.

Element deletion

Since DBPKI construction requires a dynamic accumulator, deleting operation (*AccDel*) of previously accumulated items must be provided. This operation removes element x from the accumulator and, as for the adding, update messages used to update the other witnesses will be generated too, by means of *AccWitDel* operation. Deletion can be made by replacing the leaf to be removed with $\perp$ (i.e. a null value) and recomputing values of its ancestors. Corresponding updating operation consists in replacing "the child node of the lowest common ancestor of the deleted value and x" [44] with the value contained in the received update message. However, in order to do that, some modifications should be done on the starting version of the accumulator proposed by Reyzin et al. [44]. Although those needed modifications would degrade low update frequency property and old-accumulator compatibility [44], this problem does not result in the proposed PoC, since an operation can be performed only if the author node has access to an updated version of the blockchain.

Witness verification

Membership verification (*AccVer*) for element x is made by starting from the item x itself and its witness ω^{x}. Purpose of verification is to reconstruct the Merkle Tree from the leaf related to x up to the top, checking that result of this operation matches the value of the root r_d, where d is the length of ω^{x}. This is done by recomputing the ancestors of x, starting from knowing

the value of ω^x [44].

4.5 Consensus mechanism

DBPKI model expects the use of a consensus mechanism to make nodes take decisions in a distributed fashion. The selected consensus mechanism is *Practical Byzantine Fault Tolerance* (*PBFT*). Its execution is made up of rounds and each of them is split in some phases, and it is carried on by nodes in a consensus group. In all phases, several messages are signed and then sent as multicast to all other nodes. For each running of PBFT mechanism, a node in that group acts as the *leader*, while all the others behave as *validators*. Leader node is in charge of proposing a new item (or set of items) and validators should approve it or not, according to some specifications [8]. In the proposed framework, consensus group is composed by Root and Intermediate units, so excluding Ordinary units, since they just represent the users of the PKI. In DBPKI, the leader's proposal consists in a new block to be appended to the blockchain if it has been approved by most of the validator nodes. PBFT phases are the following [8]:

1. *New round phase*: Initially, all DBPKI nodes are in `NEW ROUND` state. A new round is initiated by the user, through DBPKI User Interface commands. In this stage, leader node is selected and operation to be performed on blockchain is chosen.

2. *Pre-prepare phase*: Leader node creates a new block with one or more transactions corresponding to the operation to be performed on blockchain. Then, it signs those elements with its key pair and sends it as a multicast pre-prepare message to all validator nodes. At the end of this stage, all validator nodes should have received the pre-prepare message and their state will be set to `PRE-PREPARED`. Leader node is directly set to `PREPARED` state instead, since it already knows the content of block proposal, and it does not need to validate its own pre-prepare message.

3. *Prepare phase*: Upon receiving the pre-prepare message of current round, each validator verifies correctness and validity of leader's proposal. The former can be checked by looking at the transactions contained in the

block proposal, while validity check is made by verifying the digital signature. After that, each validator node multicasts a signed prepare message to all nodes in consensus group. Each of those messages contains the approval decision for block proposed by the leader node. At the end of this phase, all nodes in consensus group are in `PREPARED` state.

4. *Commit phase*: Nodes in consensus group wait for all prepare messages and, after analyzing their content, they can know if most of validator nodes have approved the new block. In that case, the latter can be committed to blockchain and a positive signed commit message is sent out as multicast by all nodes, stating their agreement on current operation. Moreover, the new block in the chain will also contain all the signatures of DBPKI nodes, along with their approval decisions. Otherwise, if number of validators' approvals is not high enough, blockchain will remain unchanged and negative commit messages are exchanged instead. At the end of Commit phase, states of all the nodes in consensus group will be set to `COMMITTED`.

5. *Round change phase*: After reporting current round result, leader node leaves control of the PoC to DBPKI User Interface, which will make application ready for the next round. In the meantime, all the nodes have changed their states to `ROUND CHANGE`.

This kind of consensus mechanism can tolerate up to $f = \lfloor (t-1)/3 \rfloor$ Byzantine (i.e. faulty or malicious) nodes, where t is equal to the number of nodes in consensus group. This is possible since each validator node should verify at least $2f+1$ collected messages, in both Prepare and Commit phases. Indeed, actual value of messages to be validated could be different for each round, depending on trust weights of nodes in consensus group, as explained later in section 4.7. At the end of each PBFT round all honest nodes in the consensus group has reached a common decision about approving or rejecting the new block proposal provided by the current leader node. In this way, all honest nodes will then have the same view on blockchain state [8].

4.6 Procedures

Proposed framework defines a set of procedures that can be executed by DBPKI nodes in order to create new blocks for the blockchain, or to verify validity of a certain public key. Additionally, a minor functionality simply consists in terminating the entire session, by making DBPKI User interface and all other nodes stop themselves. The main provided functionalities are [8]:

- Setup(λ): After Root nodes initialization, this function generates first accumulator object, and publishes genesis block of blockchain including transactions for all Root units. This operation is performed by first Root unit, namely R_0.
- Enroll(ID_{u_i}, pk_{u_i}, `flag`, `role`) $\rightarrow$ 1/ $\perp$: Enrolls an entity u_i with identifier ID_{u_i} and `role` $\in \{\texttt{R}, \texttt{I}, \texttt{O}\}$ in the PKI. Public key pk_{u_i} is the public key of the entity to be enrolled. Boolean value `flag` indicates if entity is being enrolled for the first time (0), or if the public key is being updated (1) (i.e. by means of execution of Update procedure). This procedure can be requested as leader node by Root (`R`) and Intermediate (`I`) nodes only (i.e. nodes belonging to the consensus group). It outputs success (1) or failure ($\perp$). Enrolling procedure is used during Setup in order to enroll Root units and, as an example, it can be used when an Intermediate unit (e.g. an organization) would like to enroll a certain device (i.e. an Ordinary unit).
- Update(ID_{u_i}, $pk^*_{u_i}$) $\rightarrow$ 1/ $\perp$: Updates the public key value corresponding to entity with identifier ID_{u_i} to the new value $pk^*_{u_i}$, returning as output success (1) or failure ($\perp$). As enrolling and revoking procedure, also updating a key can be requested by Root (`R`) and Intermediate (`I`) nodes only, and not by Ordinary (`O`) ones. Update procedure has to be used any time a node is changing its reference public key, for example when old key pair has reached is expiration time, if any.
- Revoke($\text{ID}_{u_i}, pk_{u_i}$) $\rightarrow$ 1/ $\perp$: Revokes public key pk_{u_i} of the entity with identifier ID_{u_i}, returning success (1) or failure ($\perp$). Revoking procedure can be requested as leader node, only by nodes that are currently in the consensus group. For instance, revoking process can be used by a Root

or Intermediate node which has learned that its own private key has been compromised, or if it would like to revoke key of a faulty Ordinary unit.

- Verify(ID_{u_i}, pk_{u_i}) $\rightarrow$ 1/ $\perp$: Verifies whether or not pk_{u_i} is a valid public key, meaning that it correctly corresponds to the identity of the entity with identifier ID_{u_i}, and returns success (1) if key is valid, or failure ($\perp$) otherwise. Verification process can be carried on by any unit $u \in \{\mathtt{R}, \mathtt{I}, \mathtt{O}\}$. This operation can be useful, for example, when a certain communication channel has to be instantiated and an identity validation is necessary.

All procedures, except for verification, require reaching of consensus in order to be successfully completed. Hence, most of the nodes in consensus group should have approved the new block proposed by current leader node. Those procedures are explained in more detail below, and their related pseudo-codes are available in Appendix B.

4.6.1 Setup

Setup operation is performed only once in the beginning, immediately after Root nodes are initialized, and their own key pairs (sk_{R_i}, pk_{R_i}) are created. After that, first Root node (R_0) takes on the task of generating an accumulator a_0 by means of the `AccGen` function presented in Appendix A. R_0 accumulates identifiers and hash of public keys of all Root nodes, starting from itself, into the accumulator. Thus, an updated accumulator a_1 and related witness ω_1 are generated. Then, it creates a transaction item for each of the Root units, including identifier ID_i, entire public key pk_{R_i}, `flag` set to '0' indicating that keys are being added for the first time, and `R` in the `role` field. After that, all those transactions are inserted in a block proposal and sent out by R_0 in the first pre-prepare message, so initiating PBFT consensus protocol as leader node. All other Root units act instead as validator nodes during Setup procedure. They should verify that their public keys have been correctly accumulated by R_0, and that transactions in block proposal contain the right values. At the end of Setup procedure, if everything works correctly, all Root units share the same view on blockchain state and they are ready to start

first real round of this framework. Otherwise, initialization process fails and Setup procedure has to be restarted from the beginning [8].

4.6.2 Enroll

If any new unit u_i would like to join the DBPKI, it should generate its key pair and send a signed request to unit u_j, which is already part of the consensus group. This enrollment request should contain an identifier ID_{u_i}, the public key pk_{u_i}, desired role in PKI, and it could also contain validity period or expiration date for that key if applicable, or other minor information if needed [8]. In order to simplify the process, proposed Proof of Concept assumes that node u_j in consensus group decides to enroll a new node by its own, only requiring the latter to create a key pair, and to share its public part. In this way, enrollment request is not needed anymore. Then, unit u_j verifies if received public key correctly correspond to claimed u_i identity by means of *Public key and Identity Validation* (*PIV*), which preserves *identity retention*. Finally, node u_j creates a new signed block proposal related to current enrollment operation, that will be evaluated by other nodes in consensus group following the steps of PBFT mechanism. It is important to notice that if Enroll function is used during Setup, then transactions contain entire public keys of Root units, otherwise transactions contain hash of public keys only, in order to reduce accumulator size and communication overhead.

Each node in consensus group will verify correctness and validity of proposal and, if at least $2\lfloor(t-1)/3\rfloor + 1$ of them have approved current enrollment, new block will be successfully committed to blockchain and so new unit will be part of PKI starting from the next round [8]. Indeed, block proposal approval depends not only on the number of nodes accepting it, but also on their trust weights, described in section 4.7. Note that a Root unit can be enrolled only during Setup procedure, while Intermediate and Ordinary nodes can only be enrolled after it. On the other hand, if consensus mechanism leads to a rejection decision, then enrolling fails and blockchain remains unchanged.

4.6.3 Update

Update procedure can be performed by any unit $u_i \in \{\texttt{R}, \texttt{I}\}$ that would like to update public key reference of a unit $u_j \in \{\texttt{R}, \texttt{I}, \texttt{O}\}$ (it is possible for a node to update its own keys, too). It consists in revoking the old key $pk_{u_j}^{old}$ by means of revocation procedure, and registering the new one $pk_{u_j}^{new}$ by executing Enroll(ID_{u_j}, $pk_{u_j}^{new}$, 1, `role`), where `flag` = 1 indicates that affected entity u_j was already recorded in the public ledger [8]. Hence, enrolling and revoking procedures are merged in a unique updating operation, which results in a single execution of consensus mechanism, that leads to a success if and only if a certain number of DBPKI nodes, depending on their trust weights (see section 4.7), agrees on current proposal. Unlike starting DBPKI model proposed by Toorani et al. [8], which decided to insert both the revoking and enrolling part in the block proposal as different transactions, proposed framework uses just the enrolling part (with `flag` value appropriately set to '1'). Thus, revocation of old public key remains implicit.

4.6.4 Revoke

Revocation procedure can be performed by any unit u_i which is part of DBPKI (i.e. Root or Intermediate), when key pair of unit u_j has to be revoked for some reason, like validity expiration, probable private key disclosure, etc. It is also possible that $u_i \equiv u_j$, if an entity would like to share a revocation request to consensus group for its own key pair. Node u_i creates a block proposal, inserting a transaction including identifier ID_{u_j}, hash of public key pk_{u_j} to be revoked, `flag` value set to '2' and `role` value indicating which is the role of unit u_j. Block proposal is then signed by u_i and shared with other nodes in consensus group during first step of PBFT protocol [8].

As for enrolling and updating procedures, upon completion of PBFT consensus mechanism, during which any unit in DBPKI has verified signatures of received PBFT messages, revocation request is approved if the number of nodes in consensus group accepting it is large enough. DBPKI nodes also have to verify that public key to be revoked was present in last version of the accumulator, by means of `AccVer` function. If revocation succeeds, then new

block in the chain will contain an accumulator from where revoked public key has been removed through `AccDel` functionality. Otherwise, if number of agreeing units is not large enough, then revocation is aborted [8].

4.6.5 Verify

Verification procedure is the way a unit $u_i \in \{\mathtt{R}, \mathtt{I}, \mathtt{O}\}$ can verify validity of a certain public key and its correct association to a specific identifier, and so to a certain node u_j in PKI. It should check if public key belongs to the right owner, if it is not expired yet (if an expiration time is applied), and, primarily, if it has not been revoked. For simplicity, PoC described in this document does not consider any expiration date. Given the identifier ID_{u_j} of the entity whose key has to be verified, and value of the public key pk_{u_j}, verifier u_i has to build $x = (\mathrm{ID}_{u_j}, H(pk_{u_j}))$, where H represents an hash function. Then, this proof membership algorithm verifies whether if $\mathtt{AccVer}(a_i, x, \omega^x) = 1$, where a_i is the accumulator updated to last round, and ω^x is the up to date witness corresponding to x, which was generated and published when pk_{u_j} was enrolled. If that equation is satisfied, then it means that the element to be verified is valid as it is present in the accumulator [8].

4.7 Trust weights

In order to better distribute trust among DBPKI entities, this thesis improved starting model of Toorani et al. [8], by proposing the use of dynamic trust weights. Each node in the network is assigned to a certain trust weight value, and all other nodes develop trust in it based on that weight. This means that, when PBFT consensus mechanism is activated, the threshold to successfully commit the new block proposal is not only given by the number of agreeing nodes (that should be $2\lfloor(t-1)/3\rfloor + 1$, where t is number of nodes), but also on their current trust weight values. Therefore, in Prepare and Commit phases of PBFT, when a node has received all messages from other DBPKI nodes, it verifies whether the threshold $(2\lfloor(t-1)/3\rfloor + 1) \cdot w^i_{avg}$ is reached or not. Current value of w^i_{avg} is computed in each round i as the average trust value that each node in DBPKI, except for leader, would have if they all had

the same trust weight, and if all of them were always considered not revoked. The latter requirement is necessary to compute value of w^i_{avg} because, by default, trust weight is always set to 0 for revoked nodes, as well as for the Ordinary ones, not being part of DBPKI. Moreover, trust weights are stored in blockchain, so allowing any node to stay updated about all of them.

Initially, when a node has just been enrolled in the system, a default trust weight is assigned to it, according to its role type (Root, Intermediate or Ordinary). As the rounds go by, trust weights of nodes are increased in a way that the nodes that have been part of the network for the longest time, have a greater trust weight than newly enrolled nodes. Furthermore, proposed framework implements also a reward-and-punishment mechanism that increases or decreases trust weight of nodes according to the outcome of their operations. When a Root or Intermediate node has not successfully completed an operation on blockchain, it means that it was not able to reach majority of consensus in DBPKI. For this reason, reward-and-punishment mechanism will decrease its trust weight, since that node could have become faulty or malicious. Otherwise, if operation was successfully committed, trust weight of leader node will be increased instead. This reward-and-punishment mechanism also sets directly to zero trust weight of nodes that have been revoked, because no other node should trust them anymore.

Chapter 5.6 describes usage of trust weights in more detail, and how reward-and-punishment mechanism is actually implemented, too.

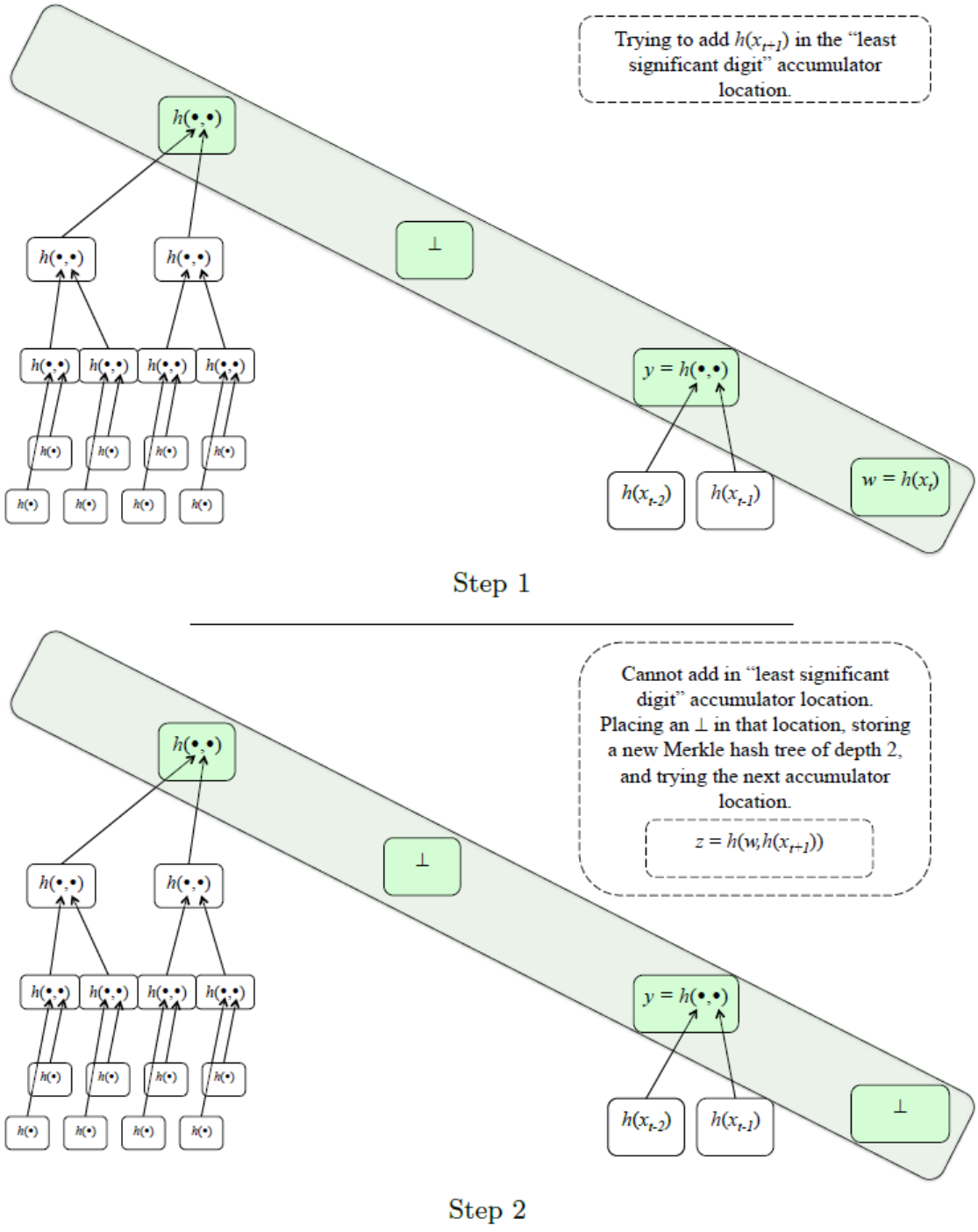


Figure 4.3: Adding operation example - part 1 [44]

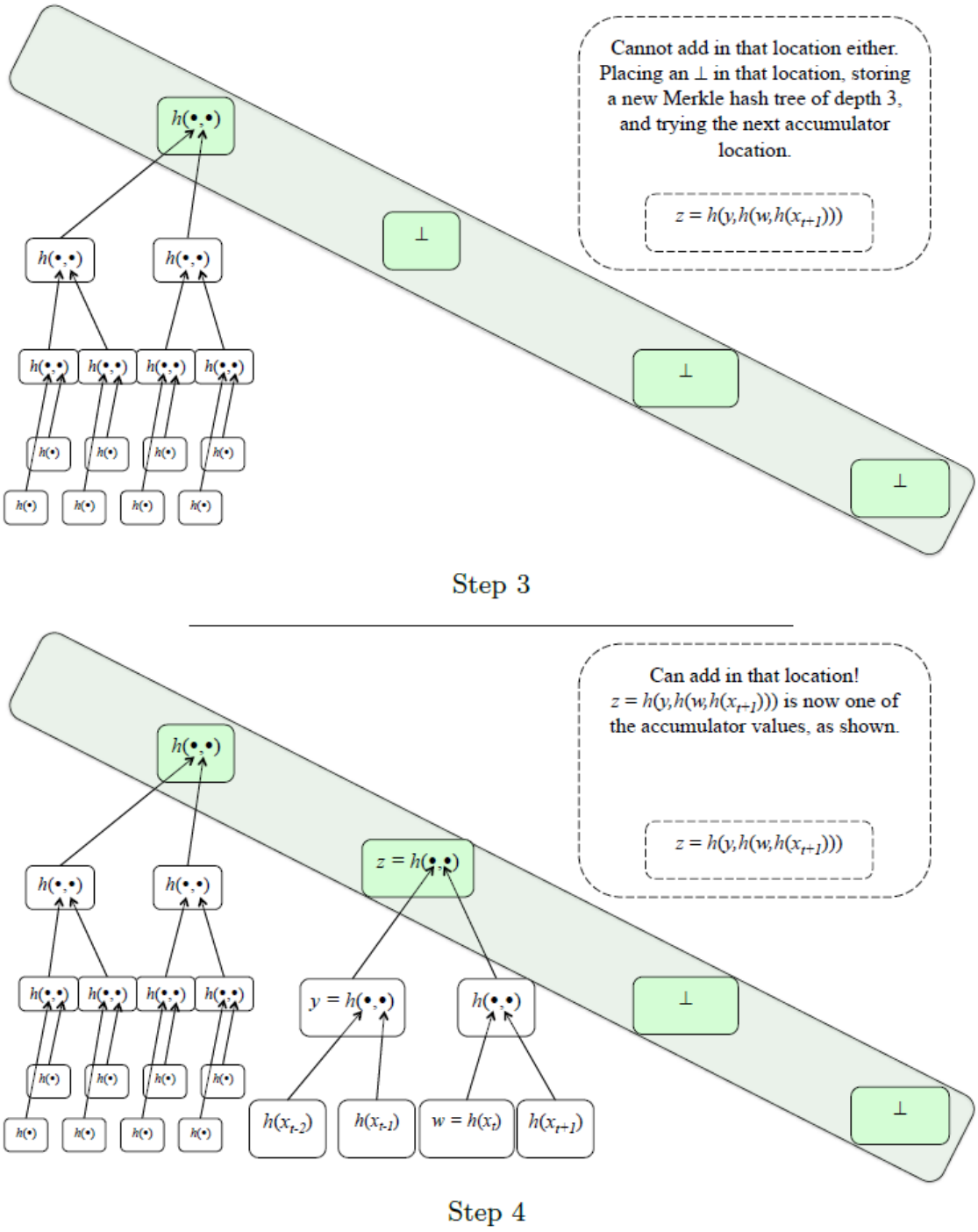


Figure 4.4: Adding operation example - part 2 [44]

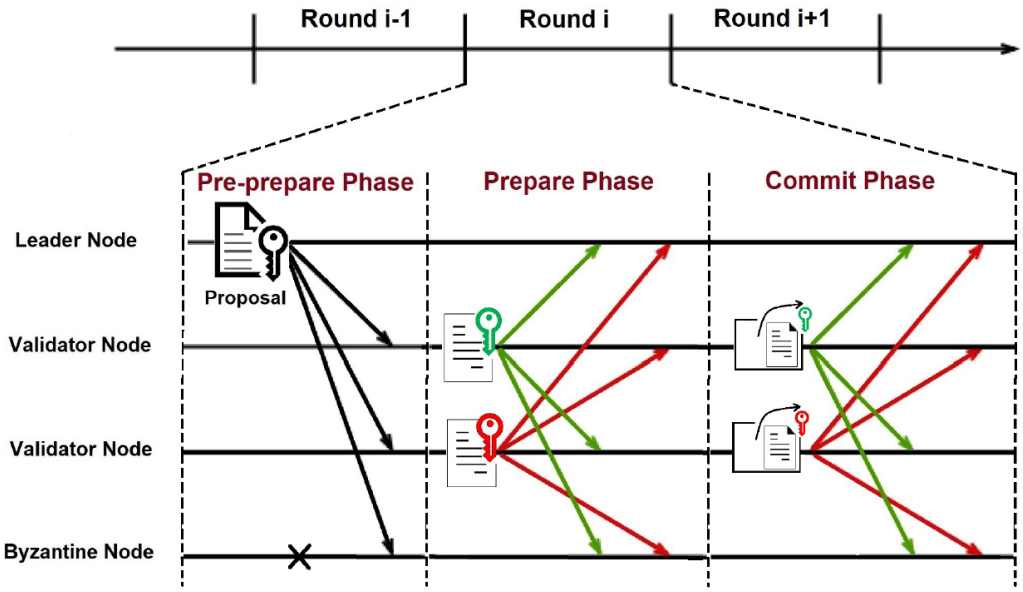


Figure 4.5: PBFT phases and messages [8]

Chapter 5

PoC implementation

In this Chapter, all implementation aspects of proposed framework are introduced. It describes actual implementation of accumulator and trust weights, and how PKCS#11 standard has been integrated to provide communication towards a Hardware Secure Module. Used libraries and implemented data structures are presented, too.

5.1 Project characteristics

This thesis project was entirely developed in Python 3.9 [68]. Python has been chosen instead of other programming languages (e.g. C, C++) since it provides a particularly suitable way for developing simple applications, in an object-oriented fashion. Moreover, nodes in DBPKI should be able to establish communication channels towards a HSM in order to use its functionalities, and this can be done by means of low-level C programming, or by means of the existing Pycryptoki Python library [69], as done in this project. Pycryptoki is later described in section 5.5.

Developed Proof of Concept has been designed to run on a single machine at the same time, as already explained in Chapter 4. For this reason, main Python script to be launched is the one containing DBPKI User Interface code that, after a brief initialization, takes care of helping the user to build its

own custom PKI. Initially, it requires the user to choose how many Root units are needed, and after that selection, it instantiates them as sub-processes (i.e. by means of `subprocess` library), each one running the DBPKI node Python script. After that, all those sub-processes wait for DBPKI UI to finish initialization and to start the sequence of rounds characterizing this framework, which was previously described.

5.2 Simulation database

Each node in DBPKI should be able to obtain some information on the network by itself. For instance, it should know, in some way, which are the public keys of other nodes. This could be done by requesting and then caching all those keys locally. For the sake of simplicity, this PoC implements a database containing all needed information, common to all the nodes, and to DBPKI User Interface, too. It is a simple MySQL [70] database (version 9.1, distribution 10.3.32-MariaDB) allowing nodes to obtain public keys and their related data, by means of specific queries. Structure of proposed solution and usage of the database is later shown in Figure 5.5. Used database contains a single table named `pubkeys`, including the following columns:

- `id`: Identifier of the row in the table. This field contains integer values used for database management but it has no role in DBPKI implementation actually. It is the primary key of the table.
- `node_id`: Identifier of the DBPKI nodes, as integer type. Each node identifier is unique.
- `cur_user_id`: Identifier of the current `User` object related to a certain node. A `User` object is used as reference of a node when working on the accumulator, as better explained in section 5.3. Its value is unique and has integer type.
- `trust_weight`: Current trust weight of the node in DBPKI, as integer value.
- `type`: Field containing a value describing the role of current node, which could be Root (`R`), Intermediate (`I`) or Ordinary (`O`).

- `value`: This field contains a string of bytes referencing current value of node's public key stored in HSM. It refers to the last value of node's public key value only, remembering that a node can have just one public key at a time, by design. So, if public key has been updated by means of Update procedure, the new enrolled key is present only, while the old one is not.

- `updated`: This field is just used as a boolean flag, letting user know whether a certain node's public key has been updated at least once or not.

- `status`: It contains a value treated as boolean, which describes current status of nodes' public key. It can assume two values indicating if a node's identity has been revoked, or if it is still valid.

Some of the columns contained in that table of the database, as `trust_weight` and `status`, are indeed redundant since their values could be simply retrieved through blockchain. However, they are included anyway in order to allow user better understanding current state of DBPKI, when working with its related UI. An example of how the values contained in `pubkeys` table are displayed to the user of the framework is available later in Figure 5.6.

5.3 Accumulator

Accumulator used in proposed PoC has been inspired to the work of Reyzin et al. [44], and then implemented in Python programming language. Its aim is to accumulate public keys and generate correspondent witnesses, used to later verify them as a proof of membership. It is basically composed by a list containing all the roots of the Merkle Trees composing the accumulator. How the roots are created and moved, and how elements are added to the accumulator is explained in Chapter 4.4. Each of the Merkle Trees are composed by a set of `AccumulatorNode` objects, of which the roots are part. Any time a new item is accumulated in one of the leaves of a Merkle Tree, a witness is generated. That proof is a list of hashes needed to reconstruct the authentication path for its corresponding element, and it is also used in order to update other witnesses. In proposed framework, witnesses are stored in

`Proof` objects and intrinsically tied to `User` objects, which are needed for collecting data about the users of the accumulator.

In addition to adding or removing a certain item, accumulator object also offers other functionalities like dichotomic search of a leaf node given its value, or the retrieving of all proofs for values accumulated in a specific Merkle Tree, or the printing of current accumulator structure. Obviously, it provides also a way for updating or verifying one or more witnesses.

Figure 5.1 shows an example of printing operation performed at a certain time on accumulator. In that representation, a root is equal to `None` (i.e. $\perp$) when its Merkle Tree is empty, while `Left` and `Right` help in drawing actual tree structure. Each accumulator node is represented as a couple, containing its hash value in hexadecimal format and the nominative of its author between brackets. The latter is optional and, if present, it is obviously shown in leaves only, since these are the Merkle-tree nodes containing the accumulated values. Moreover, each accumulator leaf corresponding to a revoked key is associated to a byte string which value is "REVOKED".

5.3.1 Accumulator node

An `AccumulatorNode` is a Python object representing a node in one of the Merkle Trees currently present in the accumulator. It could be a root, a leaf or one of the accumulator nodes between those two. It points to its ancestor and to its neighbors (i.e. children) nodes, and it owns the following attributes:

- `hash`: Current value contained inside the node. It is part of one or more authenticating path. If node is a root, it is the highest part of authenticating paths. If node is a leaf, it is the hash of an accumulated element. Otherwise, if node is in the middle, it contains a central piece of those paths. It can assume $\perp$ value if corresponding element has been removed from accumulator, or if node is a root and the tree is empty. Hashing could be performed by means of Pycryptodome Python library [71], or by means of a HSM. Each hash value in the Proof of Concept has been computed by means of SHA-256 algorithm. There are a lot of algorithms for hashing data which are more efficient than SHA-256,

```
root 0 = None

root 1 = | '\x9ah\x90K\x9d0z_...'() |
 |
 --Left---> | '\x91\x1e\xd8\r\x80\xe7\xc7\xc5...'(alice) |
 --Right--> | 'E\xdf\x13\x97r\xa8\x1b`...'(alice) |

root 2 = None

root 3 = | '\x83\x18\xe9]\xd8\xcb\xb7q...'() |
 |
 --Left---> | '*\x9e#\x1b\x12\x96\xd6\x95...'() |
         --Left---> | '\xc6\xa1\xcf\xed\xdd\xdd\xcb\x05...'() |
                 --Left---> | 'REVOKED_...'(alice) |
                 --Right--> | '\xf3\xb2\xce\x1aH{\xc8\x87...'(bob) |
         --Right--> | 'wYia\x8dt\x15C...'() |
                 --Left---> | '\xa7\x10\x056\x8a\xac\xd8\xe5...'(alice) |
                 --Right--> | '\xd9N\xad\x0c\x8b\xcd\xbf ...'(charlie) |
 --Right--> | '\xb3\xe0N\x95P\x94\n|...'() |
         --Left---> | '\xa9\xd5\xdb17\x85\xa3B...'() |
                 --Left---> | ';\xaa\xb3C[\x08<\x00...'(dave) |
                 --Right--> | '\x8b\x10\xa0\xe4\x8cR\xf4Y...'(bob) |
         --Right--> | '\xe6Qp\x98+\xbetF...'() |
                 --Left---> | 'REVOKED_...'(alice) |
                 --Right--> | '\xdd\xdb1O\xa4\x91Z\xfb...'(alice) |
```

Figure 5.1: Example of printing accumulator structure

which was selected for the sake of simplicity. Any other algorithm could be used, too.

- `left`: Pointer to the left child of current accumulator node. It is an accumulator node itself. It could be `None` if left child is not present yet.
- `right`: Pointer to the right child of current accumulator node. It is an accumulator node itself. It could be `None` if right child is not present

yet.

- `ancestor`: Pointer to the father of current accumulator node. It is an accumulator node itself. It could be `None` if current node is one of the roots of Merkle Trees.

- `author`: This attribute contains the `User` object corresponding to the author of the accumulated item. If node is not a leaf, then it is `None`.

- `order_id`: Unique identifier indicating the order in which elements are added to the accumulator. It grows sequentially and it is present in leaf nodes only.

Among others, accumulator nodes provide methods to recalculate their own hashes, in case an element has been added or deleted, and methods for computing witnesses and assigning them to the right `User` object.

5.3.2 User

As already mentioned, a `User` is a Python object referring to the author of a certain adding operation and containing its related witness. For each data to be accumulated, a new `User` object will be needed. For this reason, it also provides some attributes to distinguish the actual DBPKI node which performed the adding procedure. Its attributes are:

- `unique_id`: Sequential and unique identifier of `User` objects.

- `user_id`: Identifier of the actual user (i.e. DBPKI node) which has performed the adding procedure, so generating this `User` object.

- `name`: Alphanumeric string indicating the identity of the DBPKI node which has performed the related adding procedure. It just provides to user of this PoC an useful tool for identifying the right author of a certain adding operation, so its use is optional.

- `proof`: Witness generated at the end of the adding procedure related to this `User` object.

Since a new `User` is needed for each new adding procedure, its `user_id` and `name` parameters can be used to distinguish which DBPKI node is the real author of that operation. For instance, if DBPKI node with identifier ID= 6 and "Alice" as nominative, has performed the 17^{th} adding procedure in this framework, the related `User` item will have `unique_id` = 16 (since first `unique_id` is equal to 0), `user_id` = 6 and `name` = *Alice*. These attributes could be useful, for example, when printing the current structure of accumulator, as the one already presented in Figure 5.1.

5.3.3 Witness

As already pointed out, witnesses are represented by lists of arrays of bytes, containing a proof used to verify the membership of the related public key. Any time a witness has to be updated, due to an adding or removing operation performed on accumulator, new generated value is appended to the proof itself. For this reason, witnesses are implemented as a Python subclass of `list` default class. Indeed, having combined the use of this type of accumulator to the blockchain, only the valid proofs are saved in the latter. Moreover, a list of all currently valid witnesses is available in each block of the chain.

5.3.4 API

Accumulator has been developed individually, in order to make this thesis project independent from the chosen accumulator implementation. Thus, proposed framework offers an API allowing interoperability between the accumulator and the rest of the project. Functions provided by that accumulator API are the ones corresponding to the main accumulator functionalities:

- `acc_gen`: Generates a new accumulator object.
- `acc_add`: Given item x and an accumulator object a_i, it creates a new `User` u_x and accumulates x, by binding it to u_x. Then, it returns new accumulator a_{i+1} and the witness related to the just performed adding operation.

- `acc_ver`: Verifies whether a certain item x has been added to the accumulator or not, by knowing which is its related `User` object, and so which is its membership proof.
- `acc_ver_any`: Verifies whether a certain item x has been added to the accumulator or not, by going to trial and error with a list of valid witnesses. This functionality could be useful in case a node in DBPKI wants to verify a certain public key, without searching for the right `User` object within the accumulator.
- `acc_del`: Given an accumulator object a_i and an item x to be deleted from a_i, it searches for a leaf containing value corresponding to x, and then it removes it. In the end, it returns the updated accumulator a_{i+1}. After deletion all witnesses has to be updated accordingly.

5.4 Communication protocol

As already pointed out in Chapter 4.5, nodes in DBPKI have to reach a common decision about the current operation, by sending some messages in multicast to all other nodes in consensus group. Any communication protocol providing multicast could be chosen for that purpose. For instance, simple UDP messages could be used to establish those kind of communications. However, for reasons of low computational power, some UDP messages could be lost while DBPKI nodes are still busy finishing some operations. This is due to the fact that UDP is connectionless and delivery for the data is unreliable [72], and also because all of those nodes are run as subprocesses on a single machine, as already described in section 5.1. For all these reasons, a TCP-based communication protocol has been chosen for proposed Proof of Concept. The choice regarding the sending of PBFT messages fell on the MQTT protocol, since that is one of the most used in IoT communication [73], even in the automotive field [74]. Moreover, MQTT offers different QoS levels, according to the usage of the system. For this Proof of Concept, QoS level 2 has been selected, meaning that messages are delivered exactly once to subscribers.

MQTT is a communication protocol based on the concepts of topic publication and subscription. Any entity which would like to send some data,

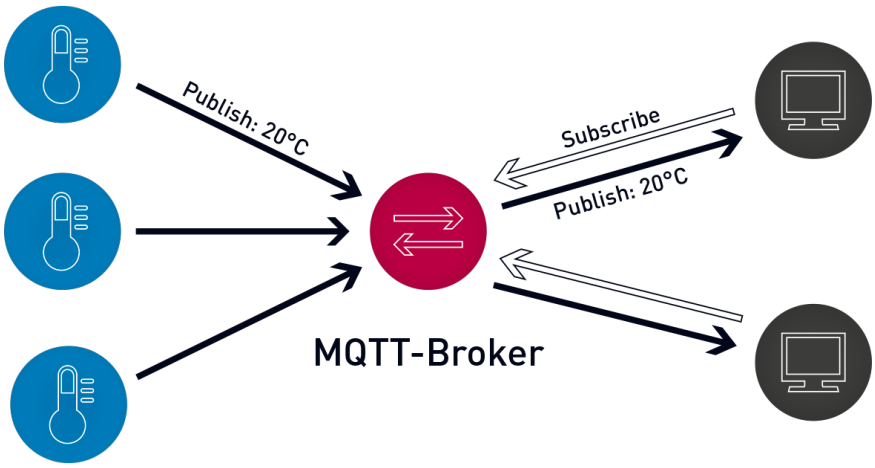


Figure 5.2: MQTT architecture example (*Source*)

has to publish to a certain posting queue (i.e. a topic) a message containing those data. Instead, any entity which would like to receive all messages related to a certain topic, has to subscribe to it. MQTT requires the use of a broker, which task is to coordinate all subscriptions and publications, making sure that each entity subscribed to a certain topic receives all the messages related to it. MQTT broker chosen for this PoC is *mosquitto* [75] (version 1.6.15), an open-source and lightweight message broker. Figure 5.2 shows an example of MQTT architecture in which some devices (e.g. digital thermometers) publish detected temperatures to a specific topic, two MQTT clients are subscribed to that topic, and the broker takes care of distributing the messages to the correct recipients.

MQTT broker has undoubtedly a central role in the structure of this communication protocol. For this reason, it introduces a single point of failure in MQTT architecture. If it stops working properly due to the malicious actions of an attacker, all MQTT clients stop receiving updates on system status. However, considering that purpose of this thesis is to create a PKI working in a distributed fashion without the need of Certificate Authorities, the choice of the communication protocol does not compromise the functionalities offered by DBPKI. That choice therefore remains independent of proposed solution, as any other multicast protocol could be used, too.

In Figure 5.3 an extract from the log file generated by MQTT broker during a round execution is presented. It represents an example situation

in which there are 3 active nodes in DBPKI (2 Root and 1 Intermediate), which are passing through PBFT steps for approving or rejecting a public key update procedure. The image shows leader node sending its new proposal in `dbpki/pbft/pre_prepare` topic and, later, it also shows MQTT broker sharing that pre-prepare message to all DBPKI nodes, since they are subscribed to that topic from enrolling time. After that, prepare and commit messages are exchanged (in `dbpki/pbft/prepare` and `dbpki/pbft/commit` topics), containing the decisions about the approval of current block proposal. In the end of the round, log file reports that DBPKI UI takes care of finishing the round by sending a specific message on `dbpki/pbft/round_change` topic. The latter will be then followed by a new message on `dbpki/pbft/new_round` topic, when the next round begins.

All messages sent within developed PoC are encoded by sender, sent by means of MQTT communication protocol, and later decoded by receivers. Encoding and decoding of a certain message depends on which of the topics it belongs to. Messages to be sent contain a lot of data like integers, strings, byte arrays, or even entire lists or data structures. Those data are organized in a single message following specific criteria, and divided by several separator items. Separators consist in different strings of bytes, each one corresponding to a specific level of the message encoding. For example, Figure 5.4 shows how the encoding of a standard pre-prepare message of a certain round in DBPKI is structured. In that image, a model of a pre-prepare message is presented. The latter contains some simple data like a timestamp, an identifier, a signature and a hash, and some other more complex structures like an entire accumulator and some Python lists. For instance, the accumulator is encoded in levels, starting from the roots of the Merkle Trees, until their leaves. The one contained in that picture is a simple summary representation, where each separator is represented by one or more special characters, corresponding to long and specific string of bytes in the real PoC implementation.

5.5 HSM and PKCS#11 integration

Regarding the creation and management of cryptographic keys and signatures, proposed framework exploits functionalities of a HSM. The used HSM is a password-authenticated Thales SafeNet Luna A750 [29], and it was provided

by Security Reply S.r.l. company. Its usage is preferred to the software library called Pycryptodome, since HSMs provide great security capabilities, anti-tampering properties and numerical generation which is not pseudo-random, but actually random. Some of provided functionalities are indeed based on the use of a True Random Number Generator (TRNG), as mentioned in Chapter 2.2.1.

As already said in Chapter 2.2.2, in order to allow this framework co-operating with a HSM, an API based on PKCS#11 standard should be used. Pycryptoki library [69] has been chosen for that purpose. This open-source Python library aims to simplify PKCS#11 library, which is written in C programming language, by providing functions using Python's *ctypes* library. Some examples of functions provided by Pycryptoki library are `c_generate_key_pair_ex`, which is used when a new key pair has to be created, or `c_sign_ex` and `c_verify`, which are used for signing some data and verify a signature, respectively. According to used standard, each of those functions has to be preceded by an initialization and the opening of a new session, and has to be followed by a finalization operation and the closing of that session. Typically, functions provided by Pycryptoki needs a *mechanism* in order to perform required operations. Proposed solution uses mechanisms based upon RSA, both for key generation and signing of data. Any other available mechanism could be used, too.

In the Proof of Concept's architecture, shown in Figure 5.5, it is possible to notice that a single HSM has been used for this thesis project. Each node in DBPKI connects itself to that Thales Luna HSM in order to request some operations, like the signing of some data. Obviously, each DBPKI node could rely on a different HSM in a real implementation of this solution.

Figure 5.5 presents a conceptual representation of implementation design of proposed solution, where:

- **User** represents the user of the developed PoC, which gives commands through scripts or command line, interacting with DBPKI UI.

- **DBPKI UI** is the User Interface presented in Chapter 4.1, which is in charge of allowing the user to interact with the system, initializing the DBPKI and managing its rounds.

- **R** (Root), **I** (Intermediate) and **O** (Ordinary) elements constitute an example representation of all the nodes in the PKI. As explained in Chapter 4.2, Root and Intermediate nodes are actually in DBPKI, while Ordinary ones are not part of consensus group.
- **MySQL DB**, **MQTT Broker** and **HSM** are the external tools used to make the proposed framework work properly. For simplicity, they are in common to all the nodes, as said before.

5.6 Trust weights

As introduced in Chapter 4.7, this thesis project proposes the use of dynamic trust weights, in order to enhance trust relationships among DBPKI entities. Any operation that can be performed on blockchain has to be approved by a minimum number of nodes in consensus group. That minimum threshold is not only a fixed value depending on current number of nodes in DBPKI, as proposed by Toorani and Gehrmann [8]. In this framework indeed, that threshold is computed at each round also considering the value of trust weight that all nodes (both leader and validators) have reached.

Starting from an initial trust weight value gained when enrolled, each node in DBPKI has its own trust weight increasing or decreasing according to the results of operations performed on blockchain, as the rounds progress. As already seen in Chapter 4.7, a new block can be successfully committed to blockchain if and only if threshold $T = (2\lfloor(t-1)/3\rfloor + 1) \cdot w^i_{avg}$ is reached, where t is the current total number of nodes in DBPKI, and w^i_{avg} is the average value of trust weight of all nodes, excluding leader one. At each round, if blockchain has to be modified, leader node checks whether the current maximum reachable trust value is equal or greater than that threshold T. If value of threshold T is not reached, operation requested by leader node will be rejected from consensus group. Moreover, if maximum trust value has not been achieved, it means that some of the nodes in DBPKI are faulty or malicious. In this way, DBPKI security is enhanced since nodes which have gained a high trust value, have a greater weight in the network.

Figures 5.6 and 5.7 show two screenshots taken from DBPKI UI, presenting example situations in which threshold T is reached and not, respectively.

These two scenarios assume that all non-revoked nodes are trustworthy (i.e. non-faulty and non-malicious). In the first case, consensus group accept the proposal made by leader node since, even if some nodes are revoked, their trust weights have low impact on the outcome of current operation. Hence, in first case the network of DBPKI nodes is able to converge to a positive decision. On the other hand, second picture presents a round in which consensus group is not able to accept leader's proposal since threshold T is not reached, because too many nodes have been revoked, and therefore the system loses its decision-taking ability. Both images contain a forecast on the result of requested procedure based on current maximum reachable trust value, a table showing status of each node in PKI, the sequence of completed consensus phases and the final result of the operation.

Proposed PoC assumes that maximum trust weight value reachable by a single DBPKI node is equal to 50, while the minimum is set to 0. The latter is also the value of trust weight of revoked nodes. Having these basic rules in mind, management of trust weights and the reward-and-punishment mechanism are explained below.

5.6.1 Initial trust weight

When enrolled, any node obtains an initial value for its trust weight in the network, according to which role it has. If node is Root (`R`), it starts with a trust weight of value 10, if it is Intermediate (`I`) its value will be 5, while if it is an Ordinary (`O`) node, its trust weight will be always equal to 0. In this way, Root nodes have a heavier weight in the beginning, thus creating a pyramidal trust system which is then flattened with the growth of the PKI, since only more Intermediate nodes will be added to consensus group. Ordinary nodes instead do not need a trust weight because they are not actually part of DBPKI, since they can not take part in PBFT consensus mechanism.

5.6.2 Trust weighted on time

Developed Proof of Concept proposes a system that proceeds in rounds, alternating the work of DBPKI UI and the nodes, as already explained in Chapter 4 of this document. At each round, a new leader node performs a certain operation on blockchain or a public key verification. Any time a new block to be appended to the chain is accepted or rejected by consensus group, trust weight value of all nodes in DBPKI (Root and Intermediate ones) is increased by 1. Thus, trust weights are weighted on time, since as rounds passes, the nodes that have been in the network from a longer time, will have a greater trust value. This feature allows nodes that have been present for the longest time to consolidate their level of trust, and also to create a time frame in which newly enrolled nodes must earn the trust of the system.

5.6.3 Reward and punishment mechanism

An additional feature implemented in proposed framework is the appliance of a custom reward-and-punishment mechanism which aims to incentivize or disincentivize nodes in DBPKI, according to the results of their performed operations on blockchain. When a DBPKI node takes the leadership of current round, it is able to propose a new block to be committed to blockchain. In that case, consensus group can approve or reject the proposed block. If proposal has been accepted, it means that leader unit was able to make itself trustworthy in the eyes of validator nodes. Hence, in that case, block is successfully committed to blockchain and leader node's trust weight value is increased by 1 as a reward for its success (in addition to the increase due to the functionality already described in section 5.6.2). Otherwise, if proposal has been refused by other nodes in DBPKI, leader is punished by means of a decrease in its trust weight of value 2. Those reward and punishment constitute the implemented custom mechanism which helps the network to exclude elements that DBPKI nodes do not trust, and to reward the ones that have earned the necessary value of trust in the framework.

```
1660299045: Received PUBLISH from auto-9262971B-1083-BC19-2371-FB8407AC3A9A (d0, q2, r0, m7, 'dbpki/pbft/pre_prepare', ... (2237 bytes))
[...]
1660299045: Sending PUBLISH to auto-F5643A67-21FA-9EE8-6AD7-EB502E83A1B7 (d0, q2, r0, m34, 'dbpki/pbft/pre_prepare', ... (2237 bytes))
1660299045: Sending PUBLISH to auto-9262971B-1083-BC19-2371-FB8407AC3A9A (d0, q2, r0, m34, 'dbpki/pbft/pre_prepare', ... (2237 bytes))
1660299045: Sending PUBLISH to auto-2E01BB04-3DB2-7F3B-5386-B1DDE7D87AE1 (d0, q2, r0, m34, 'dbpki/pbft/pre_prepare', ... (2237 bytes))
[...]
1660299047: Received PUBLISH from auto-F5643A67-21FA-9EE8-6AD7-EB502E83A1B7 (d0, q2, r0, m8, 'dbpki/pbft/prepare', ... (270 bytes))
[...]
1660299047: Received PUBLISH from auto-2E01BB04-3DB2-7F3B-5386-B1DDE7D87AE1 (d0, q2, r0, m8, 'dbpki/pbft/prepare', ... (270 bytes))
[...]
1660299047: Sending PUBLISH to auto-F5643A67-21FA-9EE8-6AD7-EB502E83A1B7 (d0, q2, r0, m35, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-9262971B-1083-BC19-2371-FB8407AC3A9A (d0, q2, r0, m35, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-2E01BB04-3DB2-7F3B-5386-B1DDE7D87AE1 (d0, q2, r0, m35, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-F5643A67-21FA-9EE8-6AD7-EB502E83A1B7 (d0, q2, r0, m36, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-F5643A67-21FA-9EE8-6AD7-EB502E83A1B7 (d0, q2, r0, m37, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-F5643A67-21FA-9EE8-6AD7-EB502E83A1B7 (d0, q2, r0, m38, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-9262971B-1083-BC19-2371-FB8407AC3A9A (d0, q2, r0, m36, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-9262971B-1083-BC19-2371-FB8407AC3A9A (d0, q2, r0, m37, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-9262971B-1083-BC19-2371-FB8407AC3A9A (d0, q2, r0, m38, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-2E01BB04-3DB2-7F3B-5386-B1DDE7D87AE1 (d0, q2, r0, m36, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-2E01BB04-3DB2-7F3B-5386-B1DDE7D87AE1 (d0, q2, r0, m37, 'dbpki/pbft/prepare', ... (270 bytes))
1660299047: Sending PUBLISH to auto-2E01BB04-3DB2-7F3B-5386-B1DDE7D87AE1 (d0, q2, r0, m38, 'dbpki/pbft/prepare', ... (270 bytes))
[...]
1660299052: Received PUBLISH from auto-F5643A67-21FA-9EE8-6AD7-EB502E83A1B7 (d0, q2, r0, m9, 'dbpki/pbft/commit', ... (270 bytes))
[...]
1660299052: Sending PUBLISH to auto-F5643A67-21FA-9EE8-6AD7-EB502E83A1B7 (d0, q2, r0, m39, 'dbpki/pbft/commit', ... (270 bytes))
1660299052: Sending PUBLISH to auto-9262971B-1083-BC19-2371-FB8407AC3A9A (d0, q2, r0, m39, 'dbpki/pbft/commit', ... (270 bytes))
1660299052: Sending PUBLISH to auto-2E01BB04-3DB2-7F3B-5386-B1DDE7D87AE1 (d0, q2, r0, m39, 'dbpki/pbft/commit', ... (270 bytes))
[...]
1660299052: Received PUBLISH from auto-2E01BB04-3DB2-7F3B-5386-B1DDE7D87AE1 (d0, q2, r0, m9, 'dbpki/pbft/commit', ... (270 bytes))
[...]
1660299052: Sending PUBLISH to auto-F5643A67-21FA-9EE8-6AD7-EB502E83A1B7 (d0, q2, r0, m41, 'dbpki/pbft/commit', ... (270 bytes))
1660299052: Sending PUBLISH to auto-9262971B-1083-BC19-2371-FB8407AC3A9A (d0, q2, r0, m41, 'dbpki/pbft/commit', ... (270 bytes))
1660299052: Sending PUBLISH to auto-2E01BB04-3DB2-7F3B-5386-B1DDE7D87AE1 (d0, q2, r0, m41, 'dbpki/pbft/commit', ... (270 bytes))
[...]
1660299057: Received PUBLISH from auto-E91CE651-D9CD-5146-D5C4-C53D0C15BE59 (d0, q2, r0, m13, 'dbpki/pbft/round_change', ... (16 bytes))
```

Figure 5.3: MQTT broker log file example

```
TIMESTAMP | BLOCK_ID | Accumulator | Proofs | Transactions | Trust weights
    | SIGNATURE | NEW_BLOCK_HASH
  Where:
     Accumulator format: - r0 - NONE - r1 - NONE - r2 - ... - rn - & NODES_NUMBER
         Where r format is:  r > f : lev1_left +++ lev1_right
             Where lev1 format is:  lev1 > f : lev2_left ++ lev2_right
                 Where lev2 format is:  lev2 > f : NO_MORE_NODES + lev3_right
                     and so on... ('f' indicates how many children the node has)

     Proofs format:          - w0 - w1 - w2 - w3 - ... - wn -
         Where w format is:  ^ p0 ^ p1 ^ ... ^ pn ^

     Transactions format:    - tr0 - tr1 - tr2 - ... - trn -
         Where tr format is:  unit_id + pk + flag + role

     Trust weights format:  - node_id0 : trust_w0 - node_id1 : trust_w1 - ... -
```

Figure 5.4: Pre-prepare message structure

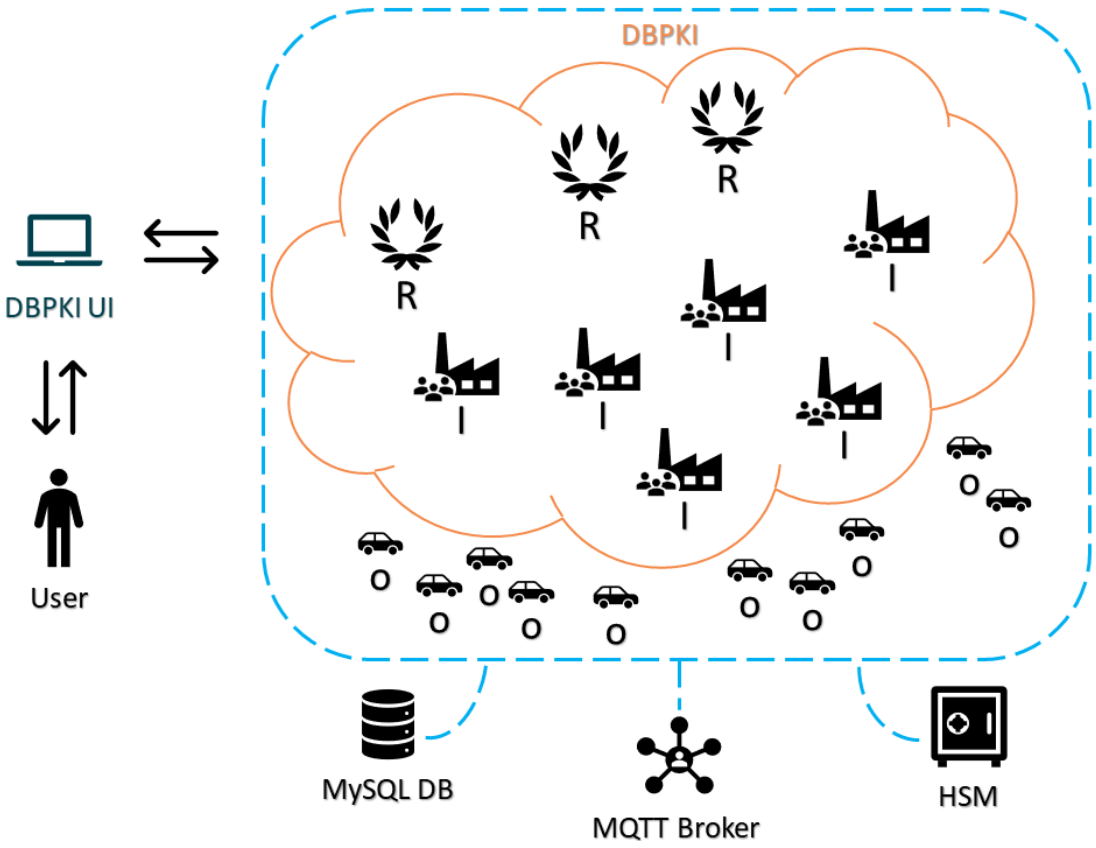


Figure 5.5: PoC model architecture

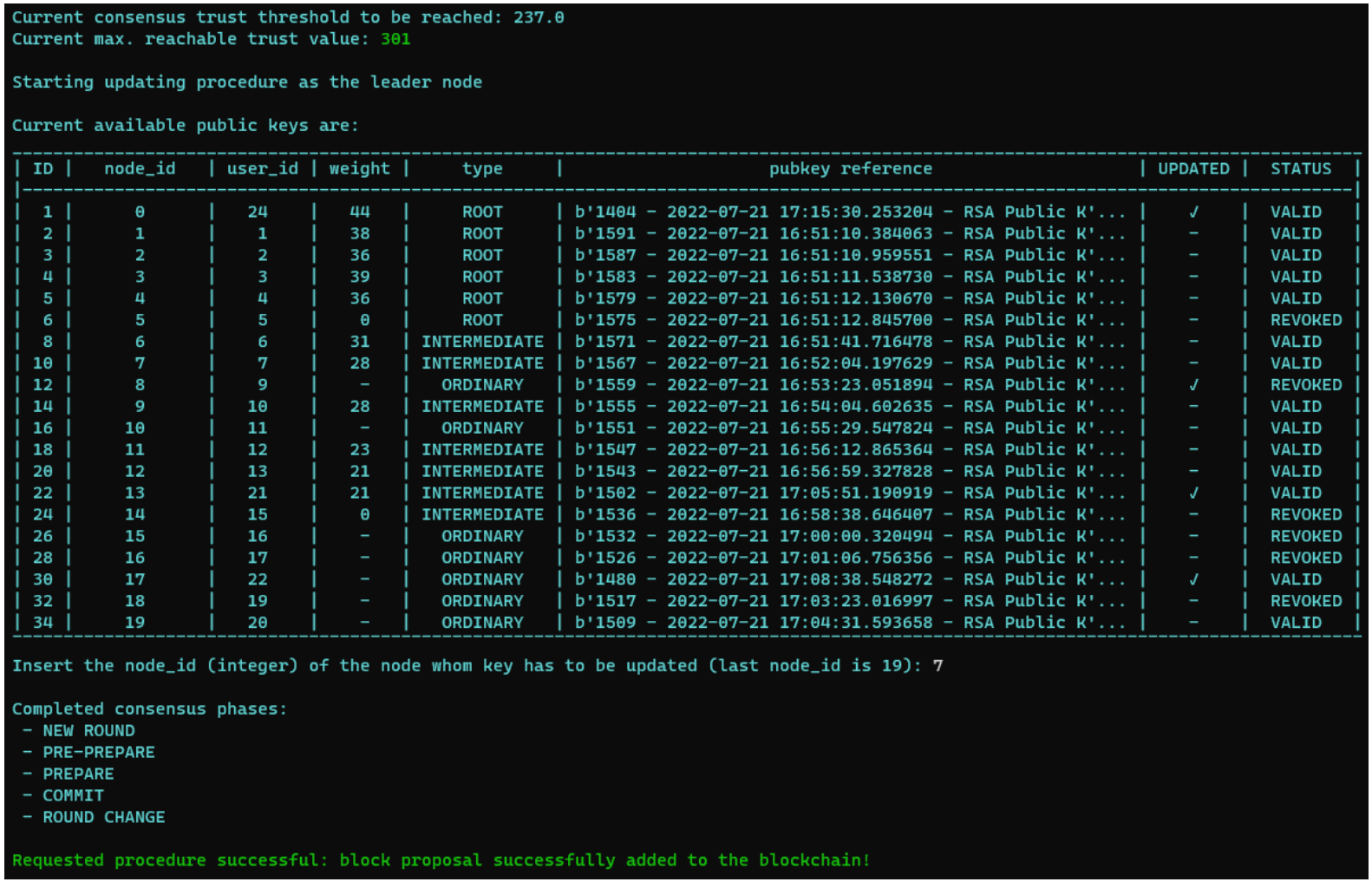


Figure 5.6: DBPKI screenshot success example

```
Current consensus trust threshold to be reached: 165.75
Current max. reachable trust value: 161

Starting revoking procedure as the leader node

Current available public keys are:
```

ID	node_id	user_id	weight	type	pubkey reference	UPDATED	STATUS
1	0	24	0	ROOT	b'1404 - 2022-07-21 17:15:30.253204 - RSA Public K'...	✓	REVOKED
2	1	1	45	ROOT	b'1591 - 2022-07-21 16:51:10.384063 - RSA Public K'...	-	VALID
3	2	2	0	ROOT	b'1587 - 2022-07-21 16:51:10.959551 - RSA Public K'...	-	REVOKED
4	3	3	0	ROOT	b'1583 - 2022-07-21 16:51:11.538730 - RSA Public K'...	-	REVOKED
5	4	4	0	ROOT	b'1579 - 2022-07-21 16:51:12.130670 - RSA Public K'...	-	REVOKED
6	5	5	0	ROOT	b'1575 - 2022-07-21 16:51:12.845700 - RSA Public K'...	-	REVOKED
8	6	6	37	INTERMEDIATE	b'1571 - 2022-07-21 16:51:41.716478 - RSA Public K'...	-	VALID
10	7	25	34	INTERMEDIATE	b'1394 - 2022-07-21 17:16:42.079055 - RSA Public K'...	✓	VALID
12	8	9	-	ORDINARY	b'1559 - 2022-07-21 16:53:23.051894 - RSA Public K'...	✓	REVOKED
14	9	10	34	INTERMEDIATE	b'1555 - 2022-07-21 16:54:04.602635 - RSA Public K'...	-	VALID
16	10	11	-	ORDINARY	b'1551 - 2022-07-21 16:55:29.547824 - RSA Public K'...	-	VALID
18	11	12	29	INTERMEDIATE	b'1547 - 2022-07-21 16:56:12.865364 - RSA Public K'...	-	VALID
20	12	13	0	INTERMEDIATE	b'1543 - 2022-07-21 16:56:59.327828 - RSA Public K'...	-	REVOKED
22	13	21	27	INTERMEDIATE	b'1502 - 2022-07-21 17:05:51.190919 - RSA Public K'...	✓	VALID
24	14	15	0	INTERMEDIATE	b'1536 - 2022-07-21 16:58:38.646407 - RSA Public K'...	-	REVOKED
26	15	16	-	ORDINARY	b'1532 - 2022-07-21 17:00:00.320494 - RSA Public K'...	-	REVOKED
28	16	17	-	ORDINARY	b'1526 - 2022-07-21 17:01:06.756356 - RSA Public K'...	-	REVOKED
30	17	22	-	ORDINARY	b'1480 - 2022-07-21 17:08:38.548272 - RSA Public K'...	✓	VALID
32	18	19	-	ORDINARY	b'1517 - 2022-07-21 17:03:23.016997 - RSA Public K'...	-	REVOKED
34	19	20	-	ORDINARY	b'1509 - 2022-07-21 17:04:31.593658 - RSA Public K'...	-	VALID

```
Insert the node_id (integer) of the node whom key has to be revoked (last node_id is 19): 6

Completed consensus phases:
 - NEW ROUND
 - PRE-PREPARE
 - PREPARE
 - COMMIT
 - ROUND CHANGE

Requested procedure failed: block proposal has not been accepted by the consensus group...
```

Figure 5.7: DBPKI screenshot fail example

Chapter 6

Results and validation

This Chapter presents an overview on experimental results of this thesis project, and a validation of the realized Proof of Concept in respect to expected requirements.

6.1 The DCS theorem

The Decentralized Consensus Scale (DCS) theorem, studied by G. Slepak and A. Petrova [76], shows how properties of any decentralized system must follow the structure defined by the DCS triangle, on whose vertices there are three different key concepts [76]:

- *Decentralization*: System has no single point of failure and, if any element of it is removed, system continues to work properly.
- *Consensus*: System uses a consensus algorithm in order to take collective decisions.
- *Scale*: System is able to handle any number of elements, and so it is capable of providing the same services of any competing system.

DCS theorem demonstrates how "decentralized consensus systems like block-chains, can have *Decentralization*, *Consensus*, or *Scale*, but not all three

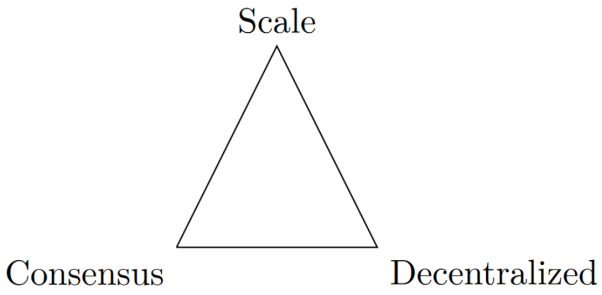


Figure 6.1: The DCS triangle [76]

properties simultaneously" [76]. Since proposed framework is decentralized and uses a consensus mechanism, it obviously focuses on Decentralization and Consensus properties, rather than on Scale.

6.2 Performance and costs

Proposed PoC has been designed to run on a single machine, which was a Linux server with 4GB of RAM and Intel Xeon Silver 4214 CPU @ 2.20GHz as processor. Due to this limitation, the Proof of Concept can instantiate a maximum of 20 Root nodes as processes during Setup. Moreover, with the increasing of number of nodes in DBPKI, the overhead of communication and consensus mechanisms consequently increases, as shown by data collected in Table 6.1, where first row contains t, which indicates the current number of nodes in DBPKI, while second row contains elapsed time for an enrolling procedure during a sample round, in seconds. Figure 6.2 also shows a more complete distribution of collected data. That measured approximate time considers creation of the key pair, the instantiating of the process, the creation of the block, the achieving of consensus and the commit in the blockchain. Presented times are quite high because, for simplicity, the Proof of Concept also subscribes the nodes to MQTT queues in which they are not interested. However, the importance of those data lies in increasing the time necessary for the convergence of the collective decisions, as the number of nodes in the consensus group increases.

t	2	4	6	8	10	12	14	16	18	20
sec.	5.95	10.76	17.94	26.28	36.13	47.28	62.33	78.51	91.63	109.40

Table 6.1: Consensus mechanism overhead

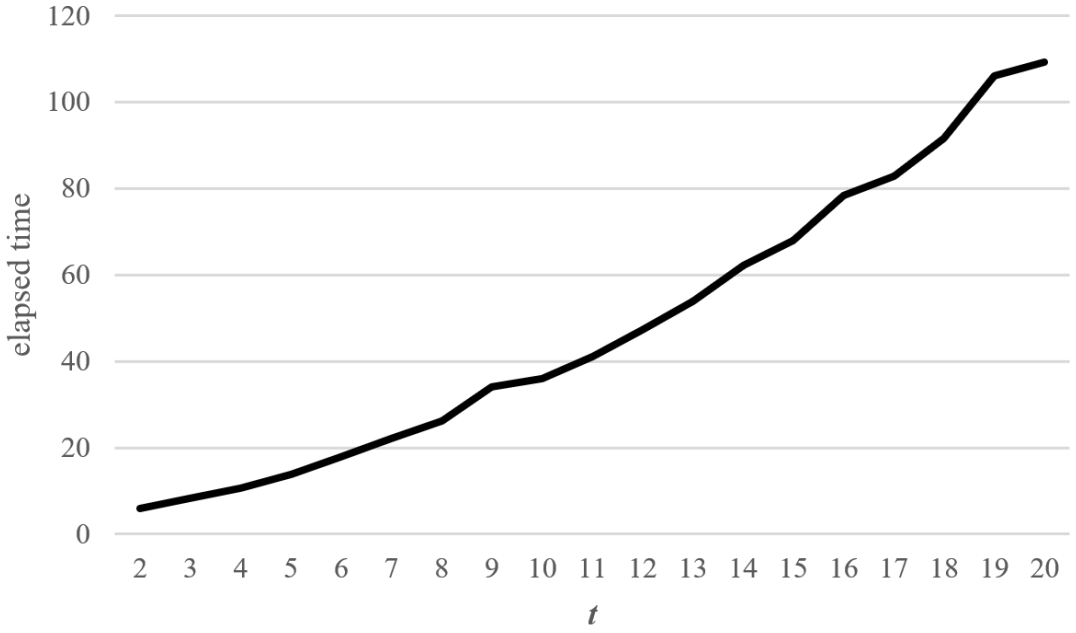


Figure 6.2: Consensus mechanism overhead graph

Consensus procedure is therefore a costly part of the framework and its cost increases with the enlarging of the DBPKI network, since each node has to send multicast messages to all other units in consensus group, as described by means of Table 6.1. Hence, PoC functionalities lie on a trade-off between security and efficiency: the more nodes in the consensus group, the better the system security, and the worse its efficiency, due to greater communication overhead [8].

Performance of proposed solution are influenced by chosen accumulator, consensus mechanism and communication protocol. Operations on accumulator deployed in the Proof of Concepts are logarithmic, as already pointed out in Chapter 4, while Table 6.2 shows computational costs of DBPKI procedures, by dividing them in needed steps. In that table, t indicates the

number of nodes in consensus group, while k represents the amount of nodes storing and validating accumulator and witnesses.

Procedure	*Sign/Veri*	*AccAdd*	*AccWitAdd*	*AccWitDel*	*AccDel*	*AccVer*
Enroll	$O(t^2)$	t	k	-	-	-
Revoke	$O(t^2)$	-	-	k	t	t
Update	$O(t^2)$	t	k	k	t	t
Verify	-	-	-	-	-	1

Table 6.2: Computational costs of DBPKI procedures [8]

6.3 Security Properties

Proposed blockchain-based PKI removes the need of digital certificates and of Certificate Authorities (CAs), since validity of cryptographic keys is granted by the blockchain itself, which is held together by a multitude of entities. There are no more central authorities able to validate certificates in a pyramidal infrastructure, as it was the traditional PKI. The lack of a validity certificate does not compromise the security of the system as it is only necessary in parallel with the deployment of CAs. Some sort of certificate could still be associated with the keys to enclose some data about them, such as an expiration time. However, the proposed solution does not provide for any expiration for the public keys, therefore, within the blockchain, only the creator of a key and its owner are stored.

Security capabilities of proposed DBPKI Proof of Concept depends on the number of nodes t in consensus group, the number f of Byzantine nodes, and all their trust weights. Assuming that deployed hash function is collision-resistant[1], that used accumulator is sound (see Chapter 2.5), and that there are no more than f faulty units, then an adversary $\mathcal{A}$ has negligible advantage in winning the following security experiments, which were defined in the

[1]A hash function is collision-resistant if it is hard to find two inputs corresponding to the same output hash.

work of Toorani et al. [8]:

- *Enroll-ValidKey-IllegitimateEntity* experiment:

$$(\text{ID}, pk) \leftarrow \mathcal{A}()$$
$$\textbf{return } 1 \textbf{ if } (\text{Enroll}(\text{ID}, pk, \texttt{flag}, \texttt{role}) = 1) \wedge (\text{Verify}(\text{ID}, pk) = \perp) \wedge (\text{PIV}(\text{ID}, pk) = 1)$$

  Adversary $\mathcal{A}$ wins the "security game" if it can enroll an illegitimate entity in DBPKI, by registering a valid public key.

- *Enroll-InvalidKey-IllegitimateEntity* experiment:

$$(\text{ID}, pk) \leftarrow \mathcal{A}()$$
$$\textbf{return } 1 \textbf{ if } (\text{Enroll}(\text{ID}, pk, \texttt{flag}, \texttt{role}) = 1) \wedge (\text{Verify}(\text{ID}, pk) = \perp) \wedge (\text{PIV}(\text{ID}, pk) = \perp)$$

  Adversary $\mathcal{A}$ wins if it can enroll an illegitimate entity in DBPKI, by registering an invalid public key.

- *Enroll-InvalidKey-LegitimateEntity* experiment:

$$(\text{ID}, pk) \leftarrow \mathcal{A}()$$
$$\textbf{return } 1 \textbf{ if } (\text{Enroll}(\text{ID}, pk, \texttt{flag}, \texttt{role}) = 1) \wedge (\text{Verify}(\text{ID}, pk) = 1) \wedge (\text{PIV}(\text{ID}, pk) = \perp)$$

  Adversary $\mathcal{A}$ wins this experiment if it can enroll a legitimate entity in DBPKI, by registering an invalid public key.

- *Update-Collision* experiment:

$$(\text{ID}, pk^*) \leftarrow \mathcal{A}()$$
$$\textbf{return } 1 \textbf{ if } (\text{Update}(\text{ID}, pk^{old}, pk^*) = 1) \wedge (\text{Verify}(\text{ID}, pk^*) = 1)$$

  Adversary $\mathcal{A}$ wins if it can update the public key of a legitimate entity with another arbitrary public key pk^*.

- *Revoke-Collision* experiment:

$$(\text{ID}, pk) \leftarrow \mathcal{A}()$$
$$\textbf{return } 1 \textbf{ if } (\text{Revoke}(\text{ID}, pk) = 1) \wedge (\text{Verify}(\text{ID}, pk) = \perp)$$

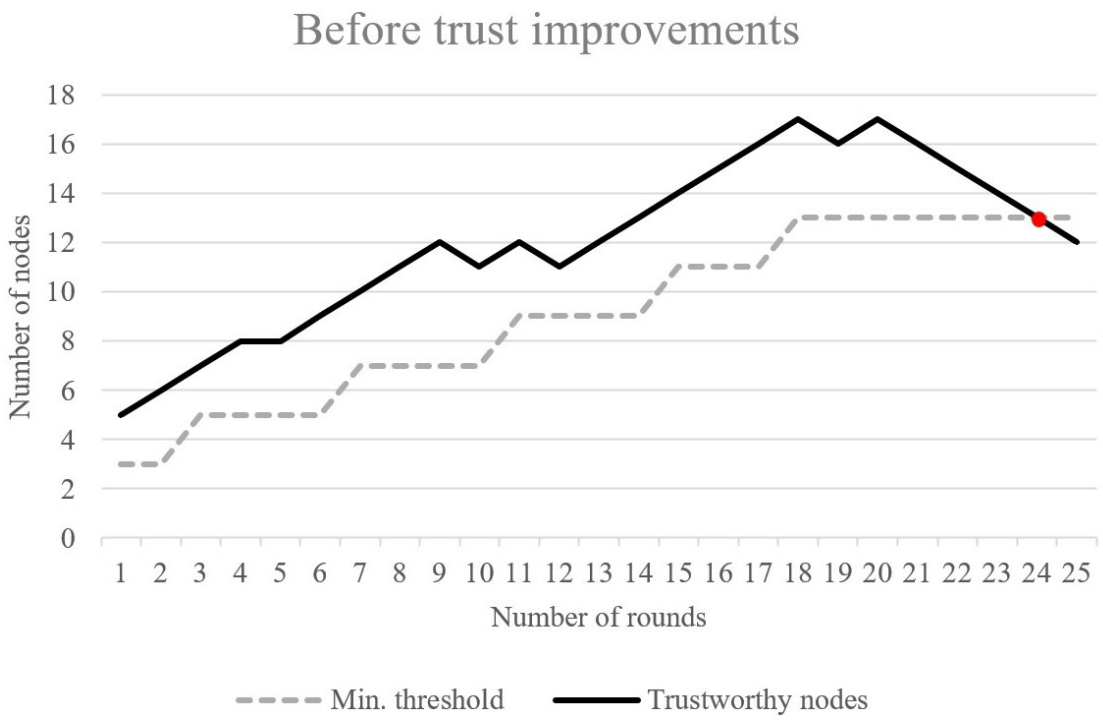
  Adversary $\mathcal{A}$ wins this game if it can revoke the public key of a legitimate entity without the consent of the network.

6.4 Trust weights analysis

The starting work of Toorani et al. [8] proposed a decentralized blockchain-based PKI, which can tolerate up to $\lfloor (t-1)/3 \rfloor$ malicious nodes inside the system. Framework described within this document proposes an enhancement of that security property. Developed PoC can tolerate a number of malicious nodes which depends on the trust value of each of the nodes in DBPKI at current time. Indeed, a trust level has to be reached in order to commit any operation on blockchain, and the value of that threshold is equal to $(2\lfloor (t-1)/3 \rfloor + 1) \cdot w^i_{avg}$, as already defined in Chapter 4.7. This developed improvement creates a mild form of centralization in the beginning, since Root units will have a higher trust level from the first round. Nevertheless, that slight centralization is quickly faded when the network grows up and the time flows, thanks to the use of trust weights and their related reward-and-punishment mechanism.

The two graphs included in Figure 6.3 show a comparison in which the same sample use case has been run without or with the use of trust weights, respectively. Those graphs present the number of rounds on the x-axis, which simulates the passage of time, and the number of nodes or value of trust on y-axis. Both graphs contain two lines: the solid line represents the trust value reached assuming that no malicious nodes are present in consensus group, while the dashed line represents the threshold to be reached to successfully commit new transactions. In the first case, before trust improvements, and so without the appliance of dynamic trust weights, solid line simply indicates the current amount of valid (non-revoked) units in DBPKI. Instead, dashed line is the curve described by $\lfloor (t-1)/3 \rfloor$ threshold, where t is the current total number of nodes participating in consensus group. On the other hand, in the second graph, solid line is defined by the actual reachable trust value, and the dashed one indicates the minimum threshold to be reached, which depends on trust weight of any single node. As shown by that comparison, the correct functioning of DBPKI version proposed by M. Toorani and C. Gehrmann strictly depends on the current amount of valid nodes and, if a lot of them are revoked (or malicious), it is soon compromised when the two lines of the graph cross. Proposed solution instead allows the system to revoke even a large number of units since, if the network is large, the difference between the threshold and the reached value of trust gets bigger and bigger

as time goes by. In this way, the relevance of a node within the network, even if it has Root role, becomes almost insignificant, thus decentralizing even more the PKI model.



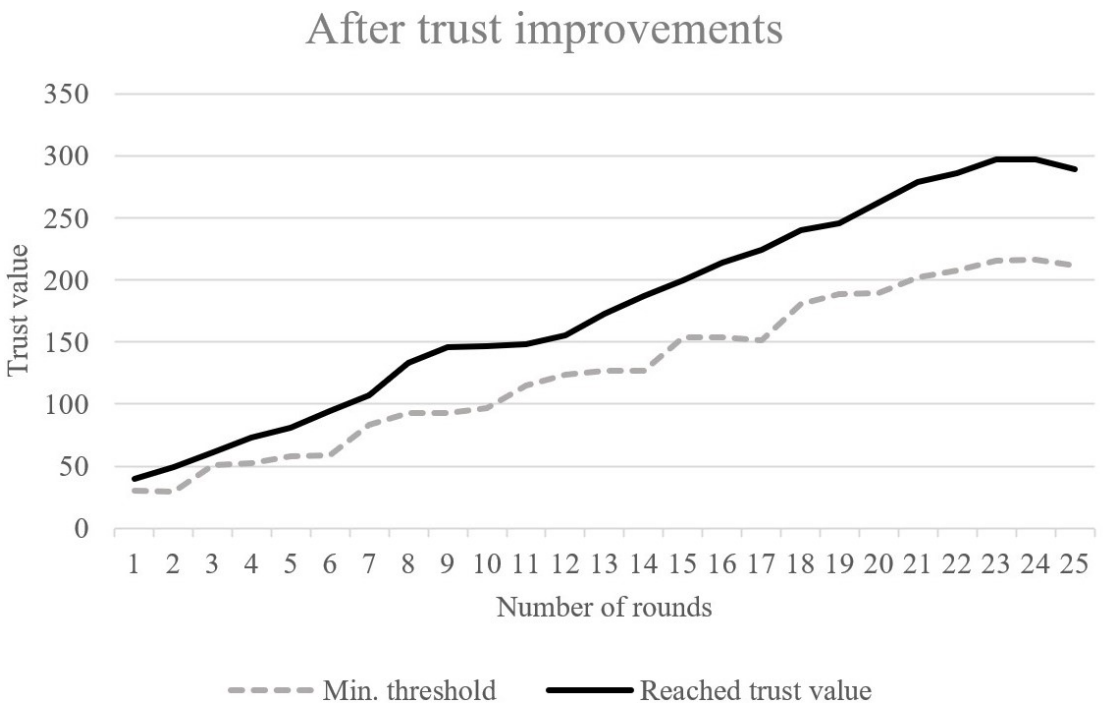


Figure 6.3: DBPKI sample case comparison

Chapter 7

Conclusion

This Master's thesis started by focusing on the research for alternatives to traditional centralized Public Key Infrastructures, in order to create a Proof of Concept for a decentralized infrastructure, that could be applied for IoT and similar distributed environments. A lot of past works have been analyzed, and Toorani and Gehrmann's one [8] was the most inspirational, since they proposed a blockchain-based PKI which is totally decentralized. After that first phase of scouting, the purpose of this thesis has therefore become to develop in a testing environment the DBPKI model that Toorani et al. had proposed only conceptually, and to possibly improve some of its features. The latter is an important contribution of this work, indeed an enhancement based on the use of dynamic trust weights has been implemented within the developed PoC.

The entire programming of the PoC was made in Python and it was based on the use of PBFT as consensus mechanism between the nodes of the network. This thesis project also involved the realization of a cryptographic accumulator, and the choice of a communication protocol which could be used in distributed systems. While carrying out this thesis, standard PKCS#11 was studied and an HSM was integrated into the proposed framework, with the aim of creating cryptographic keys and digital signatures. Furthermore, a DBPKI User Interface is provided, thus allowing the user of a single machine to run the PoC and to set up its own blockchain-based PKI.

Finally, after having analyzed the security properties offered by the DBPKI model and the experimental results obtained, it can be confirmed that the PoC has been fully completed, and that the improvement given by the use of dynamic trust weights has successfully enriched the developed infrastructure. On the other hand, the one realized is only a prototype and would therefore not be immediately usable in real applications, as experimental tests measuring PoC performance revealed. Nevertheless, future optimizations will hopefully make the use of proposed framework feasible in several scenarios, making it an effective alternative to traditional PKI.

7.1 Future improvements

This section includes limitations of proposed implementation and provides some hints for future improvements.

7.1.1 Privacy-awareness

As already pointed out by the work of Toorani et al. [8], DBPKI starting model does not consider privacy-awareness property, which requires the removal of direct link between nodes' identities and their public keys. DBPKI could be modified in order to respect this constraint, however, within this thesis, it has not been considered as it has little in common with the issue of decentralization.

This kind of improvement could be useful since features of blockchain-based PKIs contradict two rules of the GDPR (General Data Protection Regulation): the pseudonymization of personal data, and the right to erasure [77]. Indeed, identities should not be directly stored on blockchain since, once published, data become transparent, immutable and cannot be erased easily [77]. Therefore, the use of pseudonym mechanism would make entities tracking transactions to not obtain real identity of users, thus improving their anonymity [10].

7.1.2 Variants of PBFT

For the correct working of the proposed solution, DBPKI nodes have to verify at least $2f + 1$ signatures in both Prepare and Commit phases, according to PBFT consensus mechanism. This might limit the framework when a huge number of nodes are running at the same time, since their computational and communication overhead is high [8]. For this reason, standard version of PBFT could be replaced by other proposals in order to improve scalability of the system. Those alternatives could be, for instance, MirBFT [78], which allows the presence of multiple leaders, or HotStuff [79] and its variant LibraBFT [80], which uses fewer messages to achieve consensus. However, some vulnerabilities of HotStuff-based proposals should be taken into account, like the ones depicted in the work of Momose et al. [81].

7.1.3 Diffused PBFT

Another improvement that could be taken into consideration, even together with variations proposed in section 7.1.2, is the usage of a "diffused" version of the PBFT. This future enhancement can be inspired by the work of Chai et al. [14], already mentioned in Chapter 3.2. They built a framework called CyberChain in which, among other features, PBFT consensus mechanism is reinvented and applied in an even more efficient way. They called their new mechanism DPBFT, which stands for Diffused Practical Byzantine Fault Tolerance. Their work is mainly about IoV networks, so they needed to use a lightweight consensus mechanism, in order to cater for the high mobility of vehicles. DPBFT consists of the subdivision of the network, and therefore of the consensus group, into "sub-consensus areas". Each sub-area has a corresponding Edge Server (ES), whose task is to manage nodes currently in that area and to communicate with other ESs. Consensus is firstly achieved in a single sub-area, and then collective decision is spread in the rest of the network. In this way, the delay due to the total achievement of consensus and the communication overhead are drastically reduced. Therefore, the important aspects from which to take inspiration to improve the proposed solution are the division of the network into zones and the gradual spread of the consensus mechanism.

7.1.4 Existing-PKI interoperability

The proposed solution was made to work independently, but it could be useful to make it backwards compatible with the existing PKIs. A way should therefore be found to ensure that traditional digital certificates can be recognized by the proposed decentralized PKI, and vice versa, that public keys of DBPKI nodes can also be validated externally. This would make it possible to introduce interoperability with the already widespread traditional PKIs.

Appendix A

Developed accumulator algorithms

This appendix contains all algorithms used as inspiration for the development of the asynchronous accumulator. Some of these algorithms were taken from the work of Reyzin et al. [44], and later adapted to the Python implementation of proposed PoC. Among them, there is also a support function (`GetAncestors`), which is used to recompute the ancestors of a certain leaf in a tree.

Algorithm 1 `AccGen`: generating a new empty accumulator

Require: λ

1: **return** $a_0 = \perp$

Algorithm 2 `GetAncestors`: recomputing ancestors of an accumulated item

Require: p, x
1: $c = h(x)$
2: $\tilde{p} = [c]$
3: **for** $(z, \texttt{dir})$ in p **do**
4: **if** $\texttt{dir} = \texttt{right}$ **then**
5: $c = h(c||z)$
6: **else if** $\texttt{dir} = \texttt{left}$ **then**
7: $c = h(z||c)$
8: **end if**
9: append c to $\tilde{p}$
10: **end for**
11: **return** $\tilde{p}$

Algorithm 3 `AccAdd`: adding of a new element to the accumulator

Require: a_t, x
1: $a_{t+1} = a_t$ ▷ New accumulator starts out as a copy of the old one
2: $\omega^x_{t+1} = []$ ▷ Witness starts out as an empty list
3: $d = 0$ ▷ Witness' depth starts out as 0
4: $z = h(x)$
5: **while** $a_{t+1}[d] \neq \perp$ **do**
6: **if** the length of $a_{t+1} < d + 2$ **then**
7: append $\perp$ to a_{t+1}
8: **end if**
9: $z = h(a_{t+1}[d]||z)$
10: append $(a_{t+1}[d], \texttt{left})$ to ω^x_{t+1}
11: $a_{t+1}[d] = \perp$
12: $d = d + 1$
13: **end while**
14: $a_{t+1}[d] = z$
15: **return** $a_{t+1}, \omega^x_{t+1}, \texttt{updmsg} = (x, \omega^x_{t+1})$

Algorithm 4 `AccVer`: verifying of a witness

Require: a_t, x, ω^x

$\tilde{p} = \texttt{GetAncestors}(\omega^x, x)$
if a_t and r have any elements in common **then**
    return TRUE
else
    return FALSE
end if

Algorithm 5 `AccWitAdd`: updating a witness after that a new element has been added to the accumulator

Require: $y, \omega^y_{t+1}, \omega^x_t$

let d^x_t be the length of ω^x_t
let d^y_{t+1} be the length of ω^y_{t+1}
if $d^y_{t+1} < d^x_t$ **then**
    return ω^x_t ▷ Witness has not changed
else
    $d^x_{t+1} = d^y_{t+1}$
    $\omega^x_{t+1} = \omega^x_t$ ▷ New authenticating path starts out as a copy of the old one
    $\tilde{\omega}^y_{t+1} = \texttt{GetAncestors}(\omega^y_{t+1}, y)$
    append $(\tilde{\omega}^y_{t+1}[d^x_t], \texttt{right})$ to ω^x_{t+1}
    append $\omega^y_{t+1}[d^x_t + 1, ...]$ to ω^x_{t+1}
    return ω^x_{t+1}
end if

Algorithm 6 `AccDel`: deleting an element from the accumulator

Require: a_t, x

search in a_t for pointer to $\tilde{x}$ value in the accumulator such that $\tilde{x} = x$
if pointer to $\tilde{x}$ not found **then**
    return a_t ▷ No element has been removed
else
    let r_d be the root of the Merkle Tree where $\tilde{x}$ has been found
    $a_{t+1} = a_t$ ▷ New accumulator starts out as a copy of the old one
    replace $\tilde{x}$ value with $\perp$ in $a_{t+1}[d]$ corresponding leaf
    recompute hashes of all ancestors of found leaf in $a_{t+1}[d]$
    return a_{t+1}
end if

Appendix B

DBPKI procedure algorithms

This appendix contains the algorithms of functionalities provided by DBPKI proposed framework, as pseudo-codes. They are based on the work of Toorani et al. [8], and they are the basis of the Python implementation of PBFT protocol and DBPKI itself. The `AddBL` algorithm corresponds to the addition of a new block in the chain, while the "KeyGen" function refers to the method used to generate new asymmetric cryptographic keys. Each node creates its own key pairs, and then only their public part is shared with other nodes.

Algorithm 7 `AddBL`: adding a block to blockchain

Require: *Proposal*

1: **if** consensus achieved $\wedge$ proposal approved **then**
2: $\quad \mathrm{BL}_{i+1} = \{Proposal\}$
3: **end if**

Algorithm 8 `Setup`: initializing DBPKI

Require: λ

$\text{ID}_{block} = 1$ ▷ First block ID is equal to 1
$\text{TL} = []$ ▷ Transaction list starts out as empty
$\text{TW} = []$ ▷ Trust weights starts out as an empty list
$a_0 = \texttt{AccGen}(1^\lambda)$
for $i \in \{1, ..., n\}$ **do**
    $(sk_{u_i}, pk_{u_i}) = \text{KeyGen}(1^\lambda)$
    append (ID_{R_i}, pk_{R_i}, 0, `R`) to TL
    append (ID_{R_i}, `V_R`) to TW
    $x = (\text{ID}_{R_i}, h(pk_{R_i}))$
    $\texttt{AccAdd}(a_0, x)$
end for
$\texttt{AddBL}(\{Opt, \text{ID}_{block}, a_1, \omega_1, \text{TL}, \text{TW}\})$ ▷ Optional data contains a timestamp

Algorithm 9 `Enroll`: enrolling a new unit in DBPKI

Require: ID, pk, `flag`, `role`

if $\text{PIV}(\text{ID}, pk) = 1 \wedge$ consensus achieved **then**
    append (ID, $h(pk)$, 0, `role`) to TL
    if `role` = `I` **then**
        $weight$ = `V_I`
    else if `role` = `O` **then**
        $weight$ = `V_O`
    end if
    append (ID, $weight$) to TW
    $x = (\text{ID}, h(pk))$
    $(a_{i+1}, \omega_{i+1}) = \texttt{AccAdd}(a_i, x)$
    $\texttt{AddBL}(\{Opt, \text{ID}_{block}^{i+1}\ a_{i+1}, \omega_{i+1}, \text{TL}, \text{TW}\})$ ▷ *Opt* contains a timestamp
    return 1
else
    return $\perp$
end if

Algorithm 10 `Verify`: verifying a proof membership

Require: ID, pk
1: $x = (\text{ID}, h(pk))$
2: **if** `AccVer`$(a_i, x, \omega_i) = 1$ **then**
3: **return** 1
4: **else**
5: **return** $\perp$
6: **end if**

Algorithm 11 `Revoke`: revoking a public key

Require: ID, pk
1: let `role` be the role of affected node
2: **if** `Verify`(ID, pk) $\wedge$ consensus achieved **then**
3: append (ID, $h(pk)$, 2, `role`) to TL
4: $(a_{i+1}, \omega_{i+1}) =$ `AccDel`$(a, (\text{ID}, pk))$
5: `AddBL`$(\{Opt, \text{ID}_{block}^{i+1}, \text{ID}, a_{i+1}, \omega_{i+1}, \text{TL}, \text{TW}_{i+1}\})$ ▷ Opt contains a timestamp
6: **else**
7: **return** $\perp$
8: **end if**

Algorithm 12 `Update`: updating a public key

Require: ID, pk^{old}, pk^{new}
1: let `role` be the role of affected node
2: **if** PIV(ID, pk^{new}) $= 1$ **then**
3: Revoke(ID, pk^{old})
4: Enroll(ID, pk^{new}, 1, `role`)
5: **return** 1
6: **else**
7: **return** $\perp$
8: **end if**

Bibliography

[1] Priyanka Rathee. «Introduction to blockchain and IoT». In: *Advanced Applications of Blockchain Technology*. Springer, 2020, pp. 1–14 (cit. on p. 1).

[2] Ze Wang, Jingqiang Lin, Quanwei Cai, Qiongxiao Wang, Daren Zha, and Jiwu Jing. «Blockchain-based certificate transparency and revocation transparency». In: *IEEE Transactions on Dependable and Secure Computing* (2020) (cit. on pp. 2, 30).

[3] Kathleen Wilson. *Distrusting New CNNIC Certificates*. 2015. URL: `https://blog.mozilla.org/security/2015/04/02/distrusting-new-cnnic-certificates/` (visited on 06/29/2022) (cit. on p. 2).

[4] J Ronald Prins and Business Unit Cybercrime. «Diginotar certificate authority breach "operation black tulip"». In: *Fox-IT, November* (2011), p. 18 (cit. on p. 2).

[5] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen K Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. «MD5 considered harmful today, creating a rogue CA certificate». In: *25th Annual Chaos Communication Congress*. CONF. 2008 (cit. on p. 2).

[6] Peter Eckersley. *A Syrian Man-In-The-Middle Attack against Facebook*. 2011. URL: `https://www.eff.org/deeplinks/2011/05/syrian-man-middle-against-facebook` (visited on 06/30/2022) (cit. on p. 2).

[7] Mike Zusman. «Criminal charges are not pursued: Hacking PKI». In: *DEFCON 17* (2009) (cit. on p. 2).

[8] Mohsen Toorani and Christian Gehrmann. «A decentralized dynamic pki based on blockchain». In: *Proceedings Of the 36th Annual ACM Symposium On Applied Computing*. 2021, pp. 1646–1655 (cit. on pp. 2, 23, 26, 29–32, 40, 41, 44–46, 52–54, 56–58, 62, 74, 84–87, 90–92, 98).

[9] Ankush Singla and Elisa Bertino. «Blockchain-based PKI solutions for IoT». In: *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*. IEEE. 2018, pp. 9–15 (cit. on p. 3).

[10] Fengyin Li, Zhongxing Liu, Tao Li, Hongwei Ju, Hua Wang, and Huiyu Zhou. «Privacy-aware PKI model with strong forward security». In: *International Journal of Intelligent Systems* (2020) (cit. on pp. 3, 31, 91).

[11] Arijet Sarker, SangHyun Byun, Wenjun Fan, and Sang-Yoon Chang. «Blockchain-based root of trust management in security credential management system for vehicular communications». In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. 2021, pp. 223–231 (cit. on p. 3).

[12] Xavier Boyen, Udyani Herath, Matthew McKague, and Douglas Stebila. «Associative Blockchain for Decentralized PKI Transparency». In: *Cryptography* 5.2 (2021), p. 14 (cit. on p. 3).

[13] Yves Christian Elloh Adja, Badis Hammi, Ahmed Serhrouchni, and Sherali Zeadally. «A blockchain-based certificate revocation management and status verification system». In: *Computers & Security* 104 (2021), p. 102209 (cit. on p. 3).

[14] Haoye Chai, Supeng Leng, Jianhua He, Ke Zhang, and Baoyi Cheng. «CyberChain: Cybertwin Empowered Blockchain for Lightweight and Privacy-preserving Authentication in Internet of Vehicles». In: *IEEE Transactions on Vehicular Technology* (2021) (cit. on pp. 3, 32, 33, 35–37, 92).

[15] Loıc Champagne et al. «Replacing Public Key Infrastructures (PKI) by blockchain IoT devices security management». In: (2021) (cit. on p. 3).

[16] Chong-Gee Koa, Swee-Huay Heng, and Ji-Jian Chin. «ETHERST: Ethereum-Based Public Key Infrastructure Identity Management with a Reward-and-Punishment Mechanism». In: *Symmetry* 13.9 (2021), p. 1640 (cit. on pp. 3, 32, 36–39).

[17] Carlisle Adams and Steve Lloyd. *Understanding PKI: concepts, standards, and deployment considerations*. Addison-Wesley Professional, 2003 (cit. on p. 7).

[18] Kevin Stine and Quynh Dang. «Encryption basics». In: *Journal of AHIMA* 82.5 (2011), pp. 44–46 (cit. on pp. 7, 8).

[19] Jon Callas and Rest Secured. «An introduction to cryptography». In: (2006) (cit. on pp. 7, 8).

[20] IBM. *x.509 certificate revocation documentation*. 2022. URL: `https://www.ibm.com/docs/en/zos/2.2.0?topic=management-x509-certificate-revocation` (visited on 07/02/2022) (cit. on p. 12).

[21] Michael Cobb Rahul Awati. *certificate revocation list (CRL)*. 2021. URL: `https://www.techtarget.com/searchsecurity/definition/Certificate-Revocation-List` (visited on 07/02/2022) (cit. on p. 12).

[22] Ryan Sanders. *What is a Certificate Revocation List (CRL) vs OCSP?* 2020. URL: `https://www.keyfactor.com/blog/what-is-a-certificate-revocation-list-crl-vs-ocsp/` (visited on 07/05/2022) (cit. on pp. 13, 14).

[23] Rick Donato. *Certificate Revocation (CRL vs OCSP)*. 2022. URL: `https://www.fir3net.com/Security/Concepts-and-Terminology/certificate-revocation.html` (visited on 07/05/2022) (cit. on pp. 12–14).

[24] Ioannis S Iliadis, Diomidis Spinellis, Sokratis Katsikas, and Bart Preneel. «A taxonomy of Certificate Status Information mechanisms». In: *Proceedings of Information Security Solutions Europe ISSE* 2000 (2000) (cit. on pp. 12, 13).

[25] Emin Topalovic, Brennan Saeta, Lin-Shung Huang, Collin Jackson, and Dan Boneh. «Towards short-lived certificates». In: Web. 2012 (cit. on p. 13).

[26] Thales. *What is Root of Trust?* URL: `https://cpl.thalesgroup.com/faq/hardware-security-modules/what-root-trust` (visited on 07/05/2022) (cit. on p. 15).

[27] PKI Globe. *PKCS#11 Terminology*. URL: `http://www.pkiglobe.org/pkcs11_terminology.html` (visited on 07/06/2022) (cit. on pp. 15, 18).

[28] Thales. *An Introduction to PKCS#11*. URL: https://thalesdocs.com/gphsm/ptk/5.9/docs/Content/PTK-C_Program/intro_PKCS11.htm (visited on 07/06/2022) (cit. on pp. 15, 17).

[29] Thales. *Luna Newtork HSM Documentation Archive*. URL: https://thalesdocs.com/gphsm/luna/7/docs/network/Content/Product_Overview/preface.htm (visited on 07/05/2022) (cit. on pp. 16, 18, 20, 72).

[30] Mario Stipčević and Çetin Kaya Koç. «True random number generators». In: *Open Problems in Mathematics and Computational Science*. Springer, 2014, pp. 275–315 (cit. on p. 16).

[31] Jolyon Clulow. «On the security of PKCS# 11». In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2003, pp. 411–425 (cit. on p. 17).

[32] Oasis. *PKCS #11 Cryptographic Token Interface Base Specification Version 2.40*. 2015. URL: http://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html (visited on 07/07/2022) (cit. on pp. 18, 19).

[33] Rafael Martınez-Peláez, Francisco Rico-Novella, and Cristina Satizábal. «Secure smart card reader design». In: *2008 IEEE International Symposium on Consumer Electronics*. IEEE. 2008, pp. 1–3 (cit. on p. 18).

[34] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. «Blockchain technology overview». In: *arXiv preprint arXiv:1906.11078* (2019) (cit. on pp. 21, 22).

[35] Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, and Muhammad Imran. «An overview on smart contracts: Challenges, advances and platforms». In: *Future Generation Computer Systems* 105 (2020), pp. 475–491 (cit. on p. 21).

[36] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. «An overview of blockchain technology: Architecture, consensus, and future trends». In: *2017 IEEE international congress on big data (BigData congress)*. Ieee. 2017, pp. 557–564 (cit. on p. 22).

[37] Leslie Lamport, Robert Shostak, and Marshall Pease. «The Byzantine generals problem». In: *Concurrency: the works of leslie lamport*. 2019, pp. 203–226 (cit. on p. 22).

[38] Marshall Pease, Robert Shostak, and Leslie Lamport. «Reaching agreement in the presence of faults». In: *Journal of the ACM (JACM)* 27.2 (1980), pp. 228–234 (cit. on p. 22).

[39] Miguel Castro, Barbara Liskov, et al. «Practical byzantine fault tolerance». In: *OsDI*. Vol. 99. 1999. 1999, pp. 173–186 (cit. on pp. 23, 24).

[40] Germano Caronni. «Walking the web of trust». In: *Proceedings IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2000)*. IEEE. 2000, pp. 153–158 (cit. on p. 25).

[41] Simson Garfinkel. *PGP: pretty good privacy*. " O'Reilly Media, Inc.", 1995 (cit. on p. 25).

[42] David Derler, Christian Hanser, and Daniel Slamanig. «Revisiting cryptographic accumulators, additional properties and relations to other primitives». In: *Cryptographers' track at the rsa conference*. Springer. 2015, pp. 127–144 (cit. on p. 25).

[43] Ilker Ozcelik, Sai Medury, Justin Broaddus, and Anthony Skjellum. «An overview of cryptographic accumulators». In: *arXiv preprint arXiv:2103.04330* (2021) (cit. on pp. 26, 27, 46).

[44] Leonid Reyzin and Sophia Yakoubov. «Efficient asynchronous accumulators for distributed PKI». In: *International Conference on Security and Cryptography for Networks*. Springer. 2016, pp. 292–309 (cit. on pp. 27, 46–52, 60, 61, 65, 94).

[45] Daniel Maldonado-Ruiz, Jenny Torres, Nour El Madhoun, and Mohamad Badra. «Current Trends in Blockchain Implementations on the Paradigm of Public Key Infrastructure: A Survey». In: *IEEE Access* 10 (2022), pp. 17641–17655 (cit. on pp. 29–31).

[46] Clemens Brunner, Fabian Knirsch, Andreas Unterweger, and Dominik Engel. «A Comparison of Blockchain-based PKI Implementations.» In: *ICISSP*. 2020, pp. 333–340 (cit. on pp. 29, 31).

[47] Ben Laurie, Adam Langley, and Emilia Kasper. *Certificate transparency*. Tech. rep. 2013 (cit. on p. 29).

[48] Shohei Kakei, Yoshiaki Shiraishi, Masami Mohri, Toru Nakamura, Masayuki Hashimoto, and Shoichi Saito. «Cross-certification towards distributed authentication infrastructure: A case of hyperledger fabric». In: *IEEE Access* 8 (2020), pp. 135742–135757 (cit. on p. 30).

[49] Jian Zhao, Zexuan Lin, Xiaoxiao Huang, Yiwei Zhang, and Shaohua Xiang. «TrustCA: Achieving certificate transparency through smart contract in blockchain platforms». In: *2020 International Conference on High Performance Big Data and Intelligent Systems (HPBD&IS)*. IEEE. 2020, pp. 1–6 (cit. on p. 30).

[50] Rong Wang, Juan He, Can Liu, Qi Li, Wei-Tek Tsai, and Enyan Deng. «A privacy-aware PKI system based on permissioned blockchains». In: *2018 IEEE 9th international conference on software engineering and service science (ICSESS)*. IEEE. 2018, pp. 928–931 (cit. on p. 30).

[51] Bo Qin, Jikun Huang, Qin Wang, Xizhao Luo, Bin Liang, and Wenchang Shi. «Cecoin: A decentralized PKI mitigating MitM attacks». In: *Future Generation Computer Systems* 107 (2020), pp. 805–815 (cit. on p. 30).

[52] Alexander Yakubov, Wazen Shbair, Anders Wallbom, David Sanda, et al. «A blockchain-based PKI management framework». In: *The First IEEE/IFIP International Workshop on Managing and Managed by Blockchain (Man2Block) colocated with IEEE/IFIP NOMS 2018, Tapei, Tawain 23-27 April 2018*. 2018 (cit. on p. 30).

[53] Shaozhuo Li, Na Wang, Xuehui Du, and Aodi Liu. «Internet web trust system based on smart contract». In: *International conference of pioneering computer scientists, engineers and educators*. Springer. 2019, pp. 295–311 (cit. on pp. 30, 37).

[54] Elie F Kfoury, David Khoury, Ali AlSabeh, Jose Gomez, Jorge Crichigno, and Elias Bou-Harb. «A blockchain-based method for decentralizing the ACME protocol to enhance trust in PKI». In: *2020 43rd International Conference on Telecommunications and Signal Processing (TSP)*. IEEE. 2020, pp. 461–465 (cit. on pp. 30, 31).

[55] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. «A decentralized public key infrastructure with identity retention». In: *Cryptology ePrint Archive* (2014) (cit. on p. 30).

[56] LM Axon and Michael Goldsmith. «PB-PKI: A privacy-aware blockchain-based PKI». In: (2016) (cit. on p. 30).

[57] Paul Plessing and Olamide Omolola. «Revisiting Privacy-aware Blockchain Public Key Infrastructure.» In: *ICISSP*. 2020, pp. 415–423 (cit. on p. 30).

[58] Mustafa Al-Bassam. «SCPKI: A smart contract-based PKI and identity system». In: *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*. 2017, pp. 35–40 (cit. on pp. 30, 31).

[59] Thomas Sermpinis, George Vlahavas, Konstantinos Karasavvas, and Athena Vakali. «DeTRACT: a decentralized, transparent, immutable and open PKI certificate framework». In: *International Journal of Information Security* 20.4 (2021), pp. 553–570 (cit. on p. 31).

[60] Amit Dua, Siddharth Sekhar Barpanda, Neeraj Kumar, and Sudeep Tanwar. «Trustful: A decentralized public key infrastructure and identity management system». In: *2020 IEEE Globecom Workshops (GC Wkshps*. IEEE. 2020, pp. 1–6 (cit. on p. 31).

[61] Jakob Schaerer, Severin Zumbrunn, and Torsten Braun. «Veritaa-The Graph of Trust». In: *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE. 2020, pp. 168–175 (cit. on p. 31).

[62] Hiroaki Anada, Takanori Yasuda, Junpei Kawamoto, Jian Weng, and Kouichi Sakurai. «RSA public keys with inside structure: Proofs of key generation and identities for web-of-trust». In: *Journal of information security and applications* 45 (2019), pp. 10–19 (cit. on p. 31).

[63] D Rachmawati and MA Budiman. «On Using The First Variant of Dependent RSA Encryption Scheme to Secure Text: A Tutorial». In: *Journal of Physics: Conference Series*. Vol. 1542. 1. IOP Publishing. 2020, p. 012024 (cit. on p. 31).

[64] Marc Jansen, Farouk Hdhili, Ramy Gouiaa, and Ziyaad Qasem. «Do smart contract languages need to be turing complete?» In: *International Congress on Blockchain and Applications*. Springer. 2019, pp. 19–26 (cit. on p. 36).

[65] Dr Craig S Wright. «Turing Complete Bitcoin Script White Paper». In: *Available at SSRN 3160279* (2016) (cit. on p. 36).

[66] Stephanos Matsumoto and Raphael M Reischuk. «IKP: turning a PKI around with decentralized automated incentives». In: *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 410–426 (cit. on p. 37).

[67] Foteini Baldimtsi, Jan Camenisch, Maria Dubovitskaya, Anna Lysyanskaya, Leonid Reyzin, Kai Samelin, and Sophia Yakoubov. «Accumulators with applications to anonymity-preserving revocation». In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2017, pp. 301–315 (cit. on p. 46).

[68] Python. *Python 3.9.7*. 2021. URL: `https://www.python.org/downloads/release/python-397/` (visited on 08/04/2022) (cit. on p. 63).

[69] Thales Group. *Pycryptoki*. 2018. URL: `https://pycryptoki.readthedocs.io/en/latest/` (visited on 08/04/2022) (cit. on pp. 63, 73).

[70] Oracle. *MySQL*. 2022. URL: `https://www.mysql.com/` (visited on 08/04/2022) (cit. on p. 64).

[71] PyPI. *pycryptodome*. 2022. URL: `https://pypi.org/project/pycryptodome/` (visited on 08/11/2022) (cit. on p. 66).

[72] Fahad Taha AL-Dhief et al. «Performance comparison between TCP and UDP protocols in different simulation scenarios». In: *International Journal of Engineering & Technology* 7.4.36 (2018), pp. 172–176 (cit. on p. 70).

[73] Danish Bilal Ansari, A Ur Rehman, and R Ali. «Internet of things (iot) protocols: a brief exploration of mqtt and coap». In: *International Journal of Computer Applications* 179.27 (2018), pp. 9–14 (cit. on p. 70).

[74] Roshan Sedar et al. «Standards-Compliant Multi-Protocol On-Board Unit for the Evaluation of Connected and Automated Mobility Services in Multi-Vendor Environments». In: *Sensors* 21.6 (2021), p. 2090 (cit. on p. 70).

[75] Eclipse Mosquitto. *mosquitto - Eclipse Foundation*. URL: `https://mosquitto.org/` (visited on 08/12/2022) (cit. on p. 71).

[76] Greg Slepak and Anya Petrova. «The DCS theorem». In: *arXiv preprint arXiv:1801.04335* (2018) (cit. on pp. 82, 83).

[77] Gwan-Hwan Hwang, Tao-Ku Chang, and Hung-Wen Chiang. «A Semidecentralized PKI System Based on Public Blockchains with Automatic Indemnification Mechanism». In: *Security and Communication Networks* 2021 (2021) (cit. on p. 91).

[78] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. «Mir-bft: High-throughput bft for blockchains». In: *arXiv preprint arXiv:1906.05552* (2019) (cit. on p. 92).

[79] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. «Hotstuff: Bft consensus with linearity and responsiveness». In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 2019, pp. 347–356 (cit. on p. 92).

[80] Mathieu Baudet et al. «State machine replication in the libra blockchain». In: *The Libra Assn., Tech. Rep* (2019) (cit. on p. 92).

[81] Atsuki Momose and Jason Paul Cruz. «Force-locking attack on sync hotstuff». In: *Cryptology ePrint Archive* (2019) (cit. on p. 92).