POLITECNICO DI TORINO

Master Degree Course in Computer Engineering

Master Degree Thesis

Implementation of a Blockchain-based Distributed PKI for IoT using Emercoin NVS and TPM 2.0

Supervisor
prof. Antonio LIOY
prof. Diana BERBECARU

Candidate
Lorenzo PINTALDI

2022

A mio padre, radice di ogni mio successo.

Summary

Internet-of-Things is constantly expanding and one of the most relevant challenge is to secure the communications that involve these particular devices. Public Key Infrastructure (PKI) mechanisms are not well suited for IoT devices, and many new solutions to this problem leverage on the Blockchain technology for moving the "trust-anchor" from centralized root Certification Authorities (CAs) to a public distributed ledger. Starting from one of these solutions (proposed in a 2018's paper by Elisa Bertino, Ankush Singla, Jongho Won and Greg Bollella) based on a Blockchain project called Emercoin NVS, the purpose of this work is to propose an implementation that extends the original design using the TPM 2.0 technology for the identification of the IoT devices.

By using this Blockchain-based system, the certificates of the IoT devices within a network are securely stored on the Blockchain, and they can be retrieved (with simple HTTP requests, using the RPC configuration of the Emercoin Wallet) during the certificate verification step of the TLS handshake, in replacement of the standard certificate chains provided by the Certification Authorities. The registration of devices certificates on the Blockchain is performed by interacting with a special node of the network called Device Manager (DM), that runs the Emercoin Wallet and maintain a local copy of the ledger. In the original design, there are no security measures to prevent the registration of a certificate forged by a "lying" entity. Any device can in fact claim an arbitrary identity during the certificate registration protocol with the DM. Thanks to TPM 2.0 technology, the security level of the original solution was improved, by providing a strong protection against the identity theft.

Finally, some experiments based on this implementation have been conducted for measuring the TLS handshake time using the Emercoin-based distributed approach, in order to compare it with the standard centralized one. Blockchain-based TLS handshake was slower that than the standard one, but also more secure if we take into account that the revocation status check is frequently skipped in standard communications (also for web-based communications), unless the OCSP Must-Staple is enabled.

Acknowledgements

A special thanks goes to Silvia Sisinni for her patience and the huge support for everything regarding the TPM technology.

Contents

List of Figures

Chapter 1

Introduction

1.1 Background and motivation

1.1.1 PKI model

Public Key Infrastructure (PKI) is the set of hardware, software, entities, policies and procedures defined to create, distribute and revoke digital certificates for digital signature authentication and key-distribution protocol. PKI standardization (based on X.509 certificates) ensures the possibility to estabilish secure communication channels between entities, binding each entity's attributes to unique public keys.

X.509 certificates are issued by Internet Certification Authorities (CAs) - the main actors of the PKI model - following a certification hierarchy with many possible levels: each level represents those entities that certifies for the underlying level, until the bottom of this hierarchy represented by the End Entities (EEs). On the top, there's a root-of-trust represented by some special top-level CAs (Root CAs) assumed as "trustable" elements, whose public keys are locally stored nearly in every software system as configuration objects. Certification chains resolution is based on this "trust-assumption".

X.509 is the standard solution to the problem of identifying the owner of a cryptographic key. X.509 certificates are defined in ASN.1 (Abstract Syntax Notation 1) and are characterized by many attributes.

In case the private key associated with a public key certified by a CA is compromised, the certificate must be immediately revoked. When a certificate is received for authentication, the revocation status must be always checked properly. There are two possible mechanisms for revocation status check: CRL (Certificate Revocation List), and OSCP (Online Status Check Protocol).

- **CRL**: an updated list of revoked certificates issued by the CA that provides the CRL;
- **OCSP**: using this protocol, OSCP servers can be queried for a specific certificate status check. The server provides only the validity status of the certificate the request was made for;

1.1.2 PKI drawbacks

PKI trust-based model has several drawbacks. It's clear that by design this model has a single-point-of-failure represented by root CAs. Root CAs can be compromised and issue rogue certificates for bad actors on behalf of well known entities. This scenario is extremely dangerous for final users, who can't easily distinguish the malicious entity from the good one due to the fact that the provided certificate seems to be valid.

PKI is employed in a huge number of application and contexts where security is a critical feature: for this reason, a root CA compromise can be a relevant event and can also cause a denial-of-service when the failure is detected, because many entities rely on that specific CA to certify their own identity.

In this case, current implementations rely on the user to take a decision about the communication estabilishment (to trust or not the entity we are trying to communicate with). The problem with this, is that the majority of users do not understand what consequences this choice will imply.

Many attacks against root CAs have been performed successfully and in general the risks are higher when we place too much trust in one or a small number of CAs. One of the most relevant attack against a CA was performed in 2011 against DigiNotar, quickly causing bankruptcy for the company.

Moreover, as previously discussed, one of the core step of certificate verification is the certificate validity check. CRL and OSCP are in general two valid approach for this purpose, but they're still affected by some weaknesses and they can be inefficient in particular contexts.

- CRL file is usually very large and difficult to be managed by constrained-resource devices;
- OCSP is vulnerable to Denial-of-Service attacks, by overloading the OSCP responders with many requests that requires a large computational overhead for signature generations

Another important issue of PKI was brought by one of the most expanding technology of the last years, the IoT. The use of CA-based PKI is not well-suited for IoT devices for many reasons:

- It is difficult for IoT devices' owners to manage all the certificates for their own devices since there are no standard protocols or mechanism for certificate issuance and update on these devices;
- Due to this difficulty, IoT devices' keys are usually generated and installed by the manufacturers, before the deployment. This practice can lead to private key leakages;
- Modern mechanisms to easily get valid X509 certificates (e.g. ACME protocol, RFC-8555 [1]) are not well suited for typical IoT use cases/scenarios

In recent years, a lot of effort was spent on many researches about PKI drawbacks and modern solutions to solve them. Many of these researches try to remove the necessity of trusted-third-parties (TTPs) with some new PKI models. Decentralized PKIs are a possible alternative way to achieve this intent, promising an easy way for certificates to be directly managed from the entities they belong to. These solutions are mainly based on a shared database storing identities and public keys. The problem with this approach is the necessity for a consensus mechanism to ensure that what is stored on the database is valid. For this purpose, Blockchains are for sure the most interesting technology to elaborate on.

The purpose of this work is to present an experimental implementation of a solution design that's already been proposed, fixing some important issues related to the original design. The rest of the thesis will be divided into 6 chapters:

- **Related works**: in this chapter some background knowledge about Blockchain technology will be provided in order to understand the actual state-of-art for what regards the Blockchain integration with IoT and Public Key Infrastructure. Advantages and challenges of this integration will be provided and many proposed solution will be further analyzed;
- **Proposed implementation**: it will contain all the technical details about the proposed implementation, with a subsection dedicated to the TPM technology that was used for the identification of the devices during the certificate registration/update process;
- **Measurements**; a discussion about the results of some experiments conducted on the proposed solution, in order to compare it with the "standard" centralized PKI approach;

- **Conclusions**;
- **User's manual**: a simple description of the available interactions with the final application;
- **Developer's manual**: a detailed guide to compile, configure and eventually modify the application;

Chapter 2

Related works

2.1 IoT scenario

IoT connect a great amount of internet devices and aims to the transmission of huge data flows and the creation of brand new smart scenarios (smart cities, smart home, smart healthcare, smart cars, smart objects etc...) that will be able to improve the efficiency level of many processes and operations associated with the context. IoT market is constantly growing: many researches estimate that in 2025 the number of IoT devices will be approximately 35 billion, but for other researchers this number can even be greater (50 billions). With similar numbers, the estimation for the next decade is that IoT market could be potentially worth $\approx$ 19 trillions dollars.

The most important requirements for these new expanding scenarios, are low latency (especially for particular application like smart cars) and high throughput: IoT devices require a network connection that can satisfy these requirements: for this reason, they're usually featured with cellular connectivity, WiFi or they're supported by a base station connected to the network. However, most of the IoT devices are actually very limited on computational resources and storage capabilities, because their usage is typically tied to those use-cases that generally don't require an high computational power (sensors, actuators, etc...).

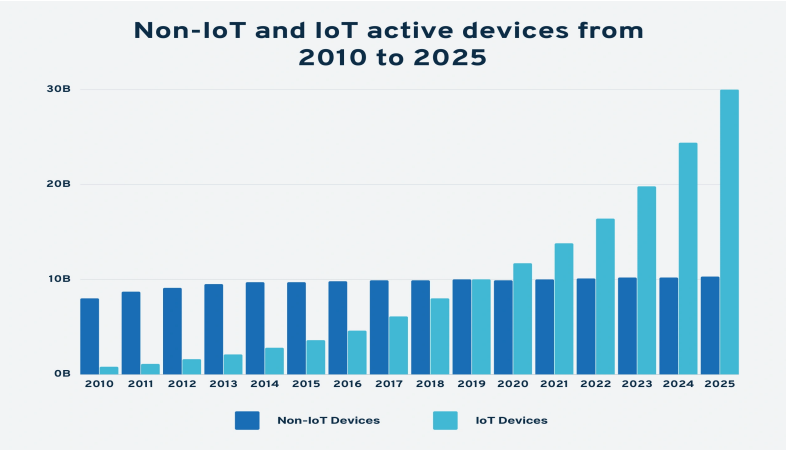

Figure 2.1: IoT growth statistics [3]

2.1.1 IoT security concerns

What clashes with this fact, is that security concerns have a great impact on some IoT applications. Among the great variety of devices within the IoT family, also actuators were previously mentioned: this category of devices can be safety-critical (for instance if we take into account the

existence of smart IoT health-care devices) and can potentially be a threat for other individuals or things within the same environment if one of these devices is not working correctly. The contradiction is that many effective security measures can't be applied to these devices exactly because of their lack of computational resources.

Moreover, the complexity of these systems is strictly tied to the great variety of different technologies and components used for their design and production: this technological heterogeneity is very difficult to deal with, even more if we consider the estimated large employment of IoT devices in the future. For this reason, ensuring an efficient and secure authentication process for these devices can be difficult. The authentication is usually performed using standard centralized approaches that are affected by some critical issues (already mentioned in 1.1.2) that many researches are trying to solve by focusing on Self-Sovereign Identity [22]

In the introduction, the integration of centralized PKI with IoT has already been discussed. CA-based PKI doesn't suit IoT because the management and maintenance of devices certificates is performed using mechanisms that were not originally designed with IoT in mind. For instance, the ACME protocol used by Let's Encrypt [2], provides a simple and automatic mechanism to get free certificates very quickly: its core functioning is based on a DNS/HTTPS challenge in order to prove the ownership of the domain for which the certificate is requested. For the most common use-cases, this protocol heavily simplifies the certificate issuance process, but it can't be applied to an IoT device that needs to be configured with a valid certificate.

The issues associated with centralized architectures are even worse if the central servers are in charge of storing and/or processing all the data coming from IoT devices: the Internet-of-Things produces a huge amount of data that could overload a centralized architecture, causing a drop in the performance and the necessity of great storage capabilities. With this approach, also the privacy and data integrity must be taken into account, especially because data processing and storing is usually demanded to external cloud provider that are not completely transparent about the management of the data contained within their infrastructure. In fact, in some cases, (Internet of Medical Things, Internet of Battlefield Things, etc...) the IoT produces sensitive data that must be properly secured and protected.

In general, the security measures applied to IoT devices are not sufficient, exposing them to an important variety of threats and possible attacks. Compared to the networks of standard devices, an additional challenge comes with Internet-of-Things: for these particular devices, it's challenging to provide host-based security because it's difficult to use anti-virus systems and automatic patching software for security monitoring and maintenance. The consequence of this, is also a dangerous increased vulnerability of these devices to those attacks that come from the internal network, where the perimeter security measures can't be applied.

Starting from the necessity of ensuring data integrity and providing a scalable and effective authentication mechanism for constrained-resource devices, many researches have been conducted in order to address these issues. With the advent of Bitcoin protocol in 2009 [12], and the widespread of Blockchain technology, most of these works have focused their attention on this technology and on distributed architectures in general. Actually, the Blockchain represent the main potential solution to many of those issues quickly discussed until now. This chapter will provide a technical background to better understand what is a Blockchain, and a state-of-the-art of its application to PKI and IoT. It will be organized in the following four subsections:

- Blockchain technology background
- Blockchain-IoT integration benefits
- Blockchain-IoT integration challenges
- State-of-the-art, proposed solutions

2.2 Blockchain technology

2.2.1 Block structure

First blockchain-like protocol was proposed in 1982 to implement a system wherein document timestamps could not be tampered with, but the first decentralized blockchain application was realized by Satoshi Nakamoto in 2008 with the publication of the Bitcoin whitepaper [12]. A blockchain is a decentralized, distributed, usually public ledger consisting of a linked list of records called "blocks". Each node of a peer-to-peer network holds and maintain a copy of the blockchain and the agreement between all of these entities about blockchain state is reached thanks to many consensus protocol (e.g. Proof-of-Work, Proof-of-Stake, Byzantine Fault Tolerance, etc...). Blocks contain transactions across the nodes of the network, but the key point of this new powerful technology is that transaction/information exchanges are performed without any intermediary or centralized third party. This property can be explained by further analyzing the structure of each block (for the following explanation, the Bitcoin block structure will be used as a model, even if the block structure can vary through different blockchain implementations):

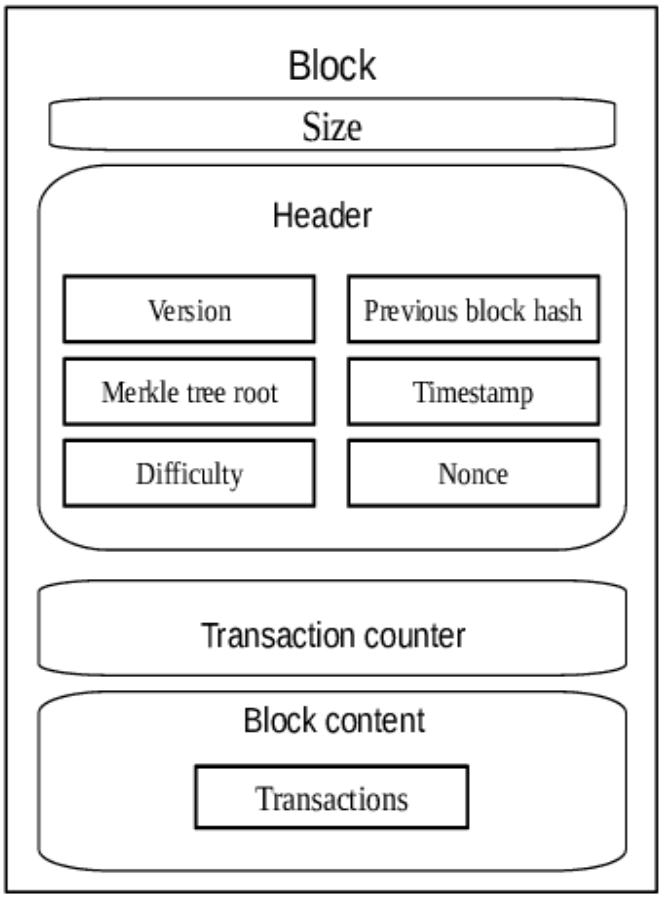

Figure 2.2: Blockchain blocks structure (source: image)

- **Size**: block size
- **Version**: overlying protocol version (e.g. Bitcoin version);
- **Merkle Tree root**: root of a Merkle Hash Tree [13] whose leafs are labelled with the hash of each transaction within the block. This is basically the "fingerprint" of the block and it's fundamental for the integrity mechanism of the blockchain. The reason why a Merkle Tree structure is used instead of simple hash of the concatenation of each transaction is that Merkle Hash Trees are very efficient structures to verify the integrity of large blocks

of data (block's transactions in this case). Merkle Trees were also fundamental for the development of many Blockchain-based protocol lightweight clients (e.g. Light Ethereum Subprotocol [58]), suited for low storage devices (e.g. smarthphones): instead of storing the entire blockchain, these clients store only the blocks header; when some information about a specific transaction is necessary, they query a full blockchain node to receive the transaction data and a "Merkle proof", the set of necessary hashes to recompute the Merkle tree root for the integrity check (Merkle proof size is $log(N_{transactions})$, that's why Merkle Trees are so efficient);

- **Difficulty**: the *difficulty* is a parameter for Proof-of-Work (PoW)-based blockchains, that can change depending on the Blockchain's *validation rate*; it represents the necessary efforts to validate a block by solving the PoW cryptographic puzzle. This is a fundamental parameter to keep PoW-based blockchains secure: due to the fast growth of hardware performance, many possible attacks to the PoW mechanism would be possible in case of too low block validation time. *Difficulty* parameter can change in time to keep the block validation time under a certain threshold,
- **Previous block hash**: a block in the chain can't be altered retroactively without the alteration of all the subsequent blocks: this feature is provided by linking each block to the previous one, thanks to the presence within the block header of the hash of the previous block. If a malicious actor would try to tamper with a transaction within a block in the middle of the chain, the Merkle tree root of that block would be automatically different, causing a chain reaction for all the subsequent blocks (whose previous block hash is now wrong).
- **Timestamp**: specifies the timestamp in which the block was validated;
- **Nonce**: the essential parameter to verify the correctness of the PoW. It is basically the PoW solution, attached to the block after the validation process so that every other node of the network can easily verify it's correctness with a simple mathematical operation (further details will be provided later in this section, when the PoW mechanism will be analyzed more in depth);
- **Transaction counter**: it tells the number of transactions within the block

  Transactions: details about a group of transactions. The content of a transaction can vary through different blockchain-based applications. The fundamental parameters are the sender and the receiver. Each transaction is digitally signed by the sender with its own private key, to prove transaction authenticity. Sender and receiver are uniquely identified by the key used to sign submitted transactions. Transactions not yet included in a block (waiting for being validated by the consensus mechanism) are called "unconfirmed";

2.2.2 Proof-of-Work consensus mechanism

Each node of the blockchain network can set up a new block from a collection of unconfirmed transactions and broadcast it to the rest of the network as a suggestion for what the next block in the chain should be. Because multiple nodes could create blocks at the same time, there could be many options to choose from. It's not possible to rely on the order in which blocks arrive because they may arrive in different orders at different points in the network, causing what is called *forking* of the chain.

Forking is when different peers receive different validated blocks from the network, generating a transition period in which different versions of the chain exists together. It's essential to establish an ordering for the incoming block in a blockchain implementation. Chain forking leads to dangerous attacks regarding the *double-spending* problem [14].

To mitigate this problem, a specific mechanism called **Proof-of-Work** is used.

PoW is basically a cryptographic proof used by one party to prove to others that a certain amount of computational effort has been spent. It was originally employed to mitigate denial-of-service attacks and to prevent email spam. This mechanism can be designed in different ways:

many PoW-blockchain application are based on **Hashcash** [15] system, originally proposed by Adam Beck in 2002 e later used by Satoshi Nakamoto for Bitcoin implementation.

In this case, the Proof-of-Work is the solution to a very difficult mathematical problem that each valid block must possess (the *nonce* mentioned in 2.2.1). This solution must satisfy a simple condition:

$$H(block||nonce) < target$$

Where H is a cryptographic hash function (e.g. SHA256), $||$ is the concatenation operator and $target$ is a threshold value that depends on the blockchain *difficulty* parameter discussed in 2.2.1.

The only way to solve the Hashcash PoW is with a brute-force approach: the validator randomly select a nonce, computes the hash and checks if the result has a certain amount of leading zeros, depending on the threshold defined by the *difficulty*. When the correct nonce is found, the solution is attached to the validated block and the latter is broadcast to the rest of the network. The other peers can immediately verify the correctness of the solution by computing the hash (using the attached *nonce*) and checking if the result is below the threshold defined by the *difficulty*.

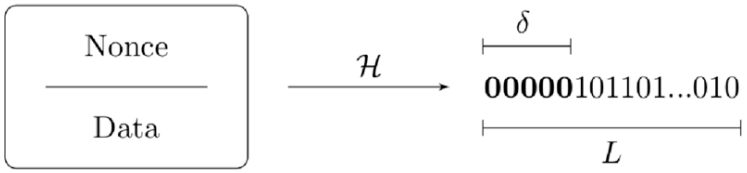

Figure 2.3: Hashcash PoW schema: δ represents the number of leading zeroes of the defined threshold (source: image)

The difficulty is an essential parameter to keep the block validation rate on a fixed value (for Bitcoin is 1 block validated each 10 minutes). If the network exceed this rate, difficulty is increased and Proof-of-Work become more difficult to be computed. The reason why *difficulty* is essential for PoW-based blockchain applications was discussed in 2.2.1.

Basically, many nodes in the network called *miners*, compete to solve the Proof-of-Work to earn a reward (cryptocurrencies) associated with the fact that the necessary computation requires a great amount of energy to be spent. To have a concrete numerical idea, a normal computer would require years to solve a single Bitcoin Proof-of-Work in order to validate an incoming block. This is why many network nodes cooperate together to solve these cryptographic puzzles, creating group of miners called *mining pools* where the final reward is shared according to members' computational effort.

In order to solve the problem of transactions ordering, thanks to the mathematical properties of this mechanism, it's unlikely that more than one PoW is solved at the same time, and it's way more unlikely that this event occurs two times in a row.

2.2.3 51% attack

But what if two blocks are validated and broadcast to the network at the same time? This case leads to the chain forking mentioned in 2.2.1. The forking issue is solved when the next block is validated: all the nodes immediately switch to the longest branch available.

As said before, the math behind this mechanism makes it unlikely for blocks to be solved at the same time, and even more rare for this to happen multiple times in a row. For this reason, blockchain quickly converges, meaning that the the network is in agreement about the order of blocks (apart from the few latest blocks).

The fact that there's some ambiguity in the end of the chain has some important consequences for transactions security. If a transaction is in one of the shorter branches, it will be moved back

to the pool of unconfirmed transactions and will be later included in another block. This scenario can be dangerous: let's suppose to have a blockchain implementation that allows users to store in a transaction a UUID (Universally Unique Identifier) and a public key, that identify a single device, and let's suppose that other users can't later store in another transaction the same UUID. We want to store our device's identity on the blockchain with a new transaction: if a malicious actor could compute Proof-of-Works faster than the rest of the network, he could broadcast a longer branch with forged (but valid) blocks that contain a fake transaction, to store our same identical UUID but with a different public key. Our original transaction would be put back in the pool of unconfirmed transactions, and later, when a new block containing our transaction will be validated, it won't be possible to store a UUID already registered. In Bitcoin protocol, for instance, this attack leads to the double-spending problem.

Luckily, this kind of attacks is practically unfeasible: a malicious actor would be able to broadcast a longer valid blockchain branch only if its computational power is greater than the 50% of the entire network's computational power (practically impossible). In the past, some mining pools have successfully validated 6 blocks in a row and, for this reason, a block is considered secure when it's at least 6 blocks deep in the chain

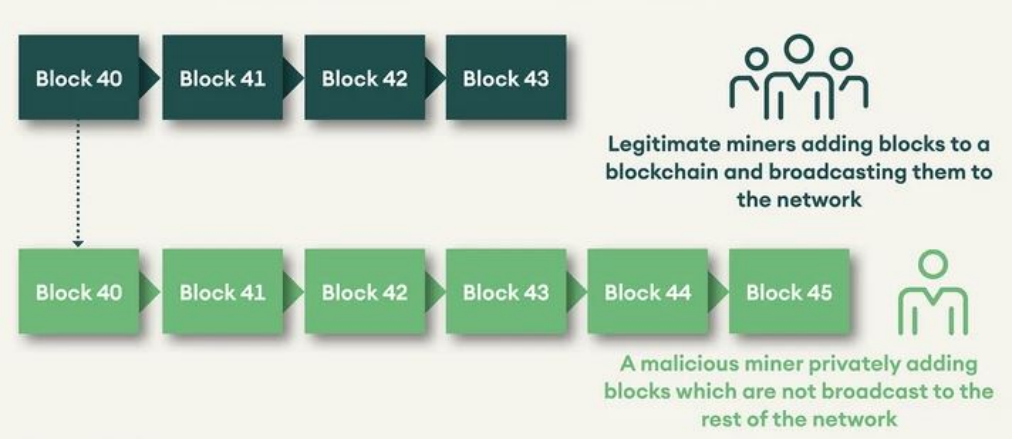

Figure 2.4: 51% attack example (source: image)

2.2.4 Proof-of-Stake consensus protocol

Proof-of-Stake is one of the most known consensus algorithm used in public Blockchains. The PoS was originally proposed to address one of the most debated issue about the PoW mechanism: the energy consumption. The high computational effort spent by miners to solve a Proof-of-Work requires a great amount of electrical energy and in the few last years, because of Bitcoin widespread, this issue came forward due to the great amount of Bitcoin miners attracted by the possibility of a passive income.

Instead of requiring to solve a complex cryptographic puzzle by spending a certain amount of energy, PoS works by providing the user who has the higher stakes in the network a higher probability to be selected to append a new block to the chain. The idea behind this mechanism is that a miner in possession of a great amount of digital assets in the network wants to keep the credibility of the ledger by avoiding fraudulent transactions.

The benefits of PoS mainly regard the energy consumption and the performance: on-chain transaction throughput is higher compared to the PoW, because the there is no heavyweight brute-force computation to be performed to create a new block, and the PoW computational lottery-like process is bypassed.

Despite of this, PoS Blockchains give too much power to the early adopters and rich people, that most likely possess the majority of the tokens of the network. Moreover, PoS is affected by the "nothing-at-stake" dilemma. Whenever a chain fork occurs, it's important for all the miners to keep mining both chains for two reasons:

Figure 2.5: Proof-of-Stake schema (source: [6])

- the validation process doesn't imply a cost for the miner
- if a miner continues to validate blocks for the same branch, he could lose all the profit from any of the time he spent mining that chain in case the alternative branch become longer. Mining both chains ensures the validator the final profit, whatever fork wins

This increases the chances to perform double-spend attacks. An attacker could create a fork in the blockchain one block before he spent some coins. If the attacker keeps mining his fork while all other peers act in their best self-interest by mining both forks, the attacker's branch would eventually become longer. [23].

	Proof-of-Work	**Proof-of-Stake**
Block validation	Computing power determines the chances of mining a block	The amount of stake determines the chance of mining a block
Competition	Competition to solve the cryptographic puzzle	An algorithm decides a winner based on the amount of its stake
Necessary equipment	ASICs and GPUs necessary to mine blocks	A standard server-grade device is sufficient
Efficiency and reliability	Less energy efficient and less expensive, but more reliable	High cost and higher energy efficiency, but less reliable

Table 2.1: Proof-of-Work vs. Proof-of-Stake

2.2.5 Practical Byzantine Fault Tolerance (PBFT) consensus mechanism

This consensus protocol uses a different approach compared with the already discussed PoW and PoS. PBFT is used to reach a consensus based on the Byzantine Fault Tolerance (BFT). Considering $3f + 1$ participants in the network, a system possesses the BFT property if it can resist to $2f + 1$ malicious node inside the network [24].

In PBFT, the involved actors during the entire process are:

- **clients**: send transaction requests
- **primary node**: collects transactions into blocks and finalize them. During each consensus-reaching process there's only one primary node
- **replica nodes**: responsible for block finalization. Each process involves many replica nodes

PBFT protocol is composed by three phases: *pre-prepare* phase, *prepare* phase, *commit* phase. [25]

- **pre-prepare phase**: the primary node verifies the requests and generates the corresponding *pre-prepare* messages to be broadcast to the replica nodes. Replica nodes will verify the legitimacy of the received *pre-prepare* messages and then broadcast a corresponding *prepare* message
- **prepare phase**: nodes collect *prepare* messages; when a certain node collects $2f+1$ *prepare* messages it will communicate to other nodes that it's ready for block submission and starts to broadcast commit messages
- **commit phase**: nodes collect *commit* messages: when a certain node collects $2f+1$ *commit* messages, it will change the system state by processing the original request (locally cached)

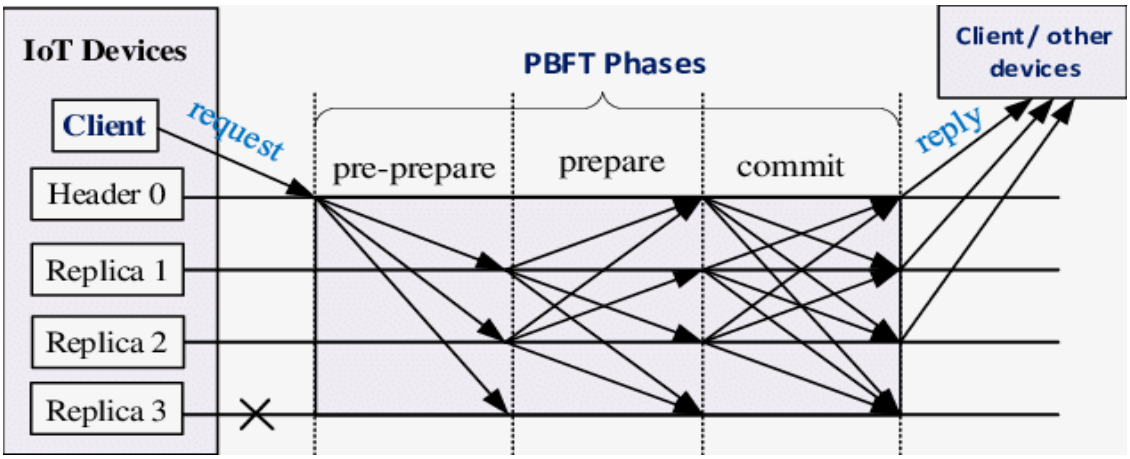

Figure 2.6: PBFT schema (source: image)

When a client receives $f+1$ identical commit messages, the consensus for its request has been reached. This is because it can be proven that $f+1$ identical commit messages contain at least one message from a non-fault node and a non fault-node will only send a commit message when $2f+1$ nodes voted for the request. When the consensus is not reached, the "view-change" protocol is executed: a new primary node is elected to achieve the consensus and respond to the client. The main reason for which this protocol is executed is that replica nodes confirm that in limited time the primary node can't reach a consensus on the request, because it's temporary unavailable or because it's a fault node.

PBFT algorithm improves BFT efficiency and allows to achieve a polinomial complexity instead of an exponential one. In this way BFT is applicable to real systems, offering a strong consistency level.

The problem with PBFT is the huge communication overhead and the limited scalability. When the number of nodes within the network is large enough, this mechanism can't be used because of huge delays to reach the consensus. For this reason, PBFT is usually used in permissioned/private blockchain like Hyperledger Fabric [26], ensuring an higher transaction throughput compared to PoW/PoS.

2.3 Blockchain usefulness for IoT

As previously discussed in the introduction, the main benefit brought by the Blockchain technology is related to its distributed and intrinsically secure nature. Thanks to this property, the research around the IoT scenario is focusing on this new technology, precisely because it's potentially able to solve many issues related to the IoT devices necessity of a continuous interaction with centralized entities for authentication purposes and stored data integrity.

IoT device authentication is one of the "hottest" research topic: many Blockchain-based solutions were proposed in the last 5 years. The main issue that has to be solved resides in the Public Key Infrastructure centralized structure: the Root-of-Trust is basically a Single Point of Failure (SPoF), because in case of attack a malicious actor could provide valid authentication credentials so that the relying party won't be able to distinguish valid certificates from rogue ones. Moreover, when the root CA compromise has been proven, accepting a certificate issued by that CA is dangerous: for this reason, proceeding in the authentication process is strongly discouraged. The natural consequence of this situation is a Denial-of-Service that temporary freezes the functioning of the entire infrastructure.

During the last years, many Blockchain applications have been developed with the aim of moving the Root-of-Trust into the Blockchain itself, thanks to its previously cited security properties and its decentralized nature which would eliminate the Single Point of Failure. A clear example is for sure **Namecoin** [16], a project that shares many properties with Bitcoin (Namecoin is a Bitcoin fork) but additionally provides an overlay that allows to register and transfer unique "names" (keys) associated with arbitrary payload of any type (at most 512 bytes). The most common Namecoin use-case is related to the creation of a decentralized DNS to ensure a secure registration of web domains independently from third centralized entities. After Namecoin, many other Bitcoin forks were developed and some of these are specifically based on Namecoin itself, like **Certcoin** [17] and **Emercoin** [56]. Certcoin is Namecoin-based protocol that aims at creating a distributed PKI using the previosly mentioned properties of Namecoin application. Emercoin provides an overlay called **Emercoin NVS**, very similar to the Namecoin protocol but with the possibility of storing greater amount of data with a set of ad-hoc use-cases specialization for many applications (EmerSSH, EmerDNS etc...).

The decentralized nature of Blockchain technology, also ensures a greater robustness and fault-tolerance to those systems and applications that rely on it. Each member of the peer-to-peer network maintain a verified copy of the public distributed ledger, so it's practically impossible to successfully perform DDoS attacks or to compromise the integrity of the data stored on the blockchain. Any external entity can retrieve the necessary information from a great number of nodes, and this ensures a great level of reliability.

Another great Blockchain's feature is to provide a sequential history of the transactions starting from the creation of the chain. Potentially, for each IoT device, it would be possible to access the list containing all the transactions directly related to it, providing a traceability system that can be employed for further verification on gathered/processed data or executed operations. This property is not irrelevant especially if applied to the expansion of Cloud Computing, which has surely supported the IoT growth by providing a solid infrastructure for data processing (improving the possibility to use this kind of devices for real-time applications with strict constraints in terms of response time) but that doesn't provide a great data transparency level.

2.3.1 Smart contracts

Smart contracts are computer programs that allow to control and execute some operations/instructions when specific contractual conditions (defined by the program) are satisfied. Smart contracts technology has become widespread thanks to the employment of this technology as a key feature of the Ethereum project [19].

Differently from the "standard" Blockchain platforms usually represented by a "ledger", Ethereum Blockchain is represented by a great distributed state machine (the Ethereum Virtual Machine [20]). Besides containing every account and transaction, the EVM is the core component that defines the rules for a state transition of the machine, every time that a new block is appended to the chain. For Ethereum, the EVM is what allows the creation and the execution of smart contracts: they're represented by special accounts (addresses) that if addressed by a message call, allow the execution of the bytecode registered at their creation time. The bytecode is usually computed starting from an high-level programming language, that for Ethereum is Solidity. Solidity is an expressive and powerful programming language that allows to write very complex decentralized applications (DApps).

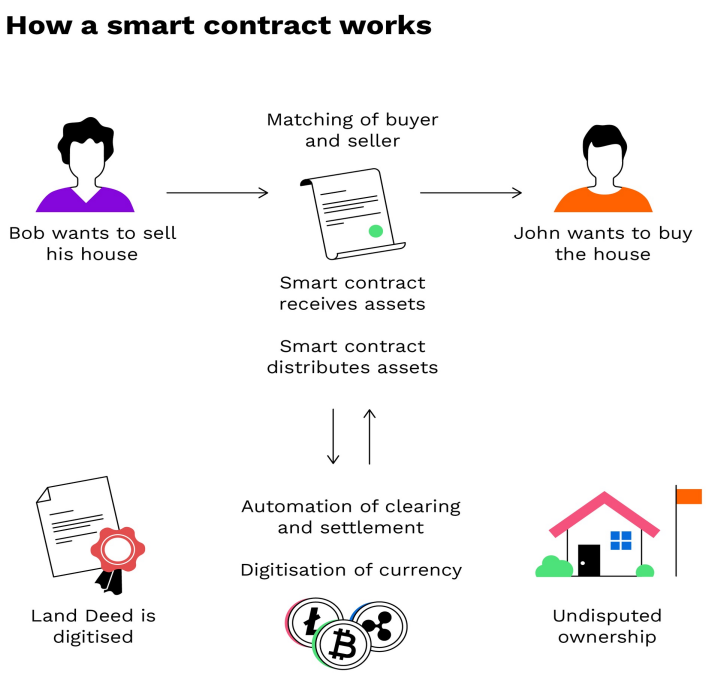

Figure 2.7: Smart contract functioning (source: image)

It's important to notice the Smart Contracts are stored in the chain by following a protocol similar to the one discussed in the Blockchain-dedicated section: for this reason, they have the same security (and especially integrity) properties of the transactions stored on a "standard" ledger. So, it's very difficult to compromise the bytecode of a Smart Contract in order to modify its instruction and functioning: when the Smart Contract is created, the code can only be modified by creating a new contract with a new address.

Smart contracts are an expanding technology and can be employed in many use-case scenarios, from real estate to voting, trade finance, digital identity, auto insurance (etc...), thanks to the fact that in most of these scenarios this technology allows to reduce the cost of contractual procedures, and it potentially ensures a greater security level without the necessity of any external intermediary. In the IoT context, a Smart Contract can help the automation of those tasks that depend from a variation of an external state: generally, IoT devices provide limited interaction options and an automation mechanism for the execution of any device operation could be relevant.

For instance, it would be possible to design a Name Value System (similar to the one proposed my Namecoin/Emercoin) with the additional feature to automatically renew an expired name-value pair (with its associated X.509 certificate) by using a Smart Contract that can be triggered by some specific time conditions. It's important to keep in mind that due to the fact Smart Contracts are represented by an address within the peer-to-peer network, a single instance of a Smart Contract can invoke any function of any other Smart Contract in the Blockchain (as if it was a normal network entity): this feature is a very powerful tool for linking many pieces of bytecode in order to design more complex solutions.

2.4 Challenges for Blockchain integration in IoT

Despite of the great amount of benefits that Blockchain could potentially bring to the IoT, there are many requirements that must be satisfied for a system to adopt it. In the initial phase of the research, a great effort was spent in evaluating which applications could potentially benefit

from Blockchain technology, but the difficulties associated with this integration if applied to some constrained environments were usually underestimated. The Internet-of-Things is a perfect representation of this problem: as previously discussed, IoT devices are usually characterized by a large variety of hardware components employed in their production, and generally they're not featured with a good level of computational power and storage capabilities.

2.4.1 Scalability

In blockchain systems there are two categories of latency: [21]

- **Block latency**: necessary time to attach a new block to the chain. This time can be different among many Blockchains, and it is usually defined by design. For instance, in Bitcoin the block generation rate must be one block every 10 minutes: if this threshold is exceeded, the difficulty of the Blockchain increases to slow down the validation process. This is an important bottleneck that can't be avoided because it's a design property of the Blockchain technology itself, necessary to provide specific features (other Distributed Ledger Technologies with mechanisms and properties different from the Blockchain ones, may not have this limitation);
- **Transaction latency**: necessary time to include a transaction in a block and attach the latter to the chain. It depends from the block latency and the number of transaction per block (can vary among many Blockchain implementations). Moreover, it's possible that our transaction is not immediately included in the next upcoming block for many reasons, for instance the fee of a transaction is too low and doesn't incentivize some miners to include it in a block to confirm.

These latencies are one of the most discussed problem regarding Blockchain technology. When the number of entities that form the peer-to-peer network grows up very fast, the transaction latency increases a lot, forcing the nodes to use very high fees to have their transactions confirmed faster. It's easy to figure out that for IoT scenario, scalability can be a real issue, because the number of IoT devices is growing exponentially.

Despite of this, in a potential distributed PKI the number of transactions is for sure strongly reduced. In this scenario, transactions would be used to register/revoke/modify an identity (and its public key) and, for this reason, it's actually essential to adopt a Blockchain implementation specifically designed for this usage. Using common Blockchain protocols designed for many purposes (like value exchange) like Bitcoin or Ethereum, is self-defeating because the block latency and the transaction cost would be too high and the final solution would be practically useless.

2.4.2 Storage size

Another very important drawback that comes up in the discussion about Blockchain technology is the storage size requirements. Each node of the network must possess a full copy of the ledger, from the genesis block (block #0) to the last one. Block size is $\approx 1MB$ (on average) and for this reason, the size of the entire ledger can be extremely heavyweight for many Blockchain applications

Blockchain	**Focus area**	**Full ledger size**
Bitcoin	Diverse	422GB
Ethereum	Diverse	863GB
Namecoin	Key-value registration	6GB
Emercoin	Key-value registration	495MB
EOS	Smart contract based applications	Undefined (many TBs)

In this table, Ethereum and EOS blockchains are extremely heavyweight because many DApps (Decentralized Applications) are hosted on these platform. Decentralized applications use Blockchain

platform to provide different services (decentralized finance, healthcare, voting and many other possible applications). For each interaction with a decentralized application based on a smart contract, a new transaction must be created and attached to the underlying ledger, so the number of transactions is far bigger compared to other Blockchains that do not provide the smart contracts feature.

It's obvious that the size of the Blockchains focused on diverse areas is much larger compared to the Blockchains designed for a restricted pool of applications, like the aforementioned Emercoin and Namecoin for name-value registration

The point is that the magnitude of these storage requirements is not suited for IoT devices and, in general, for low-storage devices. For this reason many Blockchains provide also lighter protocols (e.g. LES, Light Ethereum Subprotocol [58]), that store only the block headers. This reduces a lot the constraints on available storage but requires interactions with a full node to retrieve information about transactions within a block and in some cases the reduction of the necessary storage is not sufficient to satisfy IoT devices conditions.

2.4.3 Transactions cost

In the previous subsection about scalability issue in Blockchain-IoT integration, another important drawback was also anticipated. In recent years, cryptocurrencies and Blockchains have become widespread: due to this increasing statistic, the usage cost for major Blockchain applications has increased drastically. The usage cost related to a Blockchain application depends on the amount of required fees to submit a transaction and get it validated by miners in a reasonable time.

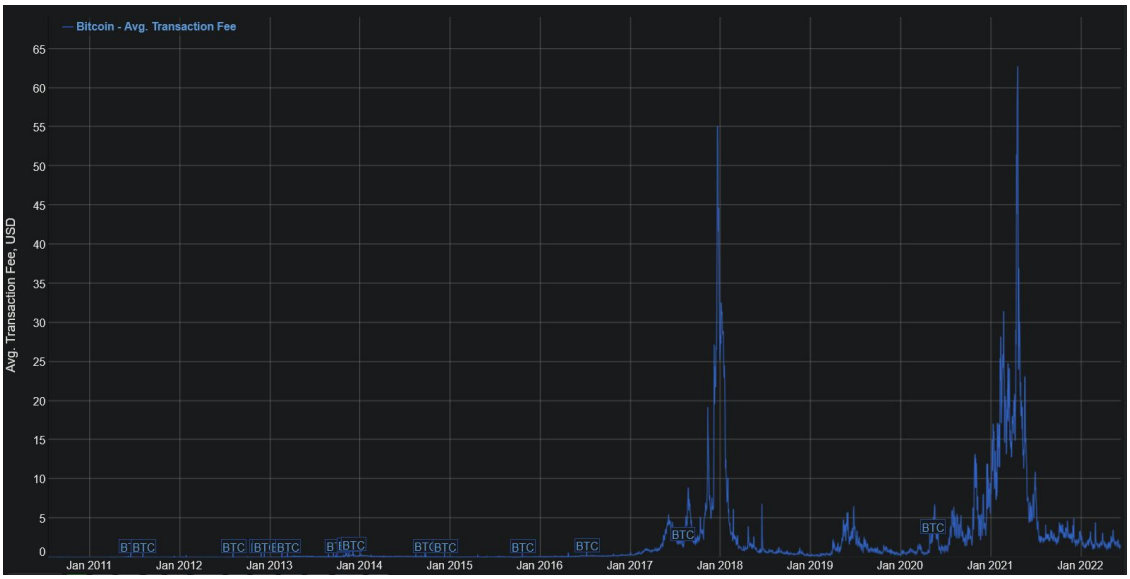

Figure 2.8: Bitcoin average transaction fee: all-time chart (source: image)

Transaction fees are an important element in public blockchain applications. Miners are incentivized to validate blocks with the special reward mentioned before: by design, public blockchain applications have a special mechanism to generate value scarcity and without this design feature, any cryptocurrency could lose its own asset's value (because of inflation). In Bitcoin, for instance, the reward for blocks validation is halved every 4 years and in 2140 no more rewards can be obtained by miners. For blockchain applications to be sustainable, the simple reward is not a sufficient incentive for miners. When miners validate a block successfully, they get the validation reward plus the sum of all the fees of each transaction within the block. Due to the fact that the reward decreases every 4 years and Proof-of-Work difficulty can increase in time, fees cost can also increase.

It's essential to carefully evaluate the cost for employing the Blockchain technology for each possible IoT scenario. An eventual widespread of these Blockchain-based architectures for IoT

would imply an increase of transaction cost, and this consideration must be taken into account when similar approaches are compared to the standard centralized ones. In our scenario, it's reasonable that a distributed PKI based on Blockchain would not generate an high transactions throughput: in fact, the main purpose of this proposal is just to move the trust-anchor from centralized entities to a distributed platform, using Blockchain as a trusted database that contains secured information about devices certificates and their validity status. In other scenarios, Blockchain could be useful, for instance, to guarantee about the integrity of data collected by IoT sensors: in this case the potential transaction throughput is incredibly higher, and for this reason projects like IOTA [28] try to propose different solutions (e.g. Tangle [29]) to minimize (or remove) transaction cost (and confirmation time).

2.5 Distributed PKI solutions

2.5.1 Public Blockchain-based PKI in IoT scenario

A distributed PKI based on blockchain is theorically feasible. Some issues come up when these concepts and designs are applied to some scenarios that differ from the common ones. In fact, our purpose is to implement a distributed PKI that can fit IoT devices (and constrained-resource devices in general) and Blockchain technology integration with IoT can be difficult for many reasons also discussed in the previous section. Blockchain applications require in general a great amount of storage to be used. As said before, the idea is that each communicating entity relies on a self-owned copy of the blockchain but constrained-resource devices can't afford this requirement.

For this reason, the focus is on proposing an efficient design solution that can overcome this obstacle. One possible approach is to delegate blockchain operations and storage to a powerful, trusted node. A communication scheme must be designed to define how the end-IoT-device can perform all the necessary Blockchain operation through the intermediate trusted node.

Many works like [37], [38] [39] have followed this direction, also describing detailed protocols for a set of operations related to IoT devices certificates. [37] and [38] describe basically the same design: [37] is based on Emercoin Blockchain and [38] it's an evolution of the original schema that uses Ethereum smart contracts. Smart contracts-based applications depend on the bytecode stored in the blockchain, so they leave more space to design more sofisticated solutions. Moreover, the scope of [38] was also to compare the two solutions in terms of costs, performance and storage requirements.

In this schema, the key actors are the **B-nodes** distributed in the Internet, powerful nodes that are capable to run a full blockchain node thanks to high storage capability and good computational resources. These nodes can be run by everyone in the network, and represent the connection between the end devices (constrained-resource devices) and the blockchain platform itself. Manufacturers, ISPs, and even individual users have the possibility to run some full B-nodes.

An IoT device uses some trusted B-nodes in the network as a bridge connection to query the Emercoin NVS platform and retrieve information about a specific key-value pair. In a possible scenario, a group of IoT devices within a network is controlled by the devices owner. The administrator hosts a pool of private B-nodes within the network also to provide more robustness to the system. In this initial state, the pre-condition is that each IoT device in the network is waiting to be configured: an IoT device in the initialization state doesn't possess any certificate that binds its identity to a specific public key, so it can't estabilish secure communications with other devices within the same network.

The sequence of necessary operations for the initial configuration is described in the device setup protocol: the basic idea is that the device owner itself is responsible for its own devices certificates, without any necessary intermediary. Each IoT device generates an identity and a self-signed certificate associated with the generated ID: these two elements are going to represent the <name, value> pair registered by the **Device Manager(DM)** (one of the trusted B-nodes within the network) on the Emercoin NVS platform. As previously discussed, the Emercoin consensus protocol ensures the uniqueness of each name stored on the ledger. With this core property, the

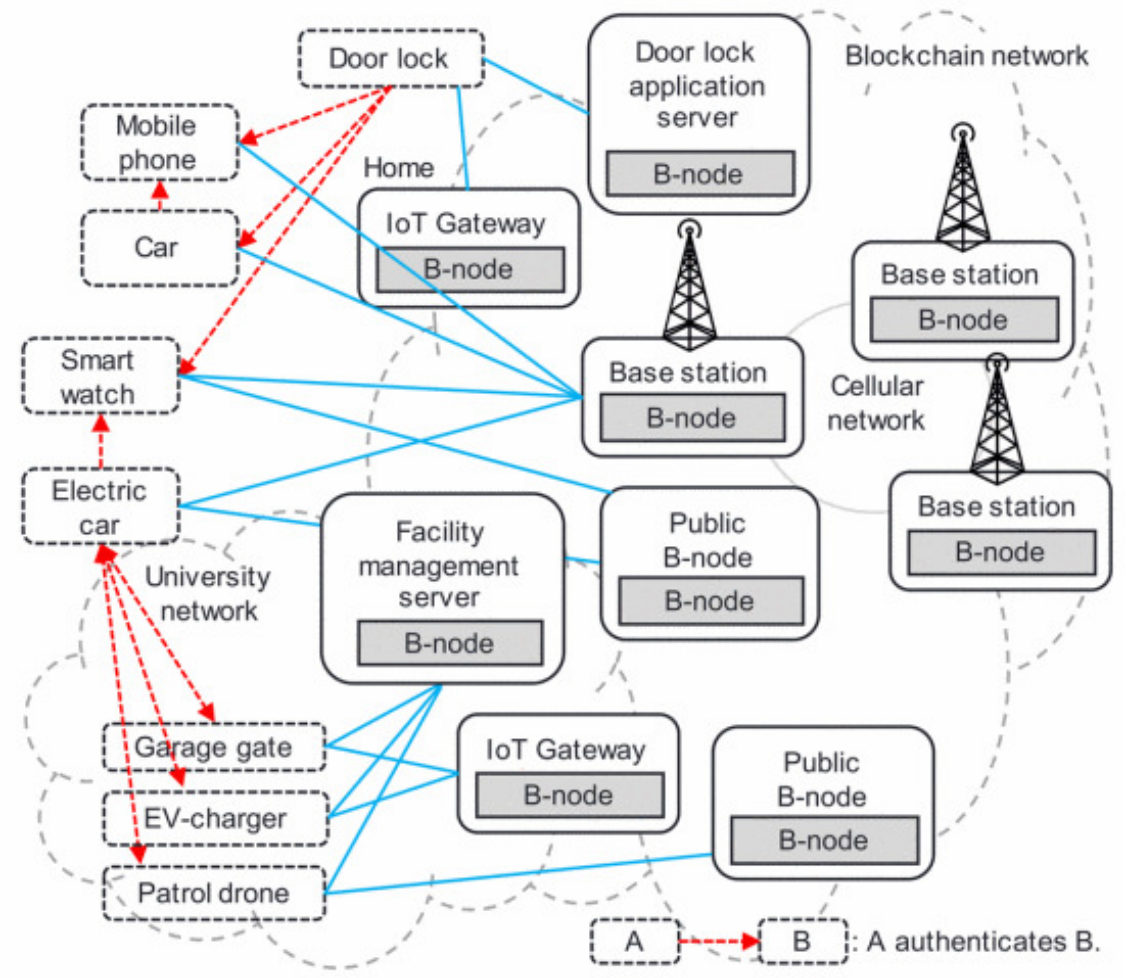

Figure 2.9: IoT-PKI architecture proposed in (source: [37])

certificate's hash associated with the unique name stored on the ledger can't be altered, thanks to the blockchain integrity properties previously discussed. Basically, the Registration Authority is represented by the Device Manager and the Root CA (the trust-anchor) is moved to the blockchain platform. When two devices are successfully configured, they're finally able to estabilish a secure communication channel (e.g. TLS channel) for data exchange: when a device receives an X.509 certificate from the other peer, instead of checking the validity of a certificate chain it has to compute the hash of the certificate and compare the result with the value stored on Emercoin NVS, whose corresponding name is defined within the certificate itself. If the two hashes are equal and the certificate is not expired, the authentication is successful.

In addition to the device setup protocol, [37] proposes three more schemas for device ownership transfer, device certificate update and revocation. Device ownership transfer protocol ends up with a transaction on Emercoin NVS where the operation type is `update`, the name involved in the transaction is the device identity and the recipient address is not equal to the sender one (the old device owner) but it's replaced with the address of the new owner. When this transaction is confirmed, the old device owner won't be able to generate transactions that modify the state of that specific ¡name, value¿ pair.

The `update` operation in Emercoin NVS is also generally used to update the information associated with a name (like the associated value or the expiration date): in this case, the recipient address and the sender address will be equal.

The last possible operation that can be performed on a <name, value> pair is the revoke operation. When a device private key is compromised, the DM can simply submit a `revoke` transaction to Emercoin NVS, without further interaction with the compromised IoT device.

Apart from physical tampering attacks, this solution is affected by few problems related to the device setup protocol. The protocol design doesn't take into account the possibility of "lying" end-entities and furthermore an IoT device in the initialization state can't distinguish a message from a real Device Manager of its network from the initialization message of an eventual malicious actor. These issues will be discussed later and some possible solutions to address them will be provided.

With this approach, the core benefit is for sure the ease of devices certificate management. Using a classical CA-based PKI, the entire process to obtain/revoke a certificate for a single device is slower and less sustainable also because dealing with certificate expiration/revocation is much more complicated if compared to the proposed solution. Moreover, the OSCP DoS attacks that affect CA-based PKI are mitigated, because it's possible to keep a pool of DM nodes without exposing their interface to the Internet. The core purpose of a DM node is to act as a reliable ledger replica for the network of IoT devices in which is operating, so basically a single device can perform all the necessary verifications for incoming certificates without interacting with external entities.

The most important challenge here is to verify that an IoT device is not lying about its claimed identity. This task is usually performed by a Certification Authority (or Registration Authority) that requires verifiable credentials before issuing a certificate for a specific entity. In this solution, each IoT device can generate its own identity and the DM does not perform any check to estabilish a binding between the claimed identity and some intrisic property of the device. The Trusted Computing Group (TCG) has published different specifications that define rules and procedures to perform the identity attestation of a device using a Trusted Platform Module (TPM) [53], but another useful technology to achieve this goal is the Physical Unclonable Function (PUF) [54]. In the next chapter, further details about the identity attestation using TPM will be provided and discussed.

A possible evolution of this design has been proposed by the same authors in [38]. The goal of this work was to provide an alternative to the NVS platform based on Emercoin blockchain by using Ethereum smart contracts, to evaluate the possible benefits/drawbacks of the two solutions. Ethereum-based approach is evaluated in two different forms: in the first case, the high-level design is basically identical to the previous one based on Emercoin NVS, with the presence of a trusted full Ethereum node. In the second case, Light Ethereum Subprotocol is used to eliminate the intermediary remote full node. A LES node stores only the header of each block of the chain, reducing the necessary storage requirements to run a blockchain node. When a LES node needs to retrieve information about specific transactions within a block, it queries an external full node and verifies the integrity of the received data using Merkle Trees theory.

The core benefit brought by Ethereum smart contracts to the distributed PKI architecture is the storage flexibility. Compared to the simple name-value storage, a smart contract provides an high-level language with the possibility to use complex data structures (e.g. maps or custom user defined structures) and to define arbitrary rules to enhance the system functionality and add new custom features. The authors of [38] propose an high-level design of the deployed smart contract, by describing the function signatures and their purpose plus the data structures used to store the information:

- `AddDevice(DeviceID, Hash)`: device data is stored in a hashmap of `<DeviceID, DeviceDetails>`, where `DeviceDetails` is a structure that contains the device owner address, the hash of the X.509 certificate and the validity status of the latter. With this function, the two input parameters are used to add a new entry to the hashmap
- `RemoveDevice(DeviceID)`: removes from the hashmap the entry associated with the input `DeviceID`
- `GetDeviceHash(DeviceID)`: returns the certificate hash corresponding to `DeviceID`.

After the deployment of the smart contract, everyone can call these functions with simple HTTP requests to an Ethereum node that hosts an RPC server (localhost in case of LES-based design), by following the ABI specifications provided by the Solidity official documentation [55].

```
contract PKI {
    struct deviceDetails {
        address deviceOwner;
        bytes32 certificateHash;
        bool valid;
    }
    mapping(bytes16 => deviceDetails) private certs;
    modifier onlyOwner(bytes16 deviceId) {
        require(msg.sender == certs[deviceId].deviceOwner,
        Only the owner of this device can perform this operation.");
        _;
    }
    modifier nonExistent(bytes16 deviceId) {
        require(certs[deviceId].certificateHash == 0, This device ID
            already exists.");
        _;
    }
    function addDevice(bytes16 deviceId, bytes32 certificateHash) public
    nonExistent(deviceId) {
        certs[deviceId].deviceOwner = msg.sender;
        certs[deviceId].certificateHash = certificateHash;
        certs[deviceId].valid = true;
    }
    function removeDevice(bytes16 deviceId) public onlyOwner(deviceId) {
        certs[deviceId].valid = false;
    }
    function getDeviceHash(bytes16 deviceId) public view returns(bytes32)
        {
        return certs[deviceId].certificateHash;
    }
}
```

Figure 2.10: One possible implementation for the smart contract proposed by [38]

The issues discussed for the original design based on Emercoin NVS are still present in this solution. Furthermore, the storage flexibility provided by the Ethereum smart contracts comes with an important issue related to the cost of this solution: actually, one of the most debated problem related to Ethereum regards the high transaction fees. Ethereum fees have reached a cost of more than 40$ during last years, and every time that a smart contract function that stores data within a structure is called, the state of the Ethereum Virtual Machine changes, and a new transaction must be created. `AddDevice()` and `RemoveDevice()` functions act on the state of the chain and every time they're called, a new transaction is generated (and some fees must be paid).

By comparing the two solutions, the potential benefits brought by the second one based on smart contracts are not enough to justify a cost far higher than the first solution based on Emercoin NVS. These considerations about the cost can be made for many other proposed solutions based on Ethereum Blockchain.

For instance, an interesting slightly different model based on Ethereum is proposed by [39]. Here, the architecture is called Distributed Public Key-store (DPK) and it consists of three elements:

- **Public Key Manager (PKM)**: authenticates DPK users and approves Blockchain storage requests. This function can be a global platform or part of a dedicated network for any device configuration. PKM sends to the client the necessary amount of Ether to store the user public key on the ledger after the approval of the PKM module itself

- **DPK Client Module**: software module installed on a device, that creates an Ethereum address and stores a generated public key on the blockchain thanks to the interaction with the PKM. Before its installation, a configuration phase is necessary to provide the client with a unique token that will be used for authentication. During the configuration, the client should select the type of DPK identity: *addressable* or *non-addressable*. In case of an *addressable* identity, the user must prove the ownership of the claimed identity, otherwise for a *non-addressable* identity a UUID is generated. The provided token is crucial to ensure that the transaction fees required to store data in Ethereum have been paid
- **DPK Smart Contract**: the core element that supports the architecture logic. In this case, the smart contract provides three functions: `addClient()`, `getClient()`, `approveClient()`

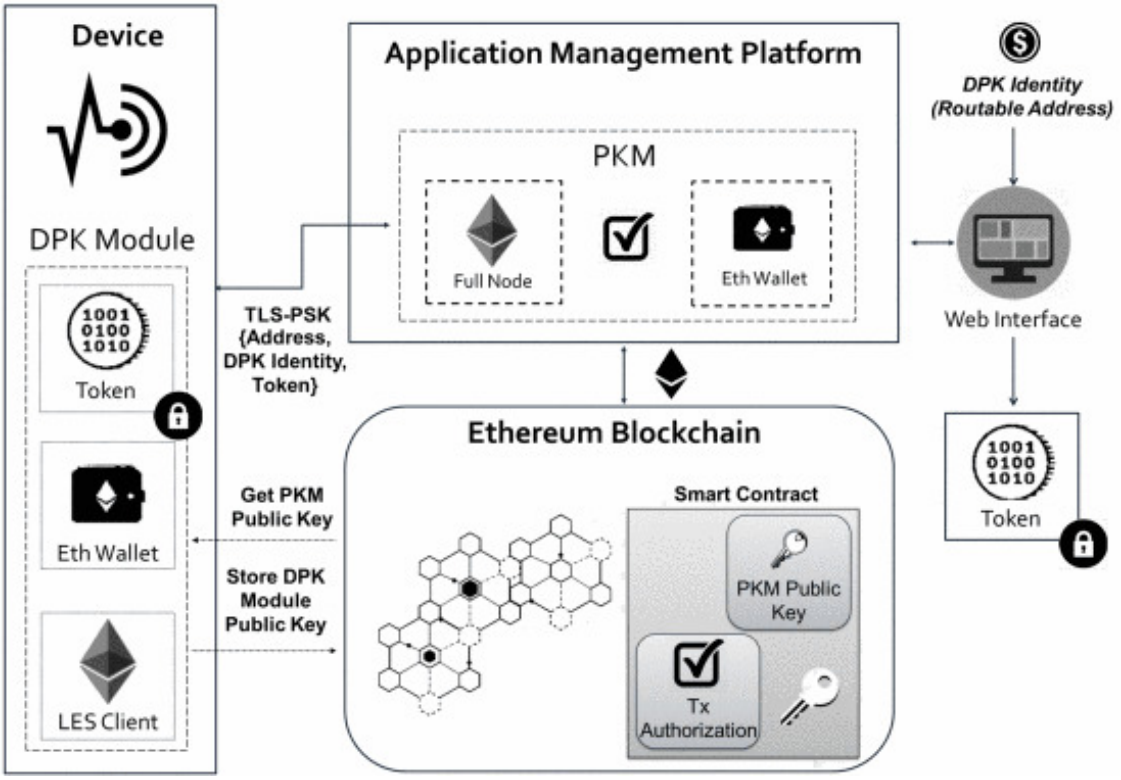

Figure 2.11: Design proposed by [39]

To approve the registration of a public key, a specific procedure must be followed:

1. The client calls `addClient()` with 3 input parameters: DPK client identity, public key and the token provided at configuration time. Now the client has been added to the pending client list
2. The client sends to the PKM an approval request providing the signed token
3. The PKM calls `approveClient()` passing as input the token associated with the requesting device: only the contract owner (the platform provider) can call `approveClient()`

Like [38], the problem here is also worsen because each device has to receive with a transaction from the Platform Provider the necessary Ether to register its own public key on the chain. After that, the client calls `addDevice()` function to append its own identity to the list of pending clients (this operation requires a certain amount of fees to be paid) and finally, the platform provider is going to approve the client with an additional invocation of a smart contract function (`approveClient()`), which also requires some fees. By complicating the smart contract logic, the main consequence is for sure the usage cost increase. Moreover, in this case, the Platform Provider isn't contextualized and further details or examples about the kind of entity that could

cover this role are not provided (it's important to take into account the operational cost that the Platform Provider is supposed to deal with).

The core advantage is to have an entity in charge of verifying each claimed (addressable) identity, that will be active and capable to provide authentication using DPK architecture only after an explicit approval from the Platform Provider. Despite of this, as previously anticipated, the cost is even greater compared to the solution proposed in [38].

2.5.2 Private/Consortium Blockchain-based solutions

A Blockchain can also be implemented as a private platform instead of a public one like Bitcoin and Ethereum. Private Blockchains can be useful in specific use cases where the number of participating nodes is limited and controlled by specific authorization rules provided by the platform host. This property allows private Blockchains to take advantage of efficient consensus protocols (like PBFT mentioned before) that solve many important Blockchain issues about scalability, cost and transaction throughput.

Consortium Blockchains are similar to the private ones, but the platform host is represented by a consortium of entities that define rules and the role of each participating node. Each consortium entity generally holds a portion of the network and join the consensus protocol together with the other hosting parties. One of the most important project related to consortium Blockchains is **Hyperledger Fabric** [26], supported by The Linux Foundation.

Hyperledger's purpose is to provide an open source enterprise-oriented DLT (Distributed Ledger Technology), to drive a massive adoption of this technology by the companies. Hyperledger addresses consortium networks, where a group of stakeholders is interested in the adoption of a decentralized, transparent network where the validation of data is not in charge of a central authority. In order to concretize this approach, Hyperledger provides a great modularity and flexibility to satisfy many potential industries and use cases. One of the core benefits of using Hyperledger instead of a public Blockchain, is that in some enterprise use-cases some privacy is necessary to protect the commercial agreements: public Blockchain transparency can be helpful for many cases, but it can also be an obstacle for these specific scenarios. In Hyperledger Network, only parties directly related to the transaction deal are updated on the ledger, thus maintaining privacy and confidentiality. This is possible thanks to the channels, virtual blockchain networks built upon a physical blockchain network, that have their own access rules.

Hyperledger blockchain runs programs called "chaincode", and each transaction is basically an invocation of the chaincode. Transaction must be "endorsed" to definitely modify the blockchain state. Transactions may be of two types:

- **Deploy transactions**: create new chaincode. After a deploy transaction, the chaincode has been "installed" on the blockchain;
- **Invoke transactions**: perform an operation by invoking one of the functions provided by the deployed chaincode. When the chaincode is executed, the blockchain state may change;

As anticipated before, an Hyperledger network is hosted by a consortium of collaborating parties. Each party controls a portion of the nodes in the network, and in Hyperledger different nodes can cover different roles:

- **Client**: is basically the end-entity. It must be connected to a **peer** for communicating with the blockchain in order to invoke the chaincode and generate new transactions;
- **Peer**: it maintains the state of the ledger, receiving ordered updates (blocks) from the **ordering service**. **Endorsing peers** are special peers whose function is to endorse transactions before they're committed. The endorsement task is related to a specific chaincode: every chaincode specifies its own "endorsement policy" and a set of endorsing peers. The endorsement policy defines the necessary conditions for a valid transaction endorsement (typically a set of endorsers' signatures);

- **Ordering service nodes (Orderers)**: provides delivery guarantees. Ordering service provides a shared communication channel to client and peers, where they can broadcast messages containing transactions. This channel supports the atomicity of the delivery so that the channel outputs the same messages to all the connected peers in the same logical order. Thanks to the atomic communication, the network is able to reach the consensus. As anticipated before, the ordering service may support multiple channels, similar to the topics of the publish/subscribe systems;

To invoke a transaction, a client sends a `PROPOSE` message to a set of endorsing peers. When an endorsing peer receives a `PROPOSE` message, it initially verifies the client's signature and then simulates a transaction. To simulate transaction execution, the endorsing peer invokes the chaincode to which the transaction refers, using the state it's currently holding. Then, the peer forwards internally the transaction proposal to the part of its logic that endorses transactions (**endorsing logic**). Basically, the endorsing logic accepts the transaction proposal and signs it, but it's possible to define arbitrary functions to achieve specific behaviours. If the endorsing logic decides to endorse the transaction, the peer sends back a message (`TRANSACTION-ENDORSED`) to the client.

The client waits until it receives enough messages (and signatures), depending on the endorsement policy, to conclude that the transaction has been endorsed. At this point, the endorsement is broadcast using the ordering service: each peer performs some checks on the endorsement, and if they pass the transaction is marked as committed and the modifications to the blockchain state are applied.

The endorsement policy for the chaincode is defined by the consortium that hosts the Hyperledger network, providing a great flexibility about the necessary conditions to achieve the consensus for transactions endorsement.

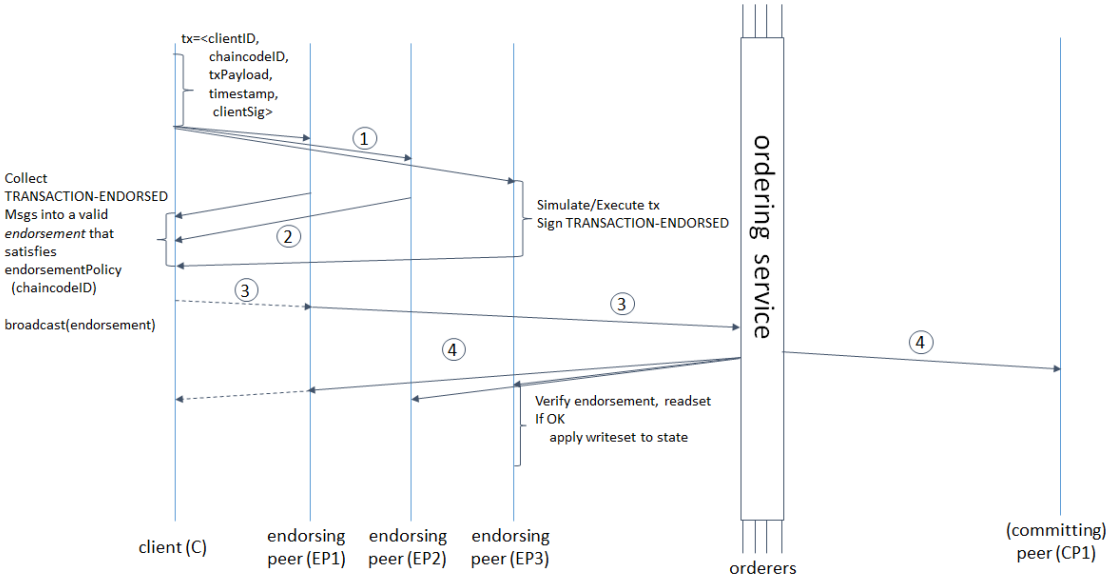

Figure 2.12: A possible Hyperledger Fabric transaction flow (source: [9])

Many proposed solutions are based on Hyperledger projects, like Fabric [26] or Sawtooth [27]. In [49] each IoT device is blockchain-enabled thanks to the consortium-based approach that drastically reduces the amount of necessary storage for the maintenance of the ledger. IoT devices are simple network clients, able to perform identity verification on their own (by retrieving data from their local updated copy of the ledger). The rest of network is composed by many peers and orderers provided by the entities that form the consortium.

The smart contract has been developed using Golang (Go) and each transaction that is going to modify the state of the blockchain is identified by 4 elements:

- **Operation**: a tag that identifies which CRUD operation will be performed;

- **Asset type**: n-letter code that identifies the type of the involved asset (e.g. "ss" for sensor devices);
- **Nonce**: an incrementing counter that ensures protection against replay attacks and potential malicious orderers;
- **Asset identifier**: identifies a specific asset within its own category;

To store the metadata associated with each specific device, a simple 32-byte string is used instead of complex structures of arbitrary length. The 32-byte string allows to store a 256-bit hash of the metadata.

This solution can be highly efficient in an Industrial-Internet-of-Things scenario, thanks to the important benefits brought by Hyperledger Fabric implementation previously discussed (high flexibility, open source code, high transaction throughput, cost free). Despite of this, the applicability of this solution is restricted to the specific case where two or more partner companies want to protect their communication channels and data exchange, using a distributed approach. In case of a single entity, Hyperledger Fabric's logic can't be applied, and many security-related benefits of public blockchains can't be guaranteed.

2.6 IOTA

IOTA [28] is a distributed ledger technology that has been originally developed with IoT in mind. The core problem that IOTA wanted to solve was related to the blockchain scalability. As previously discussed, blockchain transactions cost is also related to the transactions volume: if we consider a great amount of potential IoT devices participating to a blockchain network, the cost and the delay issues can't be ignored. The necessary cost to support transaction validators exists because the network participants can decide to use the blockchain without joining the consensus mechanism and by delegating the validation task to the other participants that decide to contribute in exchange of a reward (miners). To solve this issue, IOTA proposes a completely different way to implement a distributed ledger: the *Tangle* [29].

2.6.1 Tangle

IOTA Tangle's core principle is that each network participant must be part of the consensus mechanism and contribute to the validation of each new block. If a node wants to submit a new transaction to the Tangle, it must help the network by validating 2 other pending transaction from different nodes (in this case a "block" represents a single transaction). This key feature allows the IOTA network to speed up proportionally to the transaction volume and efficently solves the scalability issue (eliminating fees necessity). This is possible because the Tangle structure is far different from the Blockchain: if the Blockchain can be associated with a linked list, the Tangle is basically a DAG (Directed Acyclical Graph).

For this reason, a different consensus mechanism is necessary to address the forking problem: the Tangle's consensus is based on cumulative weight. The cumulative weight of a transaction (a vertex of the graph) is the sum of its own weight (proportional to the computational effort invested into the Proof-of-Work) and the weights of all the other vertexes that directly or indirectly approves it. A transaction is accepted by the whole network if its corresponding weight is above a certain threshold. When a user generates a new transaction, two *tips* (edge vertexes of the DAG) without conflict are selected according to a Markov Chain Monte Carlo (MCMC) algorithm [30] and the hash of the selected chips is included within the new transaction; a very simple PoW is solved to avoid transaction spamming and, after this, the transaction is broadcast to the rest of the network that checks its validity: the valid transaction is finally added as a new tip to the Tangle and waits for its cumulative weight to reach the consensus threshold.

To avoid the double-spending problem, Blockchain consensus mechanisms uses the longest chain as the criterion to choose between two branches after a forking, because the longest chain

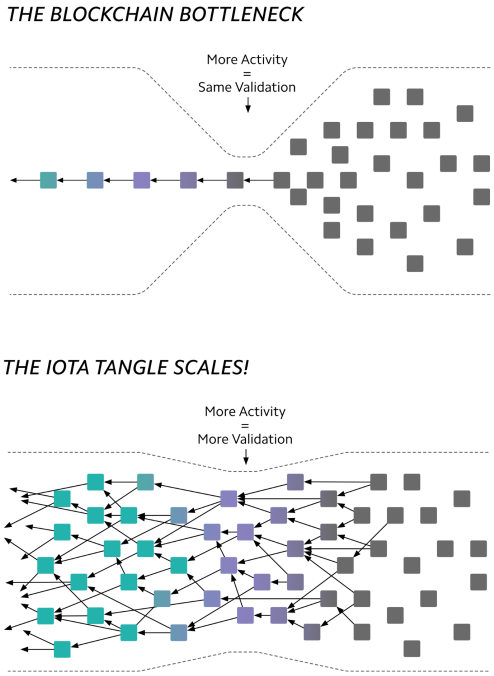

Figure 2.13: Tangle vs. Blockchain bottleneck (source: image)

has the lowest probability to be replaced with a forged branch, A similar approach is used by the Tangle, by using the MCMC tip selection algorithm to select the branch with the largest cumulative weight: in this way, the overall computing capabilities of honest IoT nodes is powerful enough to avoid double-spending while the invidual necessary computing effort is drastically reduced.

2.6.2 DAG-based consensus: challenges

The Tangle is a brilliant solution to many DLTs common concerns. Despite of this, the DAG based consensus mechanism is not perfect. From a theoretical point of view, the Markov chain based model is characterized by a significant problem regarding the transition probabilities matrix, especially in case of a huge number of system states. For this reason, Markov chain based model requires some optimizations.

Moreover, an important paramater to take into account to analyze the DAG based consensus mechanism is the transactions arrival rate. It's impossible to assume a stable transaction arrival rate (especially in IoT systems) and since the finality of a transaction is determined by its cumulative weight, the confirmation delay can increase a lot if the transactions arrival rate is too low. This problem is currently addressed using a **coordinator** [31]: in IOTA, the Coordinator is a client that sends special signed messages called *milestones*; a message in the Tangle is considered for confirmation only when it's directly of indirectly referenced by a milestone that nodes have validated. For the milestones to be recognized, all IOTA nodes on the same network are configured with the signatures of a coordinator node which they trust, so that they can validate milestones signatures to verify if a trusted Coordinator did sign them. To ensure that new messages have a chance of being confirmed even if the transactions arrival rate is too low, the Coordinator sends milestones in the network every 10 seconds. Even if the Coordinator can address the issue related to the low transactions arrival rate, on the other hand it introduces a centralized element that collide with the original decentralized nature of this technology.

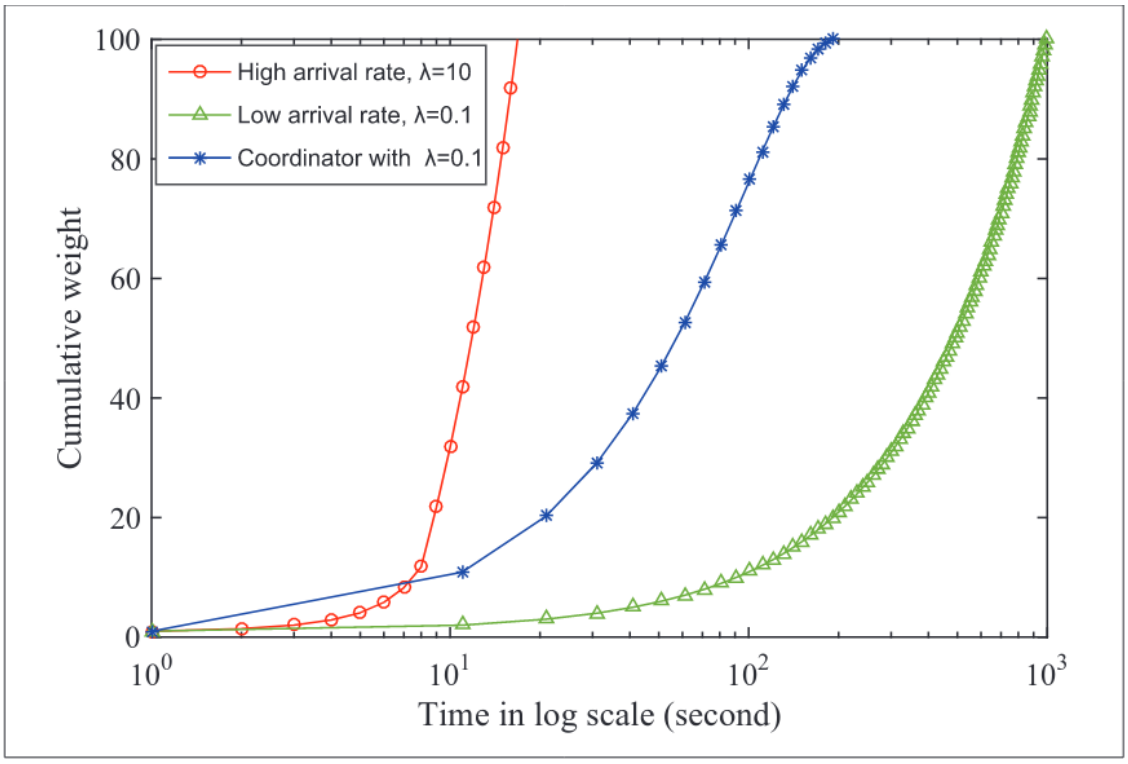

Figure 2.14: Cumulative weight growth curve in different regimes (source: [35])

	Bitcoin	**Tangle**
Transaction fee	Yes	No
Resource requirements	Huge computing power for validators	Low computing power
Throughput	7 TPS	No upper bound
Confirmation delay	60m	Variable: depends on transaction throughput
Finality	6 confirmed blocks	Cumulative weight reaches predefined threshold
Drawback	High resource consumption; low transaction throughput	Large confirmation delay when the transaction traffic is low; centralization in case of coordinator involvement

Table 2.2: Comparison of PoW and DAG based consensus (source: [35])

2.6.3 IOTA STREAMS

The IOTA foundation developed two protocols to structure and retrieve data securely stored on the Tangle. STREAMS [32] is the most recently proposed protocol, and allows any entity to exchange encrypted, unalterable and authored data using the Tangle. STREAMS protocol supports many Transport modes like HTTP or TCP, and the Tangle is just one of the possible options.

Masked Authenticating Messaging (MAM) [33] is just one example of a STREAMS application. It's basically a publish-subscribe system that involves two parties: **author** and **subscriber**. Messages can be *signed* or *tagged*: signed messages can be produced only by the author and provides a signature to verify channel owner's identity, while tagged messages can be sent both

by the author and the subscribers (it's not possible to identify the sender of a tagged message). Moreover, a message can be *public* or *masked* for a set of specific subscribers that possess the correct key for decryption.

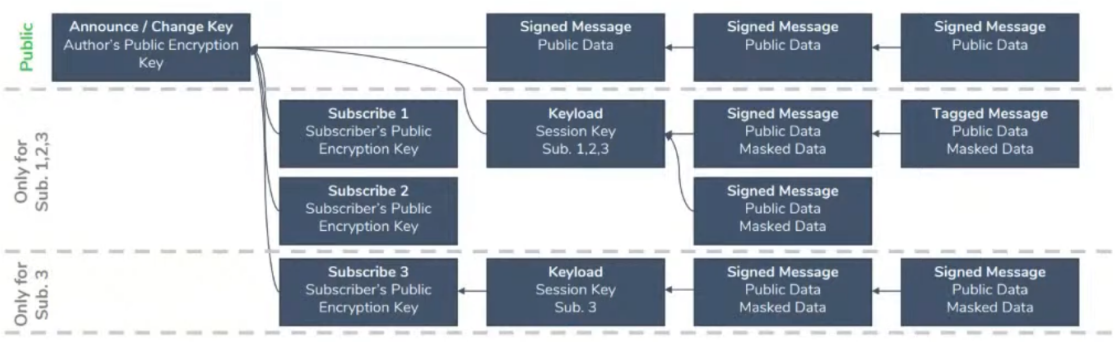

Figure 2.15: STREAMS messages example (source: [10])

STREAMS messaging system is based on "linking": a channel is generated by creating an *announce* message, whose reference is shared by the author to allow the subscribers to join the channel. When an entity wants to subscribe to that specific channel, it retrieves the author's public encryption key from the message whose reference was previously published and creates a new *subscribe* message containing its own new public encryption key. The subscriber links this new message to the original *announce* message created by the author, so that the latter knows which subscribers have joined the channel. The linking is also necessary to verify the authenticity of *signed* messages: each signed message is linked to the original *announce* message, so that subscribers can verify the correctness of the signature associated with the fresh new message. To encrypt channel messages, the **author** can also publish a *keyload*, a special message to exchange a session key with each subscriber or with a specific one: if the *keyload* message is linked to the original *announce* message, the session key is shared between all the channel subscriber; otherwise, if the linking is estabilished with a specific *subscribe* message, the session key is exclusively shared with that specific subscriber.

This messaging system provides a good level of flexibility to implement different kind of applications, but it's based on RUST language. RUST is a powerful language not suitable for constrained IoT devices: [34] proposes an alternative layer-two cryptographic protocol called **L2sec**, developed from scratch in C language and designed to be lightweight enough to run on constrained IoT devices.

2.7 Web-of-Trust

Web-of-Trust (WoT) is a decentralized alternative to the classic centralized trust model of a PKI. In WoT, there are many independent webs of trust: any user can be part of multiple webs and can be a link between two different webs. This concept is used in many OpenPGP-compatible [59] systems.

OpenPGP certificates can be digitally signed by other users that endorse the binding between that public key and the identity listed in the certificate. This is commonly done at key signing parties, meetings where many PGP users present their public keys to other users who can digitally sign the certificate containing that public key associated with the user's identity. This approach mitigates the main issues of PKI model: there is no single point of failure, because there are no central authorities (like root CAs) to be trusted. WoT model is also inspired by the theory of six degrees of separation (all people are six or fewer social connections away from each other).

The strong set is the largest set of strongly connected PGP keys. Two keys in the strong set always have a path that links them. In case of isolated and disconnected groups of key, if a single member of these groups exchange signatures with the strong set, the entire isolated group becomes part of the strong set. The most common metric to evaluate the level of trust of a given

PGP key within the strongly connected set of PGP keys that form the web of trust is the mean shortest distance (MSD)

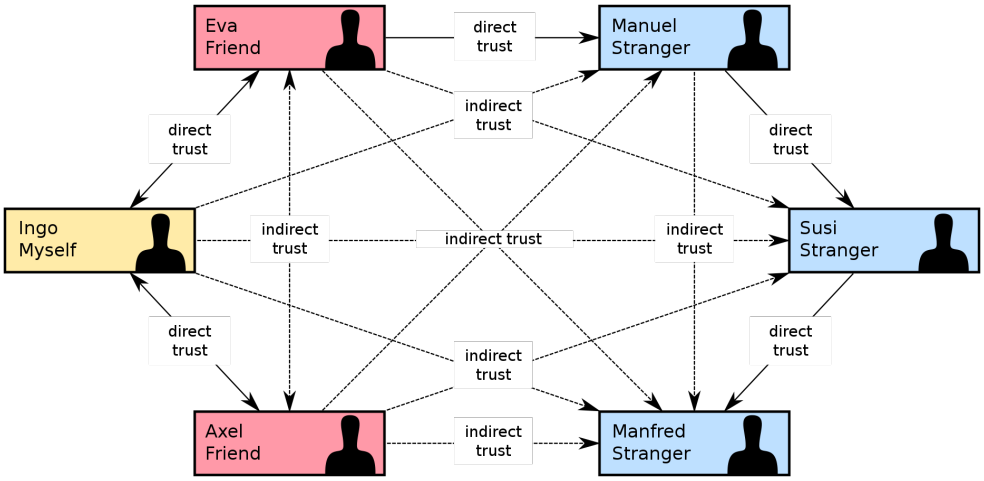

Figure 2.16: Web-of-Trust example schema (source: image)

2.7.1 WoT related problems

One of the main problems with Web-of-Trust is the public key authenticity check process. Basically, WoT can't be considered a PKI because there is no native mechanism to retrieve public keys. Currently, public keys are stored in some centralized keyservers (e.g. `pgp.mit.edu`), but this is in contradiction with the decentralized nature WoT was proposed for. Anyway, WoT also have a great adoption barrier: one of the major constraint to use PGP is to physically meet with someone (e.g. key signing party) to verify their identity and ownership of a public key and email address. For instance, a software user may need to verify lots of software produced by many developers all around the world and, in general, it would be impossible to physically meet every developers to trust their identities. It can happen in practice, that a new user's identity can't be endorsed by anyone in the peer-to-peer network. This issue can be common for user in remote areas, where there's scarcity of PGP users. In any case, a new user is not practically able to readily find someone to endorse a new certificate. This approach starts to work fine when a certain amount of endorsing signatures have been collected by a certain user.

Another important issue that affects WoT is the compromise of private keys. Initially, PGP certificates did not even include an expiration date; later, the expiration date was included to mitigate the problem, but the revocation of a compromised key is still difficult to deal with: the currently used solution is to use designated revokers, third trusted entities that have the permission to revoke the owner's key. The drawback is that revokers' misbehaviour could lead to potential attacks.

Research was active trying to integrate WoT with DLTs in order to solve these common issues. For instance, [36] proposes an Ethereum-based PKI where the smart contract logic tries to enhance the WoT mechanism. As anticipated before, WoT can't be strictly considered a PKI because there is no native mechanism to retrieve public keys: the solution proposed by [36] contains two core components: the smart contract - that defines the logic for the management of identities and attributes - and the client - which interacts with the smart contract and provide an interface to allow users to search for published attributes. The smart contract logic is based on the "entity", that publishes attributes, signatures and revocations for its identitity on the Ethereum Blockchain. Entity is representend by an Ethereum address and each attribute has an identifier, so that it's easily found by another user who wants to sign that specific attribute of another entity (because he can ensure the binding between the identity and the published attribute). Every signature that trusts a specific attribute is linked to the entity that released the signature, and everything is transparently and securely stored on the public ledger (instead of centralized PGP key servers).

The smart contract also provide a function to revoke a published signature and keeps track of all the revocations in a dedicated array.

Despite of the benefits brought by this integration, WoT approaches still suffer from a great adoption barrier that prevent its large adoption in replacement of standard PKI.

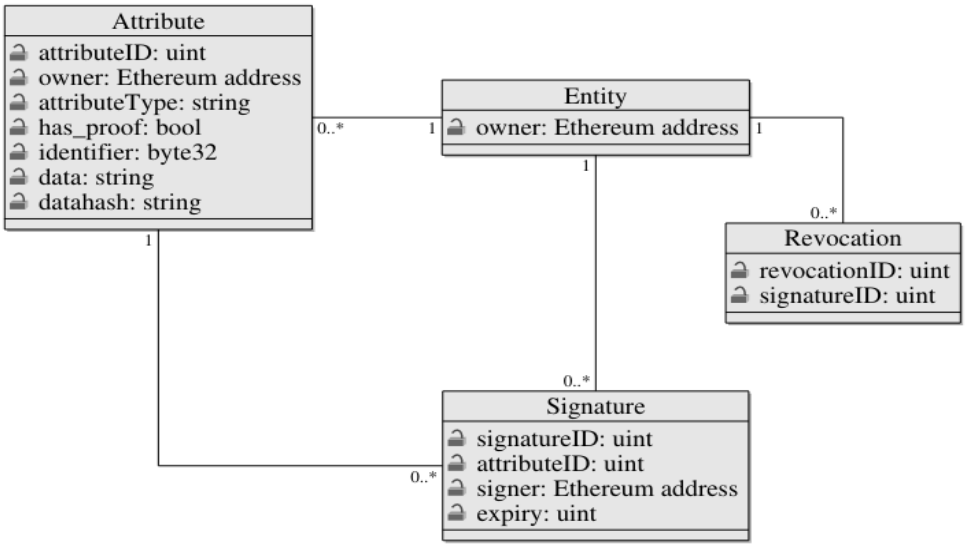

Figure 2.17: SCPKI [36] design schema

Chapter 3

Proposed implementation

3.1 High level design

The purpose to achieve with the work proposed by this thesis project is to designate and implement one possible solution among the many proposals that already exist in literature and that have been partially discussed in the second section of this document, also providing more possible enhancements where drawbacks and problematics arise. For what regards the design solution to elaborate on, the choice was to work on the proposal published by Elisa Bertino, Ankush Singla, Jongho Won and Greg Bollella [37], that has been already partially discussed in the previous section. There are many reason behind this choice and like any other solution, the brought benefits live together with the possible drawbacks, when compared to other existing solutions.

In this case, the proposed solution is based on Emercoin NVS (Name-Value System) Blockchain, a platform that benefits from Blockchain security properties to provide the possibility for storing on the ledger arbitrary values associated with unique keys, ensuring the integrity and security of this data. The idea is to use this platform to register the certificates of a group of constrained devices (like IoT devices) within the same network. To achieve this result, the proposed design requires the presence of non-constrained devices (characterized by a sufficient computational power and great storage capabilities) that can host an Emercoin node and maintain a full copy of the public ledger. The reason is that constrained devices are not compatible with the requirements that this task imposes. These powerful nodes are called **B-nodes** and they will be the interface between the IoT devices and the Blockchain platform. Logically, three actors can be defined: the client (in this case it's represented by the single IoT device), the **Device Manager (DM)** (the node that starts the communication with the client during all the protocols for certificate registration/modification/revocation and retrieve the necessary Blockchain data from a B-node) and the B-node. The DM and the B-node can also be merged in a single entity: in this implementation, the DM will host an Emercoin NVS node, so it will maintain a copy of the distributed ledger. Practically, for the implementation, two communicating portions of code have been developed, respectively for the client and the DM.

The most relevant concern of this solution is the same of every other Blockchain-based solution: the mechanism that allows to reach the consensus about the validity of a new block that is appended to the ledger is very inefficient in terms of computational resources and time. In order for a block to be appended to the chain, approximately 10 minutes are require, but this time is not sufficient for a block to be considered completely secure. Because of the possibility of 51% attacks, it's usually suggested to wait until the block depth is equal to 6 blocks: if we take into account this constraint, a group of transaction is validated after $\approx$ 60 minutes. Starting from this considerations it's possible to calculate the transaction throughput, which is on average 5-6 TPS (Transactions Per-Second). In many scenarios, a similar rate is insufficient for taking into account the possibility to choose the Blockchain as a valuable option for ensuring security and data integrity. Moreover, if we consider PoW consensus-based systems, the majority of the nodes within the peer-to-peer network do not contribute to the consensus mechanism because they can't afford the necessary computational effort due to their limited hardware components. The

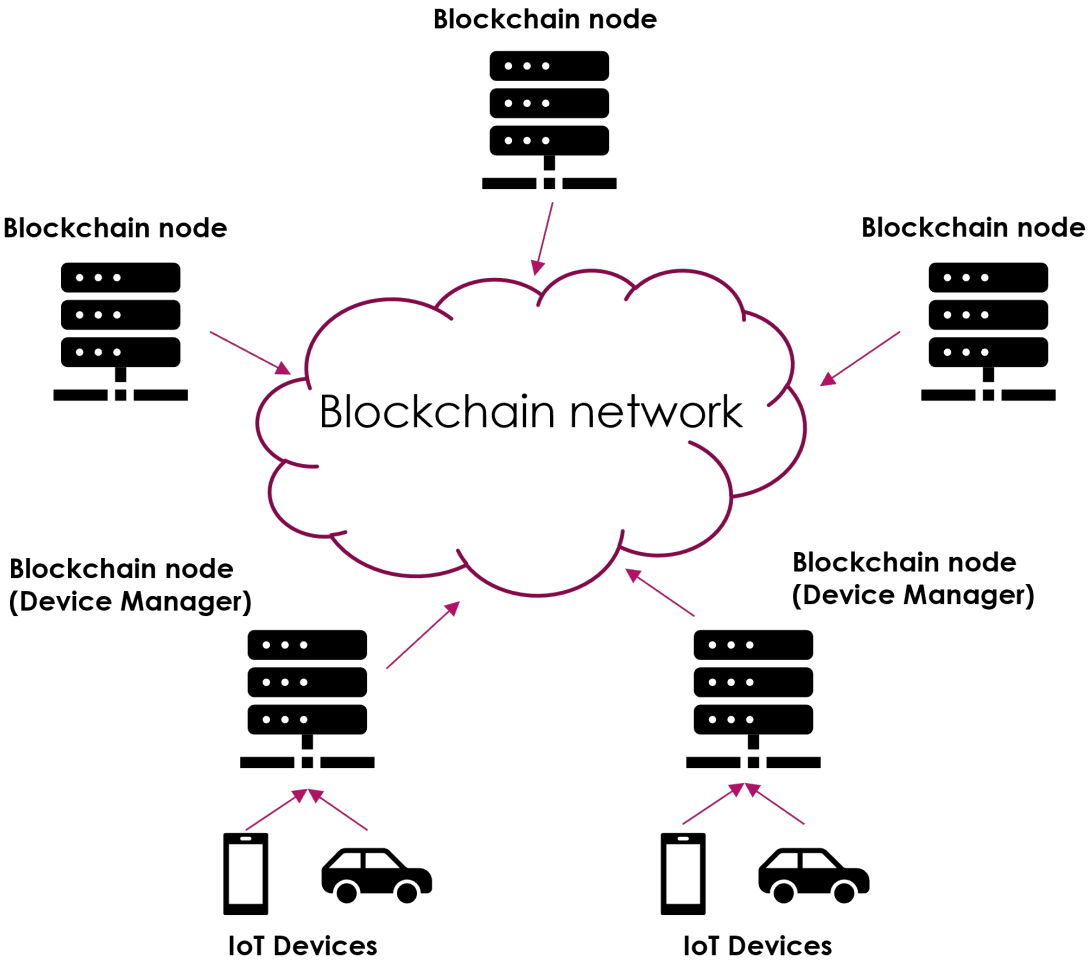

Figure 3.1: Emercoin-based solution high level design

consequence of this, is an optimistic reliance on miners, that must be economically incentivized to keep executing these tasks (and this is why fees are necessary for the creation of a new transaction in the Blockchain).

Other platforms like IOTA, provide a much higher throughput: considering IOTA, the technology proposed by this project (the Tangle) also eliminates the necessity of a cost related to the creation of new transactions. Despite of this, the reason why Emercoin-based solution was chosen resides in the type of application we are trying to implement: the core purpose is to provide a simplified access to the management of the certificates of a group (potentially very large) of devices that have difficulties in performing this task. In this scenario, an efficient mechanism that provides higher throughput is not necessary, also because the number of transactions generated by a PKI would be generally very limited when compared to other applications. Furthermore, if we compare the time required for Blockchain-based operations against the actual necessary time for the set of operations related to a "standard" PKI, we would get a good improvement anyway. For what regards the cost related to the creation of new transactions, it's important to notice that Emercoin fees are very low, especially if compared with the fees of other popular platform like Ethereum. Differently from Ethereum, the application range of Emercoin NVS is way lower: Emercoin can be useful in a restricted pool of use-cases, so the occurence of many issues also related to the financial speculation that affects Ethereum (or Bitcoin) is very unlikely for this platform. If we take into account these two compromises just analyzed, the benefit brought by the employment of Emercoin when compared to other platforms is its software layer built upon the normal Blockchain structure, that allows to register unique keys associated with arbitrary values. In Emercoin is not possible to register a key that already exists in the public ledger (if it's not been revoked): thanks to this feature, during certificate verification phase, we are sure to

retrieve from the Emercoin ledger the only existent key-value pair associated to the identity we want to authenticate.

Moreover, the PoW consensus mechanism is mature and solid: as previously discussed in the previous chapter, IOTA uses a consensus mechanism based on Markov Chain Monte Carlo algorithm and, in order for this system to properly work, it's necessary to provide a minimum transactions arrival rate; due to the fact that is difficult to constantly satisfy a similar condition, IOTA is supported by the Coordinator system, which inevitably includes in the consensus mechanism a third party (that clashes with the original decentralized nature of the technology). On the contrary, by using a PoW consensus mechanism, despite of the compromises that's been already discussed and analyzed, the decentralization is total (it's also important to notify that IOTA's Coordinator is a temporary solution, and the IOTA foundation is planning a 2.0 phase of the project, called "Coordicide", that manages to replace the Coordinator with a completely decentralized solution).

The core section of the proposed design is about the registration of a constrained device's X.509 certificate on Emercoin NVS: this operation is executed during the initialization process of a device that has just been added to the network. Compared to the original design proposed by [37], the certificate registration process of this implementation has been modified and improved in order to fix some important security issues:

- The first step executed by the DM is to send the initialization message M_{init} to the device that is waiting for a configuration. This message is digitally signed using a private key generated "on the fly" and the corresponding public key for signature verification is included within the message structure: potentially, with this approach, any entity within the network would be able to generate a keypair and to sign a valid M_{init} message that would be accepted by the other devices without any trouble. A possible solution it to provide the DM with an X.509 certificate released by a valid CA and to maintain it in time. Even if this option could be interpreted as a contradictory solution if compared to the original purpose of this work (to promote a decentralized approach), it's important to notice that this would be considered only for the DM node, which is not affected by those issues and constraints that affect IoT devices (the DM node can normally deal with "standard" PKI processes). In this case, the IoT devices must be pre-configured in order to recognize the valid DM's certificate in order to prevent any potential malicious device configuration coming from fake DMs that try to register on Emercoin NVS a different certificate hash value associated with the real device's identity (the malicious agent would be able to successfully impersonate the device);
- Another crucial modification is related to the step where (during the registration protocol) the device generates its own asymmetric key pair and the corresponding self-signed certificate: when the DM receives the message containing the identity provided by the device together with the corresponding public key, it can't verify the binding between the device and the received data. Potentially, any device could claim an arbitrary identity while the DM won't be able to detect a "lying entity". This issue has a great impact on the overall security and robustness of the proposed system: for this reason, this implementation also proposes a TPM-based solution for verifying that a device is strictly bound to a specific key/identity. In this chapter, a quick background on TPM technology will be provided together with further details about the Trusted Computing Group specification we are interested in. It's important to notice that basically, this TPM-based approach can be very useful in many other scenarios involving Digital Identity Systems [52] or e.g "Authorize-then-Authenticate" mechanisms [51].

3.1.1 Trusted Platform Module (TPM)

The TPM is a cryptographic chip that can securely store passwords, keys and certificates. This chip can be installed on high-end devices like desktop PCs, but also on smartphone and constrained devices. When a cryptographic key is stored on the TPM, the operative system won't be able to retrieve it, neither with special authorization. To use a key, the operative system can only ask to the TPM to perform cryptographic operations using that specific key. This chip enables

the possibility to develop a variety of applications that make it harder to access information on computing devices without authorization. For instance, TPM is mainly used for software attestation: when the software running on a device is altered, the chip can deny the access to a specific set of operations or storage areas, depending on a pre-defined policy. But this technology can also be used for the identification of the devices, using the pre-installed Endorsement Key (EK) that will be discussed later.

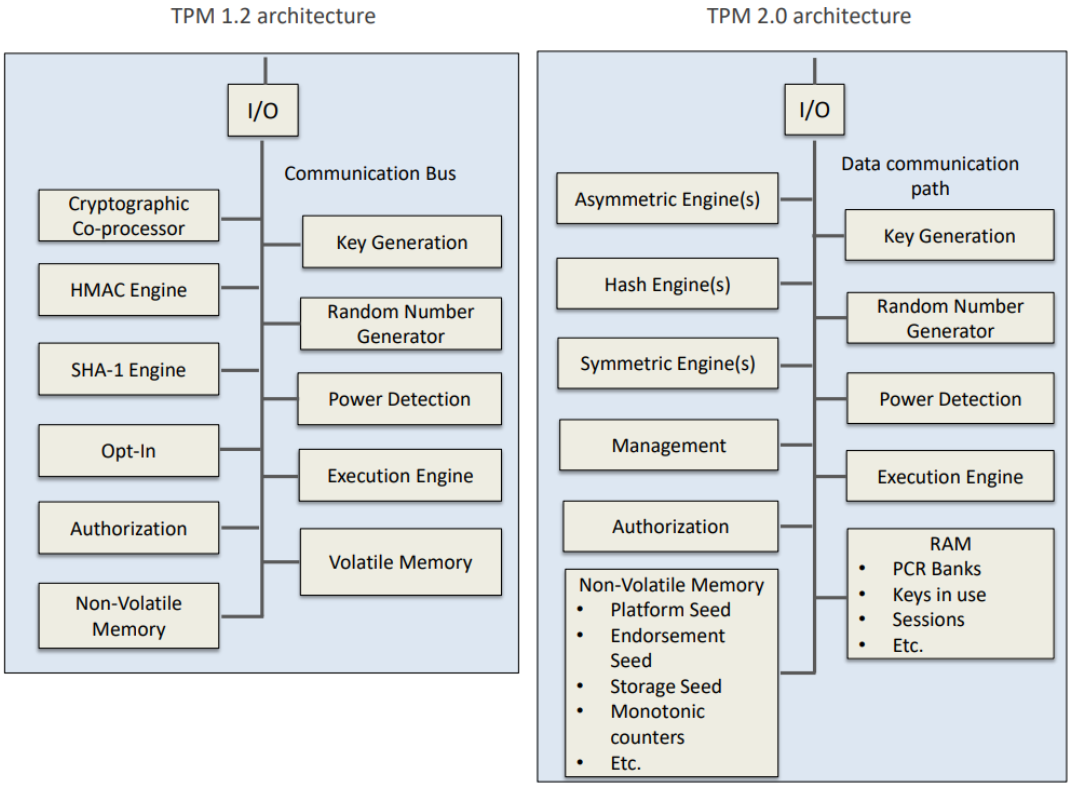

Figure 3.2: TPM 1.2 vs. TPM 2.0 architectures (source: [68])

The figure summarize TPM 2.0 architecture and compares it with the older TPM 1.2 architecture. TPM 2.0 architecture is composed by the following modules:

- **I/O Buffer**: a memory area where the data that transit from the host system to the TPM and vice-versa is stored. The input buffer contains commands-related data that will be received by the TPM, while the output buffer contains the command output data that will be sent back to the host system;

- **Cryptography Subsystem**: it is responsible for the execution of TPM's cryptographic functions. Many modules compose the Cryptography subsystem (Hash Engine, Symmetric Engine, Asymmetric Engine, Key Generation, RNG). The TCG (Trusted Computing Group, further details will be provided later) recommends specific algorithms providing a classification of three different categories: *S* (Standard), *A* (Assigned), *L* (Legacy). RSA and ECC are the only standard asymmetric algorithms: for RSA the standard signing schemes are RSASSA and RSAPSS while for ECC the set of standard signing scheme is composed by ECDSA, ECDAA and ECSchnorr.

  Differently from TPM 1.2, TPM 2.0 allows to perform symmetric encryption operation for the encrypting/decrypting command parameters and externally stored data. The standard symmetric algorithm is AES.

  Key Generation module can be used to generate two types of keys: *Primary keys* and *Ordinary keys*. Primary keys are generated starting from special seeds (*Primary Seeds*) computed by the RNG and permanently stored in the chip for next key generations. The

Attestation Key, necessary for the device authentication procedure defined by the TCG must be generated using a Primary Seed, so it will be a Primary Key;

- **Authorization Subsystem**: every time that a TPM command is executed, the Authorization Subsystem is invoked if the command wants to access shielded locations. When invoked, it performs two different operations (before and after command execution):
  - before command execution it checks that every authorization is valid for the involved TPM object;
  - after command execution it generates an acknowledge session value for the response
- **RAM**: it contains transient TPM data;
- **NVM (Non-Volatile Memory)**: it stores TPM persistent data. NV memory areas can be accessed using an handle called *NV index*;

The evolution from TPM 1.2 to TPM 2.0 brought many benefits to this technology. For instance, TPM 2.0 provides a greater flexibility for managing authorization roles; originally, TPM 1.2 was designed to provide two different types of authorization: *Owner* authorization and *Storage Root Key* authorization. Owner authorization was used in many different cases potentially different one from the other, and with TPM 2.0 this issue was solved by creating different hierarchies where every resident object is characterized by a specific set of policies and authorizations: *Platform Hierarchy*, *Storage Hierarchy*, *Null Hierarchy* and *Endorsement Hierarchy*, used by the privacy administrator in order to control the access to the Endorsement Key, that basically represents the identity of the TPM and that will be used for authenticating the device using the TCG specification procedure for Attestation Key generation;

The development of TPM-based applications is regulated by the Trusted Computing Group (TCG), an international standards body in charge of defining specifications, protocols and APIs related to the development of TPM-based interoperable applications. To achieve the device identity attestation using TPM, the TCG has published a detailed specification [60] that will be followed for the purpose of this work. The Endorsement Key previously mentioned, is an encryption key permanently stored on the TPM by the manufacturer. Together with this key, the TPM also includes an Endorsement Key Certificate, signed by the manufacturer's CA and whose validity is easily verifiable by the chip owner. Because EK has a long lifetime and is persistent, is a good candidate to be used as a device identity. Nevertheless, it's important to notice that an Endorsement Key identifies a TPM and not a device. The purpose of the specification, in fact, is to provide a detailed procedure that finally produces a *DevID* (Device Identity) certificate.

3.1.2 Device identification using TPM 2.0

The scenario described by [60] involves a Certification Authority in charge of verifying the binding between the device identity and the generated Attestation Key. In our study case, the standard CA-based PKI is replaced with a decentralized application based on Emercoin NVS, so the CA is replaced by the Device Manager, which has to verify that all the messages exchanged with a device during the registration phase really come from the device we are intended to configure. Basically, the device must prove that the generated Attestation Key resides in the same TPM of the Endorsement Key associated with the EK certificate stored by the DM. The procedure is described in section 6.1.2 of [60] and is composed as follows:

1. The device creates the Attestation Key (AK) in the Endorsement Hierarchy following the rules described in section 3.6 and 7.2 of the specification;
2. The device builds the TCG-CSR-IDEVID structure (further details will be discussed later), that contains:
   - Platform Identity Information: device model and serial number;
   - The TPM EK Certificate previously mentioned;

- The AK Public Area

3. Using the TPM, the TCG-CSR-IDEVID hash is generated and signed with the AK, to address potential MITM attacks during the transmission to the Device Manager;
4. The TCG-CSR-IDEVID is sent to the Device Manager together with the signature;
5. The DM verifies the received data:
   - Extracts the AK public key from the Public Area and uses it to verify the signature on the TCG-CSR-IDEVID structure
   - Extracts the EK certificate and verifies it using the indicated TPM manufacturer's certificate chain. Before this verification, the DM checks if the received EK certificate is equal to the certificate associated with the device model and serial number stored locally;
6. The DM issues a challenge to the device to get a Proof-of-Possession of the Endorsement Key certified by the EK certificate and to ensure that the AK is stored in the same TPM where the EK resides. To build this challenge, the DM uses the following procedure:
   - Calculate the cryptographic name of the AK ($ID_H||H(PubArea_{AK})$, where H is the hash function and ID_H is the hash algorithm ID);
   - Using the `TPM2_MakeCredential` command, create the encrypted credential blob that will be received by the device. `TPM2_MakeCredential` receives as input a nonce that must be retained by the DM for use in later steps;
   - The DM sends the generated blob to the device;
7. Using the `TPM2_ActivateCredential` command, the device decrypts the credential blob and gets the *CertInfo* data. Moreover, by using this command, the TPM verifies the AK's name using the EK. The returned *CertInfo* is basically the Proof-of-Possession (the nonce provided in step 6);
8. The device sends *CertInfo* to the CA, that checks if the received value is equal to the nonce previously retained.
9. In case of success, the AK resides in the TPM described by the EK certificate: the device is not lying about its identity and the certificate registration procedure can proceed.

The device is now able to securely authenticate the following messages of the certificate registration procedure using the Attestation Key. In the original procedure described in [37], the device generates its own identity, the private/public key pair and the self-signed X.509 certificate and signs the certificate hash (that will be sent to the DM) with the newly generated asymmetric key. As explained before, this procedure doesn't ensure a binding between the claimed identity and the real device. Now, by signing the certificate hash with the Attestation Key previously created, the DM knows that the received data come from a specific device of the network uniquely identified thanks to its TPM and the procedure described before.

3.1.3 Device certificate registration

Initially, each IoT device is waiting the DM for the configuration: in this state, the device is not active in the network. The first operation that must be performed is the registration of a device certificate using Emercoin. The DM locally stores a list containing the information about each device within the network, including IP address, model name, serial number and the EK certificate that uniquely identify a device's TPM. Potential malicious actors trying to configure malicious devices won't be able to perform the attack if the device's data is not present in this list. As discussed before, during the authentication using the TPM-based procedure, the TCG-CSR-IDEVID structure contains the device's model name, the serial number and the TPM's EK certificate: this information must match the corresponding entry in the list for the procedure to end successfully.

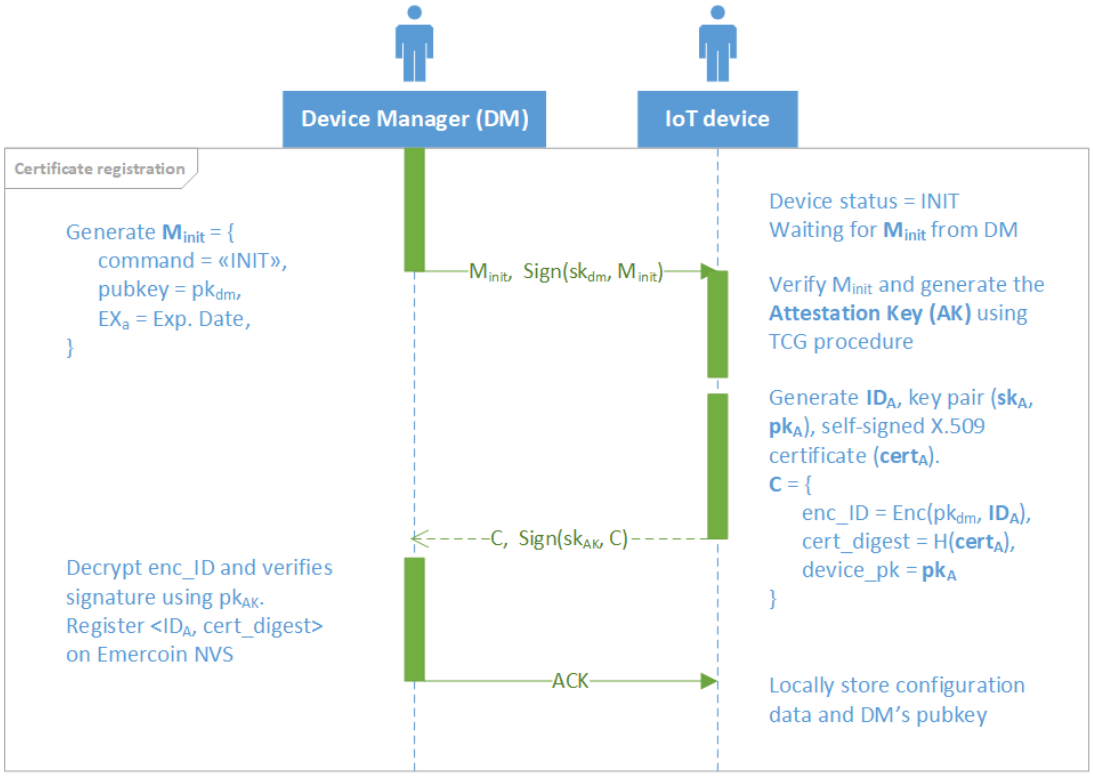

Figure 3.3: Device setup implemented schema

The device setup protocol described in [37] has been modified to integrate the TPM-based device authentication:

1. DM generates a private/public key pair sk_{DM} and pk_{DM}
2. DM sends a message $M_{init} = (C_{init}, pk_{DM}, EX_A, \sigma)$ where C_{init} is an initialization command, EX_a is the expiration date of the future certificate and σ is the signature of the concatenation of all the previous elements, signed with sk_{DM};
3. The device (D_A) is waiting for a configuration: it receives M_{init} from DM and, after signature verification, it generates the AK following the procedure described before;
4. If the AK creation procedure ends successfully, the device generates an identity ID_a, a private/public key pair sk_A/pk_A and a self-signed X.509 certificate $Cert_A$. For ID_a it's recommended to use a Universally Unique Identifier (UUID), to avoid eventual collisions with other names in the ledger;
5. D_a sends $C = Enc(pk_{DM}, Cert_A)$ to DM. The encryption of $Cert_A$ is recommended to avoid that a Man-in-the-Middle catches the unique identity and performs the name registration (with another certificate digest) before the DM. The message is finally signed using the AK previously created;
6. DM receives C, decrypts it using sk_{DM} and verifies the signature. Then, it computes $V = H(Cert_A)$ where H is the hash function and finally generates a transaction to register the $< ID_A, V >$ pair on Emercoin NVS;
7. When the transaction is successfully confirmed by the consensus procedure, DM sends a signed acknowledgement to D_A;
8. D_A receives the acknowledgment and verifies the signature using pk_{DM}. In case of success, D_A update its configuration by storing pk_{DM} and EX_A on the local storage. Now D_A is in running state and can't accept other initialization command from the DM;

3.1.4 Device ownership transfer

This procedure is applied every time that a device D_a is transferred to a different network under the control of another administrator. If this procedure is not executed, the future device administrator won't be able to submit new Emercoin transactions for that specific device name-value pair, and this privilege would be reserved to the old administrator only. In Emercoin, when a transaction related to a name-value pair is inflated with different addresses for the receiver and the sender, the management privileges of that specific pair is transferred from the sender to the receiver.

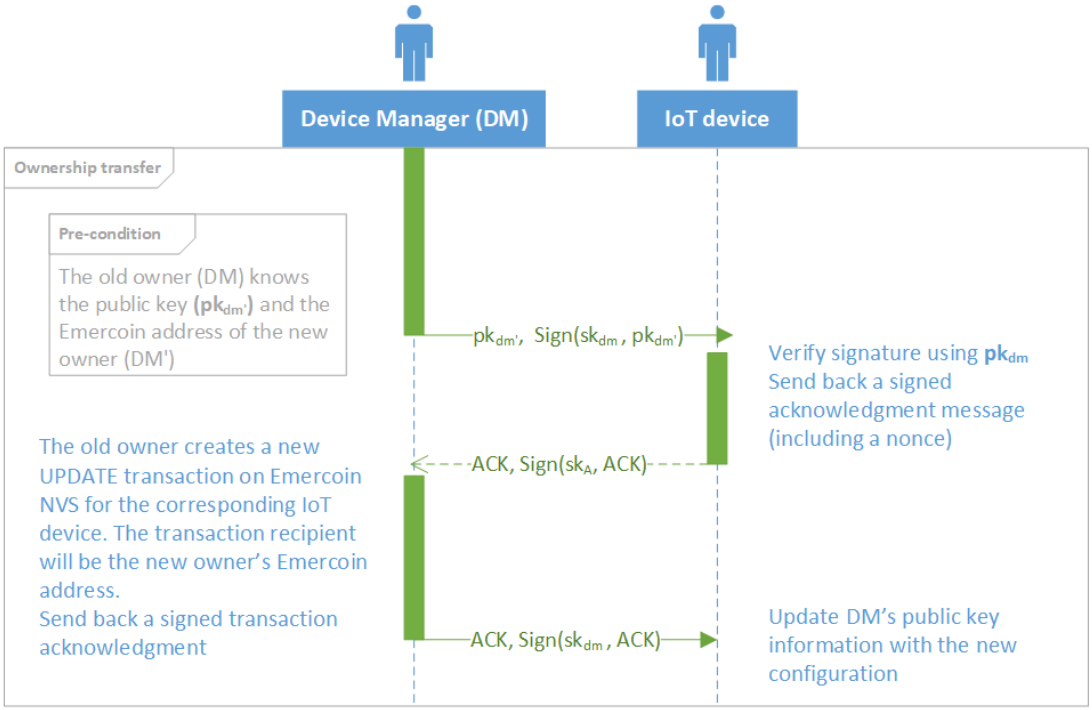

Figure 3.4: Device ownership transfer implemented schema

The procedure is composed by the following steps:

1. The new owner generates an asymmetric key pair sk_{DM2}/pk_{DM2};
2. The old owner creates a new UPDATE transaction in Emercoin, inflated with the corresponding name of the device involved in the transfer and providing the new owner Emercoin address as receiver address;
3. The old owner sends pk_{DM2} and its signature using sk_{DM}: this is necessary to prove the right to transfer the ownership of that device to another entity. The device verifies the signature using the master public key of its configuration and, in case of success, switches its master public key with the new one;

3.1.5 Device key update

When a certificate is about to expire or contains wrong information that must be updated, the application provides a way to update the name-value pair stored on the ledger. Like in the registration procedure and differently from the original protocol, the message containing the data related to the updated identity information must be signed with the AK that's been already validated.

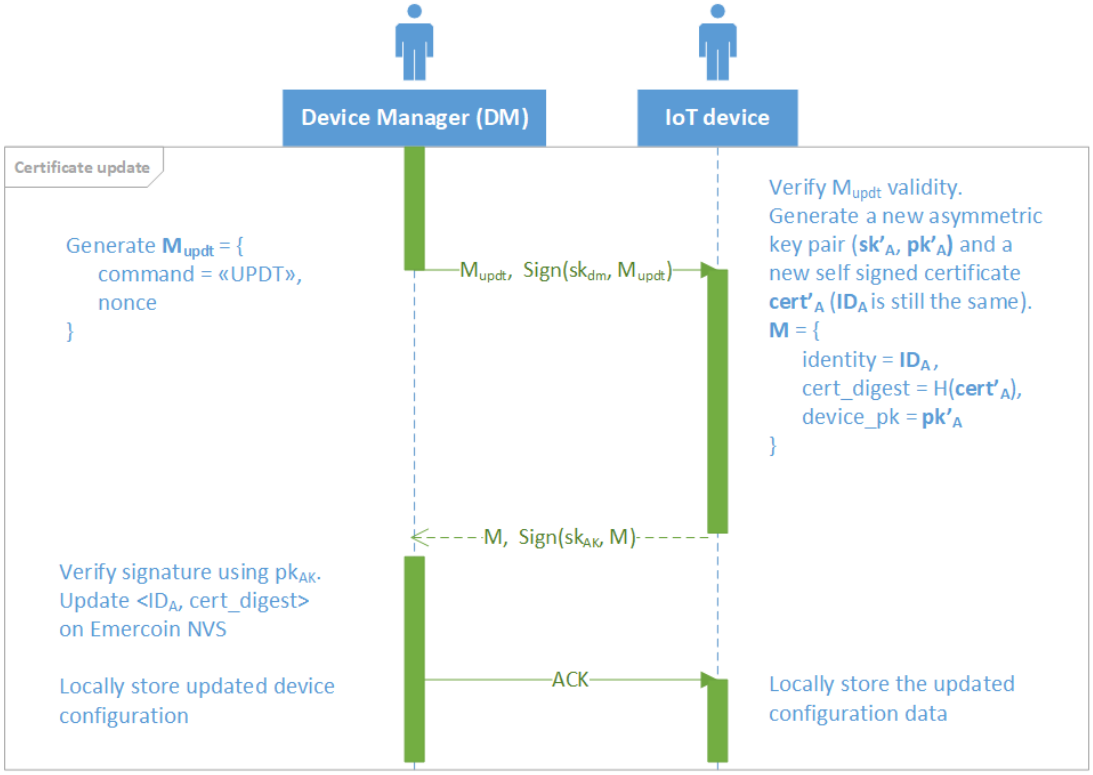

Figure 3.5: Device certificate update implemented schema

The update operation procedure is very similar to the registration one, and consists of the following steps:

1. DM sends $M_{update} = C_{update}||\eta$ together with the signature, where C_{update} is the command identifier and η is a nonce necessary to avoid replay attacks;

2. D_A checks if the signature is correct and if the nonce is greater than the previously stored one. If the verification succeeds, D_A generates a new asymmetric key pair sk'_A/pk'_A and a new self-signed X.509 certificate $Cert'_A$ (the identity is still the same ID_A already registered on Emercoin NVS);

3. D_A sends $Cert'_A$ plus a signature obtained using the AK (the encryption is not necessary in this case because no new identities are generated);

4. DM receives the message and the signature: if the signature verification succeeds, the DM creates a new UPDATE transaction for the same old name associated with the device, where the value is replaced with the hash of the newly generated certificate;

3.1.6 Device key revocation

When a device is compromised, it's important to provide the possibility to revoke its corresponding certificate stored on the ledger. In this case, the device owner itself doesn't have to ask for revocation to a Certification Authority, but it can simply creates a new DELETE transaction, providing as input the name that needs to be revoked. Whenever an entity tries to retrieve information about a specific name registered in Emercoin NVS, the returned data is always associated with the last transaction where that name is involved. If the last transaction related to a name is a revoke transaction, that record must be considered expired and not yet valid.

Figure 3.6: Name revocation in Emercoin NVS: the last transaction represent the current state of the name (revoked)

3.1.7 Authenticated Key Exchange

The final result to achieve with this work is to replace the centralized trust anchor represented by the Root CAs with a decentralized platform represented, in this case, by the Emercoin Blockchain. The standard TLS libraries, perform server/client authentication by using CA-based mechanism (certificate chain verification): with proper adjustments, our purpose is to estabilish TLS channel by using Emercoin NVS to ensure the validity of a certificate. When two devices want to securely communicate using a TLS channel, the pre-condition is that each device has been configured using the device setup procedure described in 3.1.3. Both the devices locally store their self-signed certificate containing the information about their identity and for each X.509 certificate there's a name-value pair on Emercoin NVS, where the value is the hash of the certificate and the name is the corresponding device identity.

During the TLS handshake, each device receives a certificate from the other party for the authentication: at this point, instead of providing a certificate chain that goes up until a Root CA, the certificate is self-signed and it's verified using Emercoin NVS: first the device identity is extracted from the received X.509 certificate, then the hash of the certificate is computed and compared with the value contained in the name-value pair stored on the ledger whose name is equal to the extracted identity. If the name-value pair exists, and the value is equal to the computed hash, the authentication process ends successfully and the TLS handshake can proceed. To retrieve information from the ledger, an IoT device must query a trusted B-node: to improve security and robustness, it would also be useful to query different trusted B-nodes, so that an eventual forged response can be detected.

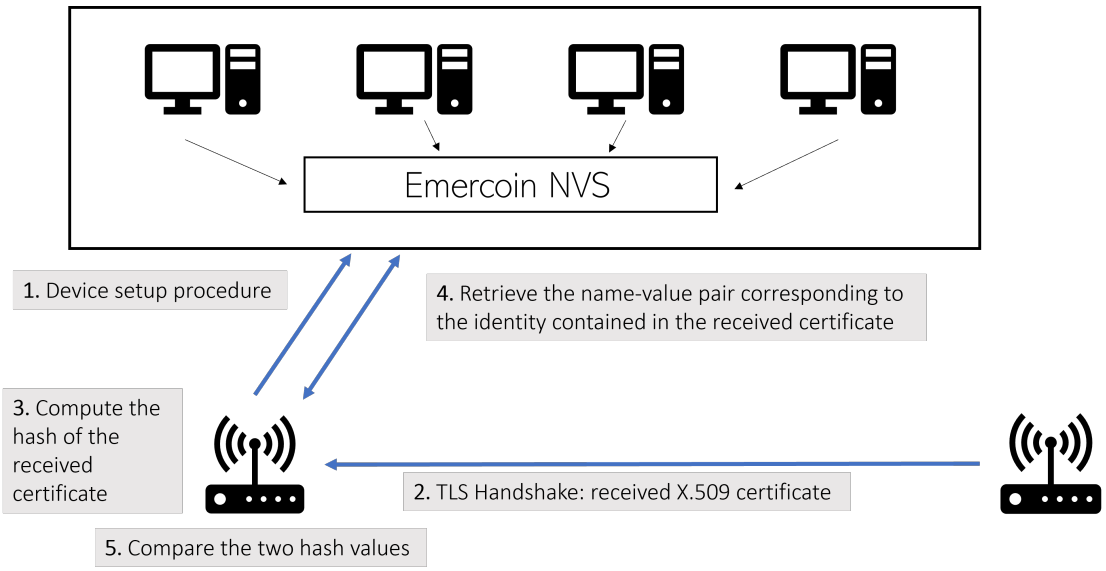

Figure 3.7: TLS authentication for authenticated key exchange

3.2 Test software

This section will provide further technical details about the implementation and the software used to build this test application.

3.2.1 Emercoin wallet

Starting from the B-node (Device Manager), the core element to run an Emercoin node is the **Emercoin Wallet**. A wallet is a software that allows users to run a Blockchain node and to manage their own account within that specific Blockchain network. When the wallet is launched for the first time, the entire blockchain will be downloaded on the storage: this process can require an important amount of time, depending on the Blockchain size, but it will be executed only the first time. The Emercoin wallet also provides a configuration file that allows a user to enable/disable specific options, like the possibility to use the *testnet* instead of the real blockchain network. The testnet is a secondary network used for experimental and test purposes, where the official cryptocurrency is replaced with a fake one. Some platforms (e.g. Ethereum) provides automatic mechanisms to get testnet coins (e.g. Ethereum testnet faucet) but, for Emercoin, the only way to get some testnet coins is to write an email to the official support, providing an Emercoin testnet address.

The home page provides a simple recap of the account, by displaying the current balance and the recent transactions. "Send" and "Receive" tabs are for simple value exchange transactions that do not involve any name-value pair. Basically, the NVS is built upon the normal Blockchain application, so that Emercoin can also be used to send/receive EMCs (the official Emercoin cryptocurrency). "Transaction" tab shows the list of all the transactions related to the active account.

"Manage Names" is the most interesting section of the wallet for our purpose. This tab provides a GUI for name-value pair creation, update and revocation: these three possible operation are identified with three specific command strings (`NAME_NEW, NAME_UPDATE, NAME_DELETE` and in case of a pair creation/update the interface allows to specify the expiration window, expressed in days but practically converted in number of mined blocks (the block mining rate is constant, so it's possible to convert the number of days in number of mined blocks starting from the block containing the involved transaction). If an address is specified, the selected name-value pair will be under the control of the entity associated with that address (this feature is used for the ownership transfer procedure described before).

The wallet can also be used without the GUI, by editing the configuration file to setup an RPC server. The configuration file can also be edited using a GUI window, that allows to specify the access credentials. To use the Emercoin RPC server, a list of API is available at [69]: this API is based on Bitcoin API with some newly added commands specific for the NVS operations. The most important functions that will be used in the implementation are:

- `name_new`: creates a new name-value pair that expires after the specified number of days. The input parameters are the following:
  - `name`: name to create;
  - `value`: value to write;
  - `days`: expiration time window (1 day $\approx$ 175 blocks);
  - `toaddress`: (optional) address of the recipient. If empty, the transaction is to yourself;
  - `valuetype`: (optional) interpretation of the string value. Can be "hex" or "base64";
- `name_delete`: delete a name (only if the user owns it). This is basically the revocation: when a delete transaction is confirmed, another entity can use `name_new` using the deleted name;
- `name_update`: it works exactly like `name_new`, but the user must own the name of which the corresponding value is going to be updated. By specifying a different `toaddress`, the ownership of that name-value pair is transferred to the recipient;

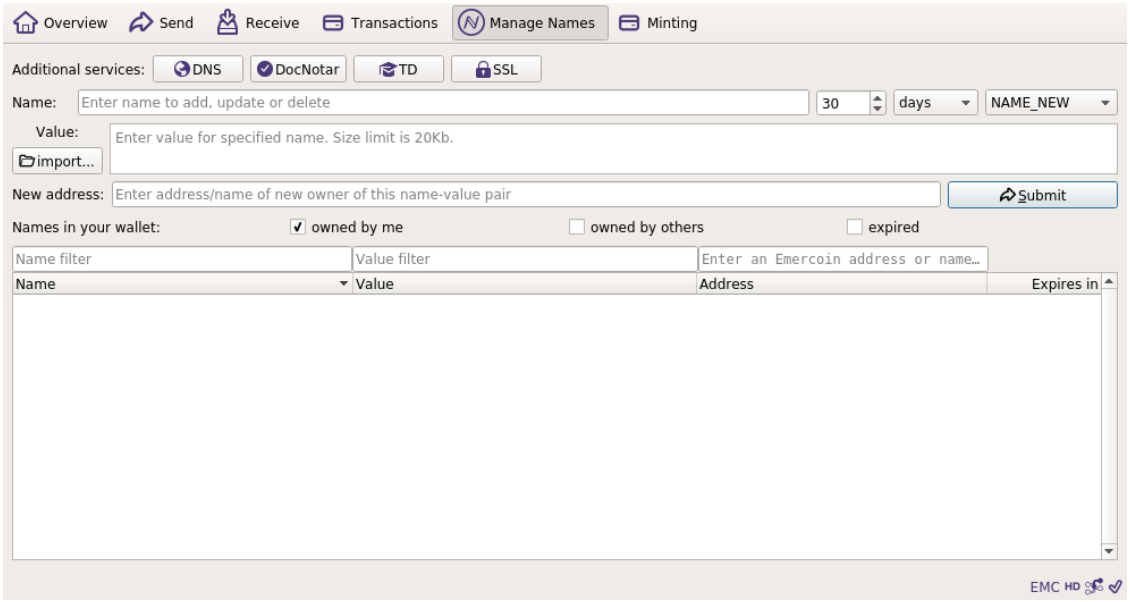

Figure 3.8: Emercoin wallet: names management tab

- `listtransactions`: returns up to `count` most recent transactions skipping the first `from` transactions (related to the specified account passed as input parameter). `listtransactions` will be used after the creation of a transaction to check if the block containing the transaction is at least 6 blocks deep in the chain (6 is the threshold to trust the validity of a block). This function receives the following input parameters:
  - `account`: (optional) DEPRECATED. This should be set to "*" to indicate the default active account;
  - `count`: (optional) the amount of transactions to return. Default is 10;
  - `skip`: (optional) the amount of transactions to skip. Default is 0;

3.2.2 TPM2 Software Stack (TSS2) library

TSS2 is an implementation of the Trusted Computing Group's TPM2 Software Stack (TSS). The stack is composed by the following layers (from top to the bottom):

- **Feature API (FAPI)**: this is the most high-level interface for TPM programming. It's designed for simple TPM-based applications and it's very easy to use. FAPI functions cover the 80% of the use cases where a TPM is involved. For more complex use cases, TSS2 provides more sofisticated APIs (ESAPI and SAPI). All these functions are grouped in a single library: `libtss2-fapi`. The documentation is available on [61]
- **Enhanced System API (ESAPI)**: this API maps all the available TPM2 commands documented in Part 3 of the TPM2 specification. It's a bit more simple to use than the SAPI and, in addition to the latter, it performs tracking of metadata for TPM object and automatic calculation of session based authorization. Finally, ESAPI functions are also available in their asynchronous version. ESAPI is documented in [62] and all its functions are grouped in the `libtss2-esys` library
- **System API (SAPI)**: ESAPI and FAPI are built upon the System API. Also for SAPI functions there's an asynchronous variant, useful for event-driven systems. The functions are entirely grouped in `libtss2-sys` library and a rich documentation is available on [63];
- **Marshaling/Unmarshaling (MU) API**: this API provides a set of functions that allow to serialize complex TSS structures into bytes (for transmission) and vice-versa. The functions are included in the `libtss2-mu` library and their documentation is available on [64];

- **TPM Command Transmission Interface TCTI**: this API is the most low-level interface for TPM interactions. There are different libraries for TCTI, depending on the used TPM platform (e.g. `libtss2-tcti-device` for hardware TPMs, `libtss2-tcti-tbs` for Windows, `libtss2-tcti-swtpm` for software TPM). The documentation is available on [65].

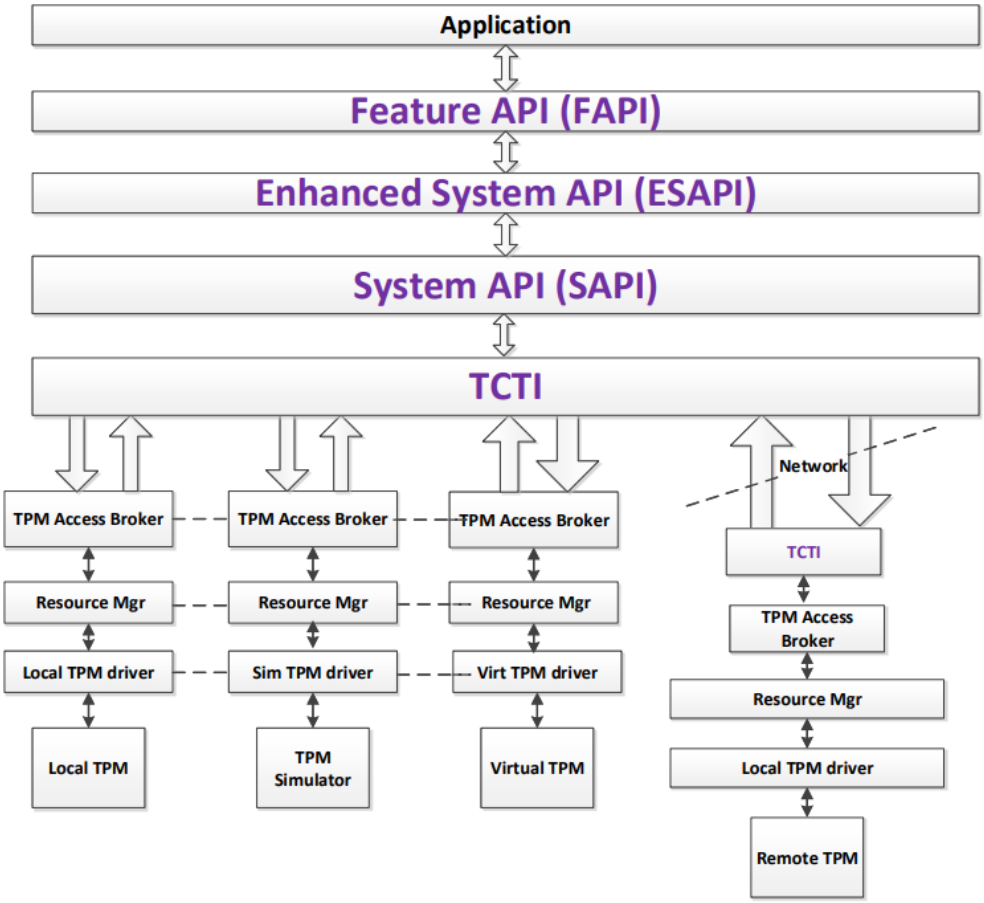

Figure 3.9: TPM2 Software Stack schema (source: [66])

For the implementation discussed in this section, the FAPI was not well suited because some important steps of the TPM-based authentication requires specific TPM commands that are not provided by the FAPI. For this reason, the ESAPI was used to implement the TCG specification for devices identity. ESAPI lies directly above the SAPI, and it's been designed to provide 100% of TPM's functionalities with some simplification when compared to the SAPI. ESAPI's main advantage is to provide cryptographic functionalities for that applications that requires to have an encrypted data stream from the library to the TPM itself (parameter encryption/decryption, secure HMAC sessions) and an enhanced session management functionality. ESAPI simplifies session starting and salting, HMAC calculation and the creation of encrypting/decrypting sessions. It's written using C99, to provide great compatibility through many potential operative systems and so that it can be easily bound to other languages (e.g. Python [77]). Compared to the System API, the ESAPI allows to execute different TPM commands using fewer function calls and to control each input parameters of the corresponding TPM commands.

The core element for the ESAPI to be used is the *context*. The `ESYS_CONTEXT` is a structure that contains all the necessary data that must be stored between each function call, so that no global state variables are necessary. It contains:

- Data structures and information that provides to the ESAPI a low-level communication channel with the TPM (e.g. TCTI context);

- Metadata associated to each `ESYS_TR` objects (handles that identify TPM Resources);
- State information;

The memory for the context is always allocated by the ESAPI (not by the programmer) and its lifecycle can be summarized as follows:

- An `ESYS_CONTEXT` is created using `Esys_Initialize()` function;
- Create (or deserialize bytes previously store on the local storage) metadata to retrieve sessions and resource information structures;
- Use sessions and resources obtained from the previous step to execute TPM commands;
- Serialize resources/sessions information on disk;
- Close the context using `Esys_Finalize()`;

Another important element stricly related to the context is the *TPM resource*. TPM resource metadata is referenced by an `ESYS_TR` handle associated with a specific `ESYS_CONTEXT`: an handle for a resource created using one `ESYS_CONTEXT` can only be used within that specific `ESYS_CONTEXT`. TPM resource metadata contains the following data:

- The TPM handle;
- `authValue`: the authentication value of the resource;
- The public area of the resource (`TPM2B_PUBLIC`);
- The **resource name**, equal to the digest of resource's public area prepended with the hash algorithm identifier;

TPM resource lifecycle is summarily described as follows:

- Create an `ESYS_TR` object (e.g. deserializing metadata from disk using `Esys_TR_Deserialize()`);
- Executed TPM commands using ESAPI function calls referencing the obtained resource;
- Serialize metadata to disk (using `Esys_TR_Serialize()`, the reverse function of `Esys_TR_Deserialize()`);
- Destroy the `ESYS_TR` handle by flushing the resource (e.g. using `Esys_FlushContext()`) or releasing only the metadata with `Esys_TR_Close()` (leaving the resource in the TPM);

The last important element related to the ESAPI use model is the *session*. Also the session is stored in an opaque structure associated with an `ESYS_CONTEXT`. This structure is referenced by an `ESYS_TR` handle and contains the following session information:

- The TPM handle;
- Session attributes that will be used in the next TPM command;
- Information related to the encrypted salt of the session and the storage of the session key;
- Session hash and symmetric algorithms information;
- Session nonces;
- Session policy information;

A session can be started using the `Esys_StartAuthSession()` function that generates the corresponding `ESYS_TR` handle. The latter is used to reference the session when a TPM command is executed through ESAPI function calls. When the application's task is finished, the session can be closed in two ways:

- using `Esys_FlushContext()` function;
- setting the `continueSession` bit to false using `Esys_TRSess_SetAttributes()` function;

These elements are essential for any TPM-based application: in order to implement the specification provided by the Trusted Computing Group for the authentication of a TPM-enabled device, the most important TPM commands that will be used are:

- `TPM2_CreatePrimary`: this command can be executed using `Esys_CreatePrimary()` function from ESAPI. It creates a Primary Object using the TPM's primary seed for generation. TCG specification explicitly requires to use `TPM2_CreatePrimary` to generate the AK. In this case, apart from the `ESYS_CONTEXT` and the session handle `ESYS_TR`, the most important input parameter is the `TPM2B_PUBLIC *inPublic`, a TSS structure that contains many fields related to the attributes of the key that we want to create. `inPublic->publicArea.type` represents the key type (e.g. RSA), `inPublic->publicArea.attributes` is an important bitstring that can be configured using a combination of pre-defined constants (using OR logical operator). This configuration must comply the specification's indicatons: for the AK, the following object attributes must be set:
  - **FixedTPM**: the key can't be duplicated;
  - **FixedParent**: the key can be copied only if the parent key can be duplicated (a FixedTPM key is also FixedParent);
  - **SensitiveDataOrigin**: the private key was generated by the TPM. This ensures that no other copies of that key exist;
  - **Restricted**: the key can only sign TPM-generated hashes (when a digest is calculated using the corresponding TPM command, the latter releases a specific ticket that can be provided to the `TPM2_Sign` command to prove that the data we are trying to sign was generated by the TPM;
  - **Signing**: the key can be used for signing;
  - **UserWithAuth**: if this attribute is set, the USER role authorizations can be provided using object's **authValue** (e.g. a password), without the necessity to satisfy a specific policy within a policy session;

  Another important field of `TPM2B_PUBLIC` structure is `inPublic->publicArea.authPolicy`: this field must be filled with a pre-defined sequence of bytes in order to retrieve the Endorsement Key with `TPM2_CreatePrimary` (the value is specified in [67, Table 1]). Generally, it contains the information related to a policy that defines the authorization rules and the constraints on the TPM object;
- `TPM2_HashSequenceStart`: hash computation must be performed using the TPM in order to sign a payload with a TPM restricted key. ESAPI provides also the possibility to compute the hash using a single function `Esys_Hash()` when the payload length is $<= 1024B$. If the payload is $> 1024B$, it has to be split in different chunks, and for each chunk the `Esys_SequenceUpdate()` function is called (after `Esys_HashSequenceStart()` function has been called). As explained before, when a digest is computed using these TPM commands, a special ticket is released to prove that the hash value has been calculated by the TPM and not externally. When this ticket is passed to the `Esys_Sign()` function, it's possible to use a *restricted* key for the signature;
- `TPM2_Sign`: this command allows to sign also an externally provided hash (except for *restricted* keys). To specify the signature properties, the `Esys_Sign()` function requires as input a `TPMT_SIG_SCHEME` structure, that contains two fields:
  - `scheme`: an identifier for the signature scheme that will be used for the signing process (e.g. `TPM2_ALG_RSAPSS`;
  - `details.<scheme>.hashAlg`: the hash algorithm that will be used for the computation of the signature;

- `TPM2_MakeCredential`: allows the TPM to create a `TPM2B_ID_OBJECT` containing an activation credential according to the methods described in section "Credential Protection" of [68]. The encrypted credential blob generated by this command is used by a TPM-enabled device to release the credential (using `TPM2_ActivateCredential`) in order to prove that a TPM object resides in the same TPM in which resides the specific Endorsement Key whose public part was used to encrypt the blob. The ESAPI function bound to `TPM2_MakeCredential` command is `Esys_MakeCredential()`. Apart from the handle of the key that is used to encrypt the credential blob, the function requires two important input parameters:
  - `credential`: a `TPM2B_DIGEST` structure that contains a buffer filled with credential data (credential data are application-dependent);
  - `objectName`: a `TPM2B_NAME` structure that wraps a buffer containing the *name* of the TPM key whose TPM residency has to be proved. The *name* is basically the hash of the public key prepended with the identifier of the hash algorithm used for the computation;
- `TPM2_ActivateCredential`: this command enables the association of a credential with an object in a way that ensures that the TPM has validated the parameters of the credentialed object. The corresponding ESAPI function's (`Esys_ActivateCredential()`) important input parameters are: the handle of the key that will be used to decrypt the credential blob, the handle of the credentialed key and the encrypted credential blob generated by `Esys_MakeCredential()`.

3.2.3 Protocol implementation

The software run by the DM and the one run by the constrained device have been developed from scratch. The DM runs a Python script organized in different files, one for each significant operation (device initialization, ownership transfer etc...). Python was chosen because is an extremely powerful language, that allows to produce complex code in very short time thanks to a great variety of libraries that provides lots of complex functionalities with minimum verbosity in the code. This choice was possible also because the DM does not have any restriction on its computational resources: in fact, Python and its libraries are not well suited for constrained resource devices.

```
Welcome to your personal Device Manager! Choose an option:
 1. Initialize new IoT device
 2. Device ownership transfer
 3. Device key update
 4. Device key revocation
 5. Exit
>>
```

Figure 3.10: Device Manager Python application: first look

`pyca/cryptography` library [71] was used to perform all the necessary cryptographic operations on DM side, except for those specific steps related to the Attestation Key creation procedure, that require to use some functions of the `tss2` library [70]. TSS2 has been developed using the C language, but also a Python binding is available (`tpm2-pytss`) [77]. Cryptographic operations are grouped in a single Python file (`device-manager/crypto.py`) that basically provides each necessary cryptographic functionality hiding extra low-level operations on the input data to make them suitable for `pyca/cryptography` library functions. On the device side, because of the hypothetical restrictions related to the computational resources, `mbedTLS` [72] library was adopted and the same approach used for the DM Python application has been replicated, so that every necessary high-level cryptographic task was wrapped into a function that hides low-level programming details. All these wrappers have been grouped in `iot-device/crypto.c`.

The following section will discuss more in detail the practical implementation of the protocols desribed in 3.1. In case of particular sections of code that require a deeper analysis, more technical details will be provided together with small code reports.

For the initial device configuration, the main function is `deviceconf()` on the DM's side, while the corresponding device-side function is called `initialization()` and can be found inside `iot-device/initialization.c` file: first, `deviceconf()` function asks to the user to type the IP address of the device that will be registered on Emercoin NVS. A connection with the device is established using Python sockets, and after this step the real registration procedure begin. The first function to be invoked is `generate_mInit()`: the purpose of this function is to build the M_{init} message described in the high-level design. It's important to specify that each message exchanged using sockets will be encoded using JSON format. The JSON M_{init} message will be structured in the following way:

- `command`: identifies the initialization message;
- `pubkey`: the public part of the RSA key previously generated, encoded using PEM format;
- `exp_date`: expiration date that the device will use to generate the self-signed X.509 certificate. This date is based on the number of validity days selected by the user;

The JSON message is sent to the device: at this point, when the message is received and verified by the device using `verify_minit()` function, the latter will execute the TPM-based authentication procedure in order to generate the Attestation Key. The entire procedure is wrapped into a single function called `create_tpm_idevid()`: in its first phase, this function generates the `TCG-CSR-IDEVID` structure and sends it to the DM. When the message containing the `TCG-CSR-IDEVID` is received, the DM completes the procedure by calling two functions: `verify_idevid()` and `make_credential()`. `verify_idevid()` extracts the device information (model name, serial number and EK certificate) in order to check if the EK certificate matches the one stored locally. Finally, the signature is extracted together with the public part of the generated Attestation Key necessary to verify it. If `verify_idevid()` ends successfully, `make_credential()` is executed: its purpose is to run the `TPM2_MakeCredential` command and send the output of the latter to the client. As discussed before `TPM2_MakeCredential` is used to produce a challenge for the device in order to verify the Proof-of-Possession of the key certified by the EK certificate and to prove that the AK resides in the same chip of the EK. To achieve this result, the DM generates a random nonce that will be used by `tpm2_pytss_utils.make_credential()` in order to produce an encrypted (using the received public EK) credential blob that can be decrypted only using the corresponding private part of the EK that resides in the device's TPM. The device extracts the nonce by decrypting the credential blob using `Esys_ActivateCredential()` and sends back the solved challenge to the DM, that verifies if the received solution matches the previously generated nonce. The input parameters of `tpm2_pytss_utils.make_credential()` will be filled with the following data:

- `public (TPMT_PUBLIC)`: this will be the public area of device's TPM EK;
- `credential (bytes)`: the nonce;
- `name (bytes)`: in this case, we want to ensure that the AK resides in the same TPM where the EK resides. So, `name` will be represented by the AK name;

The output of `tpm2_pytss_utils.make_credential()` is composed by two objects: the *credential blob* (the encrypted challenge) and the *secret*, that must be provided to `Esys_ActivateCredential()` to extract the original credential. When the client receives the encrypted challenge containing the credential blob and the secret, `Esys_ActivateCredential()` is invoked, but its functioning is a bit more complex and requires an extra explanation. This function receives (apart from the `ESYS_CONTEXT` the following required input parameters:

- `activateHandle`: the `ESYS_TR` handle of the key that will be attested to be resident in the same TPM where the EK resides. In this case this handle will reference the AK;

- `keyHandle`: the `ESYS_TR` handle of the key used to solve the challenge. In this case, it will reference the EK;
- `shandle1`: the `ESYS_TR` handle of the TPM session in which the key referenced by `activateHandle` was created (AK);
- `shandle2`: like `shandle1` but for the key referenced by `keyHandle` (EK). For our implementation, this session must satisfy some specific requirements. If `keyHandle` points to a restricted key (like the EK), the TPM grants the authorization to use it for credential decryption only if `shandle2` points to a valid *policy session*. To create a *policy session*, the `Esys_StartAuthSession()` function must be called passing the pre-defined constant `TPM2_SE_POLICY` as `session_type`. In order for a TPM command to be authorized to use a specific TPM object using a session policy, the policy hash value contained in session's metadata must be equal to `authPolicy` field value included in the `TPM2B_PUBLIC` structure passed to the `Esys_CreatePrimary()` function as a required parameter. In case of a mismatch between these two values, the TPM will deny the authorization to execute the command. The `authPolicy` field of the EK can be found in the table at section B.3.3 of [67]: the table also provide an important indication to make the policy session hash value match that specific `authPolicy`. To achieve the latter result, the `Esys_PolicySecret()` function must be invoked, passing the pre-defined handle `ESYS_TR_RH_ENDORSEMENT` as `authHandle` (a required input parameter). With these adjustments the TPM grants to the application the necessary authorization for using EK to decrypt the credential blob and retrieve the challenge solution.

```
rc = Esys_StartAuthSession(ectx, ESYS_TR_NONE, ESYS_TR_NONE, ESYS_TR_NONE,
    ESYS_TR_NONE, ESYS_TR_NONE, NULL, TPM2_SE_POLICY, &symmetric,
    TPM2_ALG_SHA256, &session2);

if (rc != TSS2_RC_SUCCESS) {
    printf("AuthSession 2 error: %s\n", Tss2_RC_Decode(rc));
    return rc;
}
TPM2B_NONCE *nonceTPM;
Esys_TRSess_GetNonceTPM(ectx, session2, &nonceTPM);
TPM2B_DIGEST cpHashA = {0};
TPM2B_NONCE policyRef = {0};
INT32 expiration = -(10*365*24*60*60); /* Expiration ten years */
rc = Esys_PolicySecret(ectx, ESYS_TR_RH_ENDORSEMENT, session2,
    ESYS_TR_PASSWORD, ESYS_TR_NONE, ESYS_TR_NONE, nonceTPM, &cpHashA,
    &policyRef, expiration, NULL, NULL);
```

Figure 3.11: Policy session configuration using Esys_PolicySecret()

When the TPM-based authentication procedure is successfully completed (the solution found by the device matches the nonce originally generated by the DM), the DM receives from the device a JSON message containing all the information about the newly generated identity (identifier, public key, self-signed certificate) signed with the verified AK. This JSON message is the output of a `initialization.c` function called `create_encrypted_identity_message()`. On the DM side, `verified_encrypted_identity()` decrypts the ciphered identifier using DM's private key (the encryption is performed to prevent a potential eavesdropper that creates a new transaction on Emercoin using that identity with a different certificate hash value) and verifies the received signature using the device's public AK. If the signature is valid, the DM calls `registerDevice()`, a function that uses `python-bitcoinrpc` library [78] to interact with the Emercoin API discussed in 3.2.1.

The registration procedure ends with an acknowledgment message from the DM to the device (produced by the function `gen_cReg()`): this message is generated when the transaction has been validated and it's considered secure (6 blocks deep in the chain). To check the status of the transaction, every 10 minutes the DM uses the `listtransactions` API call with `count = 1`. Among the different fields of the returned JSON response, `confirmations` represents the depth in the chain of the block containing our transaction. When `confirmations == 6`, the block is considered secure and the registration protocol is completed with the acknowledgment message.

The last operation performed by the DM is to store all the information about the registered device status on the local storage. For each device, there's a directory whose name is represented by the device's model name and serial number. This directory includes the EK certificate, the device's AK and a configuration file that contains the following information:

- `status`: `INIT` if the device is waiting for a certificate registration, `RUNNING` otherwise (if the status is `RUNNING`, the device won't accept any initialization messages from the DM);
- `id`: name corresponding to the newly name-value pair registered on Emercoin NVS;

Also the device stores some information on `iot-device/cli-build/configuration` using a function called `save_configuration()`: together with the `status` and the `id`, the device must store the information related to the DM's public key and additionally the certificate's expiration date.

In order to update the identity information/public key related to a specific device, the DM provides another Python script whose main function is called `update_key()`, while by the device's side all the necessary functions are grouped within `iot-device/update.c` file, whose main function is called `update()`. As discussed before, the update process steps are very similar to the registration ones. The important difference is that in this case, we won't find any Attestation Key generation phase in order to authenticate the device: this procedure is only executed when the device is configured for the first time, so that the AK can be used for the authentication of the device in every other communication between the device itself and its DM. The DM starts the communication with a message very similar to the M_{init} message seen before: now, the identifier is `UPDT` and together with the latter it would be appropriate to include a nonce, so that a malicious actor won't be able to perform replay attacks later. After the verification of this first message (generated with the `gen_mUpdt()` function), the remaining part of the procedure is identical to the registration phase except for AK generation.

The last two implemented operations are device ownership transfer and device key revocation. For what concerns device key revocation, the Python application provides a single function called `revoke_key()`: when the key revocation option is selected, `revoke_key()` is invoked and asks to the user to type the ID of the device whose key must be revoked. The ID is the name corresponding to the name-value pair stored on the public ledger, previously stored by the DM at the end of registration procedure. `revoke_key()` simply wraps another function of `device-manager/emercoin.py`, a file that contains the set of functions that uses `bitcoinrpc` library to invoke Emercoin API procedures using the pre-configured RPC server. In this case, `revoke_key()` simply invokes `revokeName()` function.

The last available operation is for transferring the ownership of a device to another entity. The functions that implement this feature are contained in `device-manager/transfer_device_ownership.py` file on DM side and in `iot-device/ownership.c` on the device side. The main DM function is `ownership_transfer()` and its execution flow is similar to the previously discussed functions: before the generation of the initial command that starts the procedure, the pre-condition is that the DM knows the Emercoin address of the new owner and has previously stored its public key so that it can be loaded using the `load_new_owner_pubkey()` function. The new owner's Emercoin address will be asked later during the protocol. If these pre-conditions are satisfied, the DM generates M_{owner} by calling `gen_mOwnr()`: the message includes the identifier (`OWNR` in this case) and the public key of the new device owner. This message must be signed using the private key

```
from bitcoinrpc.authproxy import AuthServiceProxy, JSONRPCException
import json

def registerDevice(deviceId, certHash, exptime):
    rpc_connection = AuthServiceProxy("http://user:psw@127.0.0.1:9092")
    rpc_connection.name_new(deviceId, certHash, exptime, "", "hex")

## You have to wait for the block to be mined, before the call to 'name_show'
def getDeviceValue(deviceId):
    rpc_connection = AuthServiceProxy("http://user:psw@127.0.0.1:9092")
    rpc_connection.name_show(deviceId, "hex")

def transferOwnership(deviceId, newOwnerAddress):
    rpc_connection = AuthServiceProxy("http://user:psw@127.0.0.1:9092")
    rpc_connection.name_update(deviceId, "", 0, newOwnerAddress, "hex")

def updateValue(deviceId, value):
    rpc_connection = AuthServiceProxy("http://user:psw@127.0.0.1:9092")
    rpc_connection.name_update(deviceId, value, 0, "", "hex")

def revokeName(deviceId):
    rpc_connection = AuthServiceProxy("http://user:psw@127.0.0.1:9092")
    rpc_connection.name_delete(deviceId)

def getTransactionConfirmations():
    rpc_connection = AuthServiceProxy("http://user:psw@127.0.0.1:9092")
    transaction = rpc_connection.listtransactions("*", 1, 0, True)[0]
    return transaction['confirmations']
```

Figure 3.12: Functions list from `device-manager/emercoin.py` file

of the last device owner, to ensure the validity of the ownership transfer. The device will verify this signature using `verify_mOwnr()` function.

Then, the DM requires the authentication of the device involved in the communication: it receives a message created by the device (containing its ID and a nonce to prevent replay attacks) using `create_ack_message()` function , signed with the AK obtained from the registration process. When this message is validated by the DM using `verify_identity_signature()` function, the latter creates the Blockchain transaction that will sanction the ownership transfer from the previous entity to the new one. When the transaction is confirmed (always waiting until the block containing the transaction is at least 6 block deep in the chain), the DM sends an acknowledgment to the device (`gen_ack_message()`). The latter receives the acknowledgment and finally replaces its locally stored DM's public key with the new one by calling a function called

3.2.4 Modified mbedTLS for Emercoin-based certificate verification

`mbedTLS` is a C library implementing cryptographic primitives, X.509 certificate manipulation and the SSL/TLS and DTLS protocols. Its small code footprint makes it suitable for embedded systems and constrained devices in general. When two devices within the network are successfully registered, they should be able to estabilish a secure TLS channel using their self-signed certificates previously registered on Emercoin NVS. Of course, the original mbedTLS library doesn't support a similar feature: many changes are necessary, also because mbedTLS authentication does not allow to use self-signed certificates but only valid CA chains the end up with a trusted root CA certificate. The library provides a set of programs that deal with most common operations (e.g SSL client, SSL server, X.509 certificate creation and verification etc...).

For our experimental purposes, the mbedTLS application that must be modified in order to enable the Emercoin-based certificate verification is contained in a specific C file that resides in `mbedTLS/src/programs/ssl`, called `ssl_client2.c`. This program is essentially a simple implementation of the TLS protocol that receives 2 mandatory input parameters from the command line: `server_addr/server_name` and `server_port`. The interesting feature of mbedTLS is the possibility to configure a user-defined callback function called `my_verify()` that will be invoked for every certificate of the chain received by the other party. Despite of this, the rest of the certificate verification procedure is always based on the validation of the certificate chain: the only way to change this approach is to remove the code section related to the verification of the chain against pre-loaded root CA certificates and the conditional statement that prevents the usage of self-signed certificate. In this way, the entire certificate verification procedure is delegated to the user-defined callback function. `my_verify()`.

```
static int x509_crt_verify_restartable_ca_cb( ... ) {

    ...

    /* Check the chain */
    /* Chain verification code is commented
    ret = x509_crt_verify_chain( crt, trust_ca, ca_crl,
                                 f_ca_cb, p_ca_cb, profile,
                                 &ver_chain, rs_ctx );

    if( ret != 0 )
        goto exit;
    */

    /* Merge end-entity flags */
    //ver_chain.items[0].flags |= ee_flags;

    ret = f_vrfy(NULL, crt, 0, flags);

    ...
}
```

Figure 3.13: Chain verification code section in library/x509_crt.c

In the picture above, `f_verify()` is the input parameter of `x509_crt_verify_restartable_ca_cb()` that points to the callback function that's going to be called for every certificate in the loaded chain (in our case, the chain will always be composed by a single self-signed certificate): by default, this parameter is a reference to the function `my_verify()`.

Now let's concentrate on `my_verify()`: for the code development, a great support has been provided by another similar implementation based on OpenSSL [73]. This implementation is strictly related to [37] and for this reason it already suit our final purpose: what has been done was to "translate" this implementation using mbedTLS structures and its available functions. First of all, it's important to notice that for the verification against Emercoin NVS, the retrievement of the information related to the other party is performed using the serial number of the received certificate. This is because the randomly generated pseudonym (that represents the name of the name-value pair in Emercoin NVS) produced by the device during the creation of a self-signed certificate is used to fill the serial number field of the X.509 structure.

So, the first executed operation is the extraction of the serial number from the received X.509 certificate together with some string manipulations in order to remove the colon character from the original string.

After this, a JSON request must be forwarded to the Emercoin RPC server in order to retrieve the certificate hash value associated with the serial number of the received X.509 certificate. The

```
mbedtls_x509_serial_gets(cert_sn, len, &crt->serial);
/*
 * Change the serial number to lower case
 * Remove ':' from serial number
 */
char modified_sn[1024];
memset(modified_sn, '\0', 1024);
int c = 0;
int j = 0;
while (cert_sn[c] != '\0') {
    if (cert_sn[c] >= 'A' && cert_sn[c] <= 'Z') {
        if (cert_sn[c] != ':') {
            modified_sn[j] = cert_sn[c] + 32;
            j++;
        }
    } else {
        if (cert_sn[c] != ':') {
            modified_sn[j] = cert_sn[c];
            j++;
        }
    }
    c++;
}
```

Figure 3.14: Extraction and normalization of serial number from received X.509 certificate

JSON message must be composed as follows:

- `params (array)`: an array containing the name of which we want to retrieve the corresponding value and the encoding format of the latter (`hex` in this case);
- `method (string)`: the operation we want to execute (`name_show`);
- `id (string/integer)`: identifies the request (for simplicity the used id is always `1`);

The JSON message is forwarded to the RPC server using `curl` library [75], a very useful support to deal with HTTP request. To configure the HTTP request, `curl_easy_setopt()` function is used: `EMC_CORE_URL` is a pre-defined constant that represents the URL of the RPC server and `json_request` is the JSON message already discussed. `MBEDTLS_X509_EMC_FAILED_TO_CONNECT_EMC_CORE` has been added to `mbedTLS/src/include/mbedtls/x509.h` to provide specific error codes in case of failures related to Emercoin NVS (that native mbedTLS can't recognize).

The `CURLOPT_WRITEFUNCTION` option is necessary to store the HTTP response in a buffer: to set this option, it's necessary to provide a user-defined callback function that receives the data from the HTTP response and the pointer to an outpuf buffer that will be inflated.

If the requested name exists within Emercoin NVS, the RPC server provides a valid JSON response. The most important JSON field we are interested in is `value`: JSON messages are tricky to manage for C language and, for this reason, a useful support library called `jsmn` [76] was used to speed up the code development. Using `jsmn`, the `value` field containing the requested certificate hash value has been successfully extracted.

At this point, what remains is to compute the digest of the received X.509 certificate and compare it with the hash value retrieved from Emercoin NVS. For the hash computation, the X.509 raw certificate bytes can be found within the `mbedtls_x509_crt *crt` input parameter of `my_verify()`: this structure contains a field called `tbs` that includes the buffer containing certificate's raw bytes and its length. By using mbedTLS cryptographic API for digest computation

```
curl = curl_easy_init();
if (curl) {
    chunk.memory = malloc(1);
    chunk.size = 0;
    curl_easy_setopt(curl, CURLOPT_URL, EMC_CORE_URL);
    curl_easy_setopt(curl, CURLOPT_POSTFIELDSIZE, strlen(json_request));
    curl_easy_setopt(curl, CURLOPT_POSTFIELDS, json_request);
    curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_memory_callback);
    curl_easy_setopt(curl, CURLOPT_WRITEDATA, (void *) &chunk);
    res = curl_easy_perform(curl);
    if (res != CURLE_OK) {
        *flags |= MBEDTLS_X509_EMC_FAILED_TO_CONNECT_EMC_CORE;
        return MBEDTLS_X509_EMC_FAILED_TO_CONNECT_EMC_CORE;
    }
    curl_easy_cleanup(curl);
    } else {
        *flags |= MBEDTLS_X509_EMC_FAILED_TO_CONNECT_EMC_CORE;
        return MBEDTLS_X509_EMC_FAILED_TO_CONNECT_EMC_CORE;
    }
```

Figure 3.15: Configuration of the HTTP request using curl

```
write_memory_callback(void *contents, size_t size, size_t nmemb, void *userp)
    {
    size_t realsize = size * nmemb;
    struct MemoryStruct *mem = (struct MemoryStruct *) userp;

    mem->memory = realloc(mem->memory, mem->size + realsize + 1);
    if (mem->memory == NULL) {
        printf("not enough memory (realloc returned NULL)\n");
        return 0;
    }

    memcpy(&(mem->memory[mem->size]), contents, realsize);
    mem->size += realsize;
    mem->memory[mem->size] = 0;
    return realsize;
}
```

Figure 3.16: User-defined callback function for data writing

(`mbedtls_md_xxx()`, where `xxx` can be `starts`, `update` or `finish`), the hash of the received X.509 certificate is computed and the verification is finally performed. If the two values match, the TLS handshake can successfully proceeds.

3.3 Installation of testbed

The implementation that's just been proposed was deployed in a simple testbed in order to test the functionality of the Emercoin-based distributed PKI and the creation of a TLS channel between two devices registered on Emercoin NVS using the authentication mechanism proposed in 3.2.4. For the IoT device simulation, the choice was a special Raspberry Pi 4 Model B provided by LINKS foundation, a single-board micro computer configured with a TPM 2.0 chip. The board

is featured with:

- 4GBs of RAM;
- 16GBs of micro SD storage;
- USB-C for power supply;
- micro-HDMI to connect a monitor;
- USB-A to connect a keyboard;
- Infineon Iridium TPM evaluation board (TPM 9670_Raspberry) with an OPTIGA SLI 9670AQ2.0 TPM;

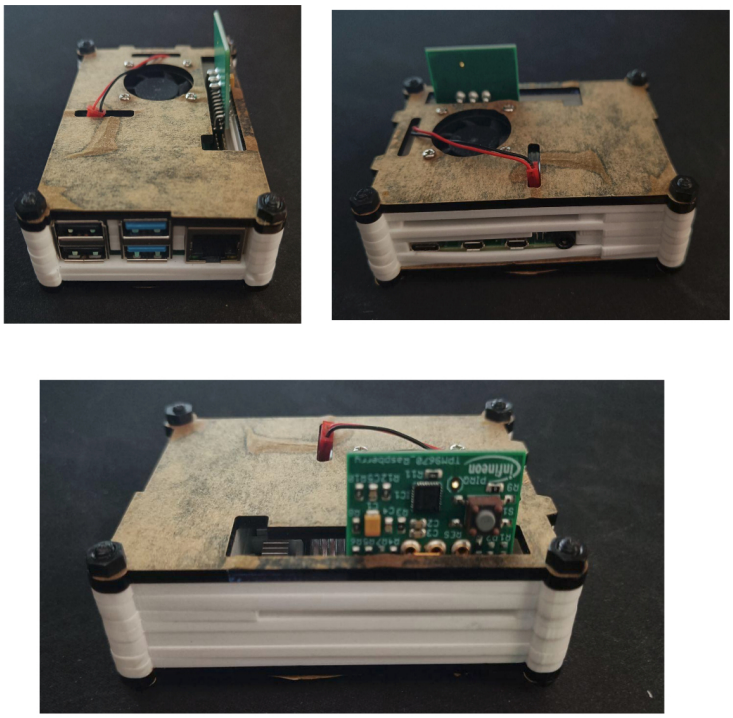

Figure 3.17: Raspberry Pi 4 with OPTIGA SLI 9670AQ2.0 TPM

This particular device was also used for another research about the addressing of software integrity attacks on IoT devices using remote attestation [50]. In fact, the most important reason behind this choice is the presence of a TPM 2.0 chip already installed on the board and configured to provide software attestation support by using uBoot boot loader to wake up the TPM and measure boot components. On the software side, the board is featured by:

- uBoot boot loader (version - 2020.04);
- Raspbian Buster Lite OS (kernel 4.19.118), no GUI;
- TPM2 TSS v.2.4.0;
- TPM2 TABRM v.2.3.1;
- TPM2 Tools v.4.2 (CLI-based TSS2 utility);
- TPM2 TSS Engine v.1.1.0;
- Cryptsetup with TPM support v.2.0.3;

Some additional software dependencies have been installed to support the implementation of the Emercoin-based PKI protocols described before: mbedTLS has been compiled from scratch with the necessary changes to support Emercoin-based certificate verification, and `curl` was installed to support HTTP requests. For the compiling process, CMake was incredibly helpful for configuring each necessary compiling option using the file `iot-device/CMakeList.txt`. The development process has been mainly conducted in a separate environment enabled to run an Integrated Development Environment (IDE). In fact, another important reason for CMake employment was its native integration with CLion IDE, used to separately develop all the code that was going to be executed by the Raspberry Pi board.

The Device Manager was deployed in a powerful desktop workstation. In our case, for sake of simplicity, the DM is also a B-node, so the Emercoin blockchain data are entirely stored in the same storage volume where the DM software resides. The workstation is featured with the following relevant hardware components:

- 16GBs RAM DDR4 3200 MHz;
- AMD Ryzen 5 5600x, 6 core CPU (max clock frequency 3632 MHz). This chip also has an internal TPM 2.0 that can be enabled from the BIOS (*fTPM*);
- 1TB Solid State Drive NVME (3400MB/s max read speed and 3000 MB/s max write speed);
- Ethernet network interface (1Gbit/s speed)

This machine runs a Linux Debian-based 64bit distribution (PopOS 21.10 [79], Linux kernel version 5.17.5-76051705-generic) and many important software dependencies necessary for running the DM software module. The most important installed modules are the `pyca/cryptography` library for cryptographic operations in Python, the `tpm2_pytss` library for TPM-based operations (this is a Python wrapper for standard TSS2 C library) and the `bitcoinrpc` Python module for interfacing the Emercoin RPC server. The Emercoin wallet was run using the `-testnet` option to enable the testnet instead of the official chain. From the wallet's interface, the RPC server was configured to accept incoming connections at port `9092`. The remaining necessary elements for the configuration procedure to work correctly in a realistic use-case scenario are the TPM Manufacturer CA certificate and the EK certificate for each device of the network (for the Raspberry Pi, the TPM manufacturer CA certificate can be retrieved from Infineon website [80]). These elements are necessary to verify the validity of the EK certificate received by the DM during the procedure.

Finally, the last element of this testbed represents the secondary device that will be configured and used to test the creation of a TLS channel between two registered devices using the modified Emercoin-based version of mbedTLS library. For this purpose, a VM was deployed using the Oracle VM VirtualBox [81] virtualization system installed on a Windows-running laptop. The laptop is a Huawei Matebook D14 with the following specifications:

- 8GBs RAM;
- 512GBs of SSD storage;
- Intel Core i3-10110U 2 cores CPU (Max clock frequency 2592 MHz);
- Intel Wireless-AC 9560 160MHz Wi-Fi network interface (Max speed 292 Mbit/s for reception and 866 Mbit/s for transmission);

The VM's guest OS is Kali-linux v2021.4 (kernel 5.14.0-kali4-amd64) with 1024 MBs of RAM. TPM2 TSS library (v3.2.0 in this case) has been installed also in this context, but in this case another TCTI (TPM Command Transmission Interface) was used. In order to provide the VM with TPM technology, a software TPM simulator was installed and configured as a background daemon. The TSS library provides different TCTI modules for different TPM implementations (hardware, virtual TPM, software TPM). For a successful emulated TPM configuration, the following software modules have been installed:

- IBM's Software TPM 2.0 [82];
- TPM2 TSS library (v 3.2.0)
- TPM2 ABRMD (Access Broker and Resource Manager) [83]: normally with a real physical TPM device, there's the resource manager (TPMRM) that manages the TPM context in a manner similar to a virtual memory manager; it swaps objects, sessions, and sequences in and out of the limited TPM memory as needed. This layer is mostly transparent to the upper layers of the TSS and is not mandatory. However, if not implemented, the upper layers will be responsible for TPM context management. In this case, since a software TPM is going to be used, an access broker implementation of it must be installed (which is what tpm2-abrmd is all about).

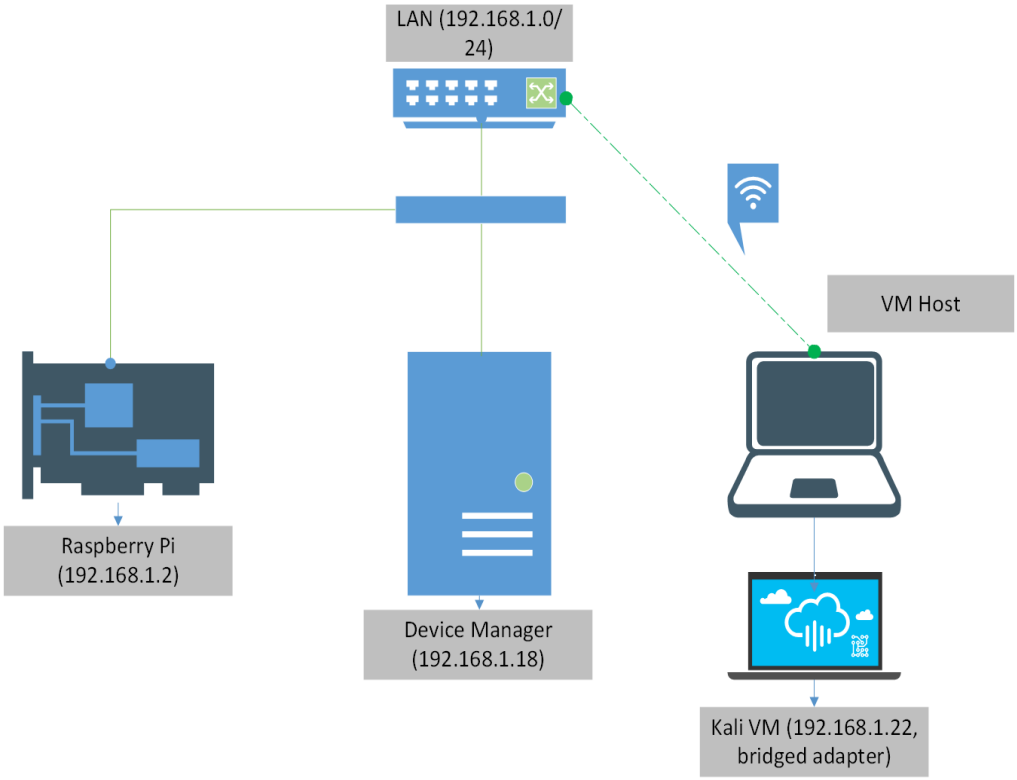

Figure 3.18: Testbed configuration

3.4 Related issues

Despite of the great improvement brought to the original design, which now provides a solid method to prevent malicious actions from "lying entities" within the network, it's still important to take into account that the background technology which enables this important feature (TPM) is not usually supported by IoT and constrained resource devices in general. Naturally, this implementation was proposed for experimental purposes, but it's also important to notice that TPM is not an expensive technology: originally, the average cost of a TPM 2.0 chip was around 10-15$. In recent times, this cost increased because Microsoft designated the TPM 2.0 as a stricly required technology for the installation of Windows 11. The consequence of this event was an incredible increase of the demand for TPM chips that caused a quadruplication of the average price in the first period. Actually, due to the widespread of TPM technology, this issue (and consequently also the price of TPM chips) has been drastically resized.

Internet-of-Things includes a huge variety of devices for many purposes, from simple sensors to more complex and expensive devices for critical applications. Not every IoT application requires

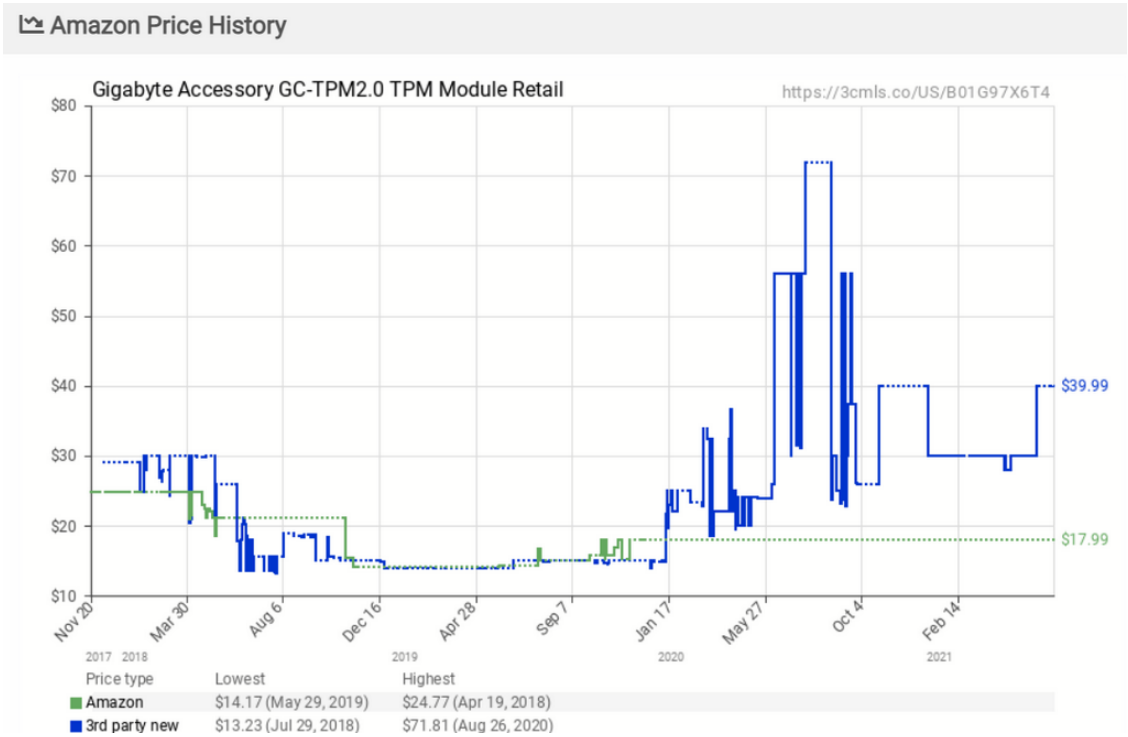

Figure 3.19: TPM 2.0 chips price variation (2018-2021) (source: [11])

high security level, especially because in many cases the data processed and transmitted by these devices do not represent a critical asset or a sensitive source of information that must be carefully protected. For other critical application like smart cars and smart healthcare, the integration of TPM technology could be a reasonable option to improve the level of security in exchange of a moderately cheap investment.

Chapter 4

Measurements and comparison

This section will present the results of the experiments conducted on the implementation proposed in the previous chapter. The purpose of the tests that have been conducted was first to provide a general impression of the necessary time for the entire certificate registration process using the proposed system (Blockchain transaction confirmation time won't be taken into account), also considering the presence of the TPM-related operations that surely require a certain amount of additional time. Additionally, the tests aim at comparing the necessary time to complete a TLS handshake in order to estabilish a secure connection for the two PKI approaches ("standard" and distributed).

It's not easy to set up a test laboratory that can define balanced conditions in which perform this kind of tests without any significant bias: in the original paper, the comparison with the centralized "standard" PKI approach was performed using a Google certificate for the verification process; this scenario is sharply different from the second one where a local Emercoin RPC server provides the information about devices' certificate, because in the first case there's an important additional time component associated with the distance of the server the provides the certificate information (an Internet connection is used). To address this issue, the "test laboratory" presented in this section has been configured to be as "fair" as possible.

4.1 Performed tests

The experiments will be divided in three subsections:

1. **Performance measurement of the certificate registration process based on Emercoin NVS**: the test will be performed first on the Raspberry Pi client using the hardware TPM 2.0 chip and then on the Kali-linux virtual machine set up with the software TPM simulator, to specifically point out the performance differences between a real and simulated technology;
2. **TLS handshake time measurement using "standard" PKI approach**;
3. **TLS handshake time measurement using distributed Emercoin-based PKI approach**;

4.1.1 Certificate registration process

For the certificate registration process, it can be useful to provide a measurement of the performance especially for comparing the two different employed implementations of the TPM technology. The software TPM emulator should only be used for experimental and study purposes: from what concerns its security level, it can't offer a good level of security when compared to the effective hardware implementation. For this reason, two different measurement were collected for

the registration process: a time measurement collected using the Raspberry Pi 4 board with the hardware TPM and another one using the Linux virtual machine configured with the software TPM emulator. Both this measurements will be performed without taking into account the necessary time for a transaction to be considered secure within the blockchain (6 blocks deep in the chain, $\approx$ 1 hour) because it would invalidate the result of the experiment. The measured process will include all the necessary operations except for the final Emercoin registration.

In the Linux virtual machine, the software TPM emulator components (the emulator and the ABRM) were configured as services using `systemctl` and started in background using `systemctl start <service-name>`. Another important consideration about the software TPM is about its initial provisioning: an hardware TPM is initially provisioned with an EK together with the corresponding certificate issued by the chip manifacturer; in the case of a software TPM, the EK is already configured at installation time, but the corresponding certificate must be manually provided. Using `tpm2-tools` [85] (a CLI tool to send TPM2 commands to a TPM 2.0 platform, very useful for scripting), the EK certificate (generated using OpenSSL) was persistently written on the virtual memory of the software TPM so that the entire certificate registration function could keep working with the original instruction flow.

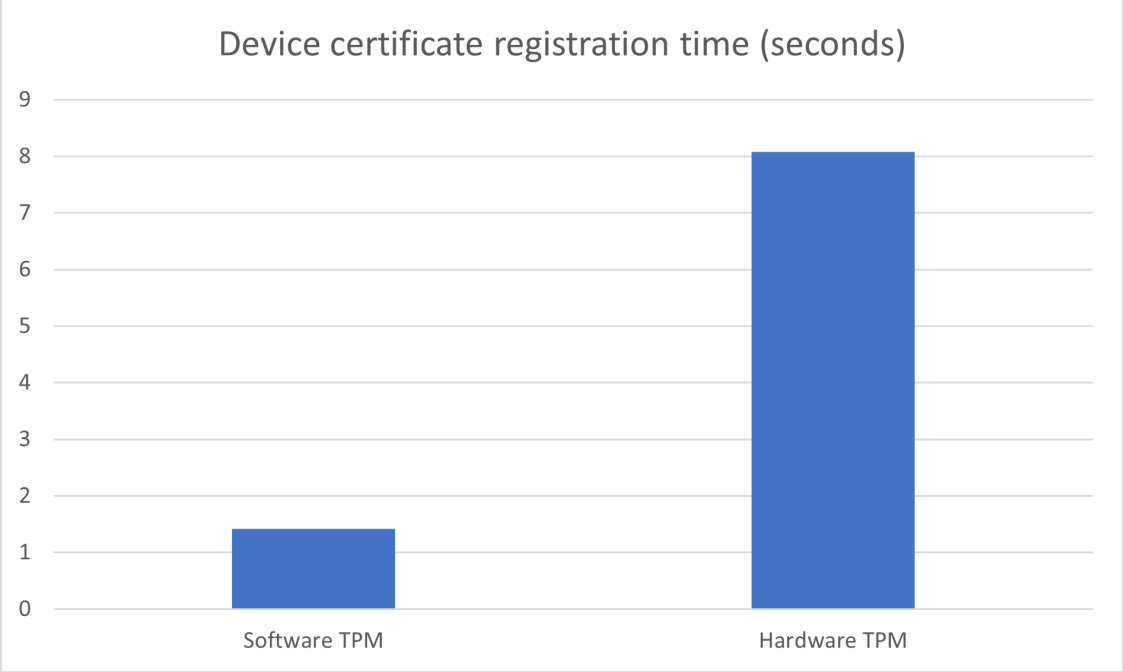

Figure 4.1: Device certificate registration time: real hardware TPM vs. software TPM emulator

As expected, the results of the same experiment on these two different TPM technology is drastically different. An important factor that influences these results is the execution platform diversity: the hardware TPM test was executed on the Raspberry Pi 4 board, while the software TPM emulator was running on a Linux virtual machine whose CPU power was limited to 50% in order to define similar conditions for both the experiments. Despite of this, this result is also one of the reason for which the employment of software TPM implementations is strongly discouraged for real applications: an emulator is forced to simplify some operations whose security level is stricly derived from the hardware presence, and these simplifications are partially responsible for the performance boost.

4.1.2 Standard TLS handshake testing environment

As anticipated in the introduction to this section, the purpose of these experiments is also to make a "fair" comparison between the alternative distributed PKI and the "standard" centralized one, by setting a test-lab where each involved actor is deployed locally. The measurement is related

to the standard OpenSSL handshake function: for this purpose, an additional modification was applied the standard version of the library in order to measure the time spent to complete the handshake process. The measurement technique is practically identical to the one used for the certificate registration process.

```
static void print_stuff(BIO *bio, SSL *s, int full) {
    clock_t start, end;
    double cpu_time_used;
    start = clock();

    ...

    verify_result = SSL_get_verify_result(s);
    BIO_printf(bio, Verify return code: %ld (%s)\n", verify_result,
        X509_verify_cert_error_string(verify_result);

    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC * 1000;
    mbedtls_printf(\Handshake time: %5f ms", cpu_time_used);
}
```

Figure 4.2: OpenSSL handshake time measurement

The modification was applied to the `print_stuff()` function within the `s_client.c` file (it can be found under `apps/`, on the official GitHub repository of OpenSSL [74]), a simple test application provided by OpenSSL for testing the creation of a TLS channel with a listening node.

For this experiment, the listening node was a Linux VM hosted by the Linux Desktop workstation already mentioned in the section dedicated to the implementation's testbed. An Apache server was deployed and configured on this machine for replying to any incoming TLS connection. The Apache server has been provided with a X.509 certificate issued by a local root CA, created using OpenSSL: for the generation of a certificate that includes the specific X.509v3 extension that points to the OCSP responder in charge of providing status information about the validity of the certificate, the local CA must be properly configured with a little adjustment on the configuration file. In this case, the original OpenSSL configuration file (`/etc/ssl/openssl.cnf` on the Linux virtual machine) was duplicated and the copy was modified by adding the following line under the tag `[usr_cert]`:

```
[ usr_cert ]
authorityInfoAccess = OCSP;URI:http://192.168.1.20:8080
```

In this case, the OCSP responder's IP address (192.168.1.20) is of a remote Linux VM hosted by the Windows laptop already mentioned in the implementation testbed section. The OCSP responder was configured on a different machine to create a use-case scenario as realistic as possible (further details on the OCSP responder configuration will be provided later in this section).

At the end of the same file, an additional tag must be appended for a correct configuration:

```
[ v3_OCSP ]
basicConstraints = CA:FALSE
keyUsage = nonRepudiation, digitalSignature, keyEncipherment
extendedKeyUsage = OCSPSigning
```

After this, the local root CA is created starting from an asymmetric RSA keypair (generated by running the command `openssl genrsa -out rootCA.key 2048`) and an X.509 certificate associated with the public part of the key (created with the command `openssl req -new -x509 -days 3650 -key rootCA.key -out rootCA.crt -config validation.cnf`, where `validation.cnf` is the modified configuration file previously discussed). After the creation of the local root CA,

the remaining steps are related to the issuance of a valid certificate for our Apache server and the creation of an OCSP responder whose certificate will also be issued by the same local root CA. The provisioning of a valid certificate for the Apache server can be performed with the following operations:

1. Create another RSA key for the Apache server (always using `openssl genrsa -out apacheServer.key 2048`);
2. Create the X.509 certificate associated with the key generated in the previous step (always using `openssl req -new -x509 -days 3650 -key apacheServer.key -out apacheServer.crt -config validation.cnf`);
3. Generate the Certificate Signing Request (CSR) for the previously generated X.509 certificate, using the command `openssl x509 -x509toreq -in apacheServer.crt -out CSR.csr -signkey apacheServer.key`;
4. Sign the Apache server certificate using the local CA previously created (the certificate will include the OCSP responder URI, thanks to the modified configuration file): the command used for this operation is

   ```
   openssl ca -batch -startdate <start-date> -enddate <end-date> -keyfile
   rootCA.key -cert rootCA.crt -policy policy_anything -config validation.cnf
   -notext -out apacheServer.crt -infiles CSR.csr;
   ```

Finally, the OCSP responder is configured bu running the two following commands:

1. `openssl req -new -nodes -out ocspSigning.csr -keyout ocspSigning.key` for generating the OCSP signing key together with the CSR;
2. `openssl ca -keyfile rootCA.key -cert rootCA.crt -in ocspSigning.csr -out ocspSigning.crt -config validation.conf` for generating an X.509 certificate issued by the local CA for the OCSP responder;

As anticipated before, the OCSP responder is running on a separate machine to propose a more realistic scenario: this machine (the Windows laptop) hosts a Linux VM properly configured with the previously generated OCSP key and certificate, that simply execute the OCSP responder function in background, thanks to a command provided by OpenSSL:

```
openssl ocsp -index demoCA/index.txt -port 8080 -rsigner ocspSigning.crt -rkey
ocspSigning.key -CA rootCA.crt -text -out log.txt &
```

(`index.txt` is a configuration file automatically generated when the local CA signs the first end-user certificate)

While the OCSP VM is running, the same machine hosts an additional Linux VM that represents the client that is going to start the TLS connection with the Apache server. The connection is started using the following OpenSSL command:

```
openssl s\_client -connect 192.168.1.21:443 -status -CAfile
    /path/to/CAfile/rootCA.crt
```

The `-status` option enables the OCSP stapling (RFC-6961 [84]) feature for the new TLS connection: OCSP stapling is used only if requested by a client, which submits the `status_request` extension in the handshake request. A server that supports OCSP stapling will respond by including an OCSP response as part of the handshake. By default, the Apache server doesn't provide the OCSP stapling feature. This option was enabled during the configuration of the Apache sever by adding the following lines:

- `SSLStaplingCache` option must be added to specify the path of the local OCSP cache file (OCSP stapling doesn't work if this option is missing);

- `VirtualHost *:433` allows the Apache server to accept connections on every network interfaces (not only `localhost`): in this case the server will be contacted on its LAN IP address 192.168.1.21;
- `SSLEngine on` enables TLS connections;
- `SSLUseStapling on` enables OCSP stapling;
- `SSLCertificateFile` specifies the path of the X.509 server certificate;
- `SSLCertificateKeyFile` specifies the path of the private key corresponding to the X.509 certificate specified before;
- `SSLCACertificateFile` specifies the path of the CA's X.509 certificate;

```
<IfModule mod_ssl.c>
        SSLStaplingCache shmcb:/var/run/ocsp(128000)
        <VirtualHost *:443>
                ...

                SSLEngine on
                SSLUseStapling on
                SSLCertificateFile /path/to/end-certificate/clientCert.crt
                SSLCertificateKeyFile /path/to/end-keyfile/clientKey.key
                SSLCACertificateFile /path/to/CAfile/rootCA.crt

                ...
        </VirtualHost>
</IfModule>
```

Figure 4.3: Apache server configuration

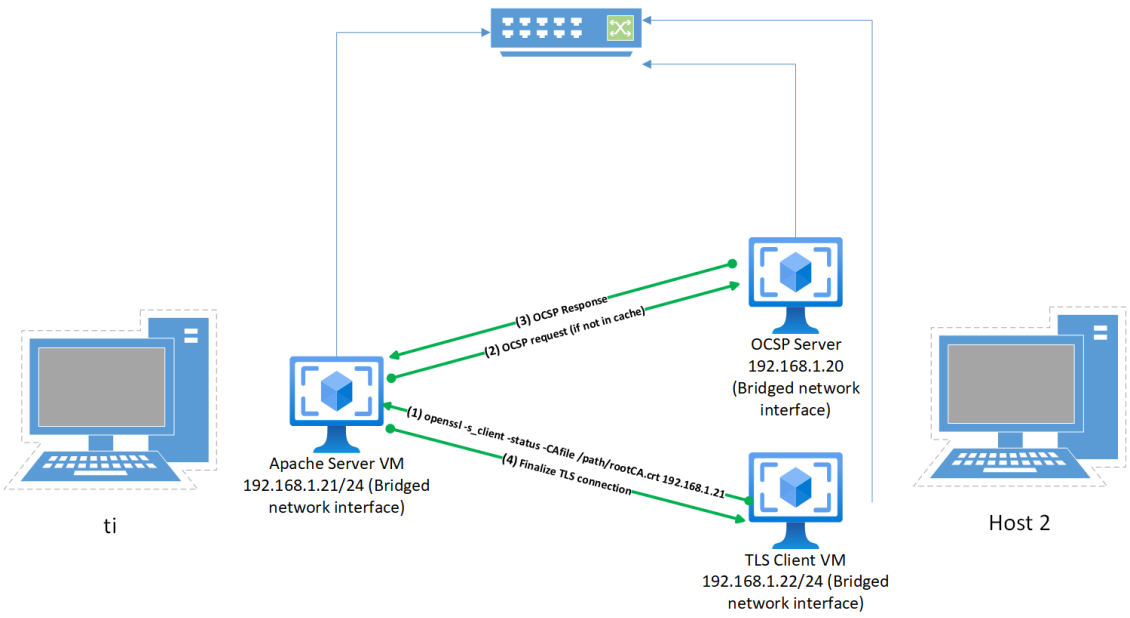

Figure 4.4: Testing laboratory configuration

With this configuration, three different experiments were conducted:

1. TLS handshake time using OCSP stapling without any cached data;
2. TLS handshake time using OCSP stapling with cached data;
3. TLS handshake time without using OCSP responder;

The result of each experiments was obtained by calculating an average time on 20 connection trials. The average time for a successful TLS handshake for the three experiements was:

- **OCSP stapling with no cached data**: 26.137 ms;
- **OCSP stapling with cached data**: 11.949 ms;
- **No OCSP**: 7.925 ms;

4.2 Comparison

The results of the experiments using the "standard" PKI approach in a local test environment must be compared with an evaluation of the performance related to the proposed distributed PKI implementation, in order to make the appropriate final considerations. In this case, the experiment configuration is far simple, because the only necessary precondition is to have two devices of the network correctly registered on Emercoin NVS, using the certificate registration procedure that's been discussed in the third chapter. After the certificate registration preliminary phase, we can consider a test environment composed by the Raspberry Pi 4 board that will represent the relying party of the TLS connection, the Linux Desktop workstation that stores a copy of the Emercoin public ledger and represents the trusted B-node, and finally a Linux virtual machine deployed in a separate host that will represent the entity that provides its own certificate for the TLS authentication.

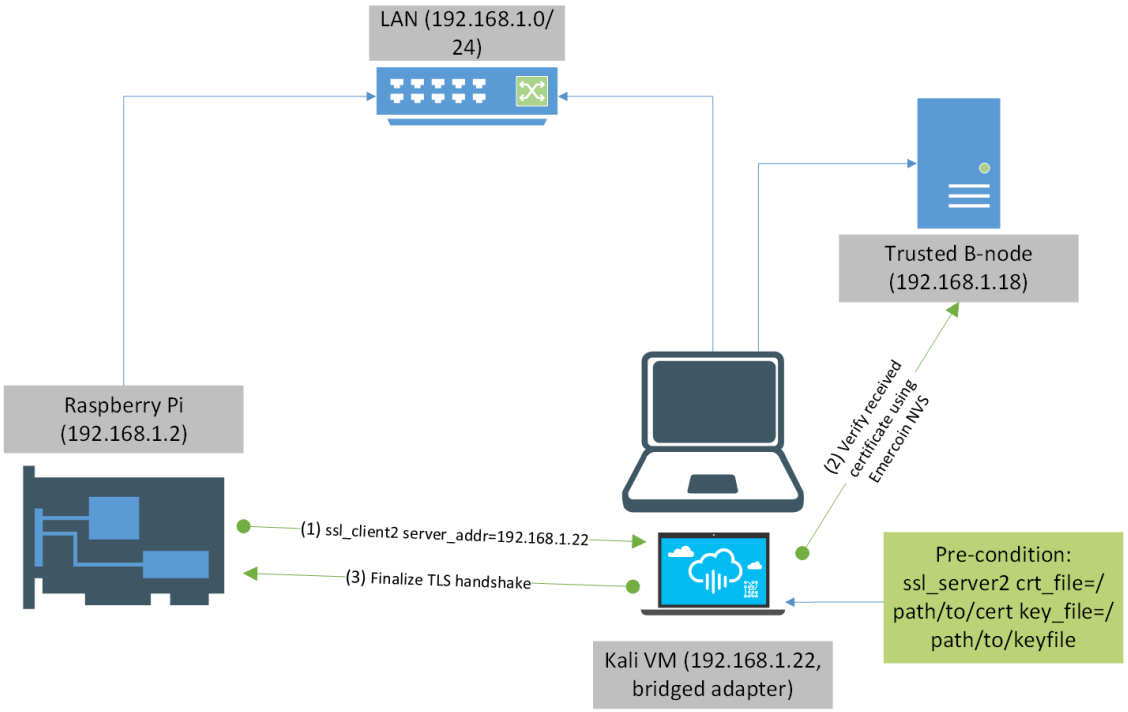

Figure 4.5: Emercoin-based PKI experiment setup

The latter is going to accept TLS connection by using the following mbedTLS command

```
/path/to/mbedTLS/programs/ssl/ssl_server2 crt_file=/path/to/certificate
    key_file=/path/to/keyfile
```

The relying party (Raspberry Pi 4) will instead execute the "client" program provided by mbedTLS:

```
/path/to/mbedTLS/programs/ssl/ssl\_client2 server\_addr=192.168.1.22
```

An additional adjustment to the modified version of the mbedTLS library (already shown in the third section) was applied to perform the time measurement like in the previous experiment with OpenSSL. In this case, the modification was applied to `mbedTLS/src/programs/ssl/ssl_client2.c` source code (there's only the `main()` function), using the same technique seen before.

```
/*
 * 4. Handshake
 */
clock_t start, end;
double cpu_time_used;
start = clock();

...

/*
 * 5. Verify the server certificate
 */
mbedtls_printf(  . Verifying peer X.509 certificate..." );
if( ( flags = mbedtls_ssl_get_verify_result( &ssl ) ) != 0 )
{
    ...

} else {
    mbedtls_printf( ok\n");
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC * 1000;
    mbedtls_printf(\Handshake time: %5f ms", cpu_time_used);
}
```

Figure 4.6: mbedTLS modification for time measurement

With this configuration, the measured handshake time in case of a successful certificate configuration is 123.132 ms (also in this case the final result represents the average time computed on 20 connection trials). From this experiment it's possible to notice a significant performance decrease if compared to the previous experiments on the normal PKI approach: the main reason behind the higher execution time when using the proposed implemented authentication schema, is that the certificate information is retrieved with a linear search on the distributed ledger. If we take into account the great amount of data to be process in order to find the name-value pair we are looking for (especially if compared to the minimum amount of entry monitored by our demo OCSP server), a relevant time increase in the handshake execution is justified.

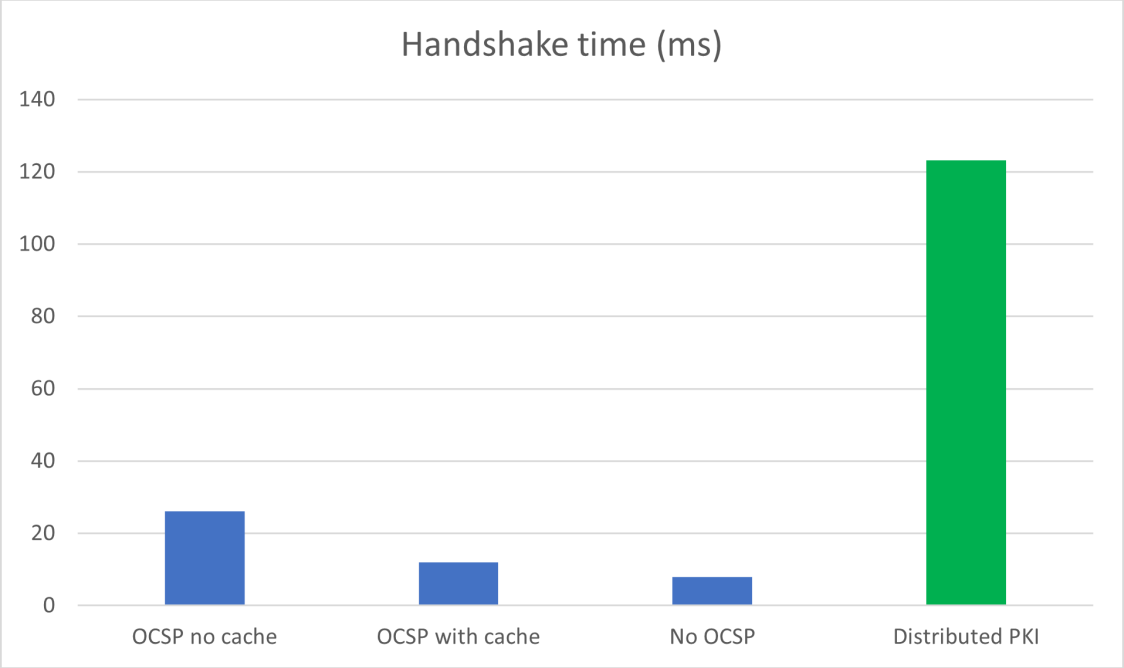

Figure 4.7: Results of the three conducted experiments

Chapter 5

Conclusion

The work carried out for this thesis project, proposes an alternative authentication mechanism that could solve many issues related to the standard centralized PKI approach when applied to particular use-case scenarios like the Internet-of-Things, that is characterized by a set of barriers and limitations that do not always allow an efficient employment of these mechanism for the creation of secure communication channels.

This alternative mechanism is based on the Blockchain technology: a preliminary research stage was necessary to underline the potential benefits this technology can bring and at the same time to analyze and discuss its limitations and the challenges for its integration with the Internet-of-Things. The most relevant obstacles to this integration are related to the low resource capabilities of IoT devices, that can't often satisfy the technological requirements necessary for the employment of the Blockchain technology.

Starting from these conditions, during the last 5 years many solutions were proposed for efficiently integrating Blockchain and IoT, in order to address the security criticalities that affect IoT devices. One of these solution, proposed in 2018 by Elisa Bertino, Ankush Singla, Greg Bollella and Jongho Won, was studied in deep, in order to propose an experimental implementation that could draw the benefits from the original solution and fix some of its security concerns. Therefore, the original solution has been implemented and improved thanks to the TPM technology, that provides a strong device identification mechanism.

Finally, some experiments have been conducted on the proposed implementation in order to underline eventual benefits/drawbacks when compared to the standard centralized PKI. From the obtained results, we can state that the Blockchain-based system has a negative impact on the performance related to the creation of a secure communication channel (e.g TLS channel), but on the other side it drastically simplifies the provisioning and the management of IoT devices keys and certificates. Moreover, this solution can benefit from the high availability and the strong integrity protection provided by the distributed design of the Blockchain in order to eliminate the Single Points of Failure represented by the Root CAs.

5.1 Future works

There are many possible further improvements that can be taken into account for future studies and proposals. Basically, the most relevant drawbacks of the Blockchain technology are:

- Transactions have a cost;
- Miners support is required;
- Low transaction validation rate;
- The size of the ledger;

Many of these issues have been addressed by the Tangle technology (discussed in 2.6) proposed by the IOTA foundation: it could be interesting to develop a software layer built upon the Tangle in order to provide a Name Value System (similar to the one provided by Emercoin) that ensures the uniqueness of each name value pair and that at the same time benefit from the Tangle technology features to achieve the elimination of miners (and the consequent elimination of the transactions cost) and a faster and more efficient transaction validation process.

Furthermore, the ledger growth can be restrained by reducing the number of transactions related to the PKI operations: a possible solution is to use the Merkle Hash Trees in order to group a set of IoT devices within a single <name,value> pair where the value is represented by the root of a Merkle Hash Tree whose leafs represent each single IoT device of the network. With this approach, the certificates of the entire IoT devices network could be stored on the Blockchain using a single transaction instead of many, but the certificate verification process during the creation of the secure communication channel must be performed using Merkle Proofs instead of the simple certificate's hash value.

Chapter 6

User's manual

6.1 Preliminary steps

The application is split into two modules: one for IoT devices and one for the Device Manager. The IoT device module must be compiled to be executed and for both the modules the `tss` library and other secondary dependencies must be installed: the description of these operations is demanded to 7.

It's important to perform some preliminary operations to successfully use the application. (Note: in this chapter we assume the presence of a TPM 2.0 chip on each device. In case of a software TPM emulator, further details for the configuration will be provided in 7)

On the device side, the file `iot-device/device_identity` must contain the model name and the serial number of the device separated by a whitespace;

On the side of the Device Manager, for every IoT device within the network, the folder `device-manager/devices` must contain a subfolder whose name must be equal to the value stored on the file `device_identity` previously configured on the corresponding device. This subfolders must contain a file called "configuration", containing the following line

```
status=INIT
```

and the Endorsement Key (EK) certificate in PEM format. The Endorsement Key certificate can be retrieved from the device by using `tpm2-tools` for the execution of the following TPM2 command:

```
tpm2_nvread 0x01c00002 -o RSA_EK_cert.bin
```

This command will save the RSA EK certificate in DER format in a file called `RSA_EK_cert.bin`. Then, it's possible to convert a DER certificate into a PEM certificate by using the following OpenSSL command:

```
openssl x509 -inform DER -in RSA_EK_cert.bin -outform PEM -out ek_cert.pem
```

Finally, the file `ek_cert.pem` must be moved from the device to the Device Manager inside the corresponding subfolder.

`tpm2-tools` and `openssl` can be installed on Debian-based Linux distributions with the following commands:

```
sudo apt install tpm2-tools
```

```
sudo apt install openssl
```

6.2 Device module

The first module must run in background on each IoT device within the network and can be executed after the compilation by running

```
sudo ./iot-device/cli-build/iot-device
```

This section of the application does not require any interaction with the final user: the device basically waits until the Device Manager sends a M_{init}, M_{ownr} or M_{updt} message. By default, the status (stored in `iot-device/cli-build/configuration`) of the device is set to `INIT`: in this condition, the device accepts the incoming M_{init} messages, but when the certificate registration procedure is completed and the status is set to `RUNNING`, the only accepted messages are M_{ownr} and M_{updt}.

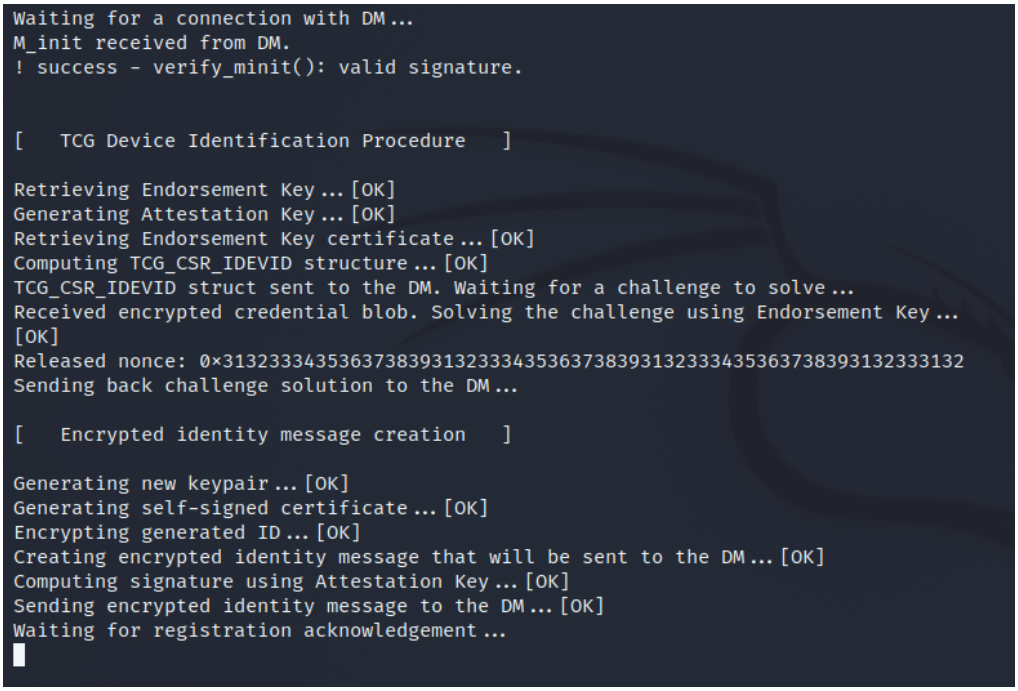

Figure 6.1: Device-side application running example: device's certificate registration

Figure 6.1 reports the example of an execution of the device software module: the application's logs summarize the entire execution flow already discussed in the previous chapters, also providing a step-by-step description of the TCG protocol used for the identification of the IoT devices. In this example, the execution is stopped because on the other side the DM has to confirm the creation of a new Emercoin transaction before sending back the acknowledgement to the device (Figure 6.3). When the transaction is created and validated by the Emercoin network, the acknowledgment is received and verified by the device: if no errors occur, the device can update its configuration file and finally store (in `iot-device/cli-build/keys/master_pubkey.pem` the DM's public key for eventual future interactions. It's important to notice that the directory `iot-device/cli-build/keys/` won't include the Attestation Key generated during the certificate registration procedure: this key is permanently stored inside the TPM and every cryptographic operation based on it, is interally managed by the application's code.

Figure 6.2 reports an example of a configuration file for a device.

The ID is the unique label associated with the certificate hash value stored on Emercoin NVS. The expiration date is calculated by adding the number of validity days to the current date at the time of the execution of the registration procedure. Practically, Emercoin measures the expiration date in number of blocks (further details in 6.3).

```
STATUS RUNNING
ID 9688180d6365a699dfc69cdd43e042c27663a79dd924654d56d0630bc46ab1fb
EXP_DATE 04/11/2022, 15:35
```

Figure 6.2: Device configuration file example

6.3 Device Manager module

The second module (`device-manager/`) is for the Device Manager node. The application can be run using Python 3, by executing the command

```
sudo python3 main.py
```

when the current working directory is `device-manager/`. In this case there's an initial interactive command-line interface: here, the user can select the operation to perform on the IoT devices of its network. When the option is selected, the CLI requires to type the IP address of the interested device and eventual additional information (e.g. the number of validity days in case of a certificate registration). Before the final Blockchain transaction is created, the application asks to the user for a final confirmation.

```
Welcome to your personal Device Manager! Choose an option:
 1. Initialize new IoT device
 2. Device ownership transfer
 3. Device key update
 4. Device key revocation
 5. Exit

>> 1
Insert the IP address of a device in your network: 192.168.1.22

Generating the initialization message M_init...
Select expiration time [days] (MAX: 2): 2


[   TCG Device Identification Procedure   ]

Extracting information from the TCG_CSR_IDEVID structure...
Device model name and serial number: kali-vm-1
Verifying Endorsement Key certificate matching for kali-vm-1...[OK]
Verifying TCG_CSR_IDEVID message signature using the provided public Attestation Key...[OK]
Generating the credential for the challenge...
Generated credential: 3132333435363738393132333435363738393132333435363738393132333132
Generating encrypted credential blob using TPM2_MakeCredential...[OK]
Sending the encrypted credential blob to the device...[OK]
Waiting for the solution of the challenge from the device...
Received challenge solution: b'12345678912345678912345678912312'
Correct credential. Successful TPM key attestation!

[   Encrypted identity message verification   ]

Extracting information from the received message...
Decrypting the ciphered generated identity using the provided public key...[OK]
Retrieving the public Attestation Key associated to the device...[OK]
Verifying signature using device's public Attestation Key...[OK]
Correct signature! (Verified using device IAK)


Please confirm registration for 9688180d6365a699dfc69cdd43e042c27663a79dd924654d56d0630bc46ab1fb [y/n]:
```

Figure 6.3: Confirmation request before creating a new Emercoin transaction

For the certificate registration process, the user must first select the IP address of the device in the network; if the device is reachable, the application asks to the user the expiration time for the certificate that is going to be created. Also in this case, the application provides some event logs, to track the most important steps of the procedures described in the third chapter. If the user confirms the registration of the device on Emercoin NVS, a notification from the Emercoin Wallet should be quickly received.

If the notification has been correctly received, the transaction's data can be consulted in the "Transactions" tab of the Wallet: the transaction will be pending until the validation after

the mining process. As long as the transaction is pending, it won't be possible to retrieve any information about it from the Blockchain.

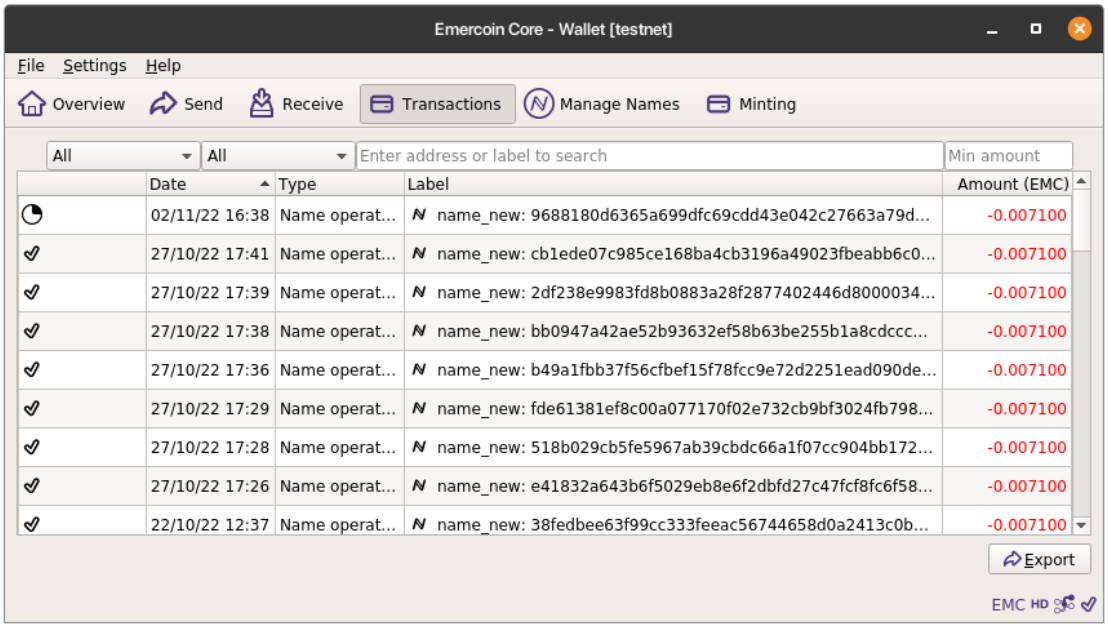

Figure 6.4: Emercoin Wallet: Transactions tab

The record on the top of this list represent the newly generated key-pair for our device. The symbol displayed on the left of the date tracks the depth of the block containing our transaction in the chain: when this symbol becomes a green check mark (like the rest of the transactions displayed in the tab), our transaction starts to be considered secure (because 6 blocks have been "mined" after the one containing the transaction). The necessary time to see the green checkmark is $\approx$ 1 hour (1 block validation each 10 minutes).

Moreover, an overview of the current registered name-value pairs is available in the "Manage Names" tab, where it's possibile to check the "expired" flag to display the information associated the expired name-value pairs. The "value" field can't be correctly displayed because during the certificate registration procedure, the hash value of each certificate is stored as a byte value in hex format (some values are not associated with printable characters). The expiration is expressed in number of days before the expiration: under the hood, Emercoin NVS translates the number of days in number of mined blocks starting from the block containing the transaction. This can be done because the blocks mining rate is always the same, so it's possible to map the number of mined blocks to the number of elapsed days. To provide a general unit of measurement, 1 days is approximately corresponding to 175 blocks.

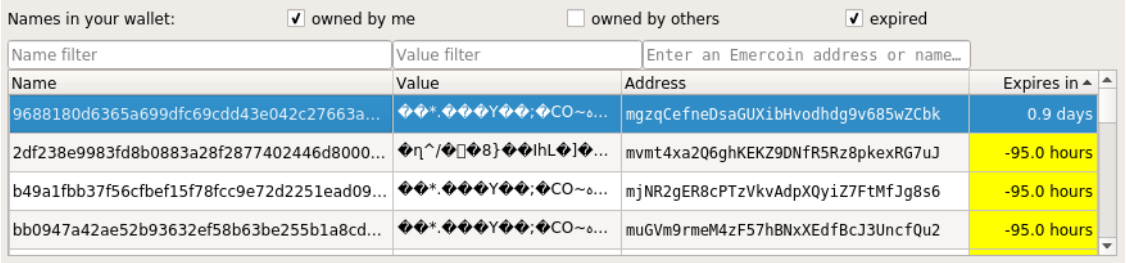

Figure 6.5: Emercoin Wallet: Manage Names tab containing the new name-value pair

When the certificate registration process is successfully completed, the subfolder corresponding to the configured device should contain the following files:

1. `configuration`: basically the same configuration file that's been already discussed for the IoT device module;
2. `ek_cert.pem`: the device's TPM Endorsement Key certificate in PEM format (this must be already included before the execution of the certificate registration process);
3. `iak.pem`: the device's Attestation Key created during the TCG procedure, in PEM format;
4. `local_pubkey.pem`: the RSA public key generated by the IoT device for the creation of the self-signed X.509 certificate that is finally stored on the Blockchain;

The other available operations (ownership transfer and certificate update) follow the same execution flow, but the ownership transfer protocol requires an additional preliminary step: the `keys` folder must include a file called `new_pubkey.pem` containing the public key of the next device's owner, in PEM format.

Finally, the revoke operation is equivalent to the creation of a revoke transaction from the Emercoin Wallet. In this case, the application will ask for the name of the <name,value> pair instead of the IP address of the device like for the other operations.

Chapter 7

Developer's manual

The entire manual will be referred to a Debian-based Linux distribution as development environment. All the commands and procedures are based on this assumption.

7.1 Required software dependencies

7.1.1 TPM 2.0 Software Stack (TSS2)

It's strongly suggested to manually build the `TSS2` library, starting from the source code available on the official GitHub repository [70]: this approach allows to select many configurable options before the installation. The installation process can be summarized by the following steps:

1. Clone the GitHub repository using

   ```
   git clone https://github.com/tpm2-software/tpm2-tss.git
   ```

2. Install all the required dependencies by running the command suggested in the `INSTALL.md` file of the GitHub repository ("Ubuntu" section);

3. Move the current working directory to the cloned repository, using

   ```
   cd /path/to/repository/tpm2-tss/
   ```

4. Run the bootstrap script (`./bootstrap`);

5. Configure the build using `./configure`. In this step it will be possible to select the configuration options mentioned before: the list of the available options can be retrieved using

   ```
   ./configure --help
   ```

6. Compile the libraries using `make`;

7. Install the libraries using `sudo make install`;

7.1.2 Device Manager module dependencies

The Device Manager software module requires Python3 to be executed. Python3 can be installed by running the following command:

```
sudo apt install python3
```

In addition to the `TSS2` library required for both the modules of the application, the Device Manager module relies on some specific Python3 packages. All the Python packages can be installed and managed using `pip` (a Python package manager). This tool can be installed by running the following command:

```
sudo apt install python3-pip
```

Using `pip`, the packages can be installed by simply running `pip install <package-name>`. The required packages for the Device Manager module of the application are listed below:

- **python-bitcoinrpc**: provides an interface to the Emercoin API (based on Bitcoin API);
- **tpm2-pytss**: Python wrapper for `TSS2` library;
- **cryptography**: Python module for cryptographic operations;

The device-manager module is going to communicate with the Emercoin Blockchain by leveraging on a RPC server: the RPC server can be configured from the Emercoin wallet (it can be downloaded from the official Emercoin website [57]):

1. From the Emercoin wallet GUI toolbar, select "Settings" > "RPC" > "emercoin.conf". This will open the Emercoin wallet configuration file;
2. Replace the content of the file with the following lines:

```
testnet=1
server=1
listen=1
rpcuser=user
rpcpassword=psw
rpcport=9092
rpcallowip=0.0.0.0/0
```

3. Save the changes and restart the wallet (note: the first time you run the wallet, the `-testnet` option must be included)

7.1.3 Device module dependencies

On the device side, in addition to the `TSS2` library, two more software dependencies must be satisfied: `mbedTLS` (further details on how to install the modified version of mbedTLS will be provided in 7.3) and `jsmn`, that simplifies the handling of JSON objects. The latter is a simple header file (`jsmn.h`) that has been already included in the application (no additional steps are required).

7.2 Enabling TPM 2.0

The `TSS2` library requires the presence of a TPM 2.0 chip to be used: for the implementation proposed in this work, the TPM 2.0 chip was already installed and configured on the Raspberry Pi 4 board. The purpose of this section is to provide the instructions to enable the TPM 2.0 technology for two distinct scenarios:

1. Testing the application on any laptop/desktop machine whose motherboard is featured with a TPM 2.0 chip;
2. Testing the application using a software TPM emulator;

In the first case, the only required operation is to enable the TPM technology from the BIOS settings of the motherboard. This procedure can vary depending on the running BIOS. Figure 7.1 shows an example of a BIOS tab where it's possible to enable/disable the TPM technology.

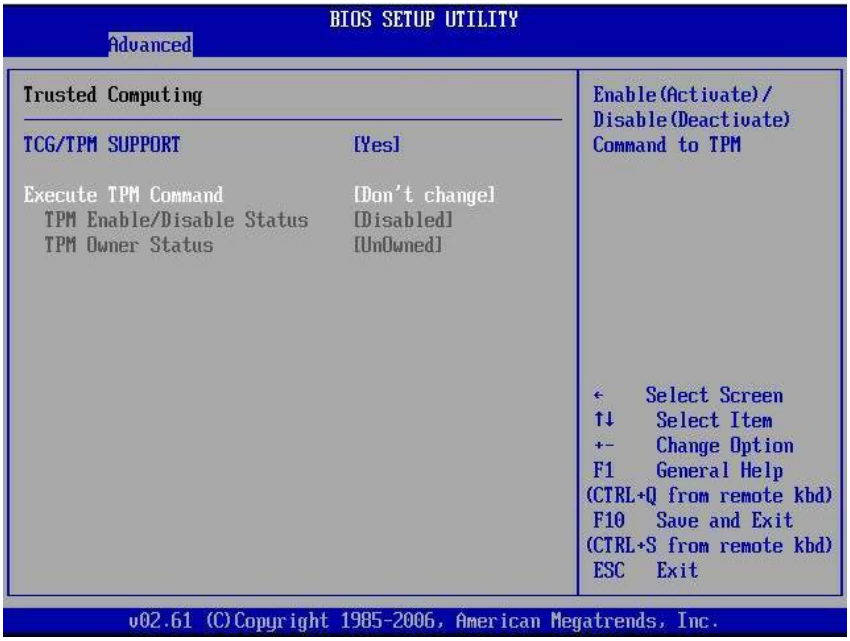

Figure 7.1: Example of a BIOS option for enabling TPM technology (source: image)

7.2.1 Configuring the software TPM emulator

The software TPM emulator is a good solution to the absence of a TPM 2.0 chip, and it's very useful for experimental and study purposes. The configuration of the TPM emulator requires the following software dependencies:

- **IBM's Software TPM 2.0**: the software TPM emulator;
- **TPM2-TSS**: TPM 2.0 Software Stack (installation steps described in 7.1.1);
- **TPM2 Access Broker and Resource Manager (ABRMD)**: for the management of the TPM context;
- **tpm2-tools**: CLI tools for executing TPM2 commands (installation: `sudo apt install tpm2-tools`)

First, the IBM's Software TPM 2.0 dependencies must be installed:

```
sudo apt install lcov \
pandoc autoconf-archive liburiparser-dev \
libdbus-1-dev libglib2.0-dev dbus-x11 \
libssl-dev autoconf automake \
libtool pkg-config gcc \
libcurl4-gnutls-dev libgcrypt20-dev libcmocka-dev uthash-dev \
```

Then, the Software TPM can be downloaded by running the following command:

```
wget https://jaist.dl.sourceforge.net/project/ibmswtpm2/ibmtpm1661.tar.gz
```

When the downloaded archive has been extracted, the `make` command must be run within the `src/` directory of the Software TPM to start the compiling process. When the compiling process is completed, the generated binary file called `tpm_server` must be moved to `/usr/local/bin`. Now, the Software TPM can be configured as a daemon service of the operative system with the following steps:

1. Create the daemon configuration file using

   (`sudo touch /lib/systemd/system/tpm-server.service`)

2. Add the following content to the file

```
[Unit]
    Description=TPM2.0 Simulator Server daemon
    Before=tpm2-abrmd.service
[Service]
    ExecStart=/usr/local/bin/tpm_server
    Restart=always
    Environment=PATH=/usr/bin:/usr/local/bin
[Install]
    WantedBy=multi-user.target
```

3. Reload daemon and start the service using the following commands:

```
systemctl daemon-reload
systemctl start tpm-server.service
```

The Software TPM is now installed and configured as a daemon service (the service's status can be checked with `systemctl status tpm-server`)

The installation of the TPM2 ABRMD follows a similar procedure:

1. Download TPM2 ABRMD from the official GitHub repository [83] using `wget`;
2. Extract the archive configure the installation:

```
cd tpm2-abrmd-2.3.1
sudo ldconfig
./configure --with-dbuspolicydir=/etc/dbus-1/system.d
    --with-systemdsystemunitdir=/usr/lib/systemd/system
```

3. Start the installation process with `sudo make install`
4. Add TPM2 ABRMD to the system services. During the previous step, a sample service definition is placed under /usr/local/share/dbus-1/system-services/. Copy it to the system services directory:

```
sudo cp
/usr/local/share/dbus-1/system-services/com.intel.tss2.Tabrmd.service
/usr/share/dbus-1/system-services/
```

5. Restart DBUS with `sudo pkill -HUP dbus-daemon`
6. Replace the content of `/lib/systemd/system/tpm2-abrmd.service` with the following lines:

```
[Unit]
    Descript=TPM2 Access Broker and Resource Management Daemon
[Service]
    Type=dbus
    Restart=always
    RestartSec=5
    BusName=com.intel.tss2.Tabrmd
    StandardOutput=syslog
    ExecStart=/usr/local/sbin/tpm2-abrmd
        --tcti="libtss2-tcti-mssim.so.0:host=127.0.0.1,port=2321"
    User=tss
[Install]
    WantedBy=multi-user.target
```

7. Run the service and check its state:

```
systemctl daemon-reload
systemctl start tpm2-abrmd.service
service tpm2-abrmd status
```

Basically, the Software TPM 2.0 is now correctly working. Despite of this, one last operation is required to make the application work with the emulator: by default, the simulated TPM is not provisioned with X.509 certificate for the Endorsement Key. To fix this issue, it's enough to run the bash script `ekc-inflater/ekc-inflater.sh`. In order to successfully run this script (note: root privileges are required), `tpm2-tools` and `openssl` must be installed; moreover, the two system services previously configured (TPM2 ABRMD and Software TPM) must be active. The script will generate an Endorsement Key certificate using a "dummy" local root CA, and permanently stores it on the proper TPM non-volatile area.

7.3 Building process

When everything is correctly configured, that last required operation is to compile the project (only for the device module and the modified mbedTLS library). The compiling process is highly simplified thanks to `CMake`. This tool can be installed with

```
sudo apt install cmake
```

and it's a cross-platform compiler that allow to configure the building process by leveraging on a text file called `CMakeList.txt`. This file will include all the necessary declarations (e.g. libraries path) to properly build the project. In order to successfully compile the device application module, the mbedTLS library is required. The modified library will retrieve data from the Emercoin Blockchain, so it's necessary to specify the correct Emercoin RPC server address in the source code. This can be done by editing the IP address defined in the constant at line 49 of `mbedTLS/src/programs/ssl/ssl_client2.c` Then, the mbedTLS library can be compiled with the following steps using CMake:

1. Move inside `mbedTLS/build_mbedtls` directory;
2. Configure the building process with `cmake ../src`
3. Build mbedTLS using `cmake --build .`

The binary files associated with each mbedTLS application, are included within `build_mbedtls/programs` subfolders. To compile the device application module using CMake, a similar procedure must be followed:

1. Move inside the `cli-build/` directory of the device module (`cd iot-device/cli-build`);
2. Configure the building process with `cmake ..`
3. Build the module with `cmake --build .`

The binary file for executing the application will be generated inside `cli-build/` and it will be named `iot-device`.

Bibliography

[1] RFC 8555, Automatic Certificate Management Environment (ACME), https://datatracker.ietf.org/doc/html/rfc8555

[2] Let's Encrypt, https://letsencrypt.org/

[3] IoT statistics (2022-2030), https://explodingtopics.com/blog/iot-stats

[4] T. L. Basegio, R. Michelin, A. F. Zorzo, R. H. Bordini, "A Decentralised Approach to Task Allocation Using Blockchain", In book: Engineering Multi-Agent Systems (pp. 75-91), May 2018, DOI: 10.1007/978-3-319-91899-0

[5] A. Meneghetti, M. Sala, D. Taufer, "A Survey on PoW-based Consensus", Annals of Emerging Technologies in Computing (AETiC), January 2020, DOI: 10.33166/AETiC.2020.01.002

[6] "What is Proof-of-Stake?", https://www.ledger.com/academy/blockchain/what-is-proof-of-stake

[7] O. Onireti, L. Zhang, M. Ali Imran, "On the Viable Area of Wireless Practical Byzantine Fault Tolerance (PBFT) Blockchain Networks", 2019 IEEE Global Communications Conference (GLOBECOM), December 2019, DOI:10.1109/GLOBECOM38437.2019.9013778

[8] Bitcoin Avg. Transaction Fee historical chart, https://bitinfocharts.com/comparison/bitcoin-transactionfees.html#3y

[9] Hyperledger Fabric architecture explained, https://hyperledger-fabric.readthedocs.io/en/release-1.3/arch-deep-dive.html

[10] IEN Workshops: from MAM to Streams, https://www.youtube.com/watch?v=EycFnTG748c

[11] TPM 2.0 chips price variation (2018-2021), %https://windowsreport.com/tpm-2-0-chip-price/

[12] Satoshi Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System", 2008, www.bitcoin.org/bitcoin.pdf

[13] Merkle Hash Trees, https://en.wikipedia.org/wiki/Merkle_tree

[14] "Understanding Double-Spending and How to Prevent Attacks", https://www.investopedia.com/terms/d/doublespending.asp

[15] Adam Back, "Hashcash - A Denial-of-Service Counter-Measure", 2002, http://www.hashcash.org/hashcash.pdf

[16] Namecoin, https://www.namecoin.org/

[17] C. Fromknecht, D. Velicanu, S. Yakoubov, "CertCoin: A NameCoin Based Decentralized Authentication System", May 2014, https://courses.csail.mit.edu/6.857/2014/files/19-fromknecht-velicann-yakoubov-certcoin.pdf

[18] Introduction to smart contracts, https://ethereum.org/en/developers/docs/smart-contracts/

[19] Vitalik Buterin, Ethereum whitepaper, 2013, https://ethereum.org/en/whitepaper

[20] Ethereum Virtual Machine (EVM), https://ethereum.org/en/developers/docs/evm/

[21] J.H. Khor, M. Sidorov, P.Y. Woon, "Public Blockchains for Resource-Constrained IoT Devices - A State-of-the-Art Survey", IEEE Internet of Things Journal (Vol. 8, Issue 15), March 2021, DOI 10.1109/JIOT.2021.3069120,

[22] Md Sadek Ferdous, Farida Chowdhury, Madini O. Alassafi, "In Search of Self-Sovereign Identity Leveraging Blockchain Technology", IEEE Access (Vol. 7), 2019, DOI: 10.1109/ACCESS.2019.2931173,

[23] "Nothing-at-stake problem", https://golden.com/wiki/Nothing-at-stake_problem

[24] K.Driscoll, B.Hall, H.Sivencrona, P.Zumsteg, "Byzantine Fault Tolerance, from Theory to Reality", SAFECOMP 2003 (Edinburgh, UK), September 2003, DOI: 10.1007/978-3-540-39878-3_19

[25] Miguel Casto, Barbara Liskov, "Practical Byzantine Fault Tolerance", Proceedings of the Third Symposium on Operating Systems Design and Implementation (New Orleans, USA) February 1999, https://pmg.csail.mit.edu/papers/osdi99.pdf

[26] The Linux Foundation, Hyperledger Fabric, https://www.hyperledger.org/wp-content/uploads/2020/03/hyperledger_fabric_whitepaper.pdf

[27] The Linux Foundation, Hyperledger Sawtooth, https://www.hyperledger.org/use/sawtooth

[28] IOTA Foundation, IOTA, https://www.iota.org/

[29] IOTA Foundation, IOTA Tangle, https://wiki.iota.org/learn/about-iota/tangle

[30] P.J. Atzberger, "The Monte-Carlo Method", http://web.math.ucsb.edu/~atzberg/pmwiki_intranet/uploads/AtzbergerHomePage/Atzberger_MonteCarlo.pdf

[31] IOTA Foundation, IOTA - The Coordinator, https://wiki.iota.org/learn/about-iota/coordinator

[32] IOTA Foundation, IOTA Streams, https://www.iota.org/solutions/streams

[33] "Introducing Masked Authenticated Messaging", https://blog.iota.org/introducing-masked-authenticated-messaging-e55c1822d50e/

[34] A.Carelli, A.Palmieri, A.Vilei, F.Castanier, A.Vesco, "Enabling Secure Data Exchange through the IOTA Tangle for IoT Constrained Devices", Sensors 2022, February 2022, DOI: 10.3390/s22041384

[35] Bin Cao, Yixin Li, Lei Zhang, Long Zhang, Shahid Mumtaz, Zhenyu Zhou, and Mugen Peng, "When Internet of Things Meets Blockchain: Challenges in Distributed Consensus", IEEE Network (Vol. 33, Issue 6), July 2019, DOI: 10.1109/MNET.2019.1900002

[36] M.Al-Bassam, "SCPKI: A Smart Contract-based PKI and Identity System", BBC '17: "Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts", April 2017, DOI: 10.1145/3055518.3055530

[37] J.Won, A.Singla, E.Bertino, G.Bollella, "Decentralized Public Key Infrastructure for Internet-of-Things", MILCOM 2018, October 2018, DOI: 10.1109/MILCOM.2018.8599710

[38] A.Singla, E.Bertino, "Blockchain-based PKI solutions for IoT", 2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC), October 2018, DOI: 10.1109/CIC.2018.00-45

[39] E.Kfoury, D.Khoury, "Distributed PKI and PSK exchange based on Blockchain technology", 2018 IEEE iThings and IEEE GreenCom and IEEE CPSCom and IEEE SmartData, 2018, DOI: 10.1109/Cybermatics_2018.2018.00203

[40] A.N. Bikos and S.A.P. Kumar, "Securing Digital Ledger Technologies-Enabled IoT Devices: Taxonomy, Challenges, and Solutions", IEEE Access (Vol. 10), April 2022, DOI: 10.1109/ACCESS.2022.3169141

[41] A. Reyna, C. MartÃn, J. Chen. E. Soler and M. DÃaz, "On blockchain and its integration with IoT. Challenges and opportunities" Elsevier "Future Generation Computer Systems" (Vol. 88, pp. 173-190), May 2018, DOI: 10.1016/j.future.2018.05.046

[42] I. Makhdoom, M. Abolhasan, H. Abbas, Wei Ni, "Blockchain's adoption in IoT: The challenges, and a way forward", Elsevier Journal of Network and Computer Applications (Vol. 125, pp. 251-279), November 2018, DOI: 10.1016/j.jnca.2018.10.019

[43] E. Kfoury, D. Khoury, "Securing NATted IoT devices using Ethereum Blockchain and distributed TURN servers", 2018 10th International Conference on Advanced Infocomm Technology (ICAIT), August 2018, DOI: 10.1109/ICAIT.2018.8686623,

[44] E. Beckwith, G. Thamilarasu, "BA-TLS: Blockchain authentication for TLS in IoT", 2020 7th International Conference on Internet of Things: Systems, Management and Security (IOTSMS), December 2020, DOI: 10.1109/IOTSMS52051.2020.9340204

[45] I. Amankona Obiri, J. Yang, Q. Xia, J. Gao, "A sovereign PKI for IoT devices based on the blockchain technology", 2021 18th International Computer Conference on Wavelet Active Media Technology and Information Processing (ICCWAMTIP), December 2021, 10.1109/ICCWAMTIP53232.2021.9674095,

[46] B. Khieu, M. Moh, "CBPKI: Cloud Blockchain-based PKI", ACM SE '19: "Proceedings of the 2019 ACM Southeast Conference", April 2019, DOI: 10.1145/3299815.3314433

[47] Y. Tu, J. Gan, Y. Hu, R. Jin, Z. Yang, M. Liu, "Decentralized identity authentication and key management scheme", 2019 IEEE 3rd Conference on Energy Internet and Energy System Integration (EI2), April 2020, DOI: 10.1109/EI247390.2019.9062013

[48] A. Dua, S. Sekhar Barpanda, N. Kumar, S. Tanwar, "Trustful: A decentralized PKI and Identity Management System", 2020 IEEE Globecom Workshops (GC Wkshps), March 2021, DOI: 10.1109/GCWkshps50303.2020.9367444

[49] A. Papageorgiou, K. Loupos, A. Mygiakis, T. Krousarlis, "DPKI: A blockchain-based decentralized PKI", 2020 Global Internet of Things Summit (GIoTS), June 2020, DOI: 10.1109/GIOTS49054.2020.9119673

[50] D.G. Berbecaru, S. Sisinni, "Counteracting software integrity attacks on IoT devices with remote attestation: a prototype", 2022 26th International Conference on System Theory, Control and Computing (ICSTCC), October 2022, DOI: 10.1109/ICSTCC55426.2022.9931765

[51] D.G. Berbecaru, A. Lioy, C. Cameroni, "On Enabling Additional Natural Person and Domain-Specific Attributes in the eIDAS Network", IEEE Access (Vol. 9 - pp. 134096 - 134121), September 2021, DOI: 10.1109/ACCESS.2021.3115853

[52] D. Berbecaru, A. Lioy, C. Cameroni, "Supporting Authorize-then-Authenticate for Wi-Fi access based on an electronic identity infrastructure", Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA), 11(2):34-54, June 2020, DOI: 10.22667/JOWUA.2020.06.30.034

[53] Trusted Computing Group, Trusted Platform Module (TPM) Summary, https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/

[54] Y.Gao, S.F.Al-Sarawi, D.Abbott, "Physical unclonable functions", 2020, https://www.nature.com/articles/s41928-020-0372-5

[55] Contract ABI Specifications, https://docs.soliditylang.org/en/v0.8.13/abi-spec.html

[56] Emercoin NVS documentation, https://emercoin.com/en/documentation/blockchain-services/emernvs/

[57] Emercoin Wallet download page, https://emercoin.com/en/for-coinholders#download

[58] Light Ethereum Subprotocol, https://github.com/ethereum/devp2p/blob/master/caps/les.md

[59] OpenPGP, https://www.openpgp.org/

[60] Trusted Computing Group, TPM 2.0 Keys for Device Identity and Attestation (1.00, Revision 2), 2020, https://trustedcomputinggroup.org/wp-content/uploads/TCG_IWG_DevID_v1r2_02dec2020.pdf

[61] Trusted Computing Group, TCG Feature API (FAPI) Documentation, https://trustedcomputinggroup.org/resource/tss-fapi/

[62] Trusted Computing Group, TCG TSS 2.0 Enhanced System API (ESAPI) Specification, https://trustedcomputinggroup.org/resource/tcg-tss-2-0-enhanced-system-api-esapi-specification/

[63] Trusted Computing Group, TCG TSS 2.0 System Level API (SAPI) Specification, https://trustedcomputinggroup.org/resource/tcg-tss-2-0-system-level-api-sapi-specification/

[64] Trusted Computing Group, TCG TSS 2.0 Marshaling/Unmarshaling API Specification, https://trustedcomputinggroup.org/resource/tcg-tss-2-0-marshalingunmarshaling-api-specification/

[65] Trusted Computing Group, TCG TSS 2.0 TPM Command Transmission Interface (TCTI) API Specification, https://trustedcomputinggroup.org/resource/tss-tcti-specification/

[66] Trusted Computing Group, TCG TSS 2.0 Overview and Common Structures Specification, https://trustedcomputinggroup.org/wp-content/uploads/TCG_TSS_Overview_Common_Structures_v0.9_r03_published.pdf

[67] Trusted Computing Group, TCG EK Credential Profile, https://www.trustedcomputinggroup.org/wp-content/uploads/Credential_Profile_EK_V2.0_R14_published.pdf

[68] Trusted Computing Group, TPM 2.0 Library Chapter 1: Architecture, https://trustedcomputinggroup.org/resource/tpm-library-specification/

[69] Emercoin API, https://emercoin.com/en/documentation/emercoin-api/

[70] OSS implementation of the TCG TPM2 Software Stack (TSS2), https://github.com/tpm2-software/tpm2-tss
[71] pyca/cryptography, https://cryptography.io/en/latest/
[72] mbedTLS library, https://github.com/Mbed-TLS/mbedtls
[73] OpenSSL fork with Emercoin-based certificate verification, Jongho Won, https://github.com/JonghoWon/openssl
[74] OpenSSL, https://github.com/openssl/openssl
[75] CURL library, https://curl.se/
[76] jsmn library, https://github.com/zserge/jsmn
[77] tpm2-software/tpm2-pytss: Python bindings for TSS, https://github.com/tpm2-software/tpm2-pytss
[78] python-bitcoinrpc, https://github.com/jgarzik/python-bitcoinrpc
[79] PopOS by System76, https://pop.system76.com/
[80] OPTIGA TPM & OPTIGA Trust certificates, https://www.infineon.com/cms/en/product/promopages/optiga_tpm_certificates/
[81] Oracle VM VirtualBox, https://www.virtualbox.org/
[82] IBM's Software TPM 2.0, https://sourceforge.net/projects/ibmswtpm2/
[83] TPM2 Access Broker and Resource Manager, https://github.com/tpm2-software/tpm2-abrmd
[84] IETF, RFC-6961: Transport Layer Security (TLS) Certificate Status Version 2 Extension, June 2013 https://www.ietf.org/rfc/rfc6961.txt
[85] tpm2-tools, https://github.com/tpm2-software/tpm2-tools