energies

Article

Blockchain-Based Services Implemented in a Microservices Architecture Using a Trusted Platform Module Applied to Electric Vehicle Charging Stations

Antonio J. Cabrera-Gutiérrez [1,2], Encarnación Castillo [2], Antonio Escobar-Molero [1], Juan Cruz-Cozar [1,2], Diego P. Morales [2,*] and Luis Parrilla [2]

1 Infineon Technologies AG, Am Campeon 1–15, 85579 Neubiberg, Germany
2 Department of Electronics and Computer Technology, University of Granada, Avda. de Fuente Nueva s/n, 18071 Granada, Spain
* Correspondence: diegopm@ugr.es

Abstract: Microservice architectures exploit container-based virtualized services, which rarely use hardware-based cryptography. A trusted platform module (TPM) offers a hardware root for trust in services that makes use of cryptographic operations. The virtualization of this hardware module offers high usability for other types of service that require TPM functionalities. This paper proposes the design of TPM virtualization in a container. To ensure integrity, different mechanisms, such as attestation and sealing, have been developed for the binaries and libraries stored in the container volumes. Through a REST API, the container offers the functionalities of a TPM, such as key generation and signing. To prevent unauthorized access to the container, this article proposes an authentication mechanism based on tokens issued by the Cognito Amazon Web Service. As a proof of concept and applicability in industry, a use case for electric vehicle charging stations using a microservice-based architecture is proposed. Using the EOS.IO blockchain to maintain a copy of the data, the virtualized TPM microservice provides the cryptographic operations necessary for blockchain transactions. Through a two-factor authentication mechanism, users can access the data. This scenario shows the potential of using blockchain technologies in microservice-based architectures, where microservices such as the virtualized TPM fill a security gap in these architectures.

Keywords: blockchain; containers; electrical vehicles; EOS.IO; hardware security modules; microservice architecture; trusted platform module; virtualized environment

Citation: Cabrera-Gutiérrez, A.J.; Castillo, E.; Escobar-Molero, A.; Cruz-Cozar, J.; Morales, D.P.; Parrilla, L. Blockchain-Based Services Implemented in Microservices Architecture Using TRUSTED Platform Module Applied to Electric Vehicles Charging Stations. *Energies* **2023**, *16*, 4285. https://doi.org/10.3390/en16114285

Academic Editor: Joao Ferreira

Received: 14 April 2023
Revised: 13 May 2023
Accepted: 21 May 2023
Published: 24 May 2023

1. Introduction

With the rise of Industrial Internet of Things (IIoT) environments, new computing and networking paradigms have emerged. This is due, among other things, to the fact that new aspects of security have started to be considered. In traditional architectures, both client–server and publish–subscribe paradigms have been dominant [1,2]. The problem with such architectures is that there are centralized entities that play an important role within the network. A typical architecture example that implements a centralized client–server architecture is the publish–subscribe message queue telemetry transport (MQTT) [3] protocol. In this model, some entities publish certain topics (publishers), and other entities that subscribe to the same topics obtain the published information (subscribers). Both publishers and subscribers are managed by a centralized entity called a broker. The broker generates a point of vulnerability in the system, since denial of service (DoS) [4] attacks can have pernicious effects on the rest of the architecture. Being centralized entities, in which all requests are made against this entity, this type of attack can collapse it and affect its availability; thus, as a consequence, the whole system is affected.

In addition, these protocols implement asymmetric cryptography to encrypt communications [5] and to prevent man-in-the-middle attacks [6]. This asymmetric cryptography is based on certificates issued by a certification authority (CA). This entity is part of what is known as a public key infrastructure (PKI) [7]. PKIs are vulnerable, among other reasons, because of the centralization of entities that play an important role, such as CAs. Attacks on these entities can lead to the loss of credentials and the potential impersonation of a user by an attacker in the system [8].

These are the main reasons why technologies such as blockchain have made a strong entry into the IIoT world in recent times. A blockchain completely breaks the established client–server paradigm, replacing it with a peer-to-peer paradigm, decentralizing the network and preventing the data it contains from being modified by third parties.

Blockchain offers trust and decentralization as its main characteristics. Based on decentralized ledger technologies (DLT) [9], blockchain distributes a database (ledger) among the entities that belong to the network, improving the security by storing a copy of the database in each entity. Furthermore, a blockchain offers privacy, since the hashes used by it make it impossible to identify the data referring to a specific user and assures data protection against other parties, by implementing access control policies. Immutability and traceability are also important features in blockchain applications, which are achieved through the transactions performed in the ledger by the different users. All the transactions are stored in the ledger, and it can never be modified. Transactions change the content of the decentralized database, so they must be approved by the entities that make up the blockchain, according to the roles established within it. In this way, any change is notified to the different entities, keeping the integrity of the database [10]. Finally, it is worth mentioning the ability of some blockchain applications to run use cases. These applications are executed through smart contracts that run on the ledger and perform actions predefined by the users who make up the blockchain [11]. Thanks to this feature, in recent times, blockchain networks have started to establish themselves in new fields of application, such as IIoT [12], healthcare [13], energy [14], business [15], financial [16], and others [17–19].

The benefits of blockchain lie, in addition to decentralization, in the use of cryptography in the transactions carried out. All transactions are signed and subsequently verified in the blockchain. All cryptographic materials (private keys, public keys, certificates) used to carry out the operations become of vital importance, since this is where many of the properties of the blockchain lie. If this cryptographic material is stolen or modified, this could have severe consequences for the integrity of the network [20].

One of the ways to effectively protect this cryptographic material is the use of hardware security modules (HSMs) [21], which securely store the keys used in a blockchain. In this way, as Figure 1 shows, a blockchain client that has an HSM integrated can sign transactions using the private keys securely stored in the HSM, since these keys can never be extracted.

HSMs fulfill a very important role as the root of trust in systems where they are applied. In fact, HSMs are used in different fields; for example, in PKIs [22], network protocols [23], database encryption [24], digital signatures [25], and cloud services [26].

In the case of cloud services, there are different applications running on the same hardware infrastructure (infrastructure as a service (IaaS)) where the HSMs are usually located. These HSMs are virtualized so that different software can make use of them, enabling providers to offer secure services to users, both at platform (platform as a service (PaaS)) and software level (software as a service (SaaS)) [27]. This virtualization of HSMs does not only occur in cloud services, but in any system where there is an orchestration of services that make use of the same resources. This is the case, for example, in an IoT gateway where different services are deployed. In fact, technologies such as edge [28] and fog computing [29] have made process virtualization a more common concept.

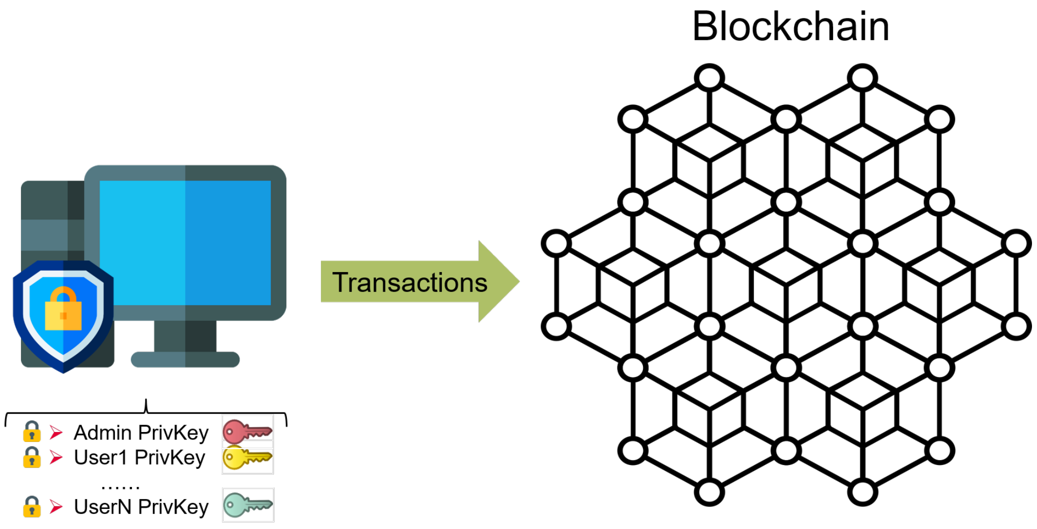

Figure 1. Blockchain with HSM.

These services can simply be placed in charge of a certain function in the system (e.g., maintain a database or run a web server). Virtual processes have given rise to microservice-based architectures [30], where the services are orchestrated in such a way that, by interacting with each other, they are able to carry out the correct functionality. This type of architecture offers a much more efficient maintenance, deployment, and reliability than traditional architectures [31].

This article proposes an HSM virtualization microservice that makes use of a special type of HSM, a trusted platform module (TPM). The virtualization of the TPM is carried out using a docker container that has a REST API to access it, offering high level functionality. In this container, different microservices delegate cryptographic operations, attesting to the status of the container beforehand. These microservices are authenticated in the TPM container utilizing Cognito Amazon Web Services (AWS) [32], in order to protect the API and establishing different permissions and roles when using the TPM. Thus, in this architecture where different microservices have to make use of cryptography, they can access the container through an authentication system that establishes different roles and permissions, which protects against unauthorized microservices. Finally, this article proposes an application of this design to an electrical vehicle (EV) charging stations use case, where the data are stored in an EOS.IO blockchain that makes use of the virtualized TPM. The main contribution over the state of the art is the virtualized TPM with security mechanisms (attestation and private key sealing) provided to preserve the integrity and the security of the sensitive cryptographic material of the container, as well as the inclusion of a mechanism based on AWS Cognito for access control.

The rest of this article is organized as follows: Section 2 describes the work previously performed regarding TPM virtualization. Section 3 presents the motivation and the importance of using HSMs in combination with blockchain. In Section 4, the proposed architecture design is described. Section 5 shows the application in an EV charging station use case. Finally, Section 6 summarizes the conclusions, emphasizing the benefits and applicability of this proposal.

2. Related Work

As a hardware resource, different cloud providers offer services where the TPM is virtualized. This virtualization is known as virtual TPM (vTPM) and is a software-based representation of a physical TPM [33]. The vTPM concept is oriented toward use in virtual machines (VMs) and it provides the functionality of a physical TPM to other VMs. In order to obtain this functionality, a vTPM is composed of a vTPM manager and different instances of the vTPM. These instances run in the VMs and implement the full trusted computing group (TCG) TPM2.0 specification. The vTPM manager creates different instances and multiplexes the requests from the VMs to the vTPM instances. The communication between the VMs and the vTPM is implemented using a driver model compound on the client side, running in the VM, and a server side, running in the vTPM. Figure 2 illustrates this architecture.

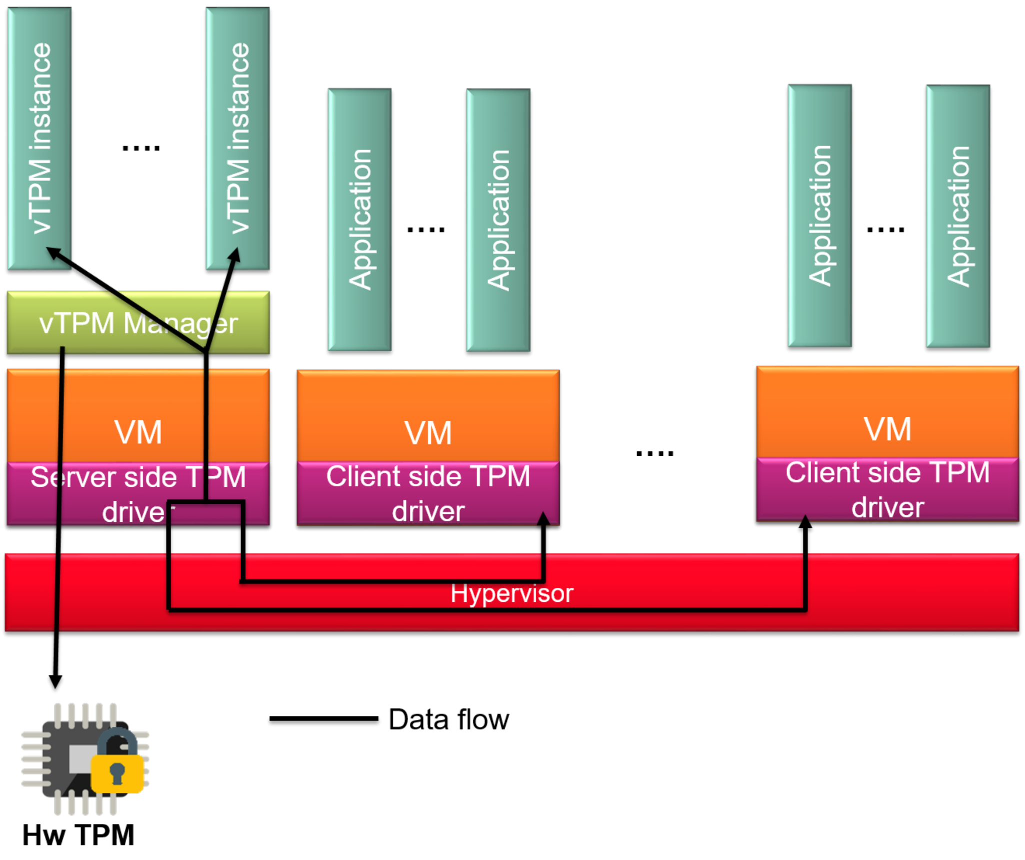

Figure 2. vTPM architecture.

As can be seen, this design requires implementing different drivers in each VM, as well as having one instance for each VM that uses the vTPM. This model does not fit properly in a microservice-based architecture, since this type of architecture rewards modulation of the implemented microservices and the independence of one from the other [34]. In fact, this type of virtualization, known as hypervisor-based virtualization, differs significantly from container-based virtualization, which is mostly used in microservice architectures, providing a much better performance than the aforementioned type [35,36]. Figure 3 shows the difference between the architectures.

In hypervisor-based virtualization, a hypervisor (virtual machine monitor) runs on the hardware, enabling different VMs which share the same hardware resources; in this way, it is possible to emulate different operating systems (OSs) within the same platform. In container-based virtualization, the container engine utilizes kernel features of the host OS to create isolated environments for applications; hence, applications built on this engine share the hardware resources and the same OS, making them much lighter than those based on a hypervisor. Each of these technologies have their own advantages and disadvantages. Container-based virtualization has lower performance overheads. As containers are lighter weight than VMs, it is faster to deploy and boot a container compared to a guest OS. On the other hand, container-based virtualization has less flexibility, in the way that it only hosts the same OS that the host platform is using [37].

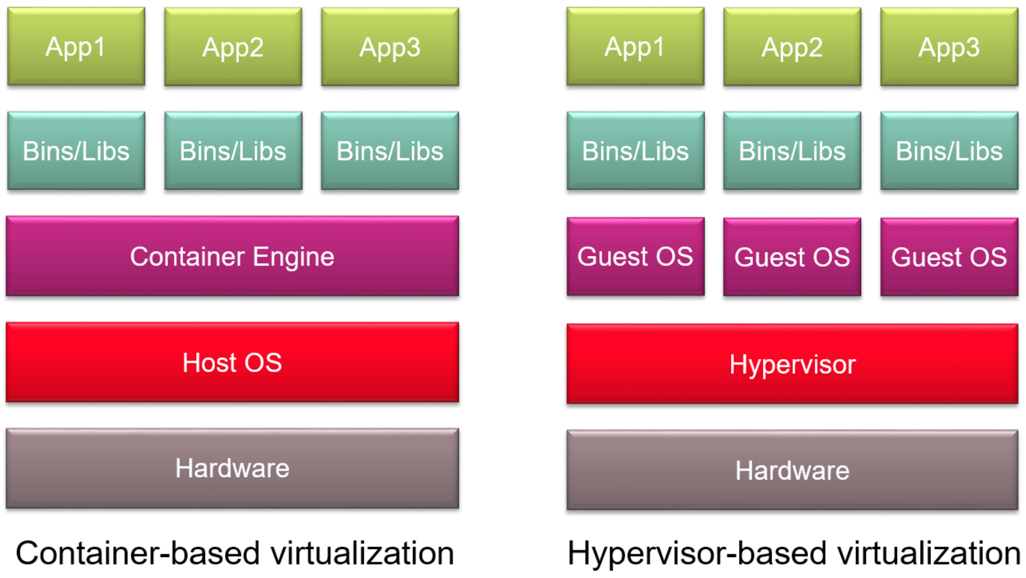

Figure 3. Hypervisor-based virtualization vs. container-based virtualization.

However, as said before, container-based virtualization brings many more benefits than hypervisor-based virtualization for microservice architectures, since they can deploy software applications as packets of modular and independent microservices, being much lighter, easier to maintain, and achieving better performance [38]. Each microservice is a container that has its own programming language, data storage, and communication mechanisms. Container-based virtualization has had a huge impact on microservice architectures, especially in IIoT applications that perform complex and new functionalities. In this kind of system, a flexible and agile deployment, as well as an efficiency of resources, is needed. In addition, these systems tend to be very volatile, so good scalability is also an important factor. In Figure 4, an example of a microservice architecture for IoT is shown. As is shown, the different microservices interact, in order to fulfill their purposes (collect data from different sensors, store them in a database, and offer different analysis and visualization techniques to external users). Each microservice is in charge of a task and communicates with the others independently. Microservices can range from managing a database to training machine learning algorithms.

However, it is not very common in the literature to find systems based on microservices that offer security aspects for IoT. In [39], the authors describe a security approach for developing microservice-based IoT systems. This approach combined microservice patterns, APIs, distribution of microservices, and access control policies. An attribute-based encryption (ABE) scheme was proposed in order to deploy identity management, authentication, and policy control in microservices.

The work proposed in [40], presented an IoT framework that incorporated security microservices for role management, authentication, access control, and identify governance.

In [41], the authors proposed a smart surveillance system based on a microservice architecture and blockchain technology. The communication between the mircroservices was encrypted using advanced encryption standard (AES) [42] and Rivest–Shamir–Adleman (RSA) [43] cryptography algorithms. The blockchain network was used to ensure the decentralization of the data exchanged by the different microservices, as well as to introduce access policies to this data through smart contracts.

The work described in [44] proposed deploying different blockchain services on edge nodes. These services were related to blockchain operations through execution of consensus algorithms, generation of new blocks, and performing mining tasks.

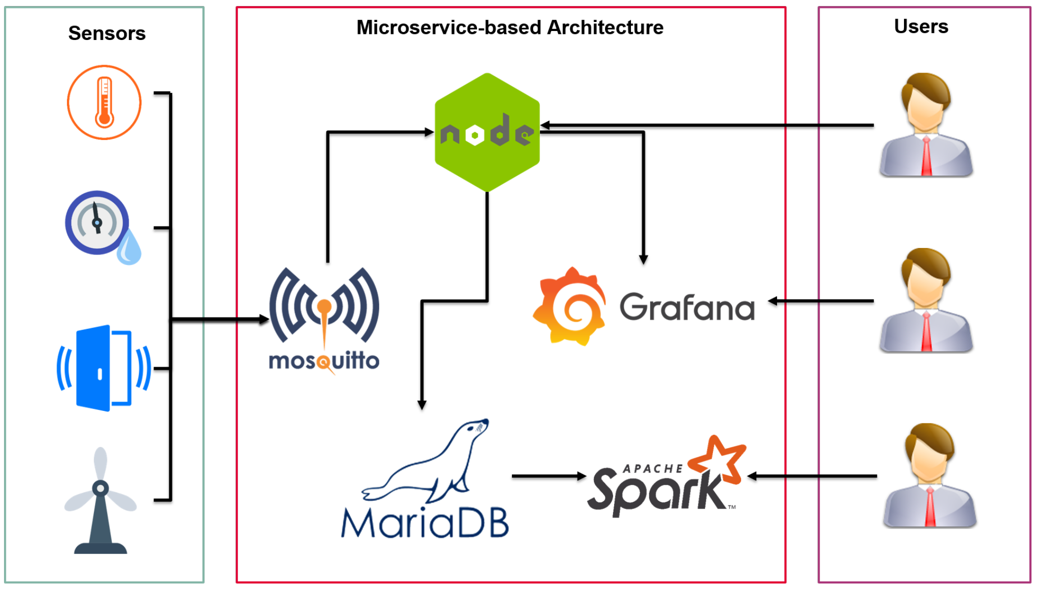

Figure 4. Example of microservice-based architecture.

In [45], the authors proposed a secure edge computing management based on microservices, with a gateway that is responsible for managing devices, data, users, and configuration security. The gateway incorporates an API used to filter all incoming requests, in order to protect the microservices running on the edge.

In [46], the security solution was a machine learning based approach. In this approach, the microservices-based model identifies strange behaviors, observes communication packets, and intercepts malicious traffic that may potentially damage the system.

The work proposed in [47] describes a solution using a traffic monitor that runs on IoT nodes, routers, etc. in the network. The traffic monitor intercepts the packages between the services and identifies the communication patterns of each data transmission. In this way, the model can classify the communication as normal or anomalous.

All of the above works offer security between microservices or prevent malicious attacks on them through different mechanisms or policies. However, none of them offer security at the cryptographic level, i.e., they do not provide any microservices in which different applications use cryptographic keys, signatures, or certificate storage through a microservice that manages this. In this article, a microservice is proposed that can be used by others to provide keys, signatures, and other cryptographic operations related to a TPM.

3. Trusted Platform Modules in Combination with Microservice-Based Architectures

In microservice architectures, hardware security has not been addressed to date. In this type of architecture, HSMs are absent in most of the literature. However, providing a root of trust in the system built on top of HSMs is of vital importance.

HSMs offer a hardware root of trust, since they generate cryptographic keys using high-entropy random number generators, and protect them through secure storage (keys stored internally are never extracted). They offer cryptographic algorithms implemented in hardware for encryption/decryption or signature generation and verification [48].

Within HSMs, there is a special subgroup formed by TPMs, which differ from other HSMs, in that the methods implemented in them are standardized by the TCG [49], as well as offering higher level mechanisms than a common HSM [50]. TPMs fit better in virtualized architectures, since they offer mechanisms such as secure boot [51] or remote attestation [52], which ensure system integrity at startup and during the execution period, detecting any software modifications. Moreover, this is ideal for use on systems with OS support, which offer virtualization support as has already discussed in the previous section introducing vTPM [33].

The problem, as mentioned above, is that vTPM does not perfectly fit into the philosophy of microservice architectures as these are implemented in container-based virtualization. To solve this problem, a few approaches have been presented in the literature [53–55], which were analyzed at the time of the designing of the proposal presented in this article.

Figure 5 presents a first approach to combining TPMs with microservices [33]. In this approach, each microservice making use of the TPM has a vTPM in the container. Therefore, in this approach, the size of the containers may grow considerably, becoming more heavy and less flexible than other proposals.

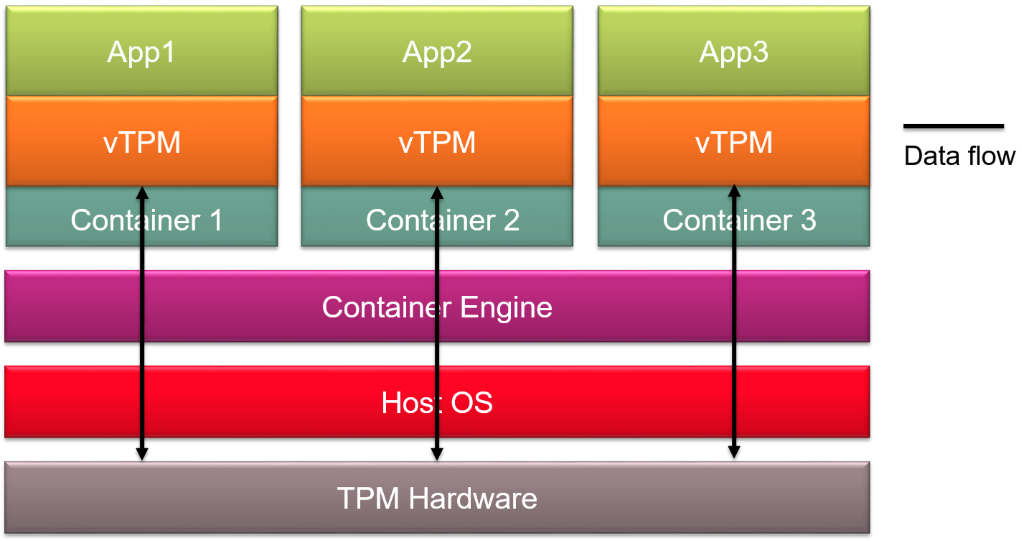

Figure 5. Approach with vTPM inside each container.

To make this approach lighter, one solution adopted is to move the vTPM to the operating system level [54]. In this way, the TPM is available to different containers by assigning a vTPM instance to a new container. The container manager asks the host OS kernel to create a new vTPM instance and then assigns it to the new container. Figure 6 shows this scheme. In this case, the kernel must be trusted, because an attack on it can lead to information leakage by the vTPM. To ensure a kernel is trusted, the solution taken must extend the root of trust from the TPM to the kernel, through mechanisms such as remote attestation or secure boot. Thus, if the kernel is considered secure, all the vTPMs attached to it are secure as well [56]. The problem with this solution, when applied to microservice architectures, is the lack of flexibility when deploying this system on any platform. As is well known, microservice systems create virtualized environments that are independent of the OS and the hardware platform implemented in the underlying layers. If this proposed solution uses kernel modules/resources for different microservices, these microservices will be more difficult to port and deploy with different OS architectures.

The other approach [55] that we will consider consists of dedicating a container exclusively to the vTPM, i.e., creating an exclusive microservice in which different containers that require the use of the functionalities provided by the vTPM must communicate with it.In this way, an appropriate flexibility and portability related microservice system is achieved. Figure 7 illustrates this solution. The difficulty of this solution is to ensure the reliability of the container carrying the vTPM. In this sense, it is necessary to make use of the attestation mechanisms offered by the TPM, both in the deployment of the container and during its use. It must also be protected from malicious microservices that try to access the vTPM with an authentication system. These aspects will be discussed in detail in the next section.

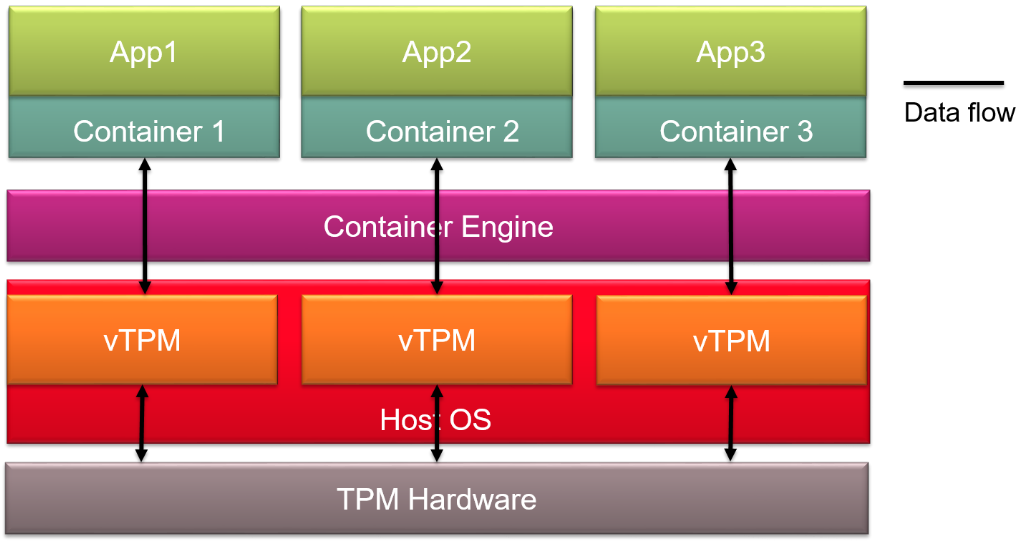

Figure 6. Approach with vTPM in the OS kernel.

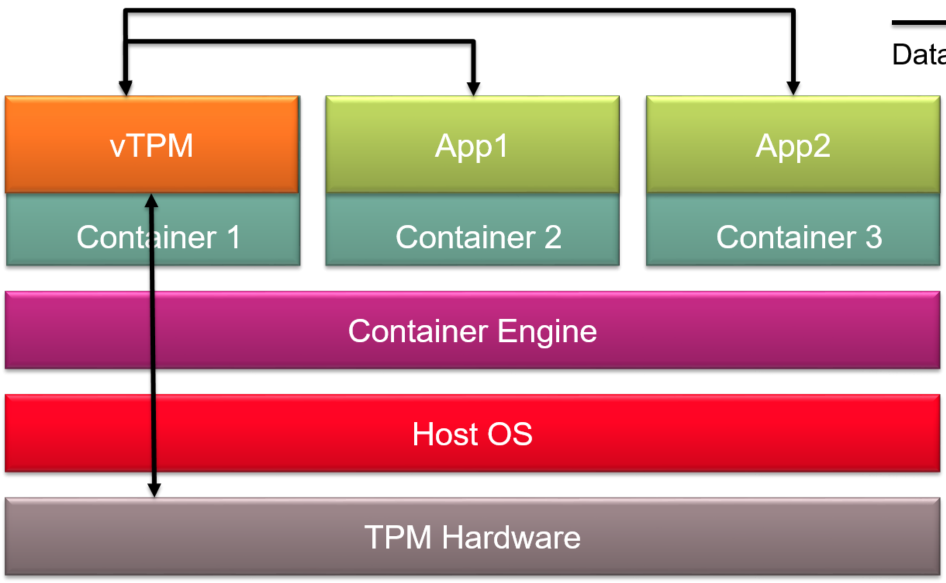

Figure 7. Approach with vTPM in a dedicated container.

Note that in this proposal any microservice system deploying a vTPM microservice will be able to realize the functionalities offered by the TPM. In this sense, many microservices implement cryptographic security in microservice-based systems; from an MQTT server to a blockchain client, going through database services, using the cryptographic capabilities to sign transactions, exchange encrypted messages using symmetric cryptography, or encrypt and decrypt files.

However, if one digs deeper into the technical aspects, not all microservices may be compatible with this vTPM microservice. TPMs are able to generate secure keys because they incorporate high-entropy true random number generators (TRNGs). These are based on an external physical phenomena, making it almost impossible for a key generated with a TRNG to be replicated. These generated keys are used to implement public-key cryptosystems based on elliptic curves. There are a wide variety of elliptic cryptographic curves (ECCs), usually supported by TPMs such as the NIST standard [57] prime256v1 or secp256k1, which is the one used in bitcoin [58] and Ethereum [59]. Since the cryptographic operations used in microservices are performed in the TPM, these two technologies must support the same cryptographic curves and algorithms, making this a mandatory property when integrating these two elements.

Instead, this type of solution allows for the possibility of abstracting the software-level functionality of the TPM. The software implemented by the microservice and making use of the TPM is independent and does not have to comply with any standard to make use of the TPM. This is a differentiating element from the traditional use of TPM by external applications, since there are software standards that are used to manage a TPM as if it were a hardware token, as in the case of PKCS11 [60]. This standard defines an API in order to interact with the TPM through external applications.

In this way, complete container-based virtualized systems can be built making use of the vTPM microservice, being flexible and easily portable to any type of platform. Based on this approach, we will proceed to explain the architecture design proposed in this article.

4. Architecture Design

The design proposed in this article is based on the third approach discussed above, in which the TPM is virtualized in a container, and where the other containers, whether they are on the same hardware platform or on a different one, can access it through an interface and a communications protocol. In this way, an architecture based on microservices can make use of cryptographic material from the TPM, creating a root of trust in this type of architecture. The proposed design improves on the third approach by adding important security features such as private key sealing, attestation, and access control to the vTPM, at the cost of a negligible loss of performance.

Architectures that implement blockchain microservices use cryptography with great frequency, since these applications base their reliability on the cryptographic operations they perform. This is the case with transactions; the messages sent to the blockchain and signed by the sender. This signature is carried out through a private key that is usually generated and stored in software. Through the use of this microservice, the keys are generated inside the TPM and all cryptographic operations are performed inside it. Thus, the external microservice that makes use of it will only receive the results of these cryptographic operations. In this way, the keys are securely stored internally, as this prevents physical attacks such as probing [61] or side channel attacks, timing attacks [62], or fault induction techniques [63]. In this way, and thanks to the TPM, hardware attacks are mitigated; however, the architecture proposed in this article involves different aspects that must be addressed in terms of security.

The vTPM container must offer security mechanisms that prevent an attacker from modifying the container and injecting malicious code inside it, and the microservices that make use of the vTPM container must follow access policies that establish different permissions and authorizations when performing functionalities within the vTPM. For example, a blockchain client may have permission to perform cryptographic signatures in the TPM, but may not have permission to generate cryptographic keys. It is logical that this operation is available to other microservices with more authority, such as a blockchain administrator.

In order to solve this issue, the vTPM microservice implements a secure attestation functionality for the microservices that make requests, so that when a microservice is going to make a request to it, the microservice first performs an attestation to check that the vTPM is correct and has not been modified. In addition to this, a secure deployment must be performed to prevent any process from interfering in the deployment and causing its failure or any malicious code injection during the deployment. In relation to the protection of the microservice from the use of other microservices without permissions, an authentication service based on Cognito AWS [32] is used, which, using different policies, establishes different access rights to other microservices that have been previously registered with it.

In the following sections these two aspects will be explored, but first we will proceed to explaining the proposed virtualization of the vTPM in a container.

4.1. Virtualization

The vTPM built in this proposal is inside a container and must communicate with others through external APIs. Typically, TPMs are the hardware base of different applications that are built on top of it. These applications use different APIs implemented in the middleware in order to use the TPM functionalities. As was said before, PKCS11 is one of these APIs.

The software used in the TPM is standardized by the TCG and is composed of diverse software: from the lower functionality used to manage the TPM resources, to the higher level functionalities through Python libraries. Figure 8 illustrates the different software components enabling the use of the TPM, which are

- TPM2 Access Broker and Resource Management Daemon (ABRMD) [64]: This software module is formed of two different parts: the access broker and the resource management. The access broker manages the synchronization between the processes that use TPM simultaneously, while the resource manager manages the TPM context in a similar way to the OS memory manager. TPMs generally have very limited memory, thus TPM data (objects, sessions and sequences) need to be swapped from the TPM to the memory, to allow the TPM commands to be executed. ABRMD works on the Unix hardware device representation of the TPM (*/dev/tpm0* and */dev/tpmrm0*).
- TPM2 Software Stack (TSS) [65]: The TSS is a compound of the following layers, from the highest level of abstraction to the lowest: feature API (FAPI), enhanced system API (ESAPI), system API (SAPI) and the TPM command transmission interface (TCTI). All these APIs offer TPM functionalities to applications written on it. These APIs are implemented in C and C++, which means a low portability and compatibility with external applications, which is why there are different libraries that are built on top of the TSS.
- TPM2 Tools [66]: This provides a set of bash binaries which interact with the different functionalities of the TSS.
- TPM2 Python [67]: This provides a Python API in order to access the TSS.
- TPM2 OpenSSL [68,69]: This implements a provider and an engine for OpenSSL v3.0, making the TPM accessible for OpenSSL API and command line tools.
- TPM2 PKCS11 [70]: This offers a PKCS11-based API, in order to make the TPM accessible as a hardware token.

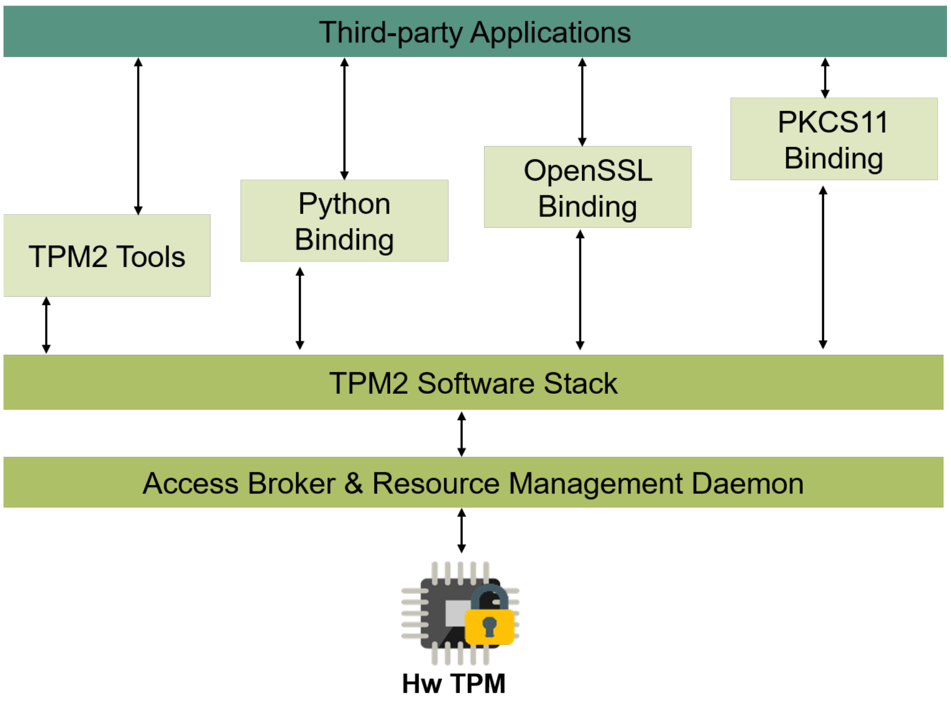

Figure 8. TPM software.

All of this software is essential to make the TPM work; however, many of the bindings mentioned are redundant for each other and can be bypassed for certain applications. For example, an application built in Python will use the libraries provided in this language for the use of the TPM, and it will not be necessary to include the command line tools. To avoid incompatibilities between applications built on top of this type of software and to make it more flexible, a REST API has been created on top of the software described above. This API only uses TPM2-Tools and OpenSSL bindings in order to interact with the TPM. In this way, the virtualized software inside the container is reduced and the TPM can be accessed by external microservices that make use of the cryptographic operations offered by the TPM through the REST API. Figure 9 shows the complete vTPM architecture, where the main component is the docker container, which has all the software modules required.

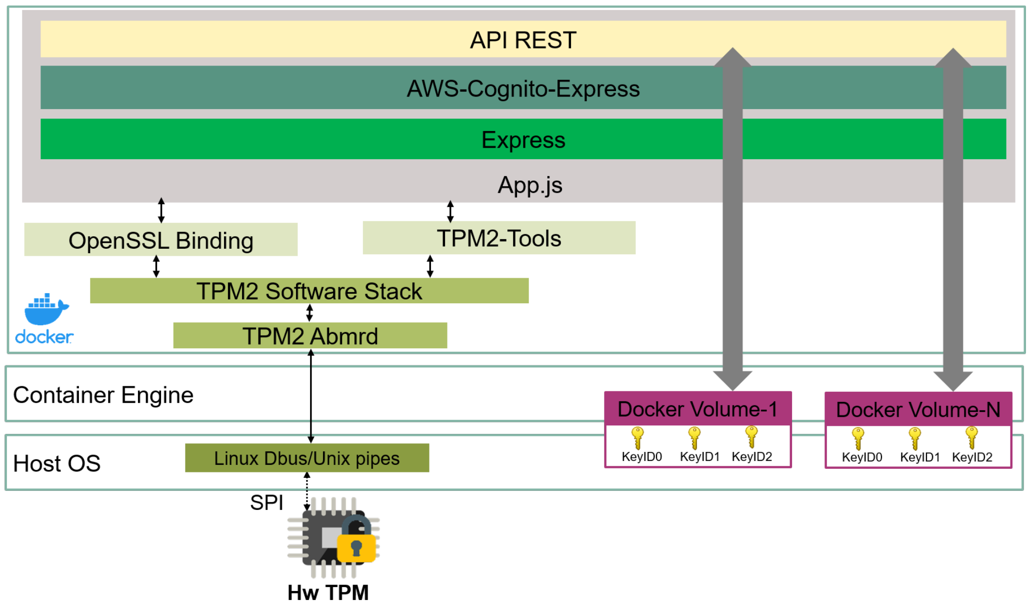

Figure 9. vTPM architecture.

The REST API is an express server built with NodeJS. This REST API implements an authentication mechanism through Cognito AWS, which will be discussed in Section 4.3. This container uses volumes to store the identifiers of the private keys created through the API, and depending on the different permissions and roles established through Cognito AWS, the keys will be created in different volumes. These identifiers are used to access the private keys that are stored inside the TPM. The application built in NodeJS is in charge of calling the underlying software (TPM2-Tools and OpenSSL bindings), which in turn communicates with the TPM through the software stack discussed above.

Therefore, the REST API is responsible for providing functionality to the container from the point of view of an external application, so that this API has different entry points, in which each performs one or more operations within the TPM. The entries available in the API are the following:

- Create key: This operation generates a private key. The name of the key is generated randomly following a binary large object (BLOB) format. This BLOB is given as a return message of the operation. The private key is stored internally in the TPM, but the identifier named with the BLOB is stored in a volume.
- Signature generation: The signature process inside the TPM is performed using this entry point. The input parameters are the BLOB identifying the private key that will make the signature and the hash that will be signed.
- Export public key: This entry point returns the public key associated with a private key identified with a BLOB.

- Signature verification: This operation verifies the signature using the public key. The entry point receives the signature, the hash, and the BLOB identifying the private key. The return value is a boolean indicating the success of the verification.
- Get random: Returns a random value using the TRNG into the TPM. The input parameter is the length in bytes of the random value requested.
- Encrypt: This operation encrypts using the private or public key, depending on the algorithm chosen in the input parameters, for a string given.The return value is the encrypted string.
- Decrypt: This operation does the inverse process to the previous operation, decrypting a given string in the input parameters.
- Hash: This computes a hash, following the algorithm given in the REST function input parameters. The return value is the hash computed in hexadecimal format.

In addition to these operations, there are other mechanisms that perform attestation and integrity validation of the microservice, since all these vTPM functionalities would be useless if an attacker intervened and accessed the microservice, modifying it and redirecting this information to another microservice and impersonating it, or even creating fakes BLOBs for the keys. These mechanisms will be explained in the next section.

4.2. Attestation and Data Sealing of the vTPM Microservice

To ensure the correct operation of the deployed microservice, different checks of the microservice status must be performed. These checks are the most important security aspects in the container, since the trustworthiness of the operations executed in the vTPM microservice may collapse due to an attack on the integrity of the container and, therefore, on the TPM functionalities.

The underlying idea is to ensure that the microservice running inside the container always provides the results it should, ensuring the immutability and integrity of the container, and transmitting that confidence to the client microservices that use that microservice. The first step to achieving this is to obtain a secure deployment on the platform where you want to make use of the microservice. To do this, it must be ensured that the underlying hardware and software do not contain any foreign elements that could be exploited in the future as a security flaw. In this sense, part of the responsibility lies with the hardware resource provider (PaaS), since if there is malicious middleware or a backdoor in any hardware component, all the hardware security mechanisms that are built at higher layers will be useless.

The next step is the deployment and the integration of the platform (continuous integration and continuous development (CI/CD)). In this regard, there are tools that check the security of the container and perform tests on it. They also check the containers for known vulnerabilities in real time and warn about them so that they can be fixed immediately [71]. This could even be deployed through a secure script, in which it is launched through checks of an external TPM that verifies that the script is correct and has not been modified by any external processes. In addition, a check of the system on which the script will be launched can be performed, as well as the different binaries used for the deployment. However, these mechanisms are external to the inner workings of the vTPM microservice and are therefore not the subject of this article's in-depth study; each vendor is responsible for offering their own mechanisms, as exist for traditional containers with microservices that are not necessarily dedicated to security.

What is of interest in this article, and where the fundamentals of the security related operation of this container are based, is the checking of the vTPM during the execution of the container. During this time, a possible lack of integrity of this container is unavoidable, some OS or even remote process can obtain access to this container through a backdoor or a software bug. The container software cannot be shielded, since, as a container, it is running in memory zones within a hardware platform where there is an OS and different containers or external processes. For this reason, the information of the vTPM must be restricted and can only be released if an integrity check has previously been performed on the container, and if this check is valid. In contrast to the container data, this information can be shielded because it is stored inside the vTPM. In this sense, when an external microservice makes a request to the vTPM microservice, it must "unlock" this information, so that it can be sent to the external microservice.

TPM2.0 provides mechanisms to carry out this functionality. Moreover, during the execution time, it can perform measurements of different elements of the operating system and the rest of the software, to check the status of these components. These measurements are stored in special TPM registers called platform configuration registers (PCRs) [51]. Depending on the value of these registers, the TPM will unlock the information required by the client microservice (the generation of a signature or the use of a private key). The PCRs are used both for the request that a microservice makes when using the TPM and for unsealing the data contained in the TPM measuring the state of the container during execution. The first is known as attestation of the container by the TPM, and the second is known as the sealing process.

4.2.1. Container Attestation

The objective of the PCRs is to measure the software state of the platform. TPM2.0 has 24 PCRs, each of which has a predefined value at the reset state. These PCR values change after performing different measurements of the OS kernel and file system. The operation that makes these values change is called the extension and the resulting value is obtained by concatenating the incoming data-digest with the current value at the PCR index, hashing the concatenated data, and replacing the PCR index value.

To check the state of the container, several of the PCRs are used by extending them with hashes from different files in the container, e.g., binaries installed inside the container or from the REST API code.

Figure 10 shows the process that allows a client container to perform the attestation and subsequent verification. If this verification is satisfactory, the client container will perform operations that it deems appropriate on this container, such as the creation of keys or obtaining random numbers. In this way, by making sure that the state of the vTPM container is correct, the client knows that the operations performed on it are reliable.

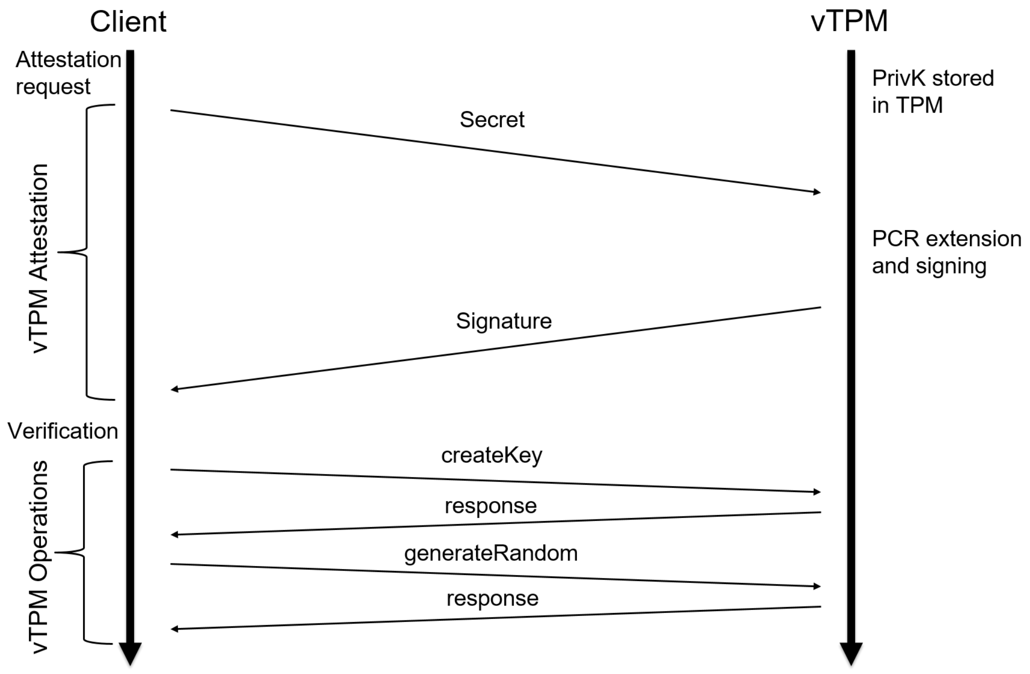

Figure 10. Container attestation mechanism.

Going deeper into the mechanism of attestation, the client sends a secret to the vTPM, this being secret randomly generated in the client and avoiding reply attacks [72]. The vTPM provides the hashes to different files in the container and extends the PCR values chosen by the vTPM, which are indicted in the vTPM policies. All the PCR values used in the operation are hashed following a combination predefined by the TPM, and this results in a single data hash. This final hash is signed using an attestation identity key (AIK) and the result is sent back to the container client, which verifies the signature and the hash. This verification process depends on the TPM provider, which owns the TPM and knows the state in which the hash returned by the vTPM is correct. In addition, it has a certificate issued from the AIK that serves to verify the signature. Therefore, when the microservice is deployed, the TPM provider must be responsible for providing both the attestation validation certificate and the correct hash for verification to the container client. This information may be made available through a microservice offered by the TPM provider or sent to client containers that make use of the vTPM microservice. PCRs are protected by different mechanisms offered internally by the TPM, implementing policies and session authorization [50] in order to make some modifications of these registers. Therefore, an attacker who wants to maliciously manipulate PCRs must first have obtained these credentials or authorization rights that are owner by the TPM itself.

Thus, this mechanism requires measurement of the container software prior to deployment. During this measurement, the TPM provider assumes that the container has not been modified, and its functionality is uncorrupted. When it is redeployed within a microservice architecture, all microservices that make use of it should have the original measurement available for verification. Thus, every time a microservice wants to access the vTPM microservice, it will know that the container status is the one certified by the provider. The same mechanism can be used by the owner of the hardware platform to check the integrity of the hardware, providing a proof that the platform is not corrupted and that the microservices can be deployed securely.

In conclusion, PCR values are reported in a signed attestation quote, permitting a external container to determine the vTPM container software's trust state through a valid authenticity and integrity proof being offered by the TPM provider.

4.2.2. Private Key Sealing

When a microservice uses a vTPM, the container stores the private keys it generates in a container volume. Actually, they are not the private keys, since these are stored inside the physical TPM, but they are files that work as identifiers of these keys; with access to these, it is possible to make use of the private key stored inside the TPM. These files are identified with a BLOB that is randomly generated at key creation. When a key is created, the API returns the BLOB to the client microservice.

However, storing this type of material in a container volume is dangerous, since a container volume is a file system that is shared with the host OS. If it is compromised by attackers, they can access these files and obtain these key identifiers. This is the reason why key identifiers stored within a volume must be protected. To solve this problem, this paper proposes to make use of a sealing process that incorporates the TPM. In this process, a file is sealed using the state of certain PCRs of the TPM. Once sealed, the file cannot be used until it is unsealed again by setting the indicated PCRs to the value they had at the time of sealing. The Figure 11 shows the sealing and unsealing processes.

The client sends a secret in a request to generate a private key. This secret, which is actually an 8-digit password, is in charge of establishing the combination of PCRs to be extended and in what order to perform the sealing. The secret is decoded, and the 8 digits are divided into four two-digit numbers, which are transformed into module 24, corresponding to one of the PCRs of the TPM. The order of extension is established, starting with the most significant number of the secret and ending with the least significant.

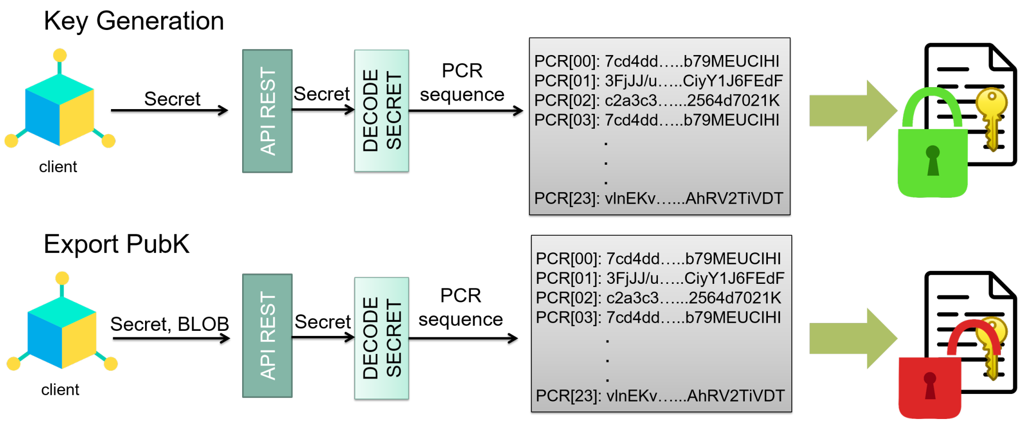
Key Generation
client
Secret
API REST
Secret
DECODE SECRET
PCR sequence
PCR[00]: 7cd4dd.....b79MEUCIHI
PCR[01]: 3FjJJ/u.....CiyY1J6FEdF
PCR[02]: c2a3c3......2564d7021K
PCR[03]: 7cd4dd.....b79MEUCIHI
PCR[23]: vlnEKv......AhRV2TiVDT
Export PubK
client
Secret, BLOB
API REST
Secret
DECODE SECRET
PCR sequence
PCR[00]: 7cd4dd.....b79MEUCIHI
PCR[01]: 3FjJJ/u.....CiyY1J6FEdF
PCR[02]: c2a3c3......2564d7021K
PCR[03]: 7cd4dd.....b79MEUCIHI
PCR[23]: vlnEKv......AhRV2TiVDT

Figure 11. Sealing and unsealing processes.

In this way, the private key is created and sealed with the contents of the PCRs. The content of these PCRs reflects the state of the container; thus, the key is sealed with a password imposed by the client and with the state of the container when the key is created.

The generation of the key operation creates a BLOB that is returned to the client; so that, in subsequent API calls that make use of that private key, the key can be referenced. For example, as illustrated in Figure 11, in the export public key operation the client must send the BLOB of the private key, which can be used to export the public key, and the secret, in order to unseal the private key. If the secret results in a correct extension of the PCRs and the PCRs contain the same value for the container as when it generated the key, the private key will be unsealed and the operation exporting the corresponding public key will be performed.

Thus, through the PCRs of the TPM, the integrity of the deployed container and its contents can be ensured; on the one hand, through measurements that are verified by the provider, and on the other hand, through passwords that serve to unlock sensitive material stored inside the container. In conclusion, for each operation that the client wants to perform in the vTPM microservice, and to ensure the reliability of the data obtained, first, the client must attest that the integrity is the one insured by the provider and, second, the private keys must be unlocked for use through a password that the client knows and that only works if the state of the container is the same as when that key was created.

Once the integrity of the container has been ensured during the use of the mechanisms explained above, it only remains to explain how to protect it from unauthorized microservices.

4.3. Cognito Amazon Web Service

Cognito AWS is a service that provides access management, authorization, and authentication of entities. Through user pools, registered containers are managed according to established policies, whereby different roles and permissions can be set. Thanks to this, secure and restricted access to the REST API exposed in the vTPM container is guaranteed.

The mechanism for ensuring this is shown in Figure 12. When a container wants to access the vTPM microservice, it must first register within an user pool. This process is performed using a username and password, and then, the container will be in the user pool and will have the rights to authenticate subsequently.

When authenticating, the container must use the username and password used in the registration. Once this has been done, Cognito AWS returns a token, which is used from now on, until the time of use expires, in all operations performed by the container on the vTPM microservice. When a request is made with this token, the vTPM microservice checks if it is valid and what permissions this token has associated with it (e.g., permission to create keys or to sign). Permissions are established through a series of associated attributes indicating the operations that the token is potentially capable of performing, such as signing, extracting a public key, or generating a random number.

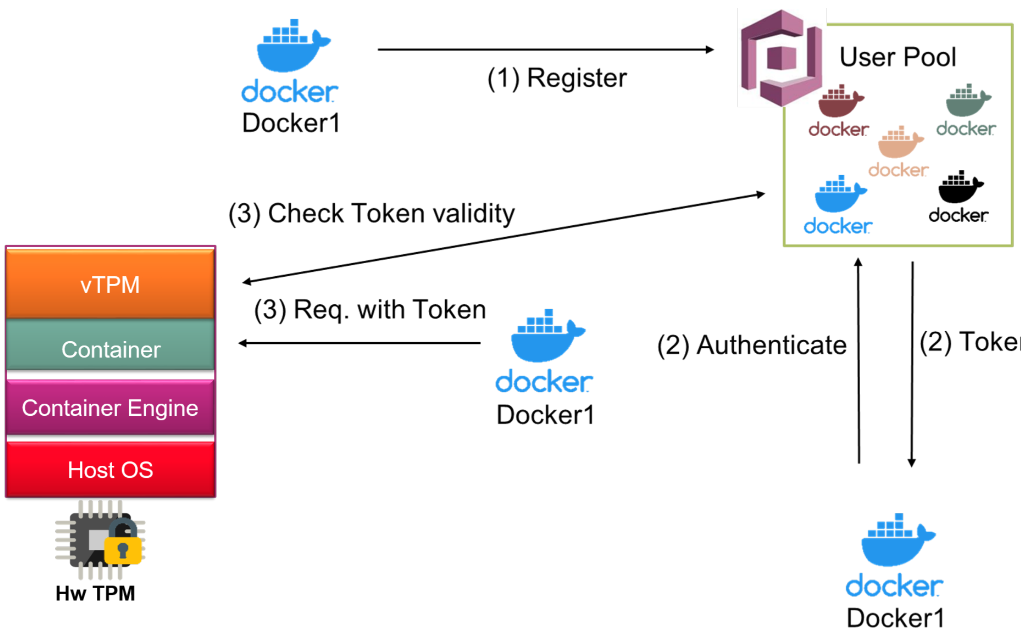

Figure 12. Authentication process with Cognito AWS.

If the token used in the request is invalid, either because it has expired or it is a fake token, the transaction will not be carried out; similarly, when the client container wants to carry out an operation for which it does not have the required permissions.

Each user pool has different permissions that are given according to the operations that the microservice performs. That is, if a blockchain client microservice needs to make use of signing, the user pool where it registers will only allow this operation; however, a blockchain administrator microservice must be able to generate private keys for clients, in this case, the user pool where it registers will allow the creation of keys in the vTPM microservice. It must be emphasized that the microservices registered in the user pool must be carried out through the pool administrator, who knows in advance that the registered microservice is reliable.

In addition, as each user pool has specific permissions associated with it, it makes sense that the clients that are registered in that user pool share the same volume in the container. Thus, a group of containers that perform blockchain operations, for example, share the same keys generated in the same volume, as these keys are used for the same purpose. It would not make sense, for example, for these keys to be shared with an MQTT client microservice. Thus, in this way, independent microservices do not share the same space, avoiding the possible mixing of private keys in the volumes.

In this way, through Cognito AWS, the API and the functionalities offered are protected against containers that are not authorized to use it, thus protecting it against possible attacks by malicious containers. In addition, setting permissions between different containers and separating the volumes used between different user pools provides better modularity of the functionalities offered, by separating the keys generated by the container between microservices.

Thanks to Cognito AWS and the attestation and sealing mechanisms of the private keys, a robust container that is secure against possible attacks on the system where it is deployed is achieved, ensuring correct functionality and detecting any unauthorized intrusions into the system.

Finally, a point to take into consideration is that the use of this type of architecture introduces a loss of performance related to the security mechanisms and the fact of using a virtualization platform; any virtualized system introduces a loss of performance in the runtime, and it is always faster to run a code closer to the hardware than in higher abstraction layers; introducing overheads in the system calls through the different layers of virtualization. The only way to mitigate this as much as possible is to optimize the code running in the container.

5. Electrical Vehicle Charging Station Use Case

The potential offered by this microservice concept is easily exploitable in different use cases; furthermore, taking advantage of the cryptographic operations that the container posses through API is ideal for use with blockchain applications, which make intensive use of cryptographic mechanisms: key generation, signing, or signature verification. There have been many implementations of blockchain in the energy field [73–75], showing the potential of this technology.

One of these cases is EV charging stations. EVs are becoming more common and charging stations are an expanding infrastructure, especially in urban areas. Thus, the connection between electric vehicles and blockchain technology is promising [76]. The microservice-based architecture proposed in this article in this case offers a charging station geolocation service, where all usage is collected in a decentralized blockchain database, preserving privacy and offering data integrity of the records. This system is shown in Figure 13. On the one hand, the information related to the EV charging stations is stored in the blockchain using a NodeJS API. The data collected are the geographical position of the charging station, the energy supplier of the station, the power it supplies, the type of vehicle that can be connected, and the kind of energy source from which the offered power is generated. This kind of data are stored in the registration process of the charging stations and are immutable until a blockchain transaction modifies them, e.g., if the type of energy changes from a renewable to a nonrenewable source. Through a charging station finder microservice, the user enters search criteria, which are the distance to the station, the estimated charging time, the money available, and the type of energy source. If the charging station ceases to be operational, it will send a revocation certificate to the blockchain, until it is operational again by re-registering within the network. User privacy is guaranteed through the blockchain, as it provides anonymity for the transactions made. Only the user identifier appears in these transactions, which is a cryptographic identifier based on a cryptographic hash function. With this information, the system offers the user different charging points that suit his preferences. In addition, the system has a history record microservice, which allows the user to query all the energy charges performed in the system. This microservice provides private information, such as a record of bank transactions, money spent, time spent, and charging stations used; thus, this must be protected through an authentication system based on Cognito AWS and a hardware token. Both this microservice and the blockchain client make use of this Cognito-based authentication system. As explained in Section 4.2, the blockchain client must be authenticated in Cognito AWS, in order to use the vTPM microservice. The blockchain client is in charge of sending information to the blockchain, as well as querying it. Both the information related to the EV charging stations and the information related to the charges made by the users are sent through the blockchain client to the blockchain network in the form of transactions previously signed by the vTPM microservice. This blockchain client microservice is an EOS.IO node.

In this way, it is possible to track the energy usage of a user, as well as all vehicles that have been charged at a given charging station. These records are immutable thanks to the blockchain. The system is deployed on a platform that has a TPM, which is virtualized through the vTPM microservice following the mechanisms discussed in Section 4.2.

In the following sections the two main parts of this systems will be explained: the blockchain client that makes use of the vTPM, and the hardware token authentication system with Cognito AWS (two-factor authentication).

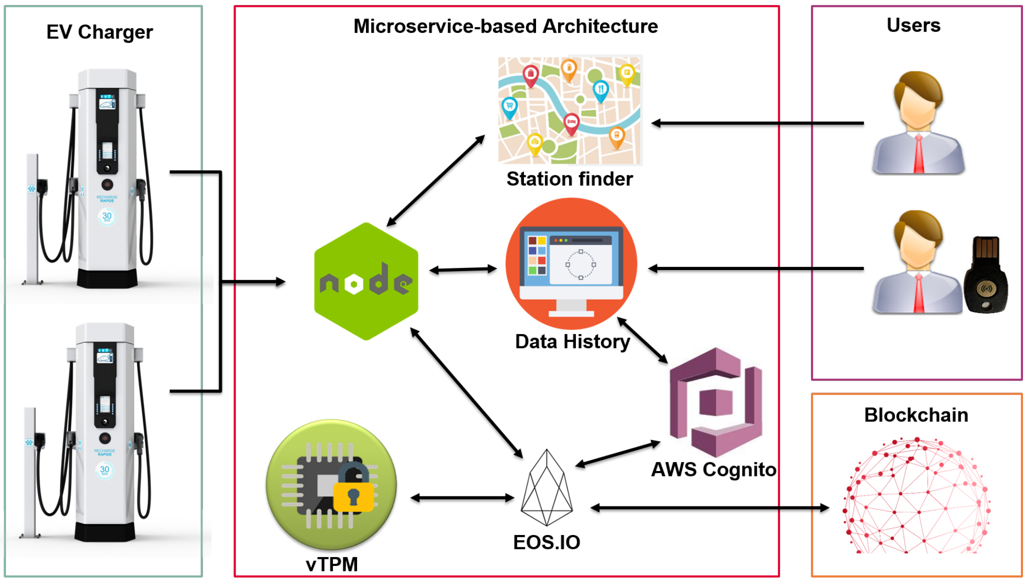

Figure 13. EV charging microservice-based architecture.

5.1. EOS.IO

EOS.IO [77] is a blockchain designed for deploying and executing decentralized applications or so-called smart contracts. This blockchain is designed to prioritize smart contract performance. EOS.IO uses a variation of the proof-of-stake algorithm known as delegated proof-of-stake (DPoS).This consensus algorithm regulates all the processes related to staking tokens, voting, vote decay, vote recording, producer ranking, and inflation pay. Smart contracts are written in C++, being the code compiled and then deployed in an EOS.IO virtual machine.

The EOS.IO client consists of the following three modules:

- Cleos: This is the command line tool that connects to the API exposed by Nodeos and manages the wallet, account, keys, transactions, and smart contracts.
- Nodeos: This functions as the central daemon that manages the EOS.IO network and can be configured as a node to produce blocks, which are be able to sign blocks.
- Keosd: This is in charge of storing and generating the keys.

EOS.IO is the blockchain microservice in charge of performing the necessary operations with the vTPM microservice and it is ideal for applications where good performance is required, this being the main reason why it has been chosen for this use case. In this scenario, instead of using Keosd as the key manager, the vTPM API is used.

It should be noted that, in the architecture depicted in Figure 13, only one EOS.IO node is shown, but additional EOS.IO nodes may appear in the use case that make use of the vTPM. It has been represented in this way for simplicity. Therefore, this EOS.IO node (or nodes) is externally connected to other nodes in a distributed network. This network can be scaled considerably in the system once deployed; typically, blockchain networks tend to have scalability issues once deployed in potentially large systems. In this case, EOS.IO introduces different types of scalability (horizontal and vertical), as well as data access. As for vertical scalability, EOS.IO improves performance by introducing enhancements to the Nodeos component. To improve the horizontal scalability, EOS.IO leverages the use of various abstraction layers for smart contracts. These abstraction layers allow multiple types of blockchain systems to easily use these smart contracts in a efficient way. Regarding data access, scalability is implemented to authenticate transactions that query and read history and state data, minimizing the exchange of data between entities.

The EOS.IO node authenticates with Cognito AWS in the vTPM microservice for use of cryptographic operations such as key generation and transaction signature. The NodeJS microservice provides the operation that EOS.IO will perform. If the operation is a transaction, the NodeJS script will provide the data as well. The transactions add data to the ledger, which can be either related to EV charging station information (geographical location, supplier, etc.) or to an energy charge (power supplied, money spent, bank details, etc.). The transaction is made with the corresponding data and signed by the vTPM microservice; before that, the blockchain client must check the status of the vTPM microservice, making an attestation and checking the values obtained with those provided by the cloud operator. In addition, in order to perform the signature, it must unseal the private key that was identified with the BLOB obtained in its generation. The transaction is sent to the blockchain and is approved by different nodes in the blockchain following the consensus algorithm implemented in EOS.IO. When using the geographical location of the charging station or the payment history microservice, they generate a query to the blockchain through the blockchain client. These microservices never write to the ledger, so operations performed on the blockchain are simply queries.

In this way, through one or more blockchain nodes running on the system that makes use of the vTPM microservice, data are stored on the blockchain, maintaining an immutable record of it and allowing building smart contracts that execute some processing logic. In addition, note that in order to be able to rely on these smart contracts, they must be verified by minimizing the risk of faults and bugs [78].

5.2. Two-Factor Authentication

The proposed mechanism for user authentication follows the same scheme as the one used in Section 4.3. This mechanism allows users who need to authenticate with the system to do so through a two-factor authentication system that is more secure than the traditional one. In this sense, the user carries a USB hardware token, where the user's private keys are securely stored. The authentication is performed through Cognito AWS.

In this scenario, user pools have different roles assigned to the users, e.g., for clients with special permissions as network administrators. In addition, the user pools assign roles depending on the functionalities assigned to the users. In this way, user pools have a similar functionality to the microservices authentication use case seen in Section 4.3.

The registration and authentication mechanism is shown in Figure 14. The user registers and logs into the system through a web interface. At registration, the user enters his/her name and password in the web form. This name and password must have been previously stored in the user pool in order to allow the registration of the user, this step is performed prior to registration through a different means (e.g., by e-mail to the system administrator). When the registration process starts, the web interface generates a challenge, which is sent to the hardware token. In the hardware token, the public and private key pair is created. Using the private key, the challenge is signed and sent to the Cognito user pool. The pool stores the user's attributes (name, email, etc.), user ID, and public key.

At this point, the user can log into the system. To do so, in this process, the user enters his/her username and password. These data are checked in Cognito AWS, to see if the user has already been registered, if so, Cognito AWS sends a challenge to the hardware token. This challenge is signed with the private key on the token and sent to Cognito AWS, which, using the previously stored public key, is able to verify the signature and check that the authenticating user is really who they claim to be. In this process, both challenge creation and challenge verification operations are performed through lambda functions deployed in Cognito AWS.

After this process, Cognito AWS sends a token to the user that will be used to authenticate with the web interface until the token expires. Through this token, the user can access the microservices that the token allows, with this token being associated with the user and the user pool that contains it. A token issued to a user belonging to the administrators' user pool will have more access rights to special functionalities (e.g., only administrators can create keys) than a normal user.

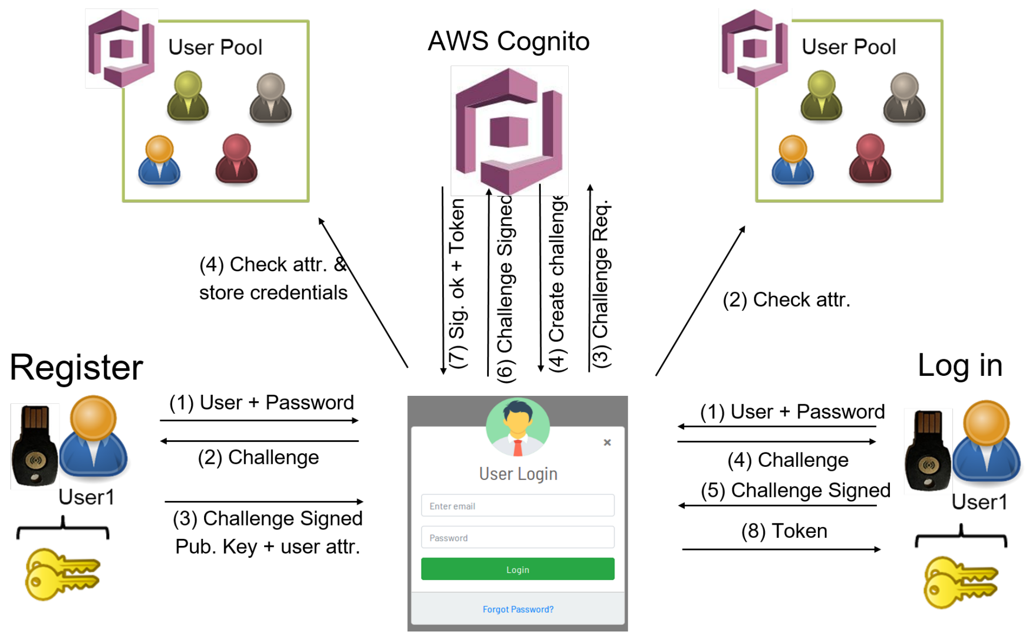

Figure 14. User authentication process with Cognito AWS.

In this way, a secure and robust authentication system is achieved, with different user roles, protecting the system from unauthorized access and storing the credentials securely in hardware.

6. Conclusions

Blockchain technologies are gaining prominence and are increasingly being used. In particular, in microservice-based architectures, the use of blockchain microservices opens up a wide range of possibilities for this type of architecture. The use of cryptography that powers blockchains is an important aspect in addressing and offering microservices of this type, improving the security of microservice-based architectures.

The use of microservices that offer cryptography through a virtualization of a hardware security module, such as the TPM, allows the hardware root of trust to be extended to other applications. This is why it is important to create a virtualization of this type of hardware resource. Through the virtualization proposed in this article, the vTPM microservice is capable of offering the functionalities that a TPM provides to other microservices, such as blockchain clients. Thanks to this microservice, the security of the rest of the microservices is increased, as the cryptographic operations are delegated to the vTPM.

On the other hand, this microservice must be secure and maintain its integrity in both deployment and use. Attestation mechanisms ensure that the container is not modified by external entities. This mechanism, through special TPM registers such as PCRs, performs different measurements inside the container, checking that the libraries, binaries, and the rest of the source code have not been modified since the initial state, which is supposed to be a secure state. In addition, private key identifiers are stored in container volumes that are protected through a sealing mechanism. Using this mechanism ensures the integrity of these identifiers and their usability by owners, as these identifiers can only be unlocked through a password or secret that unseals the identifier using the TPM's PCR. In addition to these integrity mechanisms, this container is protected with an authentication system based on Cognito AWS, which establishes permissions and roles for other containers to access the vTPM microservice. This protects against unauthorized access by untrusted third party microservices.

As a potential application of this proposal, a use case applied to EV charging stations was described, where all the information related to payments is sent to an EOS.IO blockchain, which makes use of the vTPM by signing the transactions and generating the keys used. The information stored on the blockchain is immutable, and all recorded information is accessed by users who use a two-factor authentication mechanism to gain access to sensitive information.

The future of blockchain microservices lies in the integration and virtualization of hardware security elements. This proposal has potential applications in the energy sector, such as EV charging stations, and helps these technologies to reach a higher level of maturity, without forgetting the hardware security that is required.

Author Contributions: Conceptualization, A.J.C.-G.; methodology, A.J.C.-G.; software, A.J.C.-G.; validation, A.J.C.-G.; formal analysis, A.J.C.-G.; investigation, A.J.C.-G.; writing—original draft preparation, A.J.C.-G. and J.C.-C.; writing—review and editing, E.C., A.E.-M., J.C.-C., D.P.M. and L.P.; supervision, E.C. and L.P.; project administration, A.J.C.-G. and A.E.-M.; funding acquisition, E.C., A.E.-M., D.P.M. and L.P. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by Infineon Technologies AG. Additional funding was provided by the program "Digitalisierung der Energiewende", sponsored by the "Bundesministeriums für Wirtschaft und Energie" (BMWi) within the framework of "7. Energieforschungsprogramms der Bundesregierung", through the funding project "Trusted Blockchains fur das offene, intelligente Energienetz der Zukunft (tbiEnergy)", under FKZ 03EI6029D. In addition, this work was partially funded from the European Health and Digital Executive Agency (HaDEA) program under Grant Agreement No 101092950 (EDGELESS project). The work was also funded by FEDER/Junta de Andalucia-Consejeria de Transformacion Economica, Industria, Conocimiento y Universidades under Project B-TIC-588-UGR20.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ABE	Attribute-Based Encryption
ABRMD	Access Broker and Resource Management Daemon
AES	Advanced Encryption Standard
AIK	Attestation Identity Key
API	Application Interface
AWS	Amazon Web Services
BLOB	Binary Large Object
CA	Certification Authority
CI/CD	Continuous Integration and Continuous Development
DLT	Decentralized Ledger Technologies
DoS	Denial of Service
DPoS	Delegated Proof of Stake
ECC	Elliptic Cryptographic Curve
ESAPI	Enhanced System API
EV	Electrical Vehicles
FAPI	Feature API
HSM	Hardware Security Modules
IaaS	Infrastructure as a Service
IIoT	Industrial Internet of Things
MQTT	Message Queue Telemetry Transpor
OS	Operating System
PaaS	Platform as a Service
PCR	Platform Configuration Register
PKCS	Public Key Cryptographic Standard
PKI	Public Key Infrastructure
RSA	Rivest-Shamir-Adleman
SaaS	Software as a Service

SAPI	System API
TCG	Trusted Computing Group
TCTI	TPM Command Transmission Interface
TPM	Trusted Platform Module
TRNG	True Random Number Generator
TSS	TPM Software Stack
VM	Virtual Machine
vTPM	virtual TPM

References

1. Saritha, S.; Sarasvathi, V. A study on application layer protocols used in IoT. In Proceedings of the 2017 International Conference on Circuits, Controls, and Communications (CCUBE), IEEE, Bangalore, India, 15–16 December 2017; pp. 155–159.
2. Ponnusamy, K.; Rajagopalan, N. Internet of things: A survey on IoT protocol standards. *Prog. Adv. Comput. Intell. Eng.* **2018**, *2*, 651–663.
3. Organization for the Advancement of Structured Information Standards. 2014, 1, 29. MQTT Version 3.1.1. Available online: http://docs.oasis-open.org/mqtt/mqtt/v3 (accessed on 15 October 2022).
4. Chen, F.; Huo, Y.; Zhu, J.; Fan, D. A review on the study on MQTT security challenge. In Proceedings of the 2020 IEEE International Conference on Smart Cloud (SmartCloud), IEEE, Washington, DC, USA, 6–8 November 2020; pp. 128–133.
5. Sundarrajan, M.; Narayanan, A.E.; Srithar, V. Securing the MQTT Protocol using Enhanced Cryptographic techniques in IoT Surroundings. *J. Phys. Conf. Ser.* **2021**, *1767*, 012055. [CrossRef]
6. Wong, H.C. Man-in-the-Middle Attacks on MQTT Based IoT Networks. Ph.D. Thesis, Missouri University of Science and Technology, Rolla, MO, USA, 2022.
7. Weise, J. *Public Key Infrastructure Overview*; Sun Blueprints: Palo Alto, CA, USA, 2001.
8. Talamo, M.; Arcieri, F.; Dimitri, A.; Schunck, C.H. A Blockchain based PKI Validation System based on Rare Events Management. *Future Internet* **2020**, *12*, 40. [CrossRef]
9. Sunyaev, A. Distributed ledger technology. In *Internet Computing*; Springer: Cham, Switzerland, 2020; pp. 265–299.
10. Sultan, K.; Ruhi, U.; Lakhani, R. Conceptualizing blockchains: Characteristics applications. *arXiv* **2018**, arXiv:1806.03693.
11. Khan, S.N.; Loukil, F.; Ghedira-Guegan, C.; Benkhelifa, E.; Bani-Hani, A. Blockchain smart contracts: Applications, challenges, and future trends. *Peer-to-Peer Netw. Appl.* **2021**, *14*, 2901–2925. [CrossRef] [PubMed]
12. Viriyasitavat, W.; Anuphaptrirong, T.; Hoonsopon, D. When blockchain meets Internet of Things: Characteristics, challenges, and business opportunities. *J. Ind. Inf. Integr.* **2019**, *15*, 21–28. [CrossRef]
13. Agbo, C.C.; Mahmoud, Q.H.; Eklund, J.M. Blockchain technology in healthcare: A systematic review. *Healthcare* **2019**, *7*, 56. [CrossRef]
14. Andoni, M.; Robu, V.; Flynn, D.; Abram, S.; Geach, D.; Jenkins, D.; McCallum, P.; Peacock, A. Blockchain technology in the energy sector: A systematic review of challenges and opportunities. *Renew. Sustain. Energy Rev.* **2019**, *100*, 143–174. [CrossRef]
15. Viriyasitavat, W.; Hoonsopon, D. Blockchain characteristics and consensus in modern business processes. *J. Ind. Inf. Integr.* **2019**, *13*, 32–39. [CrossRef]
16. Yu, T.; Lin, Z.; Tang, Q. Blockchain: The introduction and its application in financial accounting. *J. Corp. Account. Financ.* **2018**, *29*, 37–47. [CrossRef]
17. Jiang, L.; Zhang, X. BCOSN: A blockchain-based decentralized online social network. *IEEE Trans. Comput. Soc. Syst.* **2019**, *6*, 1454–1466. [CrossRef]
18. Raikwar, M.; Mazumdar, S.; Ruj, S.; Gupta, S.S.; Chattopadhyay, A.; Lam, K.Y. A blockchain framework for insurance processes. In Proceedings of the 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS), IEEE, Paris, France, 2 February 2018; pp. 1–4.
19. Alammary, A.; Alhazmi, S.; Almasri, M.; Gillani, S. Blockchain-based applications in education: A systematic review. *Appl. Sci.* **2019**, *9*, 2400. [CrossRef]
20. Stephen, R.; Alex, A. A review on blockchain security. *IOP Conf. Ser. Mater. Sci. Eng.* **2018**, *396*, 012030. [CrossRef]
21. Sklavos, N.; Chaves, R.; Di Natale, G.; Regazzoni, F. *Hardware Security and Trust*; Springer: Cham, Switzerland, 2017.
22. Paverd, A.J.; Martin, A.P. Hardware security for device authentication in the smart grid. In *International Workshop on Smart Grid Security*; Springer: Berlin/Heidelberg, Germany, 2012; pp. 72–84.
23. Lesjak, C.; Hein, D.; Hofmann, M.; Maritsch, M.; Aldrian, A.; Priller, P.; Ebner, T.; Ruprechter, T.; Pregartner, G. Securing smart maintenance services: Hardware-security and TLS for MQTT. In Proceedings of the 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), IEEE, Cambridge, UK, 22–24 July 2015; pp. 1243–1250.
24. Bouganim, L.; Guo, Y. Database Encryption. In *Encyclopedia of Cryptography and Security*; van Tilborg, H.C.A., Jajodia, S., Eds.; Springer: Boston, MA, USA, 2011. [CrossRef]
25. Truong, M.T.; Dang, Q.V. Digital Signatures Using Hardware Security Modules for Electronic Bills in Vietnam: Open Problems and Research Directions. In *International Conference on Future Data and Security Engineering*; Springer: Singapore, 2020; pp. 469–475.
26. Han, J.; Kim, S.; Kim, T.; Han, D. Toward scaling hardware security module for emerging cloud services. In Proceedings of the 4th Workshop on System Software for Trusted Execution, New York, NY, USA, 27 October 2019; pp. 1–6.

27. Perez, R.; Van Doorn, L.; Sailer, R. Virtualization and hardware-based security. *IEEE Secur. Priv.* **2008**, *6*, 24–31. [CrossRef]
28. Khan, W.Z.; Ahmed, E.; Hakak, S.; Yaqoob, I.; Ahmed, A. Edge computing: A survey. *Future Gener. Comput. Syst.* **2019**, *97*, 219–235. [CrossRef]
29. Yi, S.; Li, C.; Li, Q. A survey of fog computing: Concepts, applications and issues. In Proceedings of the 2015 Workshop on Mobile Big Data, Virtual, 21 June 2015; pp. 37–42.
30. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*; Springer: Cham, Switzerland, 2017; pp. 195–216.
31. Baškarada, S.; Nguyen, V.; Koronios, A. Architecting microservices: Practical opportunities and challenges. *J. Comput. Inf. Syst.* **2020**, *60*, 428–436. [CrossRef]
32. Lysakov, V.; Sievierinov, O.; Taran, I. Security of Web Applications Using AWS Cloud Provider. *Comput. Inf. Syst. Technol.* **2021**.
33. Perez, R.; Sailer, R.; van Doorn, L. vTPM: Virtualizing the trusted platform module. In Proceedings of the 15th Conference on USENIX Security Symposium, Vancouver, Canada, 31 July–4 August 2006; pp. 305–320.
34. Fazio, M.; Celesti, A.; Ranjan, R.; Liu, C.; Chen, L.; Villari, M. Open issues in scheduling microservices in the cloud. *IEEE Cloud Comput.* **2016**, *3*, 81–88. [CrossRef]
35. Li, Z.; Kihl, M.; Lu, Q.; Andersson, J.A. Performance overhead comparison between hypervisor and container based virtualization. In Proceedings of the 2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA), IEEE, Taipei, Taiwan, 27–29 March 2017; pp. 955–962.
36. Soltesz, S.; Pötzl, H.; Fiuczynski, M.E.; Bavier, A.; Peterson, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, New York, NY, USA, 21–23 March 2007; pp. 275–287.
37. Eder, M. Hypervisor-vs. Container-Based Virtualization. Chair for Network Architectures and Services, Department of Computer Science, Technische Universität München. 2016. Available online: https://search.datacite.org/works/10.2313/net-2016-07-1_01 (accessed on 2 April 2023). [CrossRef]
38. Alshuqayran, N.; Ali, N.; Evans, R. A systematic mapping study in microservice architecture. In Proceedings of the 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA), IEEE, Macau, China, 4–6 November 2016; pp. 44–51.
39. Lu, D.; Huang, D.; Walenstein, A.; Medhi, D. A secure microservice framework for iot. In Proceedings of the 2017 IEEE Symposium on Service-Oriented System Engineering (SOSE), IEEE, San Francisco, CA, USA, 6–9 April 2017; pp. 9–18.
40. Sun, L.; Li, Y.; Memon, R.A. An open IoT framework based on microservices architecture. *China Commun.* **2017**, *14*, 154–162. [CrossRef]
41. Nagothu, D.; Xu, R.; Nikouei, S.Y.; Chen, Y. A microservice-enabled architecture for smart surveillance using blockchain technology. In Proceedings of the 2018 IEEE International Smart Cities Conference (ISC2), IEEE, Kansas City, MO, USA, 16–19 September 2018; pp. 1–4.
42. Heron, S. Advanced encryption standard (AES). *Netw. Secur.* **2009**, *12*, 8–12. [CrossRef]
43. Milanov, E. The RSA algorithm. *Rsa Lab.* **2009**, 1–11. Available online: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwi-wKmU8Yz_AhV0sFYBHWHWB2kQFnoECAoQAQ&url=https%3A%2F%2Fsites.math.washington.edu%2F~morrow%2F336_09%2Fpapers%2FYevgeny.pdf&usg=AOvVaw0CedZrwLuyM7cBDvIQhMUW (accessed on 2 April 2023).
44. Xu, R.; Nikouei, S.Y.; Chen, Y.; Blasch, E.; Aved, A. Blendmas: A blockchain-enabled decentralized microservices architecture for smart public safety. In Proceedings of the 2019 IEEE International Conference on Blockchain (Blockchain), IEEE, Atlanta, GA, USA, 14–17 July 2019; pp. 564–571.
45. Jin, W.; Xu, R.; You, T.; Hong, Y.G.; Kim, D. Secure edge computing management based on independent microservices providers for gateway-centric IoT networks. *IEEE Access* **2020**, *8*, 187975–187990. [CrossRef]
46. Pahl, M.O.; Aubet, F.X. All eyes on you: Distributed Multi-Dimensional IoT microservice anomaly detection. In Proceedings of the 2018 14th International Conference on Network and Service Management (CNSM), IEEE, Rome, Italy, 5–9 November 2018; pp. 72–80.
47. Pahl, M.O.; Aubet, F.X.; Liebald, S. Graph-based IoT microservice security. In Proceedings of the NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium, IEEE, Taipei, Taiwan, 23–27 April 2018; pp. 1–3.
48. Mavrovouniotis, S.; Ganley, M. Hardware security modules. In *Secure Smart Embedded Devices, Platforms and Applications*; Springer: New York, NY, USA, 2014; pp. 383–405.
49. Mitchell, C. (Ed.) *Trusted Computing*; Iet: London, UK, 2005; Volume 6.
50. Arthur, W.; Challener, D.; Goldman, K. *A Practical Guide to TPM 2.0: Using the New Trusted Platform Module in the New Age of Security*; Springer Nature: Berlin/Heidelberg, Germany, 2015; p. 392.
51. Tomlinson, A. Introduction to the TPM. In *Smart Cards, Tokens, Security and Applications*; Springer: Cham, Switzerland, 2017; pp. 173–191.
52. Gu, L.; Ding, X.; Deng, R.H.; Xie, B.; Mei, H. Remote attestation on program execution. In Proceedings of the 3rd ACM Workshop on Scalable Trusted Computing, New York, NY, USA, 31 October 2008; pp. 11–20.

53. Guo, Y.; Yu, A.; Gong, X.; Zhao, L.; Cai, L.; Meng, D. Building trust in container environment. In Proceedings of the 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE), IEEE, Rotorua, New Zealand, 5–8 August 2019; pp. 1–9.
54. Hosseinzadeh, S.; Laurén, S.; Leppänen, V. Security in container-based virtualization through vTPM. In Proceedings of the 9th International Conference on Utility and Cloud Computing, Leicester, UK, 6–9 December 2016; pp. 214–219.
55. Sultan, S.; Ahmad, I.; Dimitriou, T. Container security: Issues, challenges, and the road ahead. *IEEE Access* **2019**, *7*, 52976–52996. [CrossRef]
56. Chandramouli, R.; Chandramouli, R. *Security Assurance Requirements for Linux Application Container Deployments*; US Department of Commerce, National Institute of Standards and Technology: Gaithersburg, MD, USA, 2017.
57. Kerry, C.F.; Gallagher, P.D. *Digital Signature Standard (DSS)*; FIPS PUB: Washington, DC, USA, 2013; pp. 186–194.
58. Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Bus. Rev.* **2008**, 21260.
59. Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Proj. Yellow Pap.* **2014**, *151*, 1–32.
60. Organization for the Advancement of Structured Information Standards. PKCS# 11 Cryptographic Token Interface Base Specification Version 2.40. Available online: https://bit.ly/3EJ18jk (accessed on 10 November 2022).
61. Anderson, R.; Kuhn, M. Tamper resistance—A cautionary note. In Proceedings of the 2nd Usenix Workshop on Electronic Commerce, Oakland, CA, USA, 18–21 November 1996; Volume 2, pp. 1–11.
62. Skorobogatov, S. Physical attacks and tamper resistance. In *Introduction to Hardware Security and Trust*; Springer: New York, NY, USA, 2012; pp. 143–173.
63. Anderson, R.; Kuhn, M. Low cost attacks on tamper resistant devices. In *International Workshop on Security Protocols*; Springer: Berlin/Heidelberg, Germany, 1997; pp. 125–136.
64. TPM2-Software. TPM2-Software/TPM2-Abrmd: TPM2 Access Broker and Resource Management Daemon Implementing The TCG Spec. GitHub. 2022. Available online: https://github.com/tpm2-software/tpm2-abrmd (accessed on 18 October 2022).
65. Arthur, W.; Challener, D.; Goldman, K. TPM Software Stack. In *A Practical Guide to TPM 2.0*; Apress: Berkeley, CA, USA, 2015; pp. 77–96.
66. TPM2-Software. TPM2-Software/TPM2-Tools: The Source Repository for the Trusted Platform Module (TPM2.0) Tools. GitHub. 2022. Available online: https://github.com/tpm2-software/tpm2-tools (accessed on 18 October 2022).
67. TPM2-Software. TPM2-Software/TPM2-Pytss: Python Bindings for TSS. GitHub. 2022. Available online: https://github.com/tpm2-software/tpm2-pytss (accessed on 18 October 2022).
68. TPM2-Software. TPM2-Software/TPM2-TSS-Engine: Openssl Engine for TPM2 Devices. GitHub. 2022. Available online: https://github.com/tpm2-software/tpm2-tss-engine (accessed on 18 October 2022).
69. TPM2-Software. TPM2-Software/TPM2-Openssl: Openssl Provider for TPM2 Integration. GitHub. 2022. Available online: https://github.com/tpm2-software/tpm2-openssl (accessed on 18 October 2022).
70. TPM2-Software. TPM2-Software/TPM2-PKCS11: A PKCS#11 Interface for TPM2 Hardware. GitHub. 2022. Available online: https://github.com/tpm2-software/tpm2-pkcs11 (accessed on 18 October 2022).
71. Rangnau, T.; Buijtenen, R.V.; Fransen, F.; Turkmen, F. Continuous security testing: A case study on integrating dynamic security testing tools in ci/cd pipelines. In Proceedings of the 2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC), IEEE, Eindhoven, The Netherlands, 5–8 October 2020; pp. 145–154.
72. Aura, T. Strategies against replay attacks. In Proceedings of the 10th Computer Security Foundations Workshop, IEEE, Rockport, MA, USA, 10 June 1997; pp. 59–68.
73. Gai, K.; Wu, Y.; Zhu, L.; Xu, L.; Zhang, Y. Permissioned blockchain and edge computing empowered privacy-preserving smart grid networks. *IEEE Internet Things J.* **2019**, *6*, 7992–8004. [CrossRef]
74. Wang, S.; Taha, A.F.; Wang, J.; Kvaternik, K.; Hahn, A. Energy crowdsourcing and peer-to-peer energy trading in blockchain-enabled smart grids. *IEEE Trans. Syst. Man, Cybern. Syst.* **2019**, *49*, 1612–1623. [CrossRef]
75. Saxena, S.; Farag, H.E.; Turesson, H.; Kim, H. Blockchain based transactive energy systems for voltage regulation in active distribution networks. *IET Smart Grid* **2020**, *3*, 646–656. [CrossRef]
76. ElHusseini, H.; Assi, C.; Moussa, B.; Attallah, R.; Ghrayeb, A. Blockchain, AI and smart grids: The three musketeers to a decentralized EV charging infrastructure. *IEEE Internet Things Mag.* **2020**, *3*, 24–29. [CrossRef]
77. EOS. IO Technical White Paper v2. EOS, Tech. Rep. 2018. Available online: https://github.com/BlockchainTranslator/EOS/blob/master/TechDoc/EOS.IO-Technical-WhitePaper-v2.md (accessed on 2 April 2023).
78. Krichen, M.; Lahami, M.; Al–Haija, Q.A. Formal Methods for the Verification of Smart Contracts: A Review. In Proceedings of the 2022 15th International Conference on Security of Information and Networks (SIN), Sousse, Tunisia, 11–13 November 2022; pp. 1–8. [CrossRef]