An Architecture for Easy Onboarding and Key Life-cycle Management in Blockchain Applications

RAFAEL GENÉS-DURÁN, *Hardapps Labs SL, UPC*

DIANA YARLEQUÉ-RUESTA, *Hardapps Labs SL*

MARTA BELLÉS-MUÑOZ, *UPF*

ANTONIO JIMENEZ-VIGUER, *SEAT SA*

JOSÉ L. MUÑOZ-TAPIA, *UPC*

22 June 2020

Abstract

New manufacturing paradigms require a large number of business interactions between multiple cyber-physical systems with different owners. In this context, public distributed ledgers are disruptive because they make it possible to securely and publicly record proofs of agreements between parties that do not necessarily trust each other. Many industry leaders have already achieved significant business benefits using this technology, including greater transparency, improved traceability, enhanced security, increased transaction speed and costs reduction. While the benefits of blockchain technologies for industrial applications are unquestionable, these technologies have an inherent complexity that might be overwhelming for many users. To decrease entry barriers for industry users to distributed ledger technologies, it is necessary to have an easy user onboarding process and a simple key life-cycle management. In this paper, we propose an architecture that facilitates these processes and simplifies how users utilize decentralized applications without sacrificing on the expected security. To achieve this goal, our architecture uses a middleware that allows us to decouple the digital signatures required for paying blockchain fees from the ones required for authorization. This approach has the advantage that users are not forced to create wallets, buy cryptocurrency, or protect their private keys. For these reasons, our solution is a promising way of enabling a reasonable transition to the integration of distributed ledger technologies in industrial business processes.

Keywords: Blockchain, Distributed Ledger, DApp, Middleware, Key Life-cycle Management, User Onboarding.

1 Introduction

Distributed ledgers are disruptive because they allow for secure record agreements between parties and devices that do not necessarily trust each other. Using a distributed ledger as a trust anchor can greatly revolutionise Industry 4.0 [1] which is made up of complex production systems of interconnected computers with sensors, instruments, robots and other devices, known together as the Industrial Internet of Things (IIoT) [2, 3]. Industry 4.0 is allowing production to be more flexible and efficient than ever, making it possible to implement individual customer wishes at costs that were previously only viable in mass production. This is thanks to new models that exploit on-demand access to a shared collection of diversified and distributed manufacturing resources to form re-configurable cyber-physical production lines [4]. These new manufacturing paradigms require a large number of business interactions, between multiple systems, belonging to different owners, and with different levels of trust.

The main benefit of using a distributed ledger in Industry 4.0 production models is that trust among parties and devices can be based on the certainty that interactions will be performed exactly as defined in the ledger [5]. Many types of agreements across the manufacturing chain, for example agreements with suppliers, transportation and warehousing providers can be conducted using a distributed ledger as a trust anchor. In this way, agreements can be made faster, for less, and with fewer errors, and yet still carry the authenticity and credibility of regular contracts.

The main technology used to build public ledgers is blockchain. Using a distributed consensus algorithm, a blockchain network creates a single sequence of blocks. The consensus algorithm immutably orders the blocks in question, each of which contains a list of digitally signed transactions. These transactions are then applied by each blockchain network participant in the specified order to deterministically create a copy of the ledger [6].

Many distributed ledgers also provide users with the ability to use smart contracts [7]. Smart contracts are deployed (uploaded) to the ledger through transactions and are used to implement certain business logic. Once a smart contract is deployed, it is possible to modify the ledger's state by sending a transaction to a function of that smart contract. In this case, the smart contract makes the corresponding state changes according to its explicit and immutable logic.

The main advantages of implementing business logic using smart contracts are that, on the one hand, the logic is publicly available and auditable and on the other hand, the logic is immutable and tamper-proof; this guarantees that it will always be executed as defined. As a result, for Industry 4.0, smart contracts can greatly reduce the time needed to complete and register business arrangements between manufacturers and their suppliers because they are binding without the need to go through a formal registration process.

Despite its evident benefits, one of the main problems for widespread blockchain adoption in industrial applications is that it entails an inherent complexity that might be overwhelming for many users. In this context, it is crucial to achieve

an easy user onboarding process in which industrial users are not forced to set up wallets, buy cryptocurrency or protect their private keys.

In this paper, we propose an enterprise-friendly architecture that enhances user onboarding and key life-cycle management when building applications over a general purpose public blockchain. In particular, we provide a middleware that users can access to easily create blockchain transactions. Transactions are signed by the middleware which means it is the middleware who pays the fees on the public blockchain. This avoids the need for users to handle cryptocurrency. In addition, we require users to also sign the transaction data in order to reduce the risk of impersonation.

To get around the need for users to manage keys to create these signatures (a complex process fraught with errors), our architecture allows them to securely delegate this task, in an error-proof way, without sacrificing on the security and decentralization required by blockchain applications. In the paper we provide a security analysis in which we show how end users can achieve application availability, while avoiding impersonation or transaction replay attacks from either the middleware or external attackers.

The rest of the paper is organized as follows. In section 2, we provide the necessary background regarding Industry 4.0 distributed applications. In section 3, we present the main existing literature regarding user onboarding and key life-cycle management in distributed applications. In section 4, we propose our architecture. In section 5, we provide implementation guidelines and a discussion about our architecture's security. In section 6, we present a comparison with related work and finally, we conclude in section 7.

2 Distributed Applications in Industry 4.0

In the classical approach, business relationships are founded on legal contracts and reputation gained through previous interactions. Traditional applications are built with a software architecture consisting of a front end that provides the user with a friendly interface (web or mobile) and a back end that implements the business logic (as well as providing an API for the front end). Typically, back ends are provided by the application owner and run either on proprietary servers or in the cloud.

Under a blockchain architecture, applications have part, or all of, their back-end logic in the form of a smart contract in a distributed ledger. These type of applications are known as Decentralized Applications or DApps [8]. An important point to note is that DApps do not have to be totally distributed in every single aspect. Some parts of the application can be distributed amongst only a few entities, and some may even be centralized. For example, the back end may be partially centralized for data privacy reasons. To protect user privacy, a common pattern is to store cryptographic fingerprints and access control logic in the blockchain, while the data itself is stored off-chain (in proprietary servers or private cloud systems). In the same way as traditional applications, DApps also need a front end for the user interface (web or mobile). The DApp frontend

can either be distributed through centralized servers or decentralized storage systems (like IPFS [9]). In addition, most DApps, like traditional applications, have an owner that governs the application and who often stands to gain some economic profit from it. In these cases, in order to build logic around which transactions should be allowed or denied, smart contracts need to identify the transaction sender as either the owner, a user, or neither one of them.

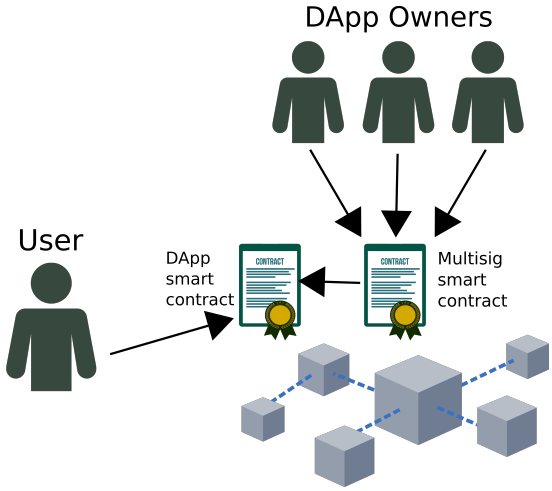

Figure 1: A Typical DApp Architecture.

Owners are responsible for running their DApp so they are expected to carefully manage their identity. Many times, the owner is not an individual, but a decentralized consortium. Figure 1 shows a typical DApp architecture. As shown in the figure, the identity of the consortium is managed with a multisig smart contract (e.g. [10, 11]).

A multisig contract works as follows: Members of the consortium send transactions to the multisig contract to approve or deny certain actions. Actions performed by owners might involve privileged tasks that change certain settings of the DApp such as adding a new identity to the list of allowed users. When enough signatures for an action are recorded by the multisig, the contact sends a transaction to the DApp contract (which contains the DApp logic). In this case, the address of the multisig contract is used as the owner's "identity".

How does a DApp manage the identity of its users? We can also use smart contracts for this. This is the approach taken by so-called decentralized identity solutions like uPort [12] or Jolocom [13]. A problem with this approach is that it forces end users to deploy smart contracts and buy cryptocurrency to manage their identities. While this scheme might be acceptable for some users, it may be not practical for general industrial users.

In this context, W3C [14] is promoting an architecture for building a global identity for end users that involves creating self-sovereign identities (SSI) [15]. In SSI, the identity is self-managed and the user is at the center of the design

with a focus on peer-to-peer interactions. As a result, the user is responsible of anchoring her identity in the blockchain and managing her cryptographic material. However, industry users are used to delegating certain aspects of their business processes (if this simplifies them and/or reduces their cost). The architecture that we present in this paper is designed to allow industrial users to delegate complex tasks related to identity management, while minimizing the risk of human error, and without sacrificing on the security and decentralization required by DApps. Our design follows the principle that complexity is acceptable for DApp owners but should be pushed away from users.

One could think that the easiest solution for managing end user identities would be plain public keys because this simple mechanism does not involve a priori anchoring identities on a blockchain. However, it is important to remark that in this case, whoever owns a private key can sign transactions on behalf of the associated public key. This fact makes the preservation of the private key a critical aspect, since if this key is leaked, anyone who knows it would be able to impersonate the affected identity. In addition, if a private key is lost, the associated cryptocurrency balance will be blocked making it no longer possible to continue performing successful transactions.

To try and get around this problem, users can utilize cryptocurrency wallets – devices, programs or services – to manage their cryptographic material. Often, cryptocurrency wallets are built as hierarchical deterministic (HD) wallets [16]. In a HD wallet, keys are generated from a single seed value instead of being randomly generated on-demand. The seed allows a user to easily back up and restore a wallet without needing any other information. Seeds are normally serialized into human-readable words in a mnemonic phrase. Users can conveniently create a single back up of the seed in a human readable format that will last the lifetime of the wallet. In addition, since the keys are generated from the mnemonic in a deterministic manner, users can restore the HD wallet in different devices, programs or services. The main problem with the wallet approach is that the user still bears a lot of responsibility. For example, it is still the user's responsibility to make sure that her keys are correctly generated and that her mnemonic is kept in a safe place. Our architecture provides a way of simplifying the key life-cycle management process by allowing business users to work with passwords that can be easily recovered instead.

Finally, another big challenge that DApps have to face in industrial environments is how to perform the *user onboarding* process. Most users do not have any specific knowledge about cryptography or previous experience sending blockchain transactions and paying their associated fees. Forcing industrial users to buy cryptocurrency creates a considerable entry barrier. To fix this issue, our architecture uses a middleware that allows for the decoupling of the digital signatures required for paying the blockchain fees, from the ones required for authorization purposes. This has the advantage that users are not forced to create wallets, buy cryptocurrency, or protect their private keys. This approach also enables a smooth integration between blockchain and traditional industrial systems. In the following sections we present the state of the art and the details of our architecture.

3 State of the Art

This section presents technical and research works that are related to different aspects of DApp usage and management.

3.1 Man4Ware

For Industry 4.0, we can find a proposal called Man4Ware [17], which is focused on building a middleware for distributed fabrication using blockchain. Man4Ware unifies all communications between blockchain and Industry 4.0 systems, such as cyber-physical systems, cloud services, sensors, actuators and so on.

The Man4Ware middleware provides a standard API which allows devices that do not have the capability of directly signing their own blockchain transactions to create requests for obtaining this service. Man4Ware creates transactions on behalf of its users, pays the corresponding fees, and finally sends these transactions to the desired blockchain.

3.2 SHOCARD

ShoCard [18] uses Bitcoin [19] as a time stamping service for sign cryptographic hashes of the user's identity information such as a passport or driver license. That means, ShoCard previously obliges each user to identify himself via presenting or scanning his or her national identifier or passport. This scanned document then generates a file which is encrypted and added to Bitcoin's blockchain.

This solution uses a central server as an essential part of its scheme. This server intermediates the exchange of encrypted identity information between users and relying parties. This means, that as a user, you don't own your keys. Rather, you need to access to your identity through the ShoCard application.

3.3 SSI and DIDs

Through the specification of decentralized identifiers (DIDs) [20], W3C [14] is promoting an architecture for building global and self-sovereign identities (SSI). DIDs are used as global permanent identifiers to reference documents called "DID Document Objects" (DDOs). A DDO is a JSON-LD[1] object that contains public information about a decentralized identity.

DID are usually anchored in a distributed ledger. The blockchain stores data that enables other users to securely find the corresponding DDO (this data is typically the fingerprint of the DDO). However, the DDO itself is normally recorded in a external distributed storage system like IPFS [9]). This resolution process is defined in detail in the DID method specification [20]. Having a publicly resolvable DDO is beneficial for users who want to auto-manage their

[1] A JSON-LD object is a JSON object with a schema to provide "context" for the meaning of properties.

identities because the DDO can include public credentials, advertise endpoints for further off-chain interactions, include credential revocation lists and more.

3.4 UPORT

The uPort project [12] is a specific implementation of a global SSI based on the Ethereum blockchain. In particular, a smart contract represents an identity that is associated with either an individual, device, entity, or institution. A core function of a uPort identity is that it can digitally sign and verify a claim, an action, or a transaction.

uPort identities are global, meaning they can be used in many different contexts and DApps. These identities are also self-sovereign, which means that they are fully owned and controlled by the creator. In addition, the uPort wallet application provides a simple consent interface for DApps to request private data from users, allowing users to approve or reject requests.

In the case of device loss, uPort allows the user to link their identity to a new public key through the intervention of a previously selected quorum of trusted people. The members of the quorum are selected by the user. They can be individuals (friends or family members for example) or institutions (like banks and credit unions).

3.5 JOLOCOM

Jolocom [13] is an open source project that is developing an identity management system based on hierarchical deterministic keys. Like uPort, it is developed on top of the Ethereum blockchain and consists of several smart contracts including a registry smart contract. Users can utilize a mobile App as a front end to interact, create, manage and share their identities.

The HD keys are generated from a known seed and enable the user to generate further child keys from the parent key (these can be recovered later if the seed is known). This enables users to generate multiple *personas* or subidentities from the parent seed.

3.6 SOVRIN

Sovrin [21] is a permissioned blockchain that is exclusively built to create a self-sovereign decentralized identity ecosystem. Sovrin's software is open-source and hosted by the Linux foundation as one of the Hyperledger [22] projects, called Indy [23].

The state of Sovrin is public, but only trusted institutions, called *stewards*, can write to the ledger. These trusted institutions can be banks, universities, governments or any entity that legally abides by the Sovrin Trust Framework. In this solution, users must be externally identified to the stewards through some kind of documentation (for example information issued by a government).

Users and organizations exchange information using agents that are addressable network points. The agent addresses are stored on the ledger to enable interaction between identities.

3.7 KMS

Regarding the key life-cycle management, many cloud providers currently offer Key Management Systems or KMS. A KMS provides back end functionality for key generation, usage, and replacement. This functionality is provided with well-defined Service Level Agreements (SLAs).

For instance, AWS-KMS [24] uses a hardware security module (HSM): a dedicated crypto processor specifically designed for the protection of the crypto key life-cycle. In particular, [24] is built with a multi chip HSM that allows each client to operate in isolation with her keys using a command line interface and a RESTful API. On the other hand, Google-KMS [25] is a scalable, automated and fast key provider that allows users to easily encrypt and sign data over a client command line, a RESTful API, or a RPC API. Finally, another KMS is Azure Vault [26] which can be configured as a key manager allowing users to automate the generation and utilization of their keys. Azure Vault audits and monitors both access and usage of each data encryption and key generation and can be easily integrated with the rest of the Azure services like central log monitoring.

4 Our Proposal

In this section, we present an enterprise-friendly architecture to facilitate the access to distributed blockchain applications (DApps) for industrial users. Our design follows the principle that, while complexity is acceptable for DApp owners, it should be pushed away from users. The two key aspects that our architecture simplifies for users are onboarding and key management.

4.1 Enhancing DApps Onboarding

Probably the biggest issue standing in the way of users embracing DApps based on public blockchains is the need to buy cryptocurrency in order to use them. This requirement comes from the fact that, in the majority of public blockchains, such as Bitcoin or Ethereum, transactions have to be digitally signed by the end user. In order to send a transaction, a fee (in cryptocurrency) is discounted from the corresponding account. In essence, an account is a public key, with the associated private key used to create the digital signature.

If we take a closer look, we would observe that the transaction signature is in fact used for two important but differing purposes:

1. To provide the identity of the account to which the transaction fee has to be charged.

2. To provide the identity that must be considered for authorization purposes to the smart contract.

In regular transactions, there is a single signature that is used for the two mentioned purposes. As a result, end users are the ones who pay the fees and consequently, they need cryptocurrency to use DApps.

However, industrial users are not used to managing cryptocurrency. They prefer to pay for services with classical payment methods (e.g. credit cards, bank transfer, etc.). To free users from managing cryptocurrency, in our architecture we decouple the two signatures by introducing a middleware, which will assume and take charge of the transaction fees.

Middlewares for blockchain (like Man4Ware) are interesting because they can provide a friendly API to interact with DApps. However, if they are not designed carefully, there is a risk that too much power might be granted to the middleware, which could end up impersonating the end user. In our design, we minimize this risk by forcing the DApp smart contracts to check not only the transaction's signature (performed by the middleware) but also the user's signature. This way, a successful transaction will be performed only if both signatures are correct. The flow to send a transaction in our architecture is shown in figure 2.

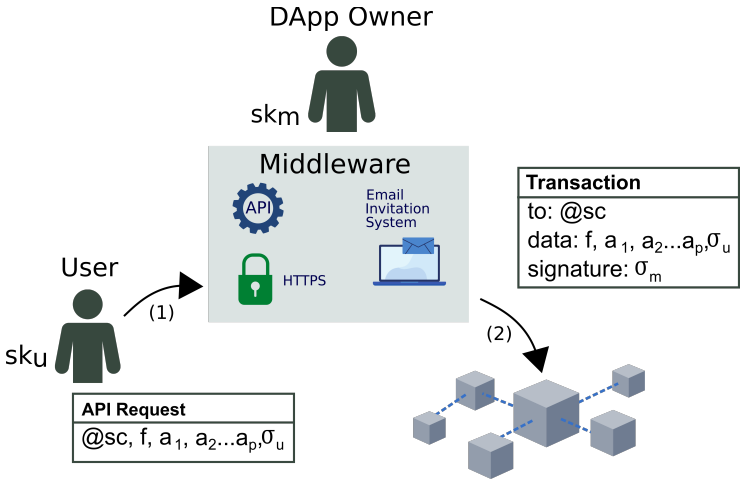

Figure 2: Middleware transactions scheme.

In step (1), the user sends the following User Transaction Data (UTD) to the middleware API:

- The address of the DApp smart contract: $@sc$
- The function of the contract to be executed: f
- The function arguments: a_1, a_2, ..., a_p

- The user's signature (using sk_u) of the previous items:

$$\sigma_u = sign_{sk_u}(@sc, f, a_1, a_2, ..., a_p)$$

In step (2), the middleware checks that the address provided through the API is a valid address pointing to a smart contract of the DApp and creates a regular transaction signed with its own secret key (sk_m) that calls the corresponding function with the arguments provided and the user's signature. As mentioned, a primary benefit of our architecture is that it provides a friendly API to interact with DApps while avoiding user impersonation by design.

Another advantage of our architecture is that it enables a clear and easy-to-implement business model for DApp owners. By charging users per API request, DApp owners can get rewarded for providing the necessary infrastructure to run the DApp: which involves implementing and distributing the front end, creating and deploying the associated smart contracts, implementing and operating the middleware, and paying for the transaction fees. Users can simply buy credit to use the DApp using a classical off-chain payment method such as a credit card, bank transfer, etc.

Another interesting feature of our architecture is that the DApp owner can tokenize this credit (as a utility token [27]) and allow users to trade it, thereby creating a market for the DApp. Finally, when the token is used, that is, a transaction is executed, the predefined amount of tokens per transaction is burned. Notice that in our architecture, it is the user's signature σ_u which authorizes the burn of the corresponding amount of tokens.

4.2 Key Life-cycle Management

A key aspect for DApps, as for any other production application, is to guarantee a correct Application Life-cycle Management (ALM) [28]. In the case of DApps, as part of their ALM, it is mandatory to plan their key life-cycl management, which defines the processes for creating, saving, changing and recovering a key in case of loss. Implementing a proper enterprise-friendly key life-cycle management is one of the main challenges that DApps need to overcome in order to achieve massive adoption by businesses and industries.

In this context, the architecture presented so far still has a critical flaw for enterprise grade DApps, that is, if a user loses her private key, then, no more transactions associated with the related identity can be executed. In our architecture, we propose taking advantage of cloud key management services to avoid this problem in production applications.

We propose to use professional Key Management Services (KMS) which have user-friendly key registration, usage and recovery procedures. The process of registering a user in a KMS typically involves setting an e-mail account, a password and other authentication factors[2] such as providing a mobile phone

[2]When there are two authentication factors this is called "two factor authentication or 2FA.

to which SMS are sent for authorizing certain operations. In general, these processes take place in a way in which end users are accustomed. The flow to send a transaction with a KMS is shown in figure 3.

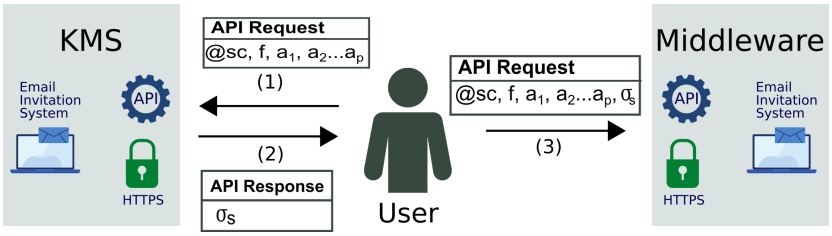

Figure 3: Architecture with a KMS.

Once the user has been registered in a KMS, she can typically log into a web or mobile application provided by the KMS using all the authentication factors required and get a token to use the API of the key provider. This token is usually a *bearer token* [29], which is a security token with the characteristic that any party in possession of the token (a *bearer*) can use the token in any way they wish (same as any other party in possession of it). This means that using a bearer token does not require a user to prove possession of cryptographic key material. This property is very useful for building software that automates API consumption without human intervention. Using this software the user agent sends the tuple ($@sc$, f, a_1, a_2, ..., a_p) to the KMS API. If the user is correctly authenticated in the KMS API, the KMS provider returns the signature σ_s, where:

$$\sigma_s = sign_{sk_s}(@sc, f, a_1, a_2, ..., a_p).$$

In general, a KMS manages the private key internally. The key (sk_s) is never in the user's hands, since it never leaves the provider. Next, the user sends the transaction data to the middleware using the corresponding API call:

$$UTD = (@sc, f, a_1, a_2, ..., a_p, \sigma_s)$$

Finally, as in the previous case, the middleware creates a signed transaction with its own private key (sk_m) and sends it to the blockchain. Notice that the two signatures included in each transaction avoid a user being impersonated by either the KMS or the middleware (a more detailed security analysis of the architecture is provided in section 5).

As mentioned earlier, the user's software can automatically interact with the KMS APIs using tokens. In our architecture, the DApp owner can also implement similar procedures for the middleware API. This architecture has the benefit that the token and password recovery procedures can be implemented as usual:

- If the tokens of an user are lost or compromised, the user can log in again in the KMS or Middleware back end application and get a new token (revoking the previous one).
- If a user loses any of her passwords, a regular password recovery procedure can be used. For example, sending an e-mail with a temporary link. In this case, if extra security is required, additional authentication factors can be added: factors such as a TOTP (Time-based One-time Password Algorithm [30] or an HOTP (HMAC-based One-time Password algorithm [31]). For these factors there are several easy-to-use applications available for both web and mobile [32, 33].

In our architecture, there is no need to setup a wallet, buy cryptocurrency, or protect the private key yourself. But there is a risk that a KMS might not be respond, preventing users from using the DApp. In this case, we can distribute the representation of an identity among several key providers. Then, to make sure the DApp is usable in spite of KMS unavailability, we can create a policy that requires only a subset of the total number of providers to create a transaction. We call k the amount of providers required to create a transaction and n the total number of providers (note that $k \leq n$). As shown in figure 4, the user now has to send the transaction data to at least k KMS. Each available KMS will sign this data with its private key ski_j to produce a signature σ_{si_j}, where:

$$\sigma_{si_j} = sign_{ski_j}(@sc, f, a_1, a_2, ..., a_p).$$

The user should receive k signatures σ_{si_j} of n possible ones. More formally,

$$\sigma_{si_1}, \ldots, \sigma_{si_k}$$

where $\{i_1 \ldots i_k\} \subseteq \{1 \ldots n\}$ and $\sigma_{i_j} \neq \sigma_{i_{j'}}$ for all $j \neq j'$.

When the user software receives the k required signatures, it sends them together to the middleware API as user transaction data:

$$UDT = (@sc, f, a_1, a_2, ..., a_p, \underbrace{\sigma_{si_1}, \sigma_{si_2}, ..., \sigma_{si_k}}_{\text{k}})$$

The exact policy defining (n, k) can be adjusted according to the standards of the particular DApp. Notice that by using this (n, k) scheme, we are protecting DApp users from fail-stop behaviours and reducing the risk of impersonation in the case any KMS exhibits a dishonest behaviour (also known as byzantine behaviour). A more detailed analysis is provided in section 5.

Furthermore, it is remarkable that the policy will be enforced by the DApp smart contracts. Remember that in our architecture, transactions transmit signatures for authorization as arguments of the smart contract function to be executed. In this case, the transaction has to be authorized by k entities. As mentioned, the smart contract defines the logic to accept or deny each transaction. This configuration will include the list of possible KMS providers (as a list of public keys) and the minimum number of necessary authorization signatures.

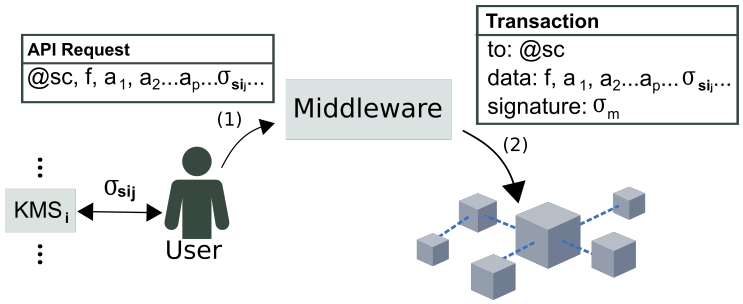

Figure 4: Architecture with several KMS.

It is worth remarking that with our architecture, users can have an identity that is valid for a DApp and that can be managed without having to deal with complex cryptographic material.

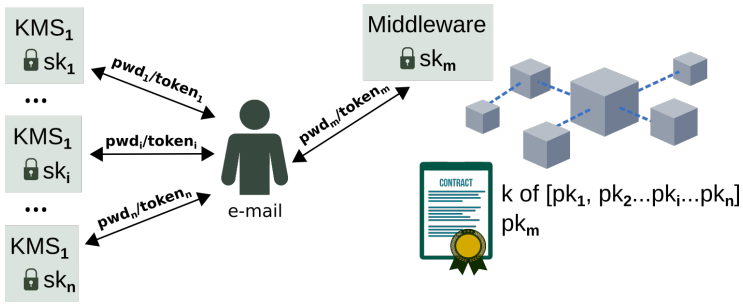

Figure 5: Passwords and keys.

As it can be observed in figure 5, the user software just needs to deal with traditional e-mails, passwords and API tokens which greatly simplifies the recovery procedures. Finally, it is important to remark that the smart contracts of the DApp must allow the user to change the KMS list and the minimum number of KMS necessary to accept transactions. To do so, the user should simply send a valid transaction to the middleware to update this policy. Obviously, this transaction for updating the policy must be valid according to the current policy defined in the smart contract.

On the other hand, remember that the design and implementation of the smart contracts and the middleware of a DApp are performed in general by its owner. In this context, DApp owners should have incentives to provide an acceptable SLA for their end users since they generally obtain benefits from operating the application. In particular, the middleware is a critical part of the architecture and must be implemented as a high availability component. Furthermore, depending on the DApp, having a single entity operating the middleware might not be acceptable for end users. In this case, the middleware can be implemented as a decentralized component.

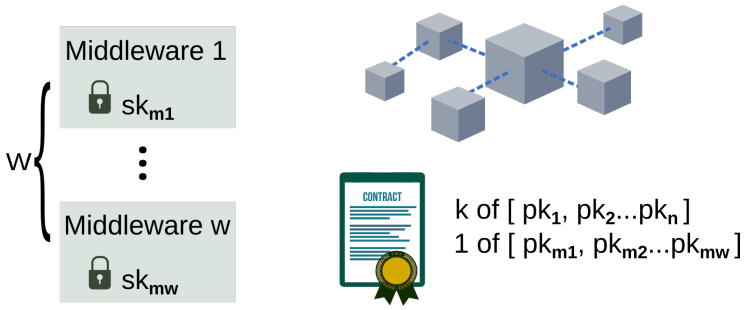

Figure 6: A Decentralized Middleware.

Figure 6 shows a DApp built with a middleware that has been decentralized among w entities. As shown in the figure, each middleware component is operated by a different entity which independently manages a private key. The DApp smart contract stores the list of public keys of authorized middlewares and will only accept a transaction if it is signed by one of these valid middlewares.

5 Implementation and Security Guidelines

Obviously, the proposed architecture has to be customized in order implement it for a specific DApp. In this section, we provide general implementation guidelines and a discussion of the architecture's security that should be useful as a best practice list for any implementation.

5.1 Smart Contracts and Transactions

The first aspect to consider in the architecture is how to implement the smart contracts and transactions. We consider that Ethereum is a very suitable technology because it is a public and well established blockchain and because we can use EIP 712 [34] to sign arbitrary messages with Ethereum. In our case, EIP 712 can be used to create the digital signatures for authorization. EIP 712 defines the signature with Keccak256 algorithm and with the following format:

- "\x19Ethereum Signed Message:\n"
- len(message)
- message

The recommended way to create signatures is to use a hash of the message as message in the previous format. This ensures `len(message)` is always 32 bytes long (the prefix is always the same). It also makes the manipulation of signatures easier, particularly on the Solidity[3] side. Notice that adding a prefix to the

[3]Solidity is the most widely used high level language to write smart contracts in Ethereum.

message makes the calculated signature recognizable as an Ethereum specific signature. This prevents misuse where a malicious DApp can sign arbitrary data (e.g. transaction) and use the signature to impersonate the victim.

In addition, the function of the smart contract to be executed should be idempotent, meaning that it can be applied multiple times without changing the result beyond the initial one. In case of non idempotent functions, the smart contract must ensure that the user does not suffer reply attacks. To do so, the user should add a timestamp in both signatures and transaction data:

$$\sigma_{si} = sign_{ski}(@sc, f, a_1, a_2, ..., a_p, ts)$$
$$UDT = (@sc, f, a_1, a_2, ..., a_p, ts, \sigma_{si}, \sigma_{sj}, ...)$$

The smart contract then checks if a timestamp has been already used for the same user and the non-idempotent function and if so rejects executing the action again. Finally, notice that since we are including the address of the DApp smart contract in each UDT, a transaction cannot be replayed to the smart contract of another DApp.

5.2 The Middleware API

A simple approach to manage bearer tokens for users of the middleware is to generate each token as a random number and store them in a list. However, storing big lists of secrets is impractical and highly risky. A better solution is to make the middleware sign each API token and send it to the users. Then, if a user wants to access the API, she presents the signed token and the middleware simply checks that the token has not been tampered with.

In this context, OWASP [35] (the Open Web Application Security Project is a non-profit foundation that works to improve the security of software) recommends the use of JSON Web Tokens (JWTs) [36] to create signed API tokens. We recommend following the OWASP recommendations for building APIs [37].

5.3 Fail-stop and Byzantine Behaviours Protection

Our scheme enables the users of a DApp to interact with the smart contract without directly handling with private keys. This is done by allowing users to authenticate themselves using k from a list of n predefined key providers. This procedure can bring about two types of security issues. On the one hand, it can present availability problems if the key providers do not respond (*fail-stop faults*), and on the other hand, if the key providers or the middleware are malicious, users are at risk of being impersonated by any of these services (*byzantine faults*). In this section, we analyse the two problems and show that parameters k and n can be adjusted by the user and the DApp owner to lower the probability of these problems happening.

Before analysing a general setup, let us consider the case $k = 3$ and $n = 5$. In this scenario, the user can only interact with the smart contract if the middleware, and 3 of 5 key providers, whose public keys have previously been

stored in a state variable of the smart contract, are operative. For the analysis, we consider the typical SLAs of the main KMS which are summarized in table 1. As shown in this table, a KMS returns to the user the full service cost if the service is below 95%. So, the probability of a key provider not being operative is very likely to be between 0 and 0.05. To be safe, we assign a probability of unavailability of 0.05 to each provider. Regarding the middleware, as argued in section 4.2, the DApp owner is responsible of keeping the middleware up and running. For the following analysis, we will assume no downtime for the middleware.

Monthly Up-time Percentage	**Service Credit Percentage**
Less than 99.9% but greater than 99.0%	10%
Less than 99.0% but greater than 95.0%	25%
Less than 95.0%	100%

Table 1: Service Level Agreements from [38–40].

We call *fail-stop fault tolerance* of the architecture the probability that a user can not send a transaction to the DApp. In this example, this is the probability that 3 of the 5 key providers are not available. Assuming unavailability is not correlated between the KMS services, the fail-stop tolerance in this case is exactly

$$\binom{5}{3} 0.05^3 = 0.00125.$$

So, there is a less than 0.2% chance that a user can not access 3 of the 5 providers. Taking a smaller k lowers this probability even more, since fewer providers are needed. On the other hand, taking a larger k increases this probability. This tendency is shown in figure 7.

Now, let us study the probability that the middleware or the key providers are able to impersonate the user and consequently, interact with the smart contract on its behalf. For that, let us assign a probability of maliciousness to each of the services:

$$p_i = \text{probability that the key provider } i \text{ is malicious,}$$
$$q = \text{probability that the middleware is malicious.}$$

These probabilities can be understood as the degree of mistrust for each of the services. As opposed to the fail-stop probabilities, these measures respond to more subjective criteria and may be more difficult to determine. Nonetheless, we can argue that the key providers will have similar maliciousness probabilities, since they respond to the same business motivations. So, instead of having different $p_1, \ldots, p_5$ probabilities for each provider, we can simplify the analysis by assuming all these probabilities are the same. We will call this probability p from now on. It makes sense to keep a different probability parameter q for the middleware, since the degree of trust on a DApp can differ very much from p.

The probability of a user of being impersonated, which we call the *byzantine fault tolerance* of the architecture, is in this case, the probability that the middleware and three of the key providers are malicious. If we assume that each service is independent of the rest, then this will happen with probability

$$P(\text{impersonate an identity}) = \binom{5}{3} p^3 q.$$

Imagine we are 80% confident the DApp is not a scam, and that the key providers are 95% reliable. Then the result of this calculation would be

$$\binom{5}{3} 0.05^3 \, 0.2 = 0.00085.$$

As opposed to the fail-stop tolerance, if k had been higher, this probability would have been smaller, as it would have required more services to be malicious. And similarly, the smaller the k, the larger the byzantine fault tolerance. As fail-stop and byzantine fault tolerances have opposite behaviours, there is an inevitable trade-off between the two. In figure 7 we have presented in logarithmic scale how these probabilities vary for all possible k when $n = 5$.

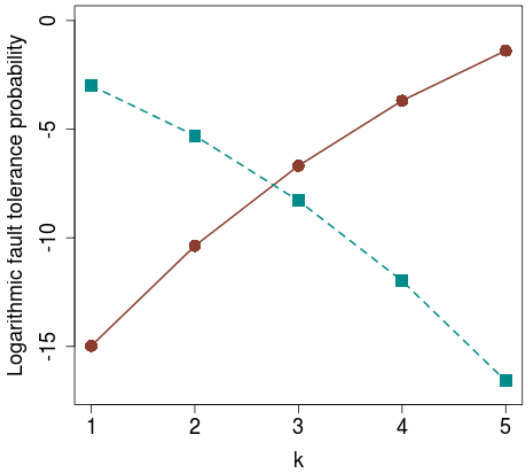

Figure 7: Graphic of the fault probabilities in logarithmic scale for all possible values of k when $n = 5$. The fail-stop fault tolerance (- - -) is computed with a 0.05 probability of a key provider being unresponsive and the byzantine fault tolerance (—) taking the parameters $p = 0.05$ and $q = 0.2$.

We can see that taking $k = 2$ and $k = 3$ leads to more balanced fault tolerances, but as we will show now, the specific choices of k and n can be adjusted

by the user and the DApp owner according to their criteria and risk-policy requirements.

In a more general setup, a user will be able to interact with the smart contract if the middleware and k of the n key providers are operative. As argued before, assuming the middleware is up all the time and calling t the probability that a key provider is not available, the fail-stop fault tolerance is

$$\binom{n}{n-k+1} t^{n-k+1}.$$

In the previous example we used $t = 0.05$, but this parameter can be adjusted according to the providers policy.

Now we assume middleware non-availabilities in our analysis. To do so, we consider the more general model depicted in figure 6, where the middleware can be decentralized and thus, the transaction signature can be performed by 1 of w entities. We consider a new probability called r that defines the availability of each independent middleware platform. This way, the fail-stop fault tolerance becomes

$$\binom{n}{n-k+1} t^{n-k+1}(1 - r^w) + r^w.$$

Note that by setting $r = 0$ we get back the previous equation.

Let us study now the byzantine fault tolerance. As before, consider the probabilities:

$$p_i = \text{probability that the key provider } i \text{ is malicious,}$$
$$q = \text{probability that the middleware is malicious.}$$

where i now runs from 1 to n. In general, the middleware will only be able to supplant the identity of a user if it has the private keys from k of the n key providers. This means, that the middleware would need to collude with k key providers to generate a valid transaction that can pass the smart contract verification. Similarly, key providers do not have access to the middleware API nor to the other provider APIs. So, in order to create a valid transaction on behalf of the user, the malicious provider would need to collude with $k - 1$ key providers and the middleware. So, a user would be impersonated if the middleware and a subset of k key providers $j_1, \ldots, j_k$ are malicious. As a result, the byzantine fault tolerance is

$$P(\text{impersonate an identity}) = \binom{n}{k} p^k q = \frac{n!}{k!(n-k)!} p^k q.$$

Note than when all key providers are required to sign ($k = n$), the architecture has maximum byzantine fault tolerance, since the user can only be impersonated if the middleware and all key providers are malicious. The probability in this case becomes

$$P(\text{impersonate an identity}) = p^k q.$$

	Our Architecture	Man4ware	ShoCard	uPort	Jo
Users do not need to handle cryptocurrency	✓	✓	×	×	
Delegated key management	✓	✓	×	×	
User cannot be impersonated	✓	×	✓	✓	
Password key recovery	✓	×	×	×	
Smart Contacts	✓	×	×	✓	

Table 2: Comparison between different solutions.

So, if we want this probability to be less than a particular security parameter ε, then k should be chosen so that

$$k < \frac{\log \varepsilon - \log q}{\log p}.$$

Note that when choosing $k = n$, the fail-stop fault tolerance is 0. As explained before, it is advised to find a trade-off between fail-stop and byzantine fault tolerance by choosing $k < n$.

In the decentralized middleware architecture, the byzantine fault tolerance is higher, since it is sufficient for 1 of the w to be malicious. More specifically, in this case

$$\begin{aligned} P(\text{impersonate an identity}) &= \binom{n}{k} p^k \binom{w}{1} q \\ &= w \, \frac{n!}{k!(n-k)!} p^k q, \end{aligned}$$

which is the same probability as before, multiplied by the total amount of entities decentralizing the middleware.

6 Comparison with Related Work

The first solution analysed is Man4ware [17]. Man4ware is a middleware-based solution. As such, this approach enables a smooth integration between blockchain and traditional systems. In particular, the middleware eliminates the need for users/devices to manage their own currency. The main problem of Man4ware is that it can impersonate users/devices which makes the middleware a highly powerful trusted party in this architecture. Like in Man4Ware, in our architecture users utilize the API provided by the middleware to send the data necessary to build the transactions. Transactions are signed by the middleware and therefore, it is the middleware who pays the fees. This avoids users having to deal with cryptocurrency. Unlike Man4Ware, the smart contracts of our architecture are prepared to process extra signatures to avoid user impersonation by the middleware.

While the ShoCard [18] approach is interesting as a method for proving facts with a network that is very secure like Bitcoin, it is not suitable for our

purposes of building DApps because Bitcoin does not have the capability of executing smart contracts as a built-in feature. Also, the identity is completely dependent on the ShoCard infrastructure. This implies that your identity has a single point of failure with respect to both availability and integrity.

Regarding uPort [12], one of its main issues for industrial environments is that its onboarding process is quite complex for an average user. This is mainly because of the global and self-sovereign nature of uPort identities. In this sense, users have to deploy identity smart contracts, manage cryptocurrency, and self-manage cryptographic material and key recovery. This fact makes identities in uPort not very suitable for enterprise scenarios, where users want easier key recovery procedures, pay with traditional payment systems and in which any kind of key loss or the incapability to recover a key, may imply a huge cost.

Like uPort, the onboarding process of Jolocom [13] is quite complex too. Again, Jolocom is a SSI solution that in this case allows the generation of sub-identities (*personas*) from a parent seed. The main problem here is that the current Jolocom description does not define any production grade procedure to manage this seed or to regenerate it in the case where it is compromised or lost.

On the other hand, unlike uPort or Jolocom, our architecture takes a different and more enterprise-friendly approach in which we do not use the concept of global SSI. We use identities that are valid in the context of a DApp. This enables us to use a middleware approach achieving great advantages in both key management and onboarding. In particular, we do not require users to setup wallets, buy cryptocurrency or protect their private keys themselves. In this context, KMS solutions backed by well established companies are a suitable approach to build an enterprise-friendly solution for identity management. These KMS solutions provide SLAs and well established procedures for registration, usage and key recovery. This minimizes the probability of losing a secret key or making it public by human error. Nevertheless, under our architecture, when building a local DApp identity, it is possible to integrate uPort or Jolocom global self-sovereign identities as *key providers*.

Finally, Sovrin [21] uses a permissioned blockchain for managing identity. Permissioned ledgers require an access control layer. In the case of Sovrin, the network depends on trusting its permissioned agents called *stewards*. In general, the security of a blockchain system heavily relies on having a large number of participants contributing to the system. In this context, public blockchains are better suited for DApps that need public transparency. Our architecture is specifically focused on this types of DApps and that is why our design is based on a general purpose and public blockchain network with smart contract capabilities like Ethereum.

We summarize the different features of each analysed architecture in table 2.

7 Conclusion

Industry 4.0 will greatly benefit from the use of blockchain technologies to manage the huge amount of interactions that this paradigm will bring. Applications that use decentralized technologies (DApps) can greatly reduce the time needed to complete and register business agreements between manufacturers and their suppliers by allowing agreements to become binding without having to go through a formal registration process.

For massive adoption of DApps, it is necessary to make the user onboarding and the key life-cycle management enterprise-friendly. In this context, we presented an enterprise viable architecture for DApps deployed on public blockchains.

Our architecture is based on a general purpose and public blockchain network with smart contract capabilities such as Ethereum. It follows the principle that complexity is acceptable for DApp owners but should be pushed away from users. Unlike other architectures that advocate creating global self-sovereign identities, our approach is to use identities that are valid in the context of a DApp. This approach enables us to use a middleware. This has great advantages for simplifying key management and user onboarding enabling a reasonable transition to this new technology in industrial environments.

In particular, we do not require users to setup wallets, buy cryptocurrency or protect private keys themselves. Users can buy services provided by DApps with traditional payment methods and manage cryptographic keys without complexity. In our opinion, this makes our architecture more enterprise-friendly than the current existing approaches.

Acknowledgements

The architecture presented in this paper is supported and developed in the context of the i3-MARKET project [41]. The i3-MARKET project is an active European H2020 project focused on developing solutions for building an European data market economy by enhancing current marketplace platforms with innovative technologies (call H2020-ICT-2019-2 with grant agreement number 871754). This work is also supported by the TCO-RISEBLOCK (PID2019-110224RB-I00), MINECO/FEDER funded project ARPASAT TEC2015-70197-R and by the Generalitat de Catalunya grant 2014-SGR-1504.

References

[1] J. Al-Jaroodi and N. Mohamed, "Blockchain in industries: A survey," *IEEE Access*, vol. 7, pp. 36 500–36 515, 2019.

[2] O. Novo, "Scalable access management in iot using blockchain: A performance evaluation," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4694–4701, 2019.

[3] H. Dai, Z. Zheng, and Y. Zhang, "Blockchain for internet of things: A survey," *IEEE Internet of Things Journal*, vol. 6, no. 5, pp. 8076–8094, 2019.

[4] A. Shukla, S. K. Mohalik, and R. Badrinath, "Smart contracts for multi-agent plan execution in untrusted cyber-physical systems," in *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, 2018, pp. 86–94.

[5] N. Mohamed and J. Al-Jaroodi, "Applying blockchain in industry 4.0 applications," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, 2019, pp. 0852–0858.

[6] T. Ali Syed, A. Alzahrani, S. Jan, M. S. Siddiqui, A. Nadeem, and T. Alghamdi, "A comparative analysis of blockchain architecture and its applications: Problems and recommendations," *IEEE Access*, vol. 7, pp. 176 838–176 869, 2019.

[7] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, and J. Weng, "An overview on smart contracts: Challenges advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, Apr 2020.

[8] K. M. Kina-Kina, H. E. Cutipa-Arias, and P. Shiguihara-Jurez, "A comparison of performance between fully and partially decentralized applications," in *2019 IEEE XXVI International Conference on Electronics, Electrical Engineering and Computing (INTERCON)*, 2019, pp. 1–4.

[9] J. Benet, "Ipfs - content addressed, versioned, p2p file system," Jul 2014, [Accessed: 30-May-2020]. [Online]. Available: https://github.com/ipfs/ipfs/blob/master/papers/ipfs-cap2pfs/ipfs-p2p-file-system.pdf

[10] ConsenSys, "Consensys/multisigwallet," Jun 2018, [Accessed: 30-May-2020]. [Online]. Available: https://github.com/ConsenSys/MultiSigWallet

[11] Gnosis, "gnosis/multisigwallet," Mar 2020, [Accessed: 30-May-2020]. [Online]. Available: https://github.com/gnosis/MultiSigWallet

[12] C. L. R. H. J. T. Z. Mitton and M. Sena, 2016, [Accessed: 30-May-2020]. [Online]. Available: uport:Aplatformforself-sovereignidentity

[13] C. Fei, J. Lohkamp, E. Rusu, K. Szawan, K. Wagner, and N. Wittenberg, "Jolocom whitepaper: A decentralized open source solution for digital identity and access management," 2018.

[14] "W3c," [Accessed: 30-May-2020]. [Online]. Available: https://www.w3.org/

[15] A. Abraham, "Self-sovereign identity," Oct 2017, [Accessed: 30-May-2020]. [Online]. Available: https://www.egiz.gv.at/files/download/Self-Sovereign-Identity-Whitepaper.pdf

[16] Bitcoin, "bitcoin/bips," [Accessed: 30-May-2020]. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

[17] N. Mohamed and J. Al-Jaroodi, "Applying blockchain in industry 4.0 applications," in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, 2019.

[18] "Identity management verified using the blockchain," 2019, [Accessed: 30-May-2020]. [Online]. Available: https://shocard.com/wp-content/uploads/2018/01/ShoCard-Whitepaper-Dec13-2.pdf

[19] "Bitcoin whitepaper," [Accessed: 30-May-2020]. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[20] "Decentralized identifiers (dids) v1.0," [Accessed: 30-May-2020]. [Online]. Available: https://www.w3.org/TR/did-core

[21] "Sovrin: A protocol and token for self-sovereign identity and decentralized trust," 2018, [Accessed: 30-May-2020]. [Online]. Available: https://sovrin.org/wp-content/uploads/Sovrin-Protocol-and-Token-White-Paper.pdf

[22] "Introduction to hyperledger." [Online]. Available: https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHypnote={[Accessed:30-May-2020]},erledger.pdf

[23] "Hyperledger indy," Apr 2020, [Accessed: 30-May-2020]. [Online]. Available: https://www.hyperledger.org/projects/hyperledger-indy

[24] "Amazon web services - key management service," [Accessed: 30-May-2020]. [Online]. Available: https://d0.awsstatic.com/whitepapers/KMS-Cryptographic-Details.pdf

[25] "Google cloud - key management service," [Accessed: 30-May-2020]. [Online]. Available: https://cloud.google.com/kms

[26] "Azure vault," [Accessed: 30-May-2020]. [Online]. Available: https://docs.microsoft.com/en-us/azure/key-vault

[27] "To token or not to token: Tools for understanding blockchain tokens," [Accessed: 30-May-2020]. [Online]. Available: https://www.zora.uzh.ch/id/eprint/157908/1/To%20Token%20or%20not%20to%20Token_%20Tools%20for%20Understanding%20Blockchain%20Toke.pdf

[28] M. Gatrell, "The value of a single solution for end-to-end alm tool support," *IEEE Software*, vol. 33, no. 5, pp. 103–105, 2016.

[29] M. Jones and D. Hardt, "The OAuth 2.0 Authorization Framework: Bearer Token Usage," RFC 6750, Oct. 2012. [Online]. Available: https://rfc-editor.org/rfc/rfc6750.txt

[30] M. View, J. Rydell, M. Pei, and S. Machani, "TOTP: Time-Based One-Time Password Algorithm," RFC 6238, May 2011. [Online]. Available: https://rfc-editor.org/rfc/rfc6238.txt

[31] M. View, D. M'Raihi, F. Hoornaert, D. Naccache, M. Bellare, and O. Ranen, "HOTP: An HMAC-Based One-Time Password Algorithm," RFC 4226, Dec. 2005. [Online]. Available: https://rfc-editor.org/rfc/rfc4226.txt

[32] "Google authenticator," [Accessed: 30-May-2020]. [Online]. Available: https://github.com/google/google-authenticator/wiki

[33] "Microsoft authenticator," [Accessed: 30-May-2020]. [Online]. Available: https://www.microsoft.com/en-us/account/authenticator?cmp=h66ftb_42hbak

[34] Ethereum, "ethereum/eips," [Accessed: 30-May-2020]. [Online]. Available: https://github.com/ethereum/EIPs/blob/master/EIPS/eip-712.md

[35] "Who is the owasp foundation?" [Accessed: 30-May-2020]. [Online]. Available: https://owasp.org

[36] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," RFC 7519, May 2015. [Online]. Available: https://rfc-editor.org/rfc/rfc7519.txt

[37] OWASP, "Owasp/api-security top 10 2019," [Accessed: 30-May-2020]. [Online]. Available: https://github.com/OWASP/API-Security/raw/master/2019/en/dist/owasp-api-security-top-10.pdf

[38] "Amazon web services - service level agreement," [Accessed: 30-May-2020]. [Online]. Available: https://aws.amazon.com/kms/sla

[39] "Azure - service level agreement," [Accessed: 30-May-2020]. [Online]. Available: https://azure.microsoft.com/en-us/support/legal/sla/key-vault/v1_0

[40] "Google cloud - service level agreement," [Accessed: 30-May-2020]. [Online]. Available: https://cloud.google.com/kms/sla

[41] "H2020 european project i3-market," [Accessed: 30-May-2020]. [Online]. Available: https://cordis.europa.eu/project/id/871754

Rafael Genés-Durán is currently a PhD student of the Information Security Group (ISG) doing research in distributed ledger technologies

and zero-knowledge proofs at Universitat Politècnica de Catalunya. He holds a B.S. degree in Telecommunications Engineering (2017) and a Master in Informatics Engineering (2019). He is also co-founder and COO at Hardapps Labs. Contact him at rafael.genes@upc.edu.

Diana Yarlequé-Ruesta, is an Industrial Engineer (2012) with post-graduate studies in commercial management, search engine optimization (SEO) and search engine marketing (SEM) (2017) and she holds a Master in Marketing from Universitat Politècnica de Catalunya. She has experience as Web developer and she lead many international projects as UX and product designer. She is co-founder and CMO at Hardapps Labs. Contact her at dyarleque@hardapps.io.

Marta Bellés-Muñoz, received her B.S. degree in Mathematics at Universitat Autònoma de Barcelona and continued her Master studies at Aarhus Universitet, where she focused on the study of elliptic curves and isogeny based cryptography. She is currently a PhD student doing research in security and efficiency of arithmetic circuits for zero-knowledge protocols at Universitat Pompeu Fabra in collaboration with iden3. Contact her at marta.belles@upf.edu.

Antonio Jimenez-Viguer, Telecommunications Engineering (2011) and M.S Blockchain-DLT technologies from Universitat Politècnica de Catalunya (2019). He holds an Executive MBA (2016) and his research interest areas are distributed ledger technologies and decentralized applications (DApps). He is currently the responsible of IT Infrastructure & Operations at

Seat, S.A., Contact him at antonio.jimenez@seat.es.

José L. Muñoz-Tapia, is a researcher of the Information Security Group (ISG) and an associate professor of the Department of Network Engineering of the Universitat Politècnica de Catalunya (UPC). He holds a M.S. in Telecommunications Engineering (1999) and a PhD in Security Engineering (2003). He has worked in applied cryptography, network security and game theory models applied to networks and simulators. His research focus has now tuned to distributed ledgers technologies and he is the director of the master program in Blockchain technologies at UPC School. Contact him at jose.luis.munoz@upc.edu.