electronics

Article

Multiprotocol Authentication Device for HPC and Cloud Environments Based on Elliptic Curve Cryptography

Antonio F. Díaz *, Ilia Blokhin, Mancia Anguita, Julio Ortega and Juan J. Escobar

Department of Computer Architecture and Technology, CITIC-UGR Research Center, University of Granada, E18071 Granada, Spain; djnib@correo.ugr.es (I.B.); manguita@ugr.es (M.A.); jortega@ugr.es (J.O.); jjescobar@ugr.es (J.J.E.)

* Correspondence: afdiaz@ugr.es; Tel.: +34-958-246127

Received: 19 June 2020; Accepted: 14 July 2020; Published: 16 July 2020

Abstract: Multifactor authentication is a relevant tool in securing IT infrastructures combining two or more credentials. We can find smartcards and hardware tokens to leverage the authentication process, but they have some limitations. Users connect these devices in the client node to log in or request access to services. Alternatively, if an application wants to use these resources, the code has to be amended with bespoke solutions to provide access. Thanks to advances in system-on-chip devices, we can integrate cryptographically robust, low-cost solutions. In this work, we present an autonomous device that allows multifactor authentication in client–server systems in a transparent way, which facilitates its integration in High-Performance Computing (HPC) and cloud systems, through a generic gateway. The proposed electronic token (eToken), based on the system-on-chip ESP32, provides an extra layer of security based on elliptic curve cryptography. Secure communications between elements use Message Queuing Telemetry Transport (MQTT) to facilitate their interconnection. We have evaluated different types of possible attacks and the impact on communications. The proposed system offers an efficient solution to increase security in access to services and systems.

Keywords: elliptic curve cryptography; authentication protocol; cryptographic devices; security keys

1. Introduction

Servers and services define authentication schemes to avoid unauthorised access. Accordingly, they are essential elements to implement a secure system. A large number of computer attacks take advantage of vulnerabilities in authentication systems. The Common Vulnerabilities and Exposures (CVE) is an updated and growing list of known security vulnerabilities. In particular, among them are those related to authentication [1]. It is important to implement mechanisms to strengthen and guarantee the authentication of users and systems.

A first approach is to use passwords to control access, but this is insufficient as they may be susceptible to attacks based on dictionaries, social engineering, or access to databases where credentials are stored [2].

Some systems have been proposed to avoid the use of user keys, such as Pico [3], where a secure authentication protocol is established.

Adding additional elements to a password allows an extra security component. Systems with multifactor authentication are often used to improve authentication, ensuring that if the security of an element is compromised, the other one can ensure that access remains secure.

Cryptographic resources such as symmetric and asymmetric encryption, hashing, and public key signing are strong elements for solving mathematical authentication. Besides, it is vital the use of true random number generators.

Vulnerabilities are often primarily related to how solutions are implemented. It is crucial to be careful in choosing the right combination of resources to prevent attacks. We can find many authentication systems based on the elements indicated previously. In particular, elliptic curve-based public key cryptography [4] has a significant impact on the development of current robust systems.

We can combine passwords with some other elements to leverage security. There are solutions based on mobile applications [5], or systems based on sending keys by SMS as Azure integrates into its portal [6]. However, this kind of authentication has well-documented shortcomings [7]. If a duplicate SIM card is obtained through social engineering or impersonating the owner, the mobile phone-based security may be compromised.

Users are progressively sensitised to the privacy of their mobile phone and do not want to install applications to access work-related resources. Moreover, if they change their phones, it involves reinstalling the software. For this reason, many users prefer to have an external authentication element.

Electronic devices with cryptographic resources allow adding extra security and are more challenging to duplicate and manipulate. For a long time, we can find solutions based on smartcards or access tokens that we will describe in the Related Work section.

The new system-on-chip devices offer a 32-bit computing capacity, integrated cryptographic resources, and various communication systems, allowing the integration of new authentication alternatives.

The authentication process can be approached in different ways, depending on how the resources are interconnected and what level of security we want to implement. The aim of the proposed system is to address some of these problems, using an electronic token (eToken) in a multifactor authentication scheme that prevents duplication and facilitates communication between elements. The system includes a transparent gateway that allows integration with the systems currently in use, which simplifies its global integration. It also uses secure communications based on Message Queuing Telemetry Transport (MQTT). This model is more flexible than the usual authentication protocols since it allows to authenticate remote systems or to be combined with multi-eToken authorisation.

This paper is organised as follows: In Section 2, we introduce related works. Section 3 describes our proposed system. Section 4 shows the security analysis of the main components based on possible attacks. Section 5 gives a detailed experiment setup and results. Finally, in Section 6 we present some conclusions.

2. Related Work

A standard solution for identifying users in High-Performance Computing (HPC) and cloud environments is to authorise access with a public/private key pair, from a trusted client node. A concern is the uncontrolled use of public/private keys. Users can forget where they have installed them, so the access to this personal computer can comprise all of them [8]. If someone accesses the original computer, access to all nodes is compromised. To resolve this, Cloudfare [9] uses short-lived certificates and single sign-on (SSO) credential-based authentication.

Furthermore, Kerberos [10] was proposed in 1988 as an authentication system and was updated to version 5 in 2005 to resolve some limitations in RFC4120 [11]. It is used as an initial authentication mechanism to access Windows and Linux systems. It can also be used in other environments [12], but it requires a "kerberisation" process, which limits its use in applications that are not foreseen. These systems were designed when there were limitations in communication networks, processors were slower, and public key cryptography was less used. The Kerberos detach authentication process from the service server creates a centralised control for different services. Additionally, Kerberos is based on the use of private key cryptography, so it is essential to select robust passwords to avoid dictionary attacks. Some security analyses [13,14] show this weakness in the communication phase which represents a real challenge for this protocol. It also requires time synchronisation between computers, since tickets are based on time stamps. Some extensions to Kerberos have been proposed, such as described by Tbatou et al. [15]. They define an authentication protocol for distributed systems based upon Kerberos V5 and Diffie–Hellman models, but they are not multifactor authentication schemes. To make Kerberos more secure, Quoc et al. [16] proposed to modify the initial exchange in Kerberos 5 by using biometric data and asymmetric cryptography. They used a .Net DLL library MCC SDK for fingerprint biometric authentication, so it is limited to Windows platforms. Moreover, these solutions do not solve how to adapt the authentication scheme to other protocols.

Additionally, we can find some multifactor authentication resources to enhance security that trust on a single component. There are some time-based one-time password (OTP) systems, such as Google Authenticator [17], which involve an approximate synchronisation to generate the access code. Further, the user needs to access another program to obtain the code, type it, or wait because the time expiration is close, so it is generally somewhat inconvenient to use. Although Google has developed this OTP scheme, it prefers that its employees use security keys based on hardware tokens [18].

Some authentication models have been proposed, such as systems based on communication latency [19], but these systems are neither applicable in clusters, since the access times between nodes are almost constant and independent of the Internet connection, nor in environments with access times.

Smartcard-based systems [20] have been used for a long time to identify in a safe way. These cards require a reader that could be integrated into a keyboard, laptop, or USB reader. The problem is that, over time, some drivers have stopped working because they have become obsolete by the operating system versions. New smartcards also support near-field communication (NFC) which facilitates access but involves using a computer as an intermediate element to authenticate. If we do not have this element, it is not possible to carry out the authentication. Furthermore, the user cannot validate or cancel a request if the card is inserted.

The use of token-based systems has grown because they are simple elements to use. It is a resource that the user owns, so it is not enough to have a password or a public/private key pair. Besides, the user validates by using a click to request access.

The Fast Identity Online Alliance (FIDO) specified two authentication frameworks and protocols: the universal authentication framework (UAF) for password-less authentication from smart devices, and the Universal Second Factor protocol (U2F) [21] for the authentication of two factors using a small hardware token to accompany a non-FIDO smart device that has a FIDO-compatible web browser. Both operate on the same underlying principle of using asymmetric encryption for authentication, and both have now been combined in the World Wide Web Consortium (W3C) Web Authentication Recommendation (FIDO2) [22].

Yubiko [23] sells different U2F security keys. According to the manufacturer, to prevent attacks on the YubiKey, which might compromise its security, the YubiKey does not permit its firmware to be accessed or altered. However, a bug in the randomness of some of these keys (FIPS Series) cause 80 bits of a nonce to be fixed. If it is used for Elliptic Curve Digital Signature Algorithm (ECDSA) signatures, it could allow an attacker who gains access to several signatures to reconstruct the private key [24]. So, if you use a device that cannot be reprogrammed and critical failure is detected, finally you have to throw it away.

Chadwick et al. [25] propose the use of mobile phones with fingerprint readers to authenticate users with UAF, instead of using users and access codes. Ciolino et al. [26] have studied the impact on the usability of security keys and the importance of using them in a broader context of web services. There are provisioning schemes for U2F-based IoT devices such as U2Fi [20], but these do not offer a direct solution to communicate between client and server independent applications.

OAuth is a solution that tries to unify the authorisation of different applications. The first version of OAuth dates from 2007, and in 2012, OAuth 2.0 was defined [27]. This protocol allows to authorise third-parties to access their server resources without sharing their credentials, and it is mainly used in web or mobile applications. According to Chae et al. [28], it has many security vulnerabilities in the third-party application certification procedure. OAuth 2.0 is not an authentication protocol, though, it can be used as a base layer for another authentication protocol such as OpenID Connect [29]. OpenID Connect 1.0 allows clients to verify the identity of the user based on the authentication performed by the authorisation server. According to Li et al. [30], real-world implementations of both schemes are often vulnerable to attack, and in particular to cross-site request forgery (CSRF) attacks.

OpenID Connect and OAuth 2.0 do not include a multifactor authentication but an external solution to provide it exists such as SAASSPASS [31], with a mobile app. Also, SecSign [32] provides an extension for two-factor authentication using a mobile phone. However, as it was commented before, users are reluctant to install applications in their phones to access work resources.

Another solution is to combine different elements like single sign-on (SSO) with two-factor authentication and Security Assertion Markup Language (SAML). We can use SAML for exchanging user authentication data as Extensible Markup Language (XML) between identity providers and service providers. SSO is a scheme that uses a single authentication to allow access to multiple applications by passing an authentication token seamlessly to configured applications.

However, at the end, if you need to implement a solution you have to use different layers provided by diverse developers, so it could be critical the compatibility of the different version of each component to get a stable solution. We should avoid using a sledgehammer to crack a nut.

In an HPC environment, we can find a very diverse ecosystem of applications. Not only we have web access to a platform, but also we need to access resources and applications located internally in the cluster, so there is usually no direct access to facilitate authentication on computers and users that are on the private intranet network.

These computing environments are continually growing with new massive data storage systems, computing models such as map/reduce, or new paradigms that arise when trying to tackle complex computing problems that can be more easily solved by new communication models. Many of these systems are based on a user and password scheme as the first level of authentication.

There are specific solutions for Hadoop such as the one proposed by Khalil et al. [33], based on Trusted Platform Module (TPM), but this solution cannot be used in other applications and depends on the TPM installed in each client node, so it locks the user to a specific client node.

In addition, another general problem in multiple protocols is that they do not include a centralised system that can easily control the revocation of communications in real-time. Typically, when client–server communication has been authorised and established, there is no direct way to cancel it. In these cases, different rules can be set at the firewall level, but these require superuser permissions.

3. Design and Implementation

The proposed system offers a multifactor authentication system that can be used in various configurations; for instance, a basic client/server model, an HPC environment where it is possible to access multiple services and servers, or in a cloud environment taking advantage of various resources such as sending messages or virtualised servers.

Although it could itself be used as a single authentication element, it is recommended to have multiple elements that strengthen authentication due to the increase in the number of cyberattacks. That is why this system is proposed as an additional element of authentication that can work transparently.

This model allows that security can be increased on critical systems with combined authentication, that is, not only authenticating with a user but also with a second user to validate access.

The proposed system allows any application based on TCP/IP (network databases, big data systems, browsers) to use this authentication model. Besides, it supports remote authentication; that is, the user that authorises does not need to be directly connected to the service server. Other systems like U2F are mainly oriented to be used with browsers and session authentication.

This system monitors communications in real-time, keeping a record of active communications, so if necessary, a specific communication can be cut instantly. Additionally, the system can be installed and configured without the need for root privileges, making it easy to deploy the system.

It is highly scalable because it is based on the Message Queuing Telemetry Transport (MQTT) model, with diverse broker implementations offering high availability and redundancy. MQTT servers are widely used in IoT infrastructures, and various cloud providers also offer MQTT-based services so they can take advantage of the services they offer.

The gateway server prevents denial of service attacks because the ports are closed and only activated when the request is required.

The protocol is open, allowing the development and implementation of various solutions, with different public/private key models.

Our eToken can verify the context; that is, it can detect which Bluetooth or WiFi resources are closed for the eToken in order to authorise in a valid environment. This operation can be remotely activated by the administrator who can control which resources are valid.

The system separates the authentication service from the system being accessed, as Kerberos does, although in our proposal, we do not need time stamps and the authentication is based on public key algorithms. In general, other authentication systems require direct access from client to authentication servers, but in our scheme, communications go through the MQTT broker, which avoids direct connections facilitating global connectivity among all the elements.

This eToken is based on ESP32 [34], a low-cost device with a 2-core processor at 240 MHz, which has support for cryptographic functions and includes WiFi and Bluetooth communications. This circuit has been used in other ECC-based cipher suites for IoT, such as [35]. Although it has a USB connection, it can work independently and does not need to be connected to USB, so it is ideal for carrying it in a portable way and working with equipment that does not have access to USB ports.

Systems based on smartcards require some type of reader, either contact or recently NFC. The proposed system is independent, so it does not need any additional reader because it uses WiFi connection.

One of the advantages of our system over cryptographic circuits or security keys that cannot update is that if a vulnerability is detected, our system can be reprogrammed and updated. The idea is to have a low-cost open system that can be used for secure authentication, and that can even be extended with the use of cryptographic circuits like ATECC608A [36].

3.1. System Overview

The proposed system consists of a set of functional elements configured on physical or virtual resources. The functional elements are:

- Users: who access the system;
- Services: offered by the programs through TCP connections;
- Servers: computers where the services are installed;
- Clients: computers from which services are accessed;
- eToken: devices that authenticate the request by the user.

All these elements can be gathered into groups to facilitate configuration. We can define some access control list (ACL) rules on these elements. Each item has a 128-bit UUID and a public/private key pair that allows the system to authenticate the requests. Each element generates its keys, so the private key never leaves its environment.

The system is hybrid because each element authenticates other elements using public keys, and is combined with a centralised model that authorises access. Since authorised public keys can be added and removed, the configuration server can send messages for all items, including eTokens, to update the list of authorised items.

There are other additional elements that are:

- Configuration server: this allows to modify the configuration of each element remotely;
- Authentication server: this authorises the request for access to services based on ACLs;

- Gateways: transparent clients and servers that redirect network connections;
- MQTT broker: server for exchanging messages.

Some of these resources can be implemented on the same node but have been independently designed, allowing a distributed model. These can be separated according to the needs of the implementation.

Table 1 shows a comparison between different authentication schemes, including our model.

Table 1. Comparation of main features among different authentication schemes.

Features	Our Proposed Model	Kerberos + Biometric Auth [16]	Kerberos + TPM → Hadoop	FIDO U2F Yubikey	Oauth + OpenID Connect + SecSign	Smart Cards
Use public/private key cryptography	Public	Private	Public (TPM)/ Private(Kerberos)	Public	Public	Public/ Private
Remote nodes authorisation	Yes	Yes	No	Yes	Yes	No
Scalable	Yes	No	No	Yes	Yes	No
Multiprotocol/ Easy to adapt to new protocols	Yes	Kerberisation/ Only Windows	Only for Hadoop	Yes	No	No
Reprogrammable	Yes	Yes	No (TPM)	No	Yes	No
Password activation	Yes	No	No	No	No	Pin
Combined authorisation	Yes	No	No	No	No	No
Indirect client to auth. server connection	Yes	No	No	No	No	No
Real-time revocation control	Yes	No	No	No	No	No
Time synchronisation-independent	Yes	No	No	Yes	Yes	Yes

3.2. System Architecture

The proposed system is oriented to be distributed and scalable, allowing the control of elements that can be placed in private infrastructures, cloud environments, or hybrid models. Furthermore, it allows any application to use this authentication scheme. Hence, it acts as a bridge between client and server, avoiding having to modify the original applications and offering a reduced impact on global performances, as described in Section 5.

The communication system is implemented with the following elements:

- Client request service (CRS): this makes the service request to the eToken;
- eToken (ET): this validates the request by the user and the signature;
- Authenticator server (AS): this server verifies the authenticity of the message and requests access to the gateway server. Another authorisation process checks the ACL rules for accessing the services;
- Gateway server (GS): accepts incoming communication to the server and monitors the communication in case of revocation;
- Gateway client (GC): allows communication with the gateway server and is integrated with the client request service.

The advantage of a centralised authentication server is that it simplifies the management of access rules and avoids the management of certificate revocation. Furthermore, AS can force the cancellation of the communication instantly, informing the SG that it must close a specific connection.

CRS, AS, GS, and GC are written in Golang. It is a language that allows applications to be portable, fast, robust, and optimised for concurrency in each architecture. Besides, the system can work with containers like Docker. ESP32 code is written in C++, using the PlatformIO toolset [37]. Figure 1 shows how the different elements are interconnected.

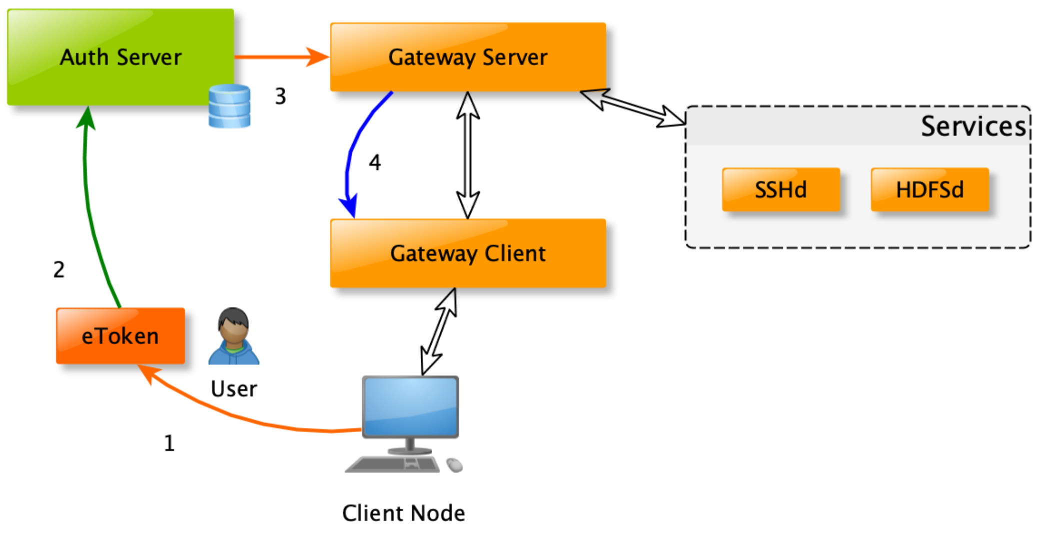

Figure 1. Main elements of the authentication and communication system.

Besides, there are two additional elements:

- MQTT broker (MB): this establishes communication between all components transparently; as shown in Figure 2.
- Configuration server (CS): the only server authorised to manage any configuration in the system;
- The MQTT protocol based on a published and subscribed system. Thanks to the centralised model with an MQTT broker, all elements are securely interconnected.

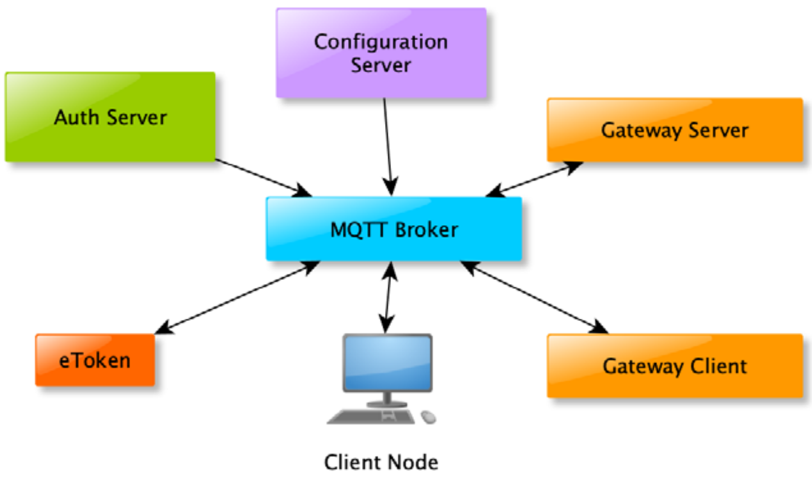

Figure 2. Message Queuing Telemetry Transport (MQTT) interconnects all the elements.

3.3. *Algorithms Used*

The algorithms we use are mainly based on: one-way hash functions, elliptic curve cryptography, Elliptic Curve Integrated Encryption Scheme (ECIES), and true random number generators (TRNG). In order not to extend the length of this work, the mathematical base of these algorithms is not included, since it is described in the bibliography here included.

3.3.1. Hash Functions

One-way hash functions are widely used in cryptographic systems [38,39]. They offer a deterministic value for an input sequence. Given an output value, it is computationally impossible to find the original input value.

By default, the system uses the SHA-256 hash function that is considered cryptographically robust, but it can work with SHA-384 and SHA-512 as well. According to Lu et al. [40], some implementations of SHA-256 are vulnerable to attacks, so it is recommended to use stronger hash functions. In any case, other secure one-way functions could be implemented and used thanks to the reprogramming capability of the eToken.

3.3.2. Elliptic Curve Cryptography

Public key algorithms grounded on elliptic curve are also frequently used in robust cryptographic communication systems such as Transport Layer Security (TLS) [41]. In particular, the Elliptic Curve Digital Signature Algorithm (ECDSA) [4] is a variant of the Digital Signature Algorithm (DSA) based on elliptic curve cryptography. These algorithms are an appealing alternative that are replacing Rivest, Shamir, and Adleman (RSA)-based systems, in part because smaller key sizes are used in the elliptic curve to provide an equivalent level of security. In symmetric key algorithms, there is a direct correspondence between the level of security and the size of the key used. In contrast, in the asymmetric key algorithms it can change depending on the algorithms used. Table 2 shows a comparison of different key sizes for the RSA and elliptic curve algorithms and their references according to the Espressif IoT Development Framework [42].

Table 2. Comparison of key size in Rivest, Shamir, and Adleman (RSA) and Elliptic Curve Digital Signature Algorithm (ECDSA).

RSA Key Size	ECDSA Key Size	ESP_IDF Curve
1024 bits	160–223 bits	secp192r1, sec192k1
2048 bits	224–255 bits	secp224r1, sec224k1
3072 bits	256–383 bits	secp256r1, secp256k1, bp256r1
7680 bits	384 bits–511 bits	secp384r1, bp384r1
15360 bits	512 $\geq$ bits	sepc512r1, bp512r1

One difference between RSA and ECDSA is that RSA allows signing and encryption, while ECDSA only allows signing. Alternatively, elliptic-curve Diffie–Hellman (ECDH) is based on the Diffie–Hellman algorithm [43] using elliptic curve and allows the exchange of a safe value between two elements that can be used later in symmetric encryption between both.

The Standards for Efficient Cryptography Group recommends the domain parameters for each curve [44]. The signature function used is based on any of the currently used elliptic curves implemented in MBED TLS [45] such as:

- FIPS 186-4: secp192r1, secp224r1, secp256r1, secp384r1, secp512r1;
- Brainpool: bp256r1, bp384r1, bp512r1;
- Koblitz: secp192k1, secp224k1, secp256k1.

We can find some implementations of elliptic curve algorithms as described by Liu et al. [46] using a TI MSP430. It is a 16-bit microcontroller with a top speed of 25 MHz. Our low-cost eToken, based on a dual core 32-bit microcontroller at 240 MHz, outperforms this implementation with reduced execution times even with larger bit sizes. In Section 4.1, we study the times of different functions for these elliptical curves used in the eToken.

3.3.3. Elliptic Curve Integrated Encryption Scheme

Elliptic Curve Integrated Encryption Scheme (ECIES) is a hybrid system that combines symmetric encryption with elliptic curve-based public key cryptography for secure message delivery. The secret key used for symmetric encryption is hidden using a key derivation function (KDF) and a key agreement (KA) key exchange function (ECDH). In Reference [47], it is shown in detail how these schemes work. Figure 3 illustrates how a session key *(SKk)* is encrypted using this scheme.

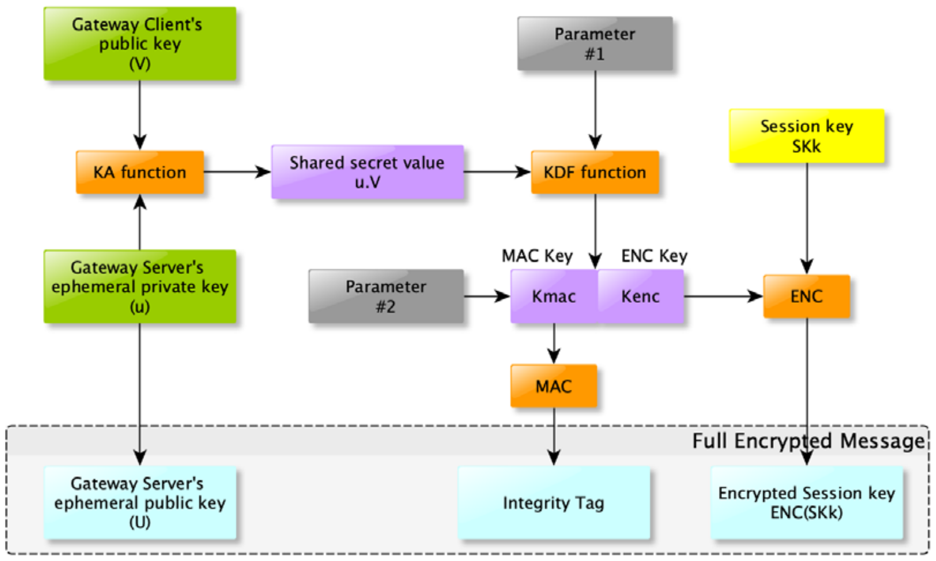

Figure 3. Scheme used to encrypt a session key.

3.3.4. True Random Number Generator

The ESP32 includes a hardware random number generator. If Bluetooth or WiFi are enabled (as in our case), it uses RF noise as an entropy source and values obtained can be considered true random numbers. Besides, random number generator functions used by all the elements pass the Dieharder random number test suite [48].

3.3.5. Combining Algorithms

The hash function is calculated for all sent messages and signed with the private key by the sender element so that the receiver can validate it.

Since the system is based on public key, it is not necessary to store symmetric encryption keys that could compromise the security of the system. Furthermore, authentication does not require multiple messages between client and server as authentication is verified at each step.

We use the notation shown in Table 3 to identify elements and primary functions used during different stages. Each of the phases involved is described below.

Table 3. Relation of the notation of the components used.

Symbol	Meaning
SKk	Sesion key
Clc	Client
Sn	Sesion
Ui	User
Tt	Token t
Tt2	Possible 2nd token
Sj	Server
GWCgc	Gateway client gc
GWSgs	Gateway server gs
Pub(Ei)	Public key element i
Priv(Ei)	Private key element i
ESk	Ephemeral session key
AS	Authentication server
SVCs	Service
SEQq	Sequence number
CS	Configuration server
h()	Hash function
Sg(hash,privK)	Sign function hash with priv key privK
Vf(msg,pubK)	Verify function msg with public key pubK

3.4. Resource Registration

The system works under a trust model based on a public key scheme, so it is essential to store the secret keys safely and correctly. To create a new configuration, we define three steps.

3.4.1. Configuration Server Registration

Initially, we initialise the configuration server. This is the only one that can modify the system configuration of any system resource. It is responsible for:

- Creating a public and private key pair to sign the configuration messages. The public key is installed on all elements and is, therefore, the only one authorised to make changes;
- Creating the root certification authority to sign the certificates used in accessing the MQTT broker. The public key of this certification authority (CA) must be installed on all computers and eTokens to establish secure TLS communication between the elements.

3.4.2. MQTT Broker Registration

Communications between the different elements are made through MQTT messages using secure TLS-based communications and avoiding possible man-in-the-middle (MitM) attacks.

This MQTT broker needs a certificate that must be signed by the CA, that is, by the configuration server. MQTT communications can be based on two models:

- Using only the MQTT server certificate, which is verified by clients who must have the public key;
- Using client and server certificates both signed by the CA.

3.4.3. Registration of Functional Elements

All functional elements (servers, clients, users, and eTokens) have to follow the next step to be included in the authentication system:

- Generate a unique 128-bit UUID;
- A public/private key pair to sign the messages. The private key must be stored safely in each of them;
- Each element has a configuration profile created by the configuration server to access the MQTT broker with a user defined for each one. This control mechanism allows cancelling access if necessary, by deleting the user associated with that element;
- Each resource sends its UUID and public key to the configuration server. Thus, each element has a set of authorised public keys. So, any request that does not come from any of these keys is directly discarded.

In the case of the eToken, the USB connection can act as an initial gateway to configure it, and the initial authorisation is only possible when the device deletes all the keys or is completely restarted, leaving it empty.

The configuration server must also store the information safely, mainly to prevent false elements from being modified or added. Thus, when the system starts, the configuration server generates the configuration of the nodes to connect using the MQTT broker.

When the configuration is changed, the configuration server sends the changes to the elements that update their information. All elements only need to store the basic configuration and keep it correctly updated.

In the case of a public cloud, we can have the authentication server and the configuration server in a safe place, outside the cloud if necessary, so we can physically implement some additional security measures.

3.5. Service Request

Once the configuration is established, the system is now available to establish communications securely. Thus, the service request process can be divided into five steps.

3.5.1. The User Requests eToken to Access Service

The client creates a new session (*Sn*) identified with a UUID, with a sequence number (*SEQq*) to the service (*SVCs*), and with the corresponding request hash validation signatures. This request is signed by the user (*Ui*) (1) and by the client node (*CLc*) (2) requesting access:

$$\text{Sgn\_user}(Sn) = Sg(h(Sn, SEQq, Ui, CLc, SVCs), Priv(Ui)) \tag{1}$$

$$\text{Sgn\_client}(Sn) = Sg(h(Sn, SEQq, Ui, CLc, SVCs), Priv(CLc)) \tag{2}$$

A robust hash function is used to get a unique value for each request. In addition, each request has a serial number, which has two advantages:

- They are different messages with different signatures;
- The number is incremental, so the same number cannot be used again.

When an element starts, the configuration server sends a random number from which the count continues. If the number is less than the previous value or greater than a small window, it sends an access revocation message informing the configuration server, so the element must identify itself again.

The new request for access to a service is sent to the eToken (*Tt*). This is an important difference from the U2F model, where the hardware key does not identify the user. Thus, an eToken that is not recognised by an authorised user will reject the request.

3.5.2. The Etoken Requests Access to the Authentication Server

After a power-up condition or after a period of time, the user has to log into the eToken to enable it sending a password as an extra security level.

The eToken has a list of authorised public keys so it can confirm if a user (*Ui*) from a node (*CLc*) has access to the system. This simplifies the configuration of the eToken since it only requires the list of authorised public keys.

The eToken verifies the signature from the node (*CLc*) using *Vf (msg, pub (CLc))*, and if it is valid, it displays on the screen the request for the user to validate it by pressing a button or by using the fingerprint sensor if it is enabled. The times required for signature validation are shown in Section 4, based on the selected elliptic curve.

Additionally, the eToken can verify the context; that is, it can detect which Bluetooth or Wifi resources are nearby. It must detect a valid environment to authorise. This operation can be remotely activated by the configuration server which can control which resources are valid.

Since the session identifier (*Sn*) is unique using a 128-bit UUID, the information is not repeated on every signature. It is only necessary to sign *Sn*.

Once authorised by the user and the context, the eToken signature is added (3):

$$\text{Sgn\_eToken}(Sn) = sg(h(Sn), Priv(Tt)) \tag{3}$$

If we define a configuration with multi-user and multi-eToken authentication, the request authorised by the first eToken is sent to the second eToken to be validated by the second user. In this case, an additional signature of the second eToken is also included (4).

$$\text{Sgn\_eToken2}(Sn) = sg(h(Sn), Priv(Tt2)) \tag{4}$$

This request is sent to the authentication server for global authorisation and to check all ACLs.

3.5.3. The Authentication Server Requests Access to the Gateway

The eToken sends the request to the authentication server. This has all the ACL rules that allow deciding if the *Ui* user from the *CLc* node with the eToken *Tt* has access to the *SVCs* service. Possible rules are described later in Section 3.8.

The authentication server verifies all signatures so it can confirm that is a valid request. Then, the authorisation process checks the ACL rules and, when it validates the request, it signs with its private key (5):

$$\mathrm{Sgn\_AS}\ (Sn) = sg\ (h\ (Sn),\ Priv\ (AS)) \tag{5}$$

This request is sent to the gateway server to enable access.

3.5.4. The Gateway Server Enables the Service

Create a random 256-bit session key (*SKk*) that will be valid for a configurable time that by default is set to 60 s. Since the server is the one that generates the session key, there is no need to timestamp, so it avoids the problem of time synchronisation between nodes.

The server opens the communication port that allows access to the service and sends the *SKk* session key using the ECIES hybrid encryption scheme (6) to the gateway client that is integrated with the initial service request system.

$$\mathrm{GwMsg} = ECIES\_Crypt\ (SKk,\ Pub\ (GWCgc)) \tag{6}$$

The message includes an ephemeral *ESk* key for a secure exchange using ECIES. The gateway client receives the message (6) to obtain the *SKk* session key (7).

$$SKk = ECIES\_Decrypt\ (GwMsg,\ Priv\ (GWCgc)) \tag{7}$$

3.5.5. Communication through the Gateway

When the client receives the session key *(SKk) (7)*, it requests communication with the server through the gateway.

The gateway server checks the session key in the list of valid keys and decides whether the connection is accepted. Invalid connections are rejected after a few seconds, preventing a brute force attack. If the session key is valid, it establishes communication with the service and the session ID (*Sn*) is included in the list of active communications.

If the authentication server receives a communication revocation request, it can send the message to the gateway server to immediately cut the communication. This revocation can come from the global supervisor.

The Figure 4 shows the message sequence chart (MSC) between the different elements:

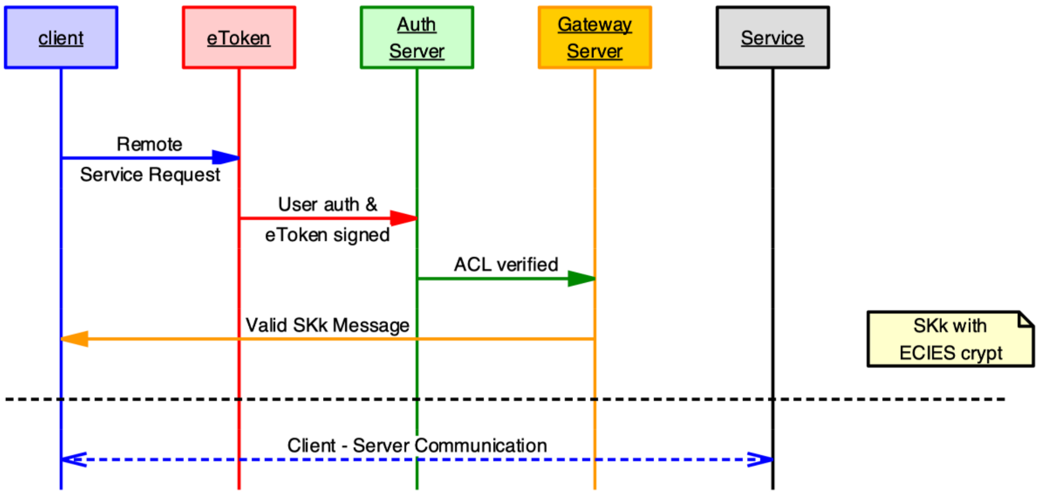

Figure 4. Message sequence chart (MSC) between the elements of the system.

3.6. Multiservice Authorisation

One of the advantages of this model is that it allows multiple communication channels to be opened simultaneously with authentication.

In this case, the client requests the service as usual. Thanks to the MQTT model, the authentication server sends multiple requests to different gateway servers. Thus, each server forwards an opening message from its communication channel, confirming client access for TCP sockets.

Suppose a Hadoop-based big data application where a client accesses multiple Hadoop servers. The client creates multiple requests for accessing the servers. In this case, multiple session keys *SKk1*, *SKk2*, ... are created, one for each gateway client that accesses the system.

Sessions can be kept open for a time, which avoids having to authenticate repeatedly. Figure 5 shows how the system performs a multiservice request with multiple servers. Yellow messages have the session keys that the client receives to access to different services.

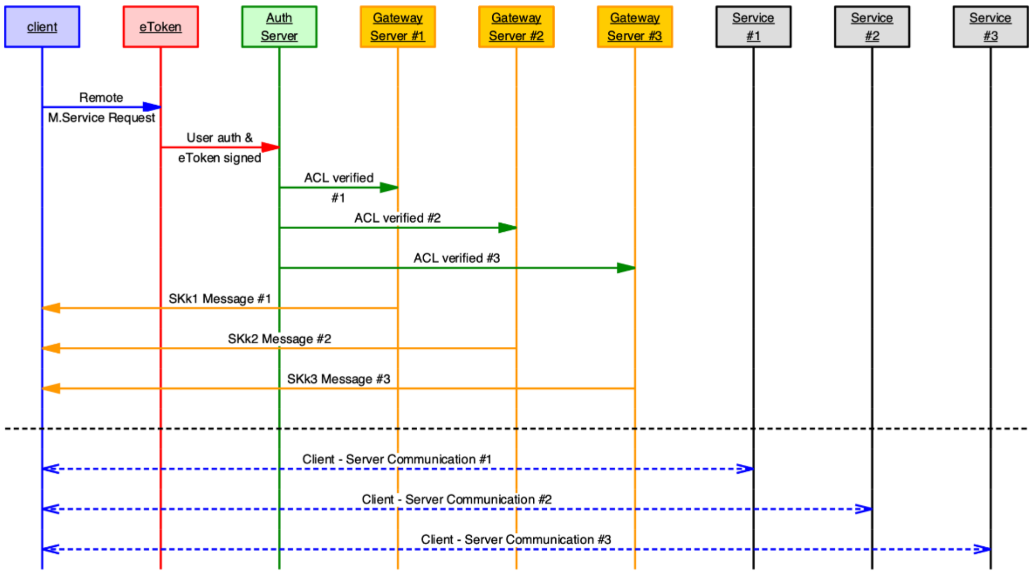

Figure 5. Message sequence chart between the elements of the system in a multiservice request.

3.7. Message Format

The messages are sent in JSON format to facilitate communication through MQTT and are processed quickly by the elements.

Every request has a sequence number that increases with each request. The verification confirms that the number must be greater than the last one received and within a window of 16 values. Values outside this window are considered anomalous.

One of the configuration parameters is the level of sequence control. In the relaxed security model, a warning message is generated to the administrator. In strict mode, the administrator must accept the restart of the numbering.

Thus, after each verification and authorisation, the elements add new JSON fields with the corresponding signature.

Since each element has its UUID, it is used to distribute the information in the topic hierarchy used in the MQTT broker. Each service request also has a unique UUID, so there is no overlap in uniquely determined requests, and there is no collision in messages sent through MQTT. The MQTT broker is in charge of sending the corresponding message to each node.

The primary communications are made using the following topic scheme:

anb/domain/dst_element_uuid/request/src_element_uuid/

Thus, the element that receives the messages has access to all the subtopics:

anb/domain/dst_element_uuid/

Instead, the sending item only has access to the subtopics:

anb/domain/dst_element_uuid/request/src_element_uuid/

The configuration server is responsible for managing the MQTT broker configuration, MQTT users, and topic access rules in the MQTT broker. It also informs the elements of the new topics they must subscribe, allowing the setting to change dynamically.

3.8. Definition of Access Rules

Flexible rules for accessing services can be established based on various combinations of the clients, users, and eToken nodes. Groups of elements can be defined to facilitate configuration. The YML configuration file consists of three main parts:

- Element definition (authentication process);
- Group and combined element definition (authentication process);
- Access rule definition (ACL) (authorisation process).

Each element is identified by an alias: its UUID and the name of the file that contains the public key in PEM format. New aliases can be created and they can contain any combination of other aliases with the "AND" and "OR" operators.

In the access rules, on the one hand, there are the elements that access: users, client nodes, and eTokens, and on the other hand, the elements access: servers and services.

It is possible to set any combination of rules that are evaluated in order to authorise access to a service on a server. As previously mentioned, multiple eTokens can be combined to define a cascading authorisation in which two users must authorise access to a service, increasing security in the system.

Authentication and authorisation processes work independently, and if necessary, can be separated in different servers and contexts. YML can be divided into each part, but for practical reasons, client requests can be solved by the same server.

4. Security Analysis

This section studies the resistance of different elements in the proposed system to frequent attacks in authentication environments.

4.1. Resistance in the MQTT Broker

In the MQTT-based message delivery model, only the MQTT broker has a listening TCP port, and all the elements connect to it using two-way TCP sockets. This design allows that the elements do not have a listening port, and therefore there is no direct way to get into them.

In any case, it is the MQTT broker that must be more robust against attacks. The MQTT protocol is quite secure and reliable; although MQTT can be considered robust because several MQTT broker implementations have their security well evaluated, the way it is implemented and configured can cause issues.

In the proposed model, we use MQTT with certificates, TLS communications, and users with access rules, which offers several advantages:

- Communications are encrypted;
- Only certificates signed by the CA are accepted. In our case, the certificates are generated by the configuration server;
- MQTT users for each element allow communication control;
- ACLs are used to access the topics of the MQTT broker so that each element has limited access only to its information.

All these resources add an extra security level to the communication between all the elements. Even if the MQTT broker is compromised, and an attacker has access to read all messages or can send new ones, the system is still reliable because:

(1) If the attacker can read all messages, they know the service requests from the users, but these messages have irrelevant information. Related to the session key *SKk*, it is encrypted using the ECIES scheme, so the attacker cannot get it to access to the gateway server in a later phase. *SKk* can only be decrypted with *Priv(GWCgc)*.

(2) If the attacker can send messages, they also need a valid private key *Priv(Ei)* to sign their false requests, but all messages with invalid signatures are rejected.

4.2. Tamper Resistance of the EToken

The eToken stores the private key and authorised public keys in its ESP32 flash memory through the non-volatile storage (NVS) library and using 256-bit AES-XTS-based NVS encryption.

ESP32 uses internal partition tables for flash memory. Since the partition is marked encrypted and the flash encryption option is enabled, the bootloader will encrypt this partition using the flash encryption key on the first boot.

When ESP32 runs the eToken program, it saves the information in encrypted form, so only the program can correctly interpret the content. So, any external attempt to modify the configuration in the eToken NVS is not possible.

If ESP32 is partially reprogrammed to try later to access the flash partition that stores the keys, it is not readable since it is encrypted.

Changing the ESP32 configuration is only possible if it receives the messages signed by the configuration server.

4.3. Theft Resistance of EToken

In the proposed scheme, if an eToken is stolen by an adversary A, the latter cannot extract any information from the memory of the stolen devices, as described in Section 4.2.

Moreover, a stolen eToken can be locked at various levels:

- After a power-on condition or an established time, a password is needed to login into the eToken and enable it. After several failed tries, the device is blocked;
- Context validation: (When it is active) The eToken only works within the detection range of validated WiFi or Bluetooth devices. So outside of its location, the eToken is not active;
- On the MQTT server: deleting the access account;
- On the authentication server: cancellation of authentication permissions;
- In the eToken WiFi access: The eToken can only communicate with the preconfigured WiFi networks.

The eToken can be reprogrammed, but as mentioned in the previous section, the operational configuration is encrypted and cannot be used to access the secure environment.

4.4. Resistance of MitM Attacks

MQTT establishes communications between nodes through TLS communications. The MQTT broker and MQTT clients use certificates signed by the CA, thus preventing a node from impersonating the MQTT broker.

TLS includes integrity checks, so communication at the TCP/IP level cannot be modified. Even, if the MQTT is compromised, messages cannot be modified because they are signed with the respective private keys in each stage.

Related to the session key *SKk*, it cannot be obtained by the attacker because even if he can obtain the message, it is encrypted with the ECIES scheme, and a private key *Priv(GWCgc)* is needed to decrypt.

The sequence number *SEQq* avoids the use of previous requests, so it needs to create new ones adequately signed.

In the case of communication between gateways, if the service offers secure communications such as SSH, it is the service itself that can detect a MitM attack, thus avoiding adding an additional layer of integrity check that could slow down communications.

4.5. Resistance of Attacks to the Client Node

Suppose a system with an SSH client where we use public/private key to access a server. If an attacker accesses the client node, they could access the SSH server in a typical configuration, but with the proposed system, access to the SSH server requires the eToken authorisation.

Furthermore, the proposed model can use a private key for the client node and a private key for the user. The administrator can select what keys can be used, one of them or both, if needed.

To tighten the private key to the client node, the client application stores the node private key encrypted with a key that is partly derived from CPU properties, such as: model, family, cache lines, and extensions. So, this private key can work only in a platform with the same physical characteristics, which makes it difficult for the private key to be used in another client node.

Thus, if an adversary accesses the client node, they could use the private key, but they need to know the password to enable the eToken and have to access it to validate the service request. If the user has the eToken in his possession, he will not authorise the request, and therefore access to the service is not possible.

Credentials used to access MQTT are encrypted, so the attacker does not have direct access to the MQTT broker. Further, due to the segmentation of the topics used, this node has access to a limited number of topics, so the MQTT is not compromised.

Likewise, if there is an attack on the client node copying the private key, this alone is not sufficient to access the server, since eToken authorisation is required.

4.6. Resistance of Attacks on the Authentication Server

The authentication server is a vital element because it decides if a request is valid, and it can request to GS a new connection. In the proposed model, the AS is safer than other systems because it does not need to have open ports outside. This is a significant difference because it is protected for direct attacks from the outside. AS must be properly isolated, as it is responsible for admitting authorisations. The proposed system has two advantages:

- Requests arrive through MQTT messages, so the authentication server does not have an explicitly open port for receiving requests.
- This model allows having the server in a different and much safer location.

Thus, in an HPC configuration, the authentication server may be in another location with a higher level of security. Likewise, in a cloud configuration, the instances can be virtualised, and the authentication server can be outside the cloud, in a secure physical environment.

4.7. Resistance of Attacks on the Configuration Server

The configuration server, like the authentication server, is an essential element since it is the only one that establishes the chain of trust in all the elements. So, it is recommended that it should be placed in a safe location.

Likewise, the configuration server communicates by MQTT messages, so it does not have an open port to receive attacks either. It stores its private key encrypted with a key that is partly derived from CPU properties, as clients do with their private keys, and partly from the administrator password. Moreover, the administrator can use his own eToken to validate modifications in the global configuration, so a direct attack without that eToken is not possible.

4.8. Resistance of Attacks Accessing Gateways

Gateway servers only accept requests through MQTT messages signed by AS and have ports closed. Only when access between gateways is requested, the access port on the server is temporarily opened.

The session key *SKk* guarantees that only a client with that value can establish a communication with the server. The ECIES scheme allows to send it from the gateway server to the gateway client in a safe way. Neither the administrator nor configuration server can access that value.

This port can remain fixed to facilitate firewall rules, or it can be dynamic, that is, in different requests, it can open different ports, so an attacker would not know which port to use to access the system. This information is transmitted in the message that the gateways have exchanged.

When the gateway server receives a request, the port listens for a limited time, and after receiving the start of a communication, the session key *SKk* is verified. If it is incorrect or not received in a while, the communication is closed.

4.9. Resistance of Denial of Service Attacks

As previously mentioned, there are only two entry points: the MQTT broker and the gateway server that is only open after a request.

If the gateway server is open for a while because a service has been requested, the server waits for the session key *SKk*, and if it is not valid, it closes the connection after a few seconds, avoiding repeated access to the system. So, the element that can bear the most pressure is the MQTT broker, but these servers are also designed to avoid denial-of-service (DoS) attacks. Additionally, other measures can be taken at the firewall level to limit the number of access attempts per second, or block IP addresses after some failed access attempts.

4.10. Resistance of Network Outages

Although current local networks are very robust, they can experience unexpected outrages during short periods. It is also essential to consider these scenarios to avoid possible attacks in conditions of network instability.

All communications with the MQTT broker use TLS, so an attacker cannot take advantage of sending false messages due to the integrity included in TLS communications. MQTT is a protocol quite robust, even under unreliable networks. In our case, all the elements, including the authentication and configuration servers, are MQTT clients and include auto reconnection, so as soon as the network re-establishes, they are ready to transmit and receive. The system uses quality of service (QoS) 1 in MQTT messages, which guarantees that a message is sent at least once to the receiver, so packets are not lost, and the sender knows if it has to retransmit the message due to loss of connection.

4.11. Real-Time Selective Block Communication

The proposed system makes it easy to control all the sockets that are in use. If unauthorised access is discovered, it is possible to cut any of the communications that are in progress, thanks to the real-time control of all communications. If necessary, the configuration server can send updated configuration messages blocking ACL access rules and preventing subsequent access to the system to the element involved.

In other systems, once access is authorised, the communication is established. To cancel the communication, the socket that has the communication has to be identified, or detect the IP address and port number to block the connection with firewall rules.

5. Performance Analysis

In our tests, we have studied the runtime of the main cryptographic functions used in different environments, and the impact of the gateway in some data transfers.

5.1. ECC Execution Times in the EToken

The eToken is based on ESP32, a low-cost system that incorporates a 240 MHz dual-core processor, with cryptographic acceleration resources and integrated communications.

An important issue is knowing if this device could process the required elliptic curve cryptographic functions at a reasonable time.

We have evaluated different elliptic curves with the MBed TLS library. Table 4 shows the runtimes obtained for key generation, signature, and verification.

Table 4. Key generation, signature, and verification runtimes for various elliptic curves on the ESP32.

Elliptic Curve	GenKey (ms)	Sign (ms)	Verify (ms)
MBEDTLS_ECP_DP_SECP192R1	112	52	175
MBEDTLS_ECP_DP_SECP224R1	140	67	231
MBEDTLS_ECP_DP_SECP256R1	224	95	347
MBEDTLS_ECP_DP_SECP384R1	331	129	497
MBEDTLS_ECP_DP_SECP521R1	574	229	842
MBEDTLS_ECP_DP_BP256R1	2057	726	2784
MBEDTLS_ECP_DP_BP384R1	3792	1132	4990
MBEDTLS_ECP_DP_BP512R1	6897	2067	9361
MBEDTLS_ECP_DP_SECP192K1	163	69	239
MBEDTLS_ECP_DP_SECP224K1	193	85	311
MBEDTLS_ECP_DP_SECP256K1	230	99	349

The SECP256R1 curve offers reasonable response times below 0.35 s for verifying and 0.1 s for signing.

5.2. Execution Times of ECDSA Functions

Various architectures have been evaluated to obtain the runtimes of key generation, verification, and signing key of a 256-bit hash value used in different client and server functions. We prefer this way to analyse because our model can be implemented with any combination of elements placed in different architectures and operating systems. Tables 5–8 show the average times obtained in microseconds using the Golang ECIES library [49] with different processors, such as: Intel i7-4980HQ @ 2.80 GHz, Intel Xeon Silver 4116 CPU @ 2.10 GHz, AMD EPYC 7571 @ 2.4 GHz, and ARMv7 on Raspberry PI 4 model B.

Table 5. Elliptic curve cryptography (ECC) execution time on Intel i7-4980HQ @ 2.80 GHz MacOS 10.15.4.

Elliptic Curve	GenKey (µs)	Sign (µs)	Verify (µs)
Elliptic.P224	763	804	1751
Elliptic.P256	17	59	114
Elliptic.P384	4417	4724	8430
Elliptic.P521	7594	7955	15,903

Table 6. ECC execution time on Intel Xeon Silver 4116 CPU @ 2.10 GHz Centos 7.7 kernel 3.10.0.

Elliptic Curve	GenKey (µs)	Sign (µs)	Verify (µs)
Elliptic.P224	2478	2327	4406
Elliptic.P256	59	182	309
Elliptic.P384	14,401	12,176	27,282
Elliptic.P521	27,484	29,694	50,466

Table 7. ECC execution time on AMD EPYC 7571 @ 2.4 GHz RHEL 8.2.

Elliptic Curve	GenKey (µs)	Sign (µs)	Verify (µs)
Elliptic.P224	1178	1228	2353
Elliptic.P256	26	65	147
Elliptic.P384	6315	6613	12,953
Elliptic.P521	10,854	11,271	20,077

Table 8. ECC execution time on Raspberry PI4b ARMv7 Raspbian GNU/Linux 10.

Elliptic Curve	GenKey (µs)	Sign (µs)	Verify (µs)
Elliptic.P224	5798	5937	7935
Elliptic.P256	2329	3525	9015
Elliptic.P384	11,2167	117,424	229,602
Elliptic.P521	211,553	209,649	416,941

In general, the elliptical curve P256 requires less processing time. Besides, Intel and AMD architectures have considerably lower execution times due to hardware optimisations.

5.3. ECIES Execution Times on Diverse Clients and Servers

Server and client gateways exchange a 256-bit secret key (*SKk*) to guarantee later access between both elements. This key is sent securely with a message with the ECIES scheme.

Table 9 shows the encryption and decryption runtimes of the 256-bit token using the ECIES library [28] with Golang. In this case, the secp256k1 curve is used in the ECIES functions related to the private/public key functions. We have evaluated different architectures.

Table 9. Elliptic Curve Integrated Encryption Scheme (ECIES) execution time on different client and server platforms.

Platform	GenKey (ms)	Crypt (ms)	Decrypt (ms)
Intel i7-4980HQ @ 2.80 GHz MacOS 10.15.4	2.55	4.54	2.39
Intel Xeon E5-2676 v3 @ 2.40 GHz Amazon Linux 2 kernel 4.14.173	3.03	5.79	2.92
AMD EPYC 7571 @ 2.4 GHz RHEL 8.2	3.14	6.34	2.83
ARM 64 bits AWS Graviton @ 2.3 GHz	4.48	8.62	4.42
Intel Xeon Silver 4116 CPU @ 2.10 GHz Centos 7.7 kernel 3.10.0	8.26	13.97	6.62
Raspberry PI4	61.93	86.06	43.21

5.4. Processing Time

As we have seen in the previous tables, the average time depends of the selected resources used. It also depends on the network topology. In order to obtain an estimated global runtime, we have used a configuration based on Intel Xeon E5-2676 v3 @ 2.40 GHz, with the MQTT broker Eclipse Mosquitto [50] version 1.6.9, widely used in Linux and other platforms, although this model allows the use of any other MQTT broker compatible with the 3.1 standard.

Table 10 shows the average times for each of the stages described in Section 3.5 with a 1 Gbps network interconnection.

Table 10. Mean time for different stages.

Stage	Average time
First request → eToken	57 ms
eToken sign verification	347 ms
User validation	Wait for user validation (touch)
eToken → authentication server	4 ms
Sign validation and ACL check in auth. server	1 ms to 5 ms (depends of ACL rules)
Send to gateway server	0.67 ms
Validation and gateway server opening	0.75 ms
ESCIES crypt, send, and decrypt session key	9 ms
Client–server gateway connection	0.1 ms

Although the process involves several stages, and the signature verification time is longer in ESP32, the central time is due to waiting for the user to validate the access. In general, authentication times are reasonable in user context and server requirements.

For these measurements, the elliptical SECP256R1 curve has been used, which offers a high security level and optimal execution times.

One of the advantages of this model is that it can work in parallel so that multiple authentication processes can be established simultaneously.

5.5. Direct Bandwidth and Latency Overhead

In this case, we use two programs written in Golang: a TCP echo, which retransmits everything it receives, and another program that transmits and receives packet sequences of different sizes. In this way, latencies and bandwidths can be evaluated considering symmetric communication. Table 11 shows the roundtrip time divided by 2 for different packet sizes with direct connection and compared using the client and server transparent gateway.

Table 11. Transfer time on Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10 GHz, CentOS 7.7.

Packet Size (Bytes)	Direct Connection (Time in µs)	Connection through Gateways (Time in µs)	% Effective BW
1	77	148	52
10	136	256	53
100	202	369	55
1,000	295	514	57
10,000	495	762	65
100,000	1148	1490	77
1,000,000	7491	8424	89
10,000,000	70,140	75,519	93

5.6. Impact on SSH Communications

We have also studied the impact on SSH communications. In our case, we have defined an SSH tunnel with the TCP echo program at the end. We have measured the roundtrip time divided by 2, as the previous test with direct connection and using our gateway.

Table 12 shows that we get an effective bandwidth better than 80% in all cases using the gateway with SSH communications.

Table 12. Secure Shell (SSH) tunnel round trip time on Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10 GHz, CentOS 7.7.

Packet Size (Bytes)	Direct Connection (Time in µs)	Connection through Gateways (Time in µs)	% Effective BW
1	169	205	82
10	283	350	81
100	406	493	82
1000	632	718	88
10,000	1043	1207	86
100,000	2160	2450	88
1,000,000	13,797	15,785	87
10,000,000	126,017	127,589	99

5.7. Impact on Accessing Data in Hadoop HDFS

Some HPCs use Hadoop for big data processing and analytics computing. We have studied the impact of accessing data stored in a Hadoop cluster using HDFS. Hadoop is in version 3.2.1, and the computer nodes have Intel E5-2620 v4 @ 2.10 GHz, CentOS 7.7, 500 GB HDDs and are connected with a 1 Gbps network. The client node uses a Golang client for HDFS [51].

5.7.1. Metadata Access

We have tested four basic functions: create file, create directory, remove file, and remove directory.

All functions have almost the same execution time and vary from 3.37 ms to 4.01 ms depending on the server load and internal Hadoop processes. The penalisation due to the gateway is only 0.05 ms, that is practically hidden in the normal fluctuation of the times obtained. It only represents 1.25%. So, the gateway has a reduced impact on these operations.

5.7.2. Data Access

As we have mentioned before, Hadoop has some fluctuations in execution time due to how it handles internal flush buffers and its disk access policy. In this case, we have executed each test 100 times and obtained the median value to avoid extreme values. The execution time includes: create/open, write/read, and close operations.

Table 13 shows the execution time obtained with these tests for direct connection and using the gateway. We can observe that the times obtained for a file with sizes from 1 byte up to 10,000 bytes are almost the same, due to remote functions overhead, so buffers cannot work properly and optimise their data transfers.

Table 13. Execution time for reading and writing access to Hadoop HFDS.

Packet Size (Bytes)	Read Direct Connection (Time in µs)	Read through Gateways (Time in µs)	Write Direct Connection (Time in µs)	Write through Gateways (Time in µs)
1 10 100 1000 10,000	1617	1837	19,916	21,014
100,000	2925	2977	19,929	21,104
1,000,000	10,578	10,949	27,913	29,849
10,000,000	87,488	88,157	109,151	110,587
100,000,000	861,559	861,737	906,371	907,872
1,000,000,000	8,581,931	8,681,764	9,625,042	9,938,063

We can observe that the impact of the gateway is reduced with different file sizes, and read and write operations.

5.8. Discussion

It is necessary to analyse the impact of the main cryptographic functions implied to determine if the proposed model can work in a real HPC environment and the overhead of the gateways. As already discussed, our model can be implemented with any combination of elements placed in different architectures and operating systems.

The results obtained confirm that the functions used in client and servers based on elliptic curve cryptography have a reduced execution time on different architectures. Moreover, our eToken can also execute them in tenths of seconds.

Concerning the gateways, they simplify the integration process with different programs and services. In general, they have a reduced overhead as we can observe in the communications test performed.

6. Conclusions

The use of security keys is as proliferating as multifactor authentication systems for secure system access. The advantage of these systems is that the user "physically" has a non-duplicable element to authorise access. Although there are standards such as those proposed by FIDO, these are subject to possible improvements.

In this work, we present a different approach to the authentication process that can be applied to HPC and cluster environments. Our proposed model is based on ESP32, a low-cost system-on-chip that allows the execution of cryptographic functions based on elliptic curve and various communication interfaces. In combination with this eToken, we propose a communication system that transparently allows authentication in client–server programs, facilitating integration in current systems. This system offers an extra level of security, flexibility in configuration, scalability in distributed systems, reduced time in the authentication process, and the gateway has a reduced impact on cluster communications.

We continue improving our system adding compatibility with other authentication methods like the Linux Pluggable Authentication Module (PAM) or authentication protocols.

Author Contributions: Conceptualization, A.F.D. and I.B.; methodology, M.A. and A.F.D.; software, A.F.D. and I.B.; validation, A.F.D., M.A. and J.J.E.; formal analysis, J.O. and A.F.D.; investigation, A.F.D., I.B. and M.A.; writing—review and editing, A.F.D, I.B., M.A., J.O. and J.J.E.; visualization, J.J.E.; supervision, J.O.; project administration, A.F.D. and J.O.; funding acquisition, A.F.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by Spanish Ministry of Science, Innovation and Universities (MICINN), grant number PGC2018-096663-B-C44 together with the European Fund for Regional Development (FEDER).

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

The following abbreviations are used in this manuscript:

ACL	Access Control List
AES	Advanced Encryption Standard
AS	Authentication Server
CA	Certification Authority
CRS	Client Request Service
CS	Configuration Server
CVE	Common Vulnerabilities and Exposures
DLL	Dynamic Link Library
DoS	Denial of Service
DSA	Digital Signature Algorithm
ECC	Elliptic curve cryptography
ECDH	Elliptic Curve Diffie-Hellman
ECDSA	Elliptic Curve Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
ESP_IDF	Espressif IoT Development Framework
FIDO	Fast Identity Online Alliance
FIPS	Federal Information Processing Standard
GC	Gateway Client
GS	Gateway Server
JSON	JavaScript Object Notation
MB	MQTT Broker
MitM	Man-in-the-Middle
MQTT	Message Queuing Telemetry Transport
MSC	Message Sequence Chart
NFC	Near Field Communication
NVS	Non-Volatile Storage
OTP	One Time Password
PAM	Linux Pluggable Authentication Module
PEM	Privacy Enhanced Mail
QoS	Quality of Service
RSA	Rivest, Shamir & Adleman
SAML	Security Assertion Markup Language
SDK	Software Development Kit
SIM	Subscriber Identity Module
SMS	Short Message Service
SSH	Secure Shell
SSO	Single Sign-On
TCP	Transmission Control Protocol
TI	Texas Instruments
TLS	Transport Layer Security
TPM	Trusted Platform Module
TRNG	True Random Number Generators
U2F	Universal Second Factor protocol
UFC	Universal Authentication Framework
UUID	Universally Unique IDentifier
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XTS	XEX Tweakable Block Cipher with Ciphertext Stealing
YML	YAML Ain't Markup Language

References

1. Common Vulnerabilities and Exposures (CVE). MITRE Corporation. Available online: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=authentication (accessed on 15 June 2020).
2. Khalil, I.; Khreishah, A.; Azeem, M. Cloud Computing Security: A Survey. *Computers* **2014**, *3*, 1–35. [CrossRef]
3. Stajano, F. *Pico: No More Passwords! In Security Protocols XIX*; Christianson, B., Crispo, B., Malcolm, J., Stajano, F., Eds.; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2011; Volume 7114, pp. 49–81. ISBN 978-3-642-25866-4.
4. Johnson, D.; Menezes, A.; Vanstone, S. The Elliptic Curve Digital Signature Algorithm (ECDSA). *IJIS* **2001**, *1*, 36–63. [CrossRef]
5. Yildirim, N.; Varol, A. Android based mobile application development for web login authentication using fingerprint recognition feature. In Proceedings of the 2015 23nd Signal Processing and Communications Applications Conference (SIU), Malatya, Turkey, 16–19 May 2015; pp. 2662–2665.
6. Microsoft SMS-based Authentication Using Azure Active Directory. Available online: https://docs.microsoft.com/en-us/azure/active-directory/authentication/howto-authentication-sms-signin (accessed on 15 June 2020).
7. Krebs, B. Reddit Breach Highlights Limits of SMS-Based Authentication. Available online: https://krebsonsecurity.com/2018/08/reddit-breach-highlights-limits-of-sms-based-authentication/ (accessed on 15 June 2020).
8. Ylonen, T. SSH Key Management Challenges and Requirements. In Proceedings of the 2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS), Canary Islands, Spain, 24–26 June 2019; pp. 1–5.
9. Rhea, S.; Johnson, E. Cloudfare: Public Keys are Not Enough for SSH Security. Available online: https://blog.cloudflare.com/public-keys-are-not-enough-for-ssh-security/ (accessed on 15 June 2020).
10. Miller, S.P.; Neuman, B.C.; Schiller, J.I.; Saltzer, J.H. Kerberos Authentication and Authorization System. In *Project Athena Technical Plan*; 1987; Available online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.7727 (accessed on 15 June 2020).
11. RFC4120: The Kerberos Network Authentication Service (V5). Available online: https://tools.ietf.org/html/rfc4120 (accessed on 6 July 2020).
12. Pereñíguez-García, F.; Marín-López, R.; Kambourakis, G.; Ruiz-Martínez, A.; Gritzalis, S.; Skarmeta-Gómez, A.F. KAMU: Providing advanced user privacy in Kerberos multi-domain scenarios. *Int. J. Inf. Secur.* **2013**, *12*, 505–525. [CrossRef]
13. Tsay, J.K. Formal Analysis of the Kerberos Authentication Protocol. Ph.D. Thesis, University of Pennsylvania, Philadelphia, PA, USA, 2008.
14. El-Emam, E.; Koutb, M.; Kelash, H.M.; Faragallah, O.S. An Authentication Protocol Based on Kerberos 5. I. *J. Netw. Secur.* **2011**, *12*, 159–170.
15. Tbatou, Z.; Asimi, A.; Asimi, Y.; Sadqi, Y.; Guezzaz, A. A New Mutuel Kerberos Authentication Protocol for Distributed Systems. I. *J. Netw. Secur.* **2017**, *19*, 889–898.
16. Le, H.Q.; Truong, H.P.; Van, H.T.; Le, T.H. A new pre-authentication protocol in Kerberos 5: Biometric authentication. In Proceedings of the The 2015 IEEE RIVF International Conference on Computing & Communication Technologies-Research, Innovation, and Vision for Future (RIVF), Can Tho, Vietnam, 25–28 January 2015; pp. 157–162.
17. Google Authenticator. Available online: https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2&hl=en_US (accessed on 15 June 2020).
18. Krebs, B. Google: Security Keys Neutralized Employee Phishing. Available online: https://krebsonsecurity.com/2018/07/google-security-keys-neutralized-employee-phishing/ (accessed on 15 June 2020).
19. Dou, Z.; Khalil, I.; Khreishah, A. A Novel and Robust Authentication Factor Based on Network Communications Latency. *IEEE Syst. J.* **2018**, *12*, 3279–3290. [CrossRef]
20. Kang, W. U2Fi: A Provisioning Scheme of IoT Devices with Universal Cryptographic Tokens. *arXiv* **2019**, arXiv:1906.06009.
21. Srinivas, S.; Balfanz, D.; Tiffany, E.; Czeskis, A.; Alliance, F. *Universal 2nd Factor (U2F) Overview*; FIDO Alliance Proposed Standard: Mountain View, CA, USA, 4 September 2015; pp. 1–5.
22. W3C Rec., "Web Authentication: An API for Accessing Public Key Credentials Level 1". Available online: https://www.w3.org/TR/webauthn/ (accessed on 15 June 2020).

23. Yubico: Protect Your Digital World with YubiKey. Available online: https://www.yubico.com/ (accessed on 15 June 2020).
24. Yubico: Security Advisory 2019-06-13. Available online: https://www.yubico.com/support/security-advisories/ysa-2019-02/ (accessed on 15 June 2020).
25. Chadwick, D.W.; Laborde, R.; Oglaza, A.; Venant, R.; Wazan, S.; Nijjar, M. Improved Identity Management with Verifiable Credentials and FIDO. *IEEE Comm. Stand. Mag.* **2019**, *3*, 14–20. [CrossRef]
26. Ciolino, S.; Parkin, S.; Dunphy, P. Of Two Minds about Two-Factor: Understanding Everyday {FIDO} U2F Usability through Device Comparison and Experience Sampling. In Proceedings of the Fifteenth Symposium on Usable Privacy and Security, Santa Clara, CA, USA, 12–13 August 2019.
27. Hardt, D. The OAuth2.0 Authorization Framework. Internet Engineering Task Force (IETF) RFC 6749. Available online: https://oauth.net/2/ (accessed on 15 June 2020).
28. Chae, C.-J.; Kim, K.-B.; Cho, H.-J. A study on secure user authentication and authorization in OAuth protocol. *Cluster Comput.* **2019**, *22*, 1991–1999. [CrossRef]
29. OpenID Foundation OpenID Connect. Available online: https://openid.net/connect/ (accessed on 6 July 2020).
30. Li, W.; Mitchell, C.J.; Chen, T. Mitigating CSRF attacks on OAuth 2.0 and OpenID Connect. *arXiv* **2018**, arXiv:1801.07983.
31. saaspass Multi-factor Authentication (mfa) with OpenID Connect Protocol. Available online: https://blog.saaspass.com/multi-factor-authentication-mfa-with-openid-connect-protocol-d6b64c49c99c (accessed on 6 July 2020).
32. Secsign: OAuth 2.0 Integration. Available online: https://www.secsign.com/developers/oauth-2-two-factor-authentication/ (accessed on 6 July 2020).
33. Khalil, I.; Dou, Z.; Khreishah, A. TPM-Based Authentication Mechanism for Apache Hadoop. In *International Conference on Security and Privacy in Communication Networks*; Tian, J., Jing, J., Srivatsa, M., Eds.; Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering; Springer International Publishing: Cham, Switzerland, 2015; Volume 152, pp. 105–122. ISBN 978-3-319-23828-9.
34. ESP32 Overview. Espressif Systems. Available online: https://www.espressif.com/en/products/socs/esp32/overview (accessed on 15 June 2020).
35. Suárez-Albela, M.; Fraga-Lamas, P.; Fernández-Caramés, T. A Practical Evaluation on RSA and ECC-Based Cipher Suites for IoT High-Security Energy-Efficient Fog and Mist Computing Devices. *Sensors* **2018**, *18*, 3868. [CrossRef] [PubMed]
36. Microchip ATECC608A. Network and Accessories Secure Authentication. Available online: https://www.microchip.com/wwwproducts/en/ATECC608A (accessed on 15 June 2020).
37. PlatformIO: A New Generation Ecosystem for Embedded Development. Available online: https://platformio.org/ (accessed on 15 June 2020).
38. Stinson, D.R. Some Observations on the Theory of Cryptographic Hash Functions. *Des. Codes Crypt* **2006**, *38*, 259–277. [CrossRef]
39. Sarkar, P. A Simple and Generic Construction of Authenticated Encryption with Associated Data. *ACM Trans. Inf. Syst. Secur.* **2010**, *13*, 1–16. [CrossRef]
40. Lu, X.; Wang, W.; Lu, Z.; Ma, J. From security to vulnerability: Data authentication undermines message delivery in smart grid. In Proceedings of the 2011-MILCOM 2011 Military Communications Conference, Baltimore, MD, USA, 7–10 November 2011; pp. 1183–1188.
41. Blake-Wilson, S.; Bolyard, N.; Gupta, V.; Hawk, C.; Moeller, B. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*; RFC4492; 2006; Available online: https://tools.ietf.org/html/rfc4492 (accessed on 15 June 2020).
42. Espressif ESP-IDF Programming Guide. Available online: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/ (accessed on 6 July 2020).
43. Diffie, W.; Hellman, M. New directions in cryptography. *IEEE Trans. Inf. Theory* **1976**, *22*, 644–654. [CrossRef]
44. SEC 2: Recommended Elliptic Curve Domain Parameters. Available online: http://www.secg.org/SEC2-Ver-1.0.pdf (accessed on 15 June 2020).
45. Mbed TLS Core Features. Available online: https://tls.mbed.org/core-features (accessed on 15 June 2020).
46. Liu, Z.; Seo, H.; Hu, Z.; Hunag, X.; Großschädl, J. Efficient Implementation of ECDH Key Exchange for MSP430-Based Wireless Sensor Networks. In Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security-ASIA CCS '15, (ACM), Singapore, 14–17 April 2015; pp. 145–153.

47. Gayoso Martínez, V.; Hernandez Encinas, L.; Sánchez Ávila, C. A Survey of the Elliptic Curve Integrated Encryption Scheme. *J. Comput. Sci. Eng.* **2010**, *2*, 7–13.
48. Brown, R.G. Dieharder: A Random Number Test Suite. Available online: https://webhome.phy.duke.edu/~{}rgb/General/dieharder.php (accessed on 15 June 2020).
49. Elliptic Curve Integrated Encryption Scheme (ECIES) Go Library. Available online: https://github.com/ecies/go (accessed on 15 June 2020).
50. Eclipse Mosquitto: An Open Source MQTT Broker. Available online: https://mosquitto.org/ (accessed on 15 June 2020).
51. Marc, C. A Native go Client for HDFS. Available online: https://github.com/colinmarc/hdfs (accessed on 15 June 2020).