Multi-Platform Wallet for Privacy Protection and Key Recovery in Decentralized Applications

Cristòfol Daudén-Esmel, Jordi Castellà-Roca, Alexandre Viejo and Ignacio Miguel-Rodríguez

Universitat Rovira i Virgili, Departament d'Enginyeria Informàtica i Matemàtiques
CYBERCAT-Center for Cybersecurity Research of Catalonia
Av. Països Catalans 26, E-43007 Tarragona, Spain

{*cristofol.dauden,jordi.castella,alexandre.viejo,ignacio.miguel*}*@urv.cat*

Abstract

In recent years, the rise of blockchain technology and its applications has led the software development industry to consider Blockchain-Powered Decentralized Applications (dApps) as Serverless REST APIs. However, to engage with dApps, users require a blockchain wallet. This tool facilitates the generation and secure storage of a user's private key and verifies their identity, among other functionalities. Despite their utility, blockchain wallets present significant challenges, such as reliance on trusted third parties, vulnerability to adversaries observing and potentially linking user interactions, key recovery issues, and synchronization of cryptographic keys across multiple devices. This paper addresses these challenges by introducing a fully decentralized multi-platform wallet that leverages blockchain and InterPlanetary File System (IPFS) technologies for managing asymmetric keys and enabling key recovery. This novel approach empowers users to interact with dApps built on blockchain smart contracts while preserving their privacy and ensuring seamless key recovery in case of device theft or damage. The proposed system is economically viable, with in-depth cost analysis, and demonstrates resilience against security and privacy attacks. A comparative analysis highlights the advantages of the new scheme over existing mainstream and state-of-the-art solutions. Finally, a preliminary prototype implementation is presented to validate the system's feasibility.

Keywords:
Decentralized applications, Crypto wallet, Smart contract, Key Recovery, Security, Privacy

1. Introduction

Over the past few years, blockchain technology and its underlying distributed ledger system have emerged as highly recognized, scalable, and dependable platforms. This infrastructure has played a pivotal role in reducing reliance on traditional centralized architectures across a wide array of Internet-based services, while also addressing inherent security complexities in distributed systems. Notable examples include the use of blockchain-powered Decentralized Applications (dApps) as serverless REST APIs [1, 2]; the adoption of Self-Sovereign Identity (SSI) as an emerging paradigm for establishing and managing individuals' digital identities [3, 4]; the management of users' personal data [5, 6] and medical health records [7, 8] under GDPR and other legislative frameworks; and, last but not least, its application in Intelligent Transportation Systems (ITS) [9, 10, 11]. These examples highlight the far-reaching potential and impact of this disruptive technology.

The blockchain platform offers significant advantages such as immutability, accessibility, auditability, and resilience. Consequently, all stored information remains unalterable and accessible solely to authorized users. Moreover, data transactions are thoroughly tracked, with data replicated across all nodes within the distributed network, facilitating detection and tracing of potential attackers while ensuring secure data backup.

However, the blockchain ecosystem introduces its own vulnerability: individuals must utilize a blockchain *wallet* for platform engagement. This tool serves as a user interface bridging the gap between blockchain networks and real-world applications. Specifically, a blockchain wallet: i) facilitates the generation and storage of a user's private key; ii) authenticates a blockchain user by confirming possession of said private key without reliance on third-party

intervention; and iii) grants a user access to their personal digital assets stored on the blockchain (e.g., cryptocurrency ownership).

Given the considerations above, it becomes evident that the relevance of blockchain wallets cannot be overlooked. Depending on their design and functionalities, these wallets can pose substantial security and privacy risks to users. Among the most commonly used blockchain wallets, two primary types can be identified: i) *online wallets* such as Binance [1] Coinbase [2] and Kraken [3]; and ii) *web browser extension wallets* such as Metamask [4], Solflare (solana) [5] and Myeherwallet (Ethereum) [6].

Online wallets present a significant drawback: users must rely on a third party to safeguard their private keys and manage their assets. Web browser extension wallets mitigate this issue by storing private keys within the user's browser, thereby reducing the risk of data theft, though not entirely eliminating it. Nevertheless, these wallets still face critical challenges related to privacy concerns and the limited capacity to share and manage private keys across multiple personal devices.

Concerning threats to user privacy, the repeated use of the same key pair for blockchain interactions introduces significant risks. Adversaries could potentially link a wallet to a specific individual, gaining access to sensitive information. For example, an attacker might identify a cryptographic key pair used across multiple dApps by the same user and subsequently exploit those dApps as pseudo-identifiers in *record linkage attacks* to uncover personal data. To mitigate this risk, it is essential to employ fresh key pairs for each dApp interaction or blockchain activity. Therefore, an effective privacy-preserving solution would involve a wallet that automatically and transparently generates and manages new key pairs for each use.

In addition to this, the envisioned privacy-preserving wallet must address the challenges posed by the multitude of devices that users integrate into their daily lives. Ensuring the protection and synchronization of cryptographic key pairs across these diverse devices presents a non-trivial task. While utilizing the Cloud for synchronization may offer a direct solution, it introduces a centralization point that undermines the claimed benefits of full decentralization. Consequently, the forthcoming privacy-preserving wallet design must be capable of dealing with the complexities of the current multi-platform ecosystem while upholding its inherently advantageous decentralized characteristics.

Last but not least, providing key recovery functionality in blockchain wallets is paramount for ensuring user confidence, convenience, and security in the decentralized ecosystem. The inherent nature of blockchain technology, with its emphasis on decentralization, also brings forth the challenge of individual responsibility for key management. In this context, offering key recovery mechanisms becomes essential to address the risk of losing access to digital assets due to forgotten or leaked keys. By enabling users to recover their keys in the event of loss or damage to their devices, blockchain wallets can enhance user experience and reduce the fear of irreversibly losing valuable assets. Moreover, key recovery features can encourage broader adoption of blockchain technology by providing a safety net for users who may be hesitant to engage with decentralized systems due to concerns about key management.

1.1. Related work

As previously mentioned, blockchain wallets serve as foundational components within this technology, prompting numerous proposals in the literature to design and create effective and secure solutions. These efforts often revolve around the design aspects concerning the generation, storage, recovery, and operation of blockchain's cryptographic keys. Based on these criteria, blockchain wallets can be categorized into three main categories: i) trusted third-party-dependent wallets, ii) multi-signature wallets, and iii) seed-derived wallets.

The first category, *trusted third-party-dependent wallets*, represents perhaps the most extensively explored realm, primarily focusing on the storage of cryptographic keys (or core data facilitating their recovery) in external repositories. These repositories may consist of servers owned by service providers or other third parties chosen directly by users themselves. Some relevant proposals that fall within this category are [12, 13, 14, 15, 16, 17, 18, 19, 20].

[1] Binance: `https://www.binance.com/`
[2] Coinbase: `https://www.coinbase.com/`
[3] Kraken: `https://www.kraken.com/`
[4] Metamask: `https://metamask.io/`
[5] Solflare: `https://solflare.com/download`
[6] Myeherwallet: `https://www.myetherwallet.com/`

It is interesting to analyze the evolutionary trajectory of these solutions. For instance, [12] introduces a two-server password-authenticated secret sharing (2PASS) scheme, wherein users initially distribute secret shares of their password and secret key to both servers. Later, they can retrieve these shares by sending new secret shares of their password to the same servers, which collaboratively determine whether all received secrets constitute shares of the same password. Similarly, [16] presents a secret key backup and recovery protocol that relies on the threshold secret sharing algorithm, through which the wallet splits user's cryptographic keys into multiple shares and distributes them among carefully selected third parties. Along the same lines, [18] proposes HasDPSS, a blockchain-based key management system tailored for decentralized storage. Acknowledging the diverse reliability of participants, this system incorporates a hierarchical access structure to enable fine-grained key management, building upon the principles of secret sharing. Finally, works such as [19] and [20] enhance security by incorporating biometrics to encrypt the shares of private keys. However, all proposals in this category inherently rely on external entities to safeguard users' cryptographic materials, a significant limitation discussed in the introduction concerning widely used online wallets, impeding their ability to meet the requirements outlined in this paper.

Works falling into the second category, *multi-signature wallets*, share similarities with those introduced earlier, as they partition their secrets and enlist external parties to store the resulting fragments. Nonetheless, their primary emphasis lies in safeguarding cryptographic keys against misuse rather than prioritizing recoverability, which is presumed due to the distribution of keys or key shares among a group of participants within the system. Some proposals within this category are [21, 22, 23, 24, 25]. Notably, [22] exemplifies this approach with a threshold-optimal signature algorithm, where the secret key is distributed among n parties, and at least t of them are required to sign a transaction. Additionally, [23] presents an advancement over [22] by leveraging homomorphic encryption to reduce computational costs. While the focus on safeguarding against misuse is a relevant aspect of these solutions, two significant drawbacks emerge: firstly, correct key recovery depends on the involvement of a sufficient number of participants, potentially resulting in temporary unavailability or prolonged recovery times if participants are intermittently offline; and secondly, there exists a clear reliance on external entities, which diverges from the requirements outlined in this paper.

In the solutions that fall into the third and last category, *seed-derived wallets*, cryptographic material is generated by deriving it from an initial seed. A common method involves generating this seed from mnemonic words, which users must commit to memory to enable recovery of their cryptographic keys. This approach is notably employed in popular web browser extension wallets like Metamask, as introduced earlier in the paper. However, should a user forget these mnemonic words, all associated data on the blockchain linked to the corresponding secret keys would be irrevocably lost. Moreover, writing down these mnemonic words introduces a potential security vulnerability. To mitigate these concerns, alternative approaches leveraging biometric data or personal information have been devised by the scientific community. These proposals offer key recovery without the need for third-party storage of cryptographic keys or related core information used for recovery. Noteworthy schemes within this category are [26, 27, 28, 29]. Given their direct alignment with the requirements outlined in this paper, these are discussed in further detail.

In [26], the authors propose an innovative approach to key generation and recovery in a healthcare blockchain, utilizing biosensors alongside *fuzzy vault* techniques. Prior to transmitting physiological signals to the healthcare blockchain, a biosensor node generates a key using additional physiological signals and fuzzy vault technology. This key is then employed to encrypt the respective signals. Later, when a user seeks to retrieve her physiological data from the blockchain, she can use her biosensor node to recover the encryption key and utilize it for data restoration. However, a significant drawback of this method arises if a user experiences drastic changes in physiological signals, such as those resulting from accidents or illnesses, rendering key recovery impossible. Furthermore, there exists a potential vulnerability wherein an adversary capable of intercepting the physiological signals used in seed generation could gain access to the user's keys.

Authors in [27] introduce the "Partial Knowledge Recovery Scheme" (PKRS), which facilitates the retrieval of an encrypted private key by leveraging personal security questions. When a user intends to secure a private key, she is prompted with a series of questions. Next, the answers to these questions are utilized to encrypt fragments of the private key, resulting in the creation of a secured private key. This secured private key comprises an array of tuples, each containing a question alongside a private key fragment encrypted with the corresponding answer. Upon requesting the recovery of the associated private key, the user must provide answers to a subset of the stored security questions. These answers, stored within the secured private key, activate and combine the fragments, ultimately enabling key recovery. Notably, this scheme requires no external entity during the recovery process, as all necessary

information is contained within the secured private key itself. However, it is imperative to acknowledge that a third party is still necessary for storing the secured private key. In addition to that, an adversary may use social engineering techniques to gather user information and deduce the answers to the security questions.

Seo et al. [28] present a comparable method but, in this case, images are used instead of questions. In particular, the authors propose a novel approach to private key generation and recovery, based on the concept that users retain lasting memories from distinct images. Cryptographic keys are generated through a process involving these images and their associated locations, chosen by the user to evoke natural memories. Later, the user's private key is derived based on the specified picture locations. During key recovery, the user must recall the locations of the chosen pictures from the initial key generation phase. However, a significant drawback of this approach surfaces: adversaries with knowledge of the picture locations may exploit this information to retrieve the corresponding private key. Moreover, the effectiveness of this method heavily relies on the user's long-term memory capabilities, thereby posing challenges similar to those encountered by wallets utilizing mnemonic words for seed generation.

The authors of [29] advocate for the implementation of hardware wallets to safeguard cryptographic material. Their proposal involves employing an Elliptic-Curve Diffie-Hellman (ECDH) key agreement protocol to back up hardware wallets. Through this method, the root of private keys is transferred between hardware wallets, addressing potential Man-In-The-Middle attack vulnerabilities associated with ECDH through side-channel human visual verification facilitated by the display screen on a hardware wallet. Consequently, the user possesses two hardware wallets with identical private keys, with one serving as the primary wallet and the other as a backup. However, in our operational context, where a new key pair is generated for each service access, adopting this scheme would impose a significant overhead on the user. In particular, each key pair would need manual securing, rendering the process cumbersome and impractical.

1.2. Contributions and organization of the paper

This work tackles the challenges previously outlined by presenting a comprehensive solution: a fully decentralized multi-platform wallet leveraging blockchain and InterPlanetary File System (IPFS) technologies for managing asymmetric keys and facilitating key recovery.

In essence, our novel approach empowers users to engage with dApps built on smart contracts (SCs) deployed on the blockchain, all the while safeguarding their privacy against potential attackers seeking to uncover sensitive information. Additionally, our proposed system ensures that users can seamlessly recover their keys in the event of device theft or damage.

The new proposal falls within the category of *seed-derived wallets*. As previously discussed, this type of scheme generates cryptographic keys by deriving them from an initial seed, which may consist of a list of mnemonic words, as seen in the widely used Metamask wallet, or from biometric data and personal information, as referenced in [26, 27, 28, 29]. Specifically, the new system adopts the same approach as Metamask, utilizing mnemonic words as the initial seed. While safeguarding these sensitive words poses its own security challenges, we consider, as discussed in the related work section, that relying on biometric data and personal information presents even greater problems.

The rest of this paper is organized as follows: Section 2 offers a brief overview of the technologies and concepts used in our contribution. Section 3 elaborates on the new scheme, detailing the design requirements, system architecture, governing smart contract, and a high-level description of the protocols employed. Section 4 formalizes the protocols that comprise our solution. Section 5 assesses the compliance with functional requirements. Section 6 is dedicated to analyzing the security and privacy aspects of the proposal. Section 7 provides a comparative analysis between the new system and existing counterparts in the industry and literature. Section 8 focuses on implementing a preliminary prototype for validating the system's feasibility. Finally, Section 9 presents the concluding remarks.

2. Background

This section introduces a brief overview of the primary technologies and concepts employed in the new contribution.

2.1. ECDSA

Elliptic Curve Digital Signature Algorithm (ECDSA) [30] consists of a variant of the Digital Signature Algorithm (DSA) using elliptic-curve cryptography.

The signature process begins by having the two parties, Alice and Bob, agree on the curve parameters ($CURVE$, G , n). In addition to the field and equation of the curve, they need G, a base point of prime order on the curve; being n the multiplicative order of the point G. Assuming that every nonzero element of the ring $\mathbb{N}/n$ is invertible, $\mathbb{N}/n$ must be a field. It implies that n must be prime according to Bézout's identity.

Once the parties have agreed on the parameters, Alice creates a key pair. This consists of a private key integer d_A, randomly selected in the interval $[1, n-1]$; and a public key curve point $Q_A = d_A \times G$ ($\times$ denotes the elliptic curve point multiplication by a scalar).

Then, for Alice to sign a message m, she performs the following steps:

1. Calculate $h = H(m)$, where $H()$ is a cryptographic hash function with output converted into an integer.
2. Let z be the L_n leftmost bits of h, where L_n is the bit length of the group order n (z can be greater than n, but it cannot be longer).
3. Select a cryptographically secure random integer k from $[1, n-1]$.
4. Calculate the curve point $(x_1, y_1) = k \times G$.
5. Calculate $r = x_1 \bmod n$. If $r = 0$, go back to step 3.
6. Calculate $s = k^{-1}(z + rd_A) \bmod n$. If s = 0, go back to step 3.
7. The signature is the pair (r, s). (And ($r, -s \bmod n$) is also a valid signature.)

Once the signature process has been completed and Alice has sent m, (r, s) to Bob, he can authenticate the signature by using a copy of her public-key curve point Q_A. But first, Bob must verify that Q_A is a valid curve point as follows:

1. Check that Q_A is not equal to the identity element O, and its coordinates are otherwise valid.
2. Check that Q_A lies on the curve.
3. Check that $n \times Q_A = O$.

After that, Bob authenticates the signature as follows:

1. Verify that r and s are integers in $[1, n-1]$. If not, the signature is invalid.
2. Calculate $h = H(m)$, where $H()$ is the same function used in the signature generation.
3. Let z be the L_n leftmost bits of h.
4. Calculate $u_1 = zs^{-1} \bmod n$ and $u_2 = rs^{-1} \bmod n$.
5. Calculate the curve point $(x_1, y_1) = u_1 \times G + u_2 \times Q_A$. If $(x_1, y_1) = O$ then the signature is invalid.
6. The signature is valid if $r \equiv x_1 \bmod n$, invalid otherwise.

Note that an efficient implementation would compute inverse $s^{-1} \bmod n$ only once. Also, using Shamir's trick, a sum of two scalar multiplications $u_1 \times G + u_2 \times Q_A$ can be calculated faster than two scalar multiplications done independently.

On the signature process, it is not only required for k to be secret, but it is also crucial to select different k for different signatures. Otherwise, an attacker can find d_A as: given two signatures (r, s) and (r, s'), employing the same unknown k for different known messages m and m', an attacker can calculate z and z'. Since $s - s' = k^{-1}(z - z')$, the attacker can find $k = \frac{z-z'}{s-s'}$. Finally, using the equation in step 6, $s = k^{-1}(z + rd_A)$, the attacker can calculate the private key $d_A = \frac{sk-z}{r}$. All operations in this paragraph are done modulo n.

From now one we denote the key pair (d_a, Q_A) as (SK, PK) for the private and the public key respectively.

CURVE	the elliptic curve field and equation used
G	elliptic curve base point, a point on the curve that generates a subgroup of large prime order n
n	integer order of G, means that $n \times G = O$, where O is the identity element
d_A	the private key (randomly selected)
Q_A	the public key $d_A \times G$ (calculated by elliptic curve)
m	the message to send

Table 1: ECDSA Parameters

2.2. Crypto wallets

In the domain of blockchain transactions, as previously stated, the individual initiating the transaction validates it by signing with a private key, while other nodes utilize public keys for transaction verification. These private keys serve as the "passwords" for the transaction address and must be securely stored to prevent unauthorized access and misuse of linked assets. Crypto wallets serve as the means to achieve this security.

Crypto wallets are hardware or software applications used for storing private keys essential for transactions while also functioning as client software enabling blockchain operations. By means of these wallets, individuals can seamlessly manage their assets published on the blockchain, including cryptocurrencies and smart contracts, leveraging their private keys to authorize transactions. Furthermore, selected crypto wallets extend their utility by granting access to dApps and assorted decentralized services, thereby offering users a comprehensive ecosystem for managing and interacting with their digital resources.

A wallet comprises private keys, serving as both the repository for these keys and the primary interface of crypto clients. This interface traditionally integrates various functionalities beyond key storage, including transaction history, address book management, and currency exchange rate tracking.

According to [13] and [31], there are various wallet implementations leveraging different technologies. This diversity permits us to categorize them into the following groups:

- *Web (online) wallets*: Services accessible to users through a browser interface, eliminating the need for downloading or installation. One advantageous aspect of these services is their universal accessibility, enabling users to connect from any location and device via the internet. However, while many web wallets offer users control over their keys, the management and storage of cryptographic keys typically rest with a third party. Consequently, the security of these web wallets relies on the provider's handling of these keys and the level of trust users place in them.

- *Mobile wallets*: Mobile applications available on Android and iOS platforms where cryptographic keys are generated and securely stored. These wallets are often considered more secure than online counterparts, offering users greater control over their keys. However, they require regular data backups to prevent the complete loss of cryptographic keys in the event of device malfunction, loss, or security breach.

- *Desktop wallets*: Applications installed on personal computers provide users with complete control over their cryptographic keys. These keys are typically stored within an encrypted file, safeguarded by a private password known only to the user. Maintaining the secrecy and safety of this password is paramount, as its loss would entail losing access to all private keys and associated assets stored within the blockchain.

- *Hardware wallets*: Offline devices, such as USB drives or physical sheets with printed private keys, offer enhanced security. By storing private keys offline, they mitigate the risk of hacking. Additionally, many of these devices are equipped for online transactions, featuring compatibility with various internet interfaces.

On the other hand, [32] and [33] further categorize wallet implementations into more specific classifications, considering not only the devices used to store keys but also the methods by which these keys are managed:

- *Keys in local storage*: Software manages multiple private keys by storing them in the device's local storage, often within a file or database. When initiating a new transaction, a blockchain client can access these keys and broadcast the transaction across the network. This approach offers several advantages, including the elimination of cognitive overhead for users, as the software manages key access. Moreover, this system can generate and store a virtually unlimited number of keys due to their small size. However, the reliance on local storage poses various security threats to consider, such as unauthorized access by other applications, potential malware attempting to breach the keys file, and risks associated with physical theft, loss, or equipment malfunction.

- *Password-protected (encrypted) wallets*: Private keys are stored locally, similar to the preceding type of wallet, yet the wallet file performs encryption using a key derived from a user-selected password or passphrase. This measure mitigates the risk of physical theft, although it requires the user's recall of the password used for encrypting the keys. If the password is forgotten, access to all blockchain assets associated with the aforementioned keys is lost. Additionally, it is worth mentioning that a user cannot access the keys on a new device by simply entering the password, she must also transfer the encrypted wallet file to this new device.

- *Offline storage of keys*: Keys are stored offline on some form of portable media, such as a USB or a piece of paper (as in previously explained *Hardware wallets*). While this approach offers the advantage of resilience against malware attacks, it renders the wallet inaccessible for immediate software use, preventing users from accessing their assets in a easy and comfortable way.

- *Air-gapped key storage*: Similar to the previous implementation, wallets are stored on a secondary device capable of generating, signing, and exporting transactions without ever being connected to a network. This method enhances resistance to key theft, as the keys are never directly exposed on an internet-connected device.

- *Password-derived keys*: Cryptographic keys are generated based on a password chosen by the user. However, a limitation of this approach is that it only creates one keypair, requiring the user to choose a new password for each new keypair or adopt a more robust solution, such as the *Bitcoin Improvement Proposal 32 (BIP-32)* [34], i.e., hierarchical deterministic wallets.

- *Hosted wallet*: Private keys are managed by a third-party entity, allowing users to access their keys or transactional functionalities via a hosted-wallet web service using standard web authentication methods like passwords or two-factor authentication. Additionally, providers of hosted wallet web services may offer smartphone applications as clients, which offer the advantage of simplicity compared to web-based interfaces. However, similar to Web wallets, the primary concern with this approach lies in the security of cryptographic keys, which relies on the provider's practices and the level of trust users place in them.

2.2.1. Bitcoin Improvement Proposal 39 (BIP-39)

BIP-39 [35] describes the implementation of a mnemonic code, which is a series of easily memorable words, for generating deterministic wallets. Employing mnemonic codes or phrases enhances the usability of managing a wallet seed for humans compared to dealing with raw binary or hexadecimal representations. The protocol consists of two main steps: i) generating a mnemonic code; and ii) converting these words into a binary seed. This seed can later be used to generate deterministic wallets using BIP-32 or similar methodologies.

The aforementioned steps work as follows:

1. For the generation of the mnemonic code a 32 bits multiple entropy variable ENT is generated (size between 128 and 256 bits): $Gen(128 - 256) \rightarrow ENT$.
2. Next, a checksum is generated by calculating the *SHA256* hash of ENT and taking the first size-of-ENT divided by 32 bits: $CS = Parse_{(|ENT|/32)}(Hash256(ENT))$
3. This checksum is appended to the end of the initial entropy and the resulting concatenated bits are split into groups of 11 bits: $[MS] = split_{11bits}(ENT|CS)$. Each one of these groups encodes a number from 0-2047, serving as an index into a wordlist.
4. Finally, these numbers are converted into words and used to form a mnemonic sentence. The mnemonic can be protected with a passphrase; if not, an empty string is used instead.

Table 2 illustrates the relation between the initial entropy length (ENT), the checksum length (CS), and the resulting length of the generated mnemonic sentence (MS) in words.

CS = ENT / 32
MS = (ENT — CS) / 11

ENT	CS	ENT+CS	MS
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

Table 2: Length relation between ENT, CS and MS

To generate a binary seed, the PBKDF2 function is used with a mnemonic sentence (in UTF-8 NFKD) as the password, while using the mnemonic along with the passphrase (also in UTF-8 NFKD) as the salt. The iteration count is set to 2048, and HMAC-SHA512 serves as the pseudo-random function. The resulting derived key has a length of 512 bits (equivalent to 64 bytes).

2.2.2. Bitcoin Improvement Proposal 32 (BIP-32)

BIP-32 [34] establishes a standard for hierarchical deterministic wallets, enabling seamless interoperability across various clients while fully leveraging all generated key pairs. This specification consists of two parts: i) the methodology for deriving a hierarchical structure of key pairs from a single seed; and ii) a comprehensive illustration of constructing a wallet framework on top of this hierarchical structure.

In the protocol explanation, we assume the use of elliptic curve cryptography, specifically adhering to the field and curve parameters delineated in secp256k1 [36]. The summation of two coordinate pairs is defined as application of the EC group operation, while the employment of the HMAC-SHA512 function adheres to the specifications outlined in RFC 4231 [37]. Moreover, we adopt the following standard conversion functions:

- $point(p)$: returns the coordinate pair resulting from EC point multiplication (repeated application of the EC group operation) of the secp256k1 base point with the integer p.
- $ser_{32}(i)$: serialize a 32-bit unsigned integer i as a 4-byte sequence, most significant byte first.
- $ser_{256}(p)$: serializes the integer p as a 32-byte sequence, most significant byte first.
- $ser_P(P)$: serializes the coordinate pair $P = (x, y)$ as a byte sequence: $(0x02\ or\ 0x03)||ser_{256}(x)$ (i.e., EC1's compressed form), where the header byte depends on the parity of the omitted y coordinate.
- $parse_{256}(p)$: interprets a 32-byte sequence as a 256-bit number, most significant byte first.

The protocol comprises a function tasked with deriving a set of child keys from a parent key. To mitigate dependence solely on the parent key during derivation, both private and public keys are extended with an additional 256 bits of entropy. This extension, denoted as the chain code, remains consistent for corresponding private and public keys, spanning 32 bytes. The extended private key is denoted as (SK, c), while the extended public key is represented as (PK, c), where SK denotes the standard private key, $PK = point(SK)$, and c signifies the chain code. Each extended key accommodates 2^{31} normal child keys and 2^{31} hardened child keys, each with an assigned index. Normal child keys are allocated indices ranging from 0 to $2^{31} - 1$, whereas hardened child keys utilize indices from 2^{31} to $2^{32} - 1$. To simplify notation for hardened key indices, i_H symbolizes $i + 2^{31}$.

Regarding the child keys, a vulnerability has been identified within BIP-32-compliant wallets. More specifically, if an attacker gains access to the master public key along with any child private key, she can then reconstruct the master private key. However, BIP-32 introduces a safeguard against this vulnerability by permitting "hardened" child private

keys, which can be compromised without jeopardizing the master private key. Additionally, these hardened keys lack the property of the master public key, which would otherwise enable anyone to derive a public child key from it. This property exposes a vulnerability wherein an attacker could discern the owner of a public child key through a brute force attack on the master public key.

The Child Key Derivation (CKD) algorithm depends on whether the child key is hardened or not (alternatively, whether $i \geq 2^{31}$), as well as whether we are referring to private or public keys. From a parent extended private key, a child extended private key and/or a child extended public key can be derived using CKD. Additionally, CKD can derive a child extended public key from another parent extended public key. However, our focus in this study lies on deriving private child keys from private parent keys exclusively, generating only hardened child private keys to circumvent the aforementioned vulnerability and to sidestep the master public key property. Accordingly, child private keys are computed via the function $CKDpriv((SK_{par}, c_{par}), i) \rightarrow (SK_i, c_i)$ as follows:

1. Check whether $i \geq 2^{31}$ (i.e., whether the child key is a hardened).
   (a) If it is hardened: let $I = HMAC - SHA512(Key = c_{par}, Data = 0x00||ser_{256}(SK_{par})||ser_{32}(i))$ (the 0x00 pads the private key to make it 33 bytes long).
   (b) If it is not hardened (it is a normal child key): do $i+ = 2^{31}$ and go back to step 1.
2. Split I into two 32-byte sequences, I_L and I_R.
3. The returned child key SK_i is $parse_{256}(I_L) + SK_{par}\ mod(n)$.
4. The returned chain code c_i is I_R.
5. In case $parse_{256}(I_L) \geq n$ or $SK_i = 0$, the resulting key is invalid, and it is required to proceed with the next value for i. Note that this situation can happen with a probability lower than 1 in 2^{127}.

To start the execution of the derivation protocol, an initial key pair known as the master key is required. The generation of the master key involves the use of a seed value, employed in the following manner:

1. Generate a seed byte sequence S of a chosen length (between 128 and 512 bits; 256 bits is advised) from a pseudo-random generator. In this paper, we use the seed obtained from the BIP-39 protocol, as explained in Section 2.2.1.
2. Calculate $I = HMAC - SHA512(Key = "Bitcoinseed", Data = S)$.
3. Split I into two 32-byte sequences, I_L and I_R.
4. Use $parse_{256}(I_L)$ as master secret key, and I_R as master chain code. If $parse_{256}(I_L) = 0$ or $parse_{256}(I_L) \geq n$, the master key will be invalid.

2.3. InterPlanetary File System (IPFS)

The InterPlanetary File System[7], introduced by Juan Benet in [38], is a protocol that defines a file sharing peer-to-peer network for storing and sharing data in a distributed file system. Employing content-addressing, IPFS assigns a unique identifier to each file within a global namespace, interlinking IPFS hosts. Unlike conventional centralized servers, IPFS operates on a decentralized model, wherein user-operators collectively host portions of the overall data, thereby offering resilience in file storage and sharing. Any participant within the network can store a file using its content address, enabling any other peer to locate and request the content from any node possessing it via a distributed hash table (DHT).

In addition to its primary function as a data storage system, IPFS is frequently integrated with various other technologies, including blockchain. For instance, in [39], Kumar et al. introduce an IPFS-based blockchain storage model. Here, transactions are stored within the IPFS distributed file system, with only the transaction hash recorded in the blockchain block. This approach effectively addresses the storage challenges arising from the increasing volume of blockchain transactions.

However, in conventional P2P file-sharing network systems like IPFS, data stored in nodes remains immutable, as it can only be removed by other nodes and not by the data owner. Such a characteristic may raise concerns when managing certain types of data. To address this challenge, Yeh et al. proposed a solution in [40]: a monitorable P2P file-sharing system built upon a consortium blockchain, enabling file revocation within a decentralized environment. Their approach leverages a trusted execution environment (TEE), such as Intel Software Guard Extensions (SGX). This TEE ensures the integrity of the P2P file-sharing system's executables and generates a file authentication code for each IPFS node, thereby ensuring correct synchronization of the system.

[7]InterPlanetary File System (IPFS): `https://docs.ipfs.tech/`

3. The proposed solution

In this section, we describe in detail our proposed fully decentralized multi-platform wallet, leveraging blockchain and IPFS technologies to deliver its functionalities.

Initially, we introduce the set of requirements pivotal to the design of the new scheme. Next, we present the architecture of the proposed system. Following this, we describe the smart contract utilized for storing cryptographic keys. Finally, we provide a high-level overview of how the proposed system operates.

3.1. Requirements

In our comprehensive review of existing literature and in light of the identified weaknesses, we have identified some points that our multi-platform wallet must fulfill to effectively address these challenges.

Primarily, the new scheme must facilitate the generation and access of key pairs across various devices while ensuring convenient management for users. Furthermore, emphasizing the paramount importance of trust, our objective is to develop a solution that operates autonomously, thereby eliminating dependence on third-party entities. Additionally, our approach prioritizes user privacy and security, carefully addressing concerns regarding key recovery procedures and potential cyber-attacks. This holistic strategy not only ensures seamless functionality but also provides a robust defense against vulnerabilities, fostering a secure and trustworthy environment for users.

Aligned with these objectives, we have crafted a series of functional, security, and privacy requirements, which are introduced in the subsequent sections.

3.1.1. Functional requirements

The proposed system must fulfill a set of functional requirements in order to address the challenges highlighted in the current literature:

R1.1. Users should have the capability to generate keys from various devices utilizing the provided wallet.

R1.2. Key pairs should always be accessible to users, including the option to export them from the wallet for use in other applications if desired.

R1.3. Users should have the capability to recover all cryptographic keys in the event of loss of access to the wallet.

R1.4. The solution must be economically viable for users.

3.1.2. Security and privacy requirements

The proposed system must fulfill a set of security and privacy requirements in order to be robust enough. These requirements are built on the premise that the software components of the proposed system are correctly implemented and run on personal devices (i.e., smartphones, laptops, etc.) whose operating systems natively apply protection measures. These measures prevent adversaries with physical access to those devices from logging in, retrieving or altering sensitive data, or modifying the application code in any way.

R2.1. Keys must be kept confidential from adversaries.

R2.2. Keys should be safe against tampering by attackers.

R2.3. Keys should not be kept by a third party, thus ensuring that the system does not rely on external trust.

R2.4. Keys should remain unlinked to each other or to a specific user, ensuring that all generated keys are independent.

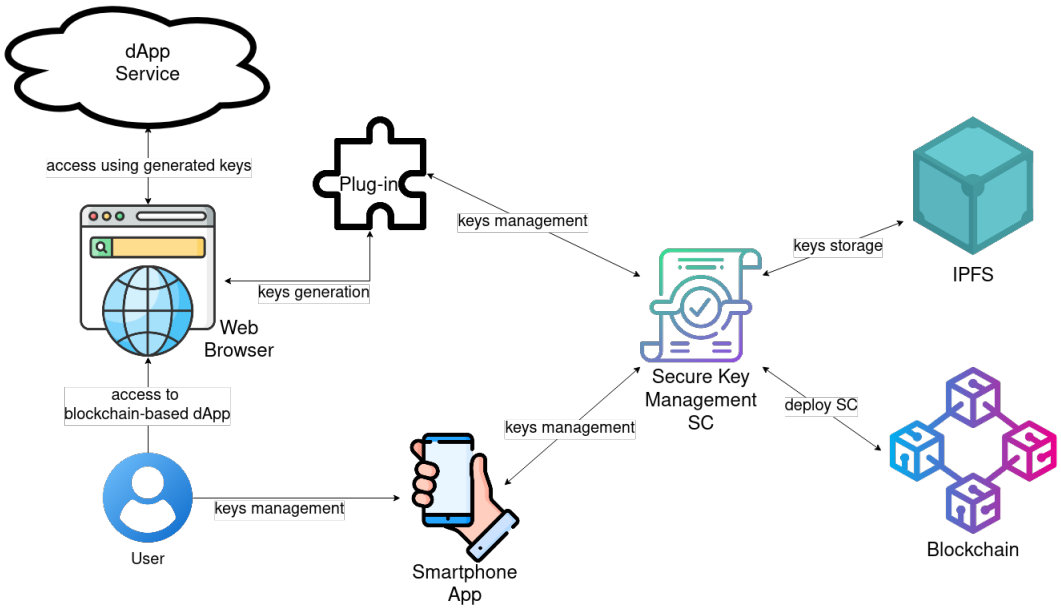


Figure 1: System's architecture

3.2. System architecture

Figure 1 depicts the general overview of the proposed system, where the *Plug-in* and the *Smartphone App* components are especially relevant and serve as access points for the users.

The *Plug-in* component is required to be installed on web browsers through which users access blockchain-based *dApps*. This plug-in is responsible for seamlessly generating cryptographic keys and deploying/utilizing smart contracts automatically and transparently for the user, facilitating the use of *dApps*.

The *Smartphone App* component serves as a smartphone application through which users can add and configure a new *Plug-in* installed on their device's browser. Additionally, this app allows users to recover all generated keys and access all deployed smart contracts, enabling seamless interaction with them.

The cryptographic key pairs generated by the *Plug-in* components are securely stored within the *Key Storage* component. This storage system leverages IPFS technology, making it fully decentralized. The IPFS solution enables key recovery by offering a distributed and secure storage layer that eliminates reliance on a central authority, without imposing any cost on users.

Interactions among the *Smartphone App*, the different *Plug-ins*, and the *Key Storage* system are facilitated through blockchain-based smart contracts. Specifically, the setup of the *Key Storage* and all related key management interactions are governed by the *Secure Key Management Smart Contract*. Further details about this smart contract will be provided in the subsequent section.

3.3. Secure Key Management Smart Contract

The *Secure Key Management Smart Contract (SKM SC)* is a smart contract deployed on the blockchain by each user of the proposed system. Its purpose is to enable user access to cryptographic keys stored on IPFS and facilitate information exchange between the browser plug-in and the smartphone during initial setup. To provide these functionalities, the smart contract comprises a single class that includes:

- *Smartphone app identifier*: This refers to the root public key of the smartphone ("smartphoneID" argument). This value is automatically configured during the deployment of the smart contract, with the public key associated with the private key used to sign the deployment transaction. The public key serves as an encryption mechanism for all cryptographic keys generated by the various plug-ins.

- *Authorized plug-ins*: This comprises a list of public keys belonging to plug-ins authorized to interact with the smart contract, thereby adding references to new cryptographic keys stored in IPFS ("whiteList" argument).
- *References to cryptographic keys*: This mapping correlates the public keys of the plug-ins in the "whiteList" list with hash references to files stored in IPFS. These files contain encrypted cryptographic keys generated by each plug-in ("refsList" argument).
- *Temporal value*: This temporal data field serves as a medium for exchanging information between the Smartphone app and the plug-ins ("temp" argument).

Once the *SKM SC* instance is created (Algorithm 1), the following methods are available for interaction:

- The *addDevice()* and *removeDevice()* methods (Algorithms 2 and 3) facilitate the addition or removal of a new public key to the "whiteList" list. These methods are exclusively accessible via the smartphone application, employing the private key associated with the public key specified in "smartphoneID" for transaction signing.
- The *storeRef()* method (Algorithm 4) updates the value in the mapping corresponding to the public key associated with the private key used for transaction signing. This value represents a reference to a file in IPFS where the keys generated by a plug-in are stored. Access to this method is limited to devices whose public keys are stored in the "whiteList" list or to the smartphone application.
- The *modTemp()* method (Algorithm 5) updates the data within the "temp" field. Access to this method is restricted to the smartphone application.

The size of the smart contract grows with the addition of plug-ins into the system. Specifically, for each registered plug-in, the "whiteList" argument acquires a new 20-byte public key, while the "refsList" argument includes a boolean value and an IPFS reference, totaling 33 bytes. In aggregate, these components require two 32-byte memory slots (i.e., 64 bytes) within the smart contract. Note that, the contract's size remains unaffected by the number of cryptographic keys generated by these plug-ins. It is considered that this increase in size does not pose scalability concerns because each user has their own smart contract, which in turn manages only a limited number of devices used for internet browsing. According to Statista[8], the global average of devices and connections per person in 2023 was approximately 3.6.

Algorithm 1: *Constructor* generates a new Secure Key Management Smart Contract instance

Input: smartphone ID PK_{sp}, digitalSignature s_{sp}
Output: res
Initialisation: $res \leftarrow error$
if $verify(PK_{sp}, s_{sp})$ **then**
    $smartphoneID \leftarrow PK_{sp}$;
    $whiteList \leftarrow [\,]$;
    $refsList \leftarrow < K, V >$;
    $temp \leftarrow null$;
    $res \leftarrow success$;
Return: res

3.4. The proposed system in a nutshell

The proposed wallet operates through four distinct protocols, which are briefly summarized below and will be explained in detail in the following section.

[8]Statista: `https://www.statista.com/statistics/1190270/number-of-devices-and-connections-per-person-worldwide/`

Algorithm 2: *addDevice* adds a new public key to the "whiteList" list

Input: device ID PK_{device}, digitalSignature s_{sp}
Output: res
Initialisation: $res \leftarrow error$
if $verify(smartphoneID, s_{sp})$ **then**
    $whiteList.add(PK_{device})$;
    $res \leftarrow success$;
Return: res

Algorithm 3: *removeDevice* removes an existing public key from the "whiteList" list

Input: device ID PK_{device}, digitalSignature s_{sp}
Output: res
Initialisation: $res \leftarrow error$
if $verify(smartphoneID, s_{sp})$ **then**
    if PK_{device} *in whiteList* **then**
        $whiteList.remove(PK_{device})$;
        $res \leftarrow success$;
Return: res

Algorithm 4: *storeRef* updates the IPFS reference in the mapping corresponding to the device that has invoked the method, or the position associated with the specified device if the method is invoked by the smartphone app

Input: device ID PK_{device}, digitalSignature s, IPFS reference $IPFSref$
Output: res
Initialisation: $res \leftarrow error$
if $verify(PK_{device}, s)$ *OR* $verify(smartphoneID, s)$ **then**
    if PK_{device} *in whiteList* **then**
        $refsList[PK_{device}] \leftarrow IPFSref$;
        $res \leftarrow success$;
Return: res

Algorithm 5: *modTemp* modifies the data contained in the "temp" field

Input: temporal value $newTemp$, digitalSignature s_{sp}
Output: res
Initialisation: $res \leftarrow error$
if $verify(smartphoneID, s_{sp})$ **then**
    $temp \leftarrow newTemp$;
    $res \leftarrow success$;
Return: res

1. *Set-up*: The setup of the proposed wallet involves two steps. First, users must install and configure the provided *Smartphone App* on their smartphones. During this process, users receive a personal password and 24 mnemonic words that they must remember. Additionally, the Root cryptographic keys are generated, which are crucial for creating all subsequent cryptographic keys. Following this, users must install and configure the *Plug-in* in the browsers where they intend to use the wallet.
2. *Session key pair generation*: After completing the setup, users who wish to access blockchain-based *dApps* must follow this protocol to generate the necessary key pairs. During this process, the browser's *Plug-in* automatically detects the intent to interact with a *dApp*, and transparently generates and stores the key pairs. These key pairs are securely stored in IPFS, enabling their recovery in case of loss. By utilizing a fully distributed storage technology like IPFS, the wallet eliminates the need for third-party services that may require trust.
3. *Key inventory*: Users utilize the *Smartphone App* installed on their smartphones to execute this protocol and manage all cryptographic key pairs generated by the wallet to date. Specifically, the *Smartphone App* displays each key pair associated with a particular *Plug-in* and *dApp*. Additionally, the wallet offers options to export a selected key pair in PKCS#12 format, send it via email, or transfer it to an external device.
4. *Key recovery*: This protocol enables users to recover all their cryptographic keys if their smartphones running the *Smartphone App* are compromised, lost, or damaged. To do this, users must reinstall the *Smartphone App* and use their personal password and 24 mnemonic words to regenerate their original Root cryptographic keys. With these Root keys, they can access the fully distributed IPFS to recover all the derived keys used to interact with each *dApp*. If the files stored in IPFS are unavailable, the proposed wallet can use the Root keys to generate new keys and store them again in distinct locations within IPFS.

4. Protocols

In this section, we establish the formal protocols governing the proposed system, providing enough detail to facilitate their implementation. These protocols are: *Set-up*, *Session key pair generation*, *Key inventory*, and *Key recovery*.

4.1. Set-up

This protocol aims to install and configure both the *Smartphone App* and the web browser *Plug-in*. It is divided into two distinct subprotocols, each devoted to one of these components. It is worth mentioning that a *Smartphone App* is expected to handle multiple *Plug-in* components.

4.1.1. Setting-up the Smartphone App

Installing the *Smartphone App* involves the *User* and the app component itself to follow these steps (see Figure 2):

1. The *User* installs the *Smartphone App* in her/his smartphone and creates a new secure log-in, by specifying a username and a password *pw*.
2. The *Smartphone App* executes the BIP-39 protocol using *pw* as input, generating 24 mnemonic words and a 512-bit seed, known as the *BIP-39 seed*.
3. The *User* securely stores the 24 mnemonic words, as they are essential for key recovery in the event of compromise to either the smartphone or the key storage.
4. The *Smartphone App* executes the BIP-32 protocol using the *BIP-39 seed* as input. This process generates the Root Secret Key SK_0 and the Root Chain Code C_0.
5. The *Smartphone App* stores SK_0 in the smartphone's secure storage.
6. The *Smartphone App* derives the Root Public Key PK_0 as follows: $PK_0 = SK_0 \times G$ (see Section 2.1).
7. The *Smartphone App* uses the Root Key pair (SK_0, PK_0) to deploy a new *SKM SC* onto the blockchain, resulting in the generation of a hash h_{SC} that uniquely identifies the published smart contract.

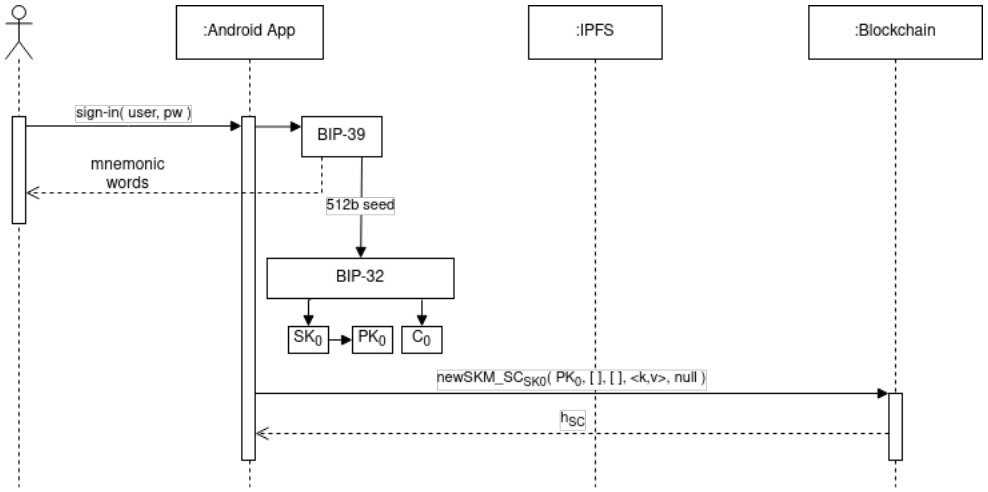


Figure 2: Set-up the Smartphone App

4.1.2. Setting-up the web browser Plug-in

Installing and configuring the *Plug-in* component in the *User*'s chosen browser requires the collaboration of these entities along with the already deployed *Smartphone App* to complete the following steps (see Figure 3):

1. The *User* installs the provided *Plug-in* in a browser of her/his choice.
2. The *User* introduces the h_{SC} (i.e., the *SKM SC* identifier) into the *Plug-in*.
3. The *Plug-in* retrieves PK_0 from the smart contract.
4. The *Plug-in* uses a secure symmetric cryptosystem (e.g., AES [41]) to generate a new session key K_i.
5. The *Plug-in* generates a QR image containing the computed session key K_i and the browser's identifier i.
6. The *Smartphone App* scans the QR image shown in the web browser by the *Plug-in* and retrieves K_i and i.
7. The *Smartphone App* employs SK_0, C_0, and the hashed browser's identifier, denoted as $SHA256(i)$, as inputs to the BIP-32 protocol. This process generates the Master Secret Key SK_i and the Master Chain Code C_i specific to that particular *Plug-in*.
8. The *Smartphone App* derives the corresponding Master Public Key PK_i as follows: $PK_i = SK_i \times G$ (see Section 2.1). This element unequivocally identifies that specific *Plug-in* component.
9. The *Smartphone App* adds PK_i to the authorized *Plug-in* components list of the *SKM SC* (referred to as the *whileList* argument) using the *addDevice()* method.
10. The *Smartphone App* uses the session key K_i to encrypt the set (SK_i, PK_i, C_i). This is, $Enc_{K_i}(SK_i, PK_i, C_i)$.
11. The *Smartphone App* stores the pair $(Enc_{K_i}(SK_i, PK_i, C_i), i)$ in the temporal data field of the *SKM SC* (referred to as the "temp" argument) using the *modTemp()* method.
12. The *Smartphone App* deletes the set (SK_i, PK_i, C_i) from its memory.
13. The *Plug-in* retrieves the pair $(Enc_{K_i}(SK_i, PK_i, C_i), i)$ from the "temp" argument of the *SKM SC*.
14. The *Plug-in* uses the session key K_i to decrypt $Enc_{K_i}(SK_i, PK_i, C_i)$, obtaining the set (SK_i, PK_i, C_i).

Note that each web browser the *User* intends to utilize requires its specific *Plug-in* component. Consequently, this sub-protocol must be performed for each necessary web browser *Plug-in*.

4.2. Session key pair generation

Upon completion of the *Set-up protocol*, users can seamlessly access various blockchain-based *dApps*. To facilitate these accesses, the installed *Plug-in* in the web browser must automatically and transparently generate and manage the needed cryptographic keys. The steps involved in this process are next detailed (see Figure 4):

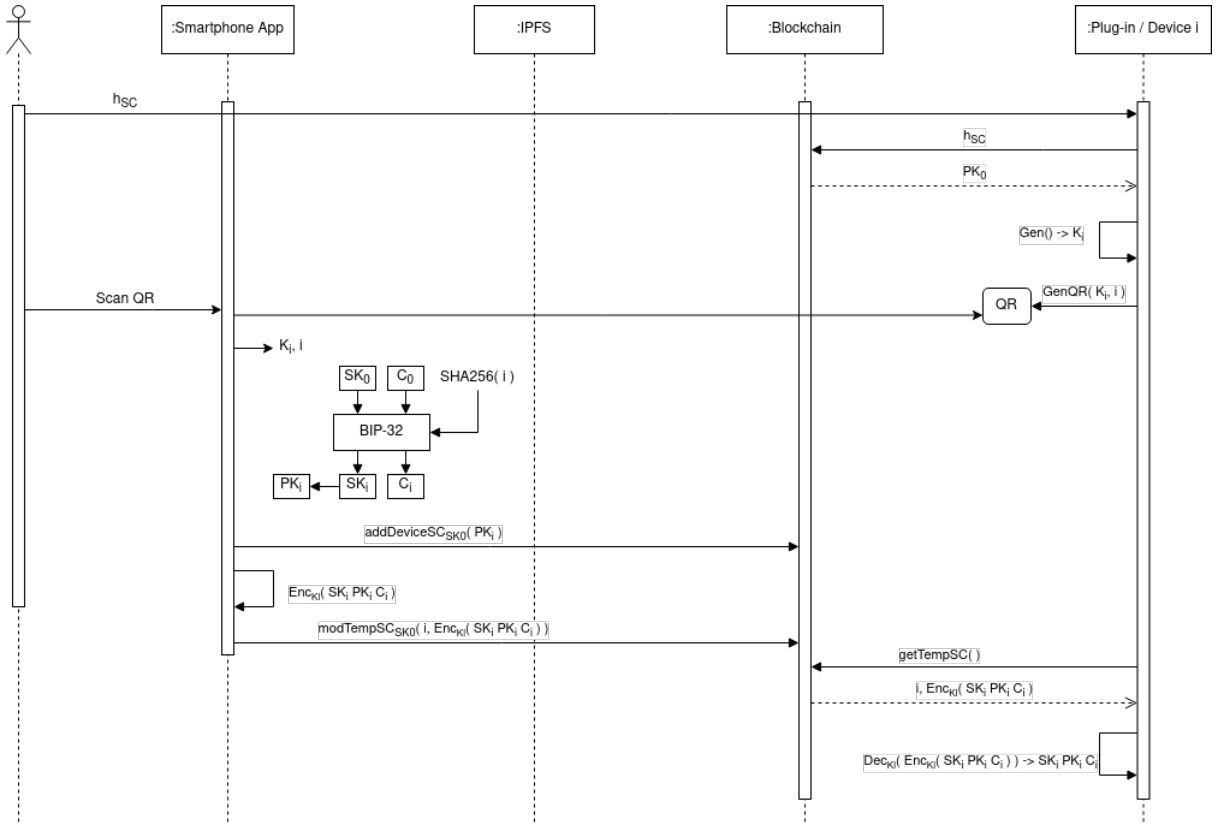


Figure 3: Set-up the web browser Plug-in

1. The *User* accesses a new blockchain-based *dApp*.
2. The *Plug-in* computes a hash value based on the domain name j of the *dApp* service provider, which serves to uniquely identify the specific *dApp*. This hash value is generated using the function $SHA256(j)$.
3. The *Plug-in* uses the browser's Master Private Key SK_i, Master Chain Code C_i, and the *dApp* identifier $SHA256(j)$ as inputs to the BIP-32 protocol. This process generates a Secret Key SK_{ij} and a Chain Code C_{ij} specific to that particular *dApp*. Then, the *Plug-in* derives the corresponding Public Key PK_{ij} as follows: $PK_{ij} = SK_{ij} \times G$ (see Section 2.1).
4. The *Plug-in* provides the key pair (SK_{ij}, PK_{ij}) to the *User*, facilitating seamless interaction with the dApp. While the detailed implementation of this aspect falls outside the scope of this paper, one possible approach could emulate the user interface of browser plugins such as Metamask, presenting the key pair to the *User* and offering the choice between exporting it in [9] or uploading it to the *Key Storage* system.
5. The *Plug-in* uses a secure symmetric cryptosystem (e.g., AES [41]) to generate a new session key K_{ij}.
6. The *Plug-in* uses the session key K_{ij} to encrypt the set (SK_{ij}, PK_{ij}, j). This is, $Enc_{K_{ij}}(SK_{ij}, PK_{ij}, j)$.
7. The *Plug-in* uses the Root Public Key PK_0 to encrypt the session key K_{ij}. This is, $Enc_{PK_0}(K_{ij})$.
8. The *Plug-in* stores the set ($Enc_{K_{ij}}(SK_{ij}, PK_{ij}, j)$, $Enc_{PK_0}(K_{ij})$) in the IPFS. This process entails a series of automated sub-steps that facilitate subsequent access to the cryptographic keys by the *Smartphone App*.
   (a) The *Plug-in* obtains the hash reference h_i of the file stored in the IPFS, which contains all the encrypted cryptographic keys generated by the *Plug-in*. This retrieval is fulfilled by leveraging its identifier PK_i within the "refsList" mapping contained in its associated *SKM SC*. If the "refsList" field is empty, the procedure goes directly to step 8c, where the plug-in generates a new file with a new IPFS reference.

[9] Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1: `https://datatracker.ietf.org/doc/html/rfc3447`

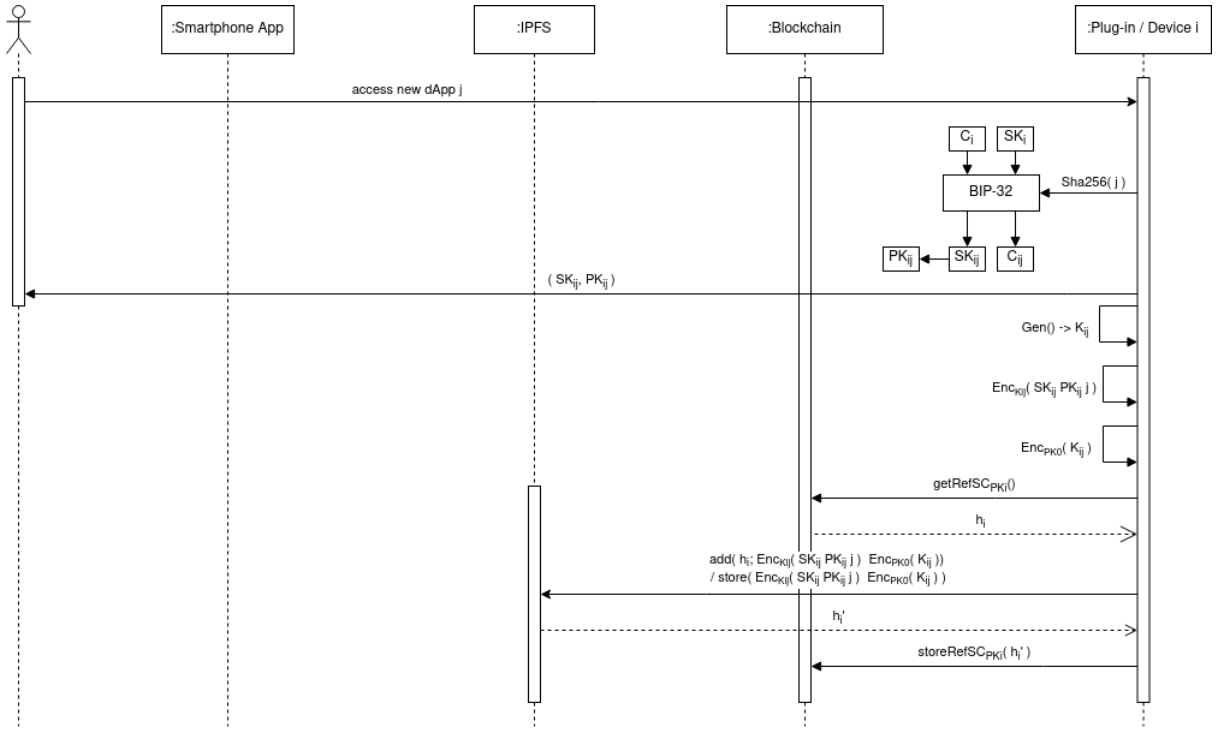


Figure 4: Session key pair generation

(b) The *Plug-in* gathers the $file_{h_i}$ from the IPFS.
(c) The *Plug-in* updates $file_{h_i}$ by appending the set $(Enc_{K_{ij}}(SK_{ij}, PK_{ij}, j), Enc_{PK_0}(K_{ij}))$. This process generates an updated reference h_i' for the stored file.
(d) The *Plug-in* uses the *storeRef()* method of its associated *SKM SC* to update the "refsList" mapping with the new reference.

4.3. Key inventory

This process enables the *User* to manage and monitor all cryptographic key pairs she has generated. To achieve that, the *User* uses the *Smartphone App* component, which follows the next steps (see Figure 5):

1. The *Smartphone App* periodically retrieves the "refsList" value from the *SKM SC*, which includes the hash references h_i to the files stored in IPFS, each one linked to a certain *Plug-in i* and storing all encrypted cryptographic keys generated by this entity.
2. If a certain IPFS h_i has been modified, such as the reference h_i changing or a new file being added (when a *Plug-in* generates a new key pair for the first time), the *Smartphone App* retrieves the corresponding file $file_{h_i}$ from the IPFS. Else, the protocol goes to the first step.
3. The *Smartphone App* gets the encrypted keys linked to the updated $file_{h_i}$. This is, $Enc_{K_{ij}}(SK_{ij}, PK_{ij}, j)$, $Enc_{PK_0}(K_{ij})$.
4. The *Smartphone App* decrypts the session key K_{ij} using the Root Secret Key SK_0. This is, $Dec_{SK_0}(Enc_{PK_0}(K_{ij}))$.
5. The *Smartphone App* decrypts the set (SK_{ij}, PK_{ij}, j) using the session key K_{ij}. This is, $Dec_{K_{ij}}(Enc_{K_{ij}}(SK_{ij}, PK_{ij}, j))$. The decrypted set is stored in the *Smartphone App*'s file system.
6. The *Smartphone App* shows to the *User* each generated key pair (SK_{ij}, PK_{ij}) associated with a particular *Plug-in i* and a specific *dApp j*. Additionally, the system provides options for exporting a selected key pair in PKCS#12 format, sending it via email, or transferring it to an external device.

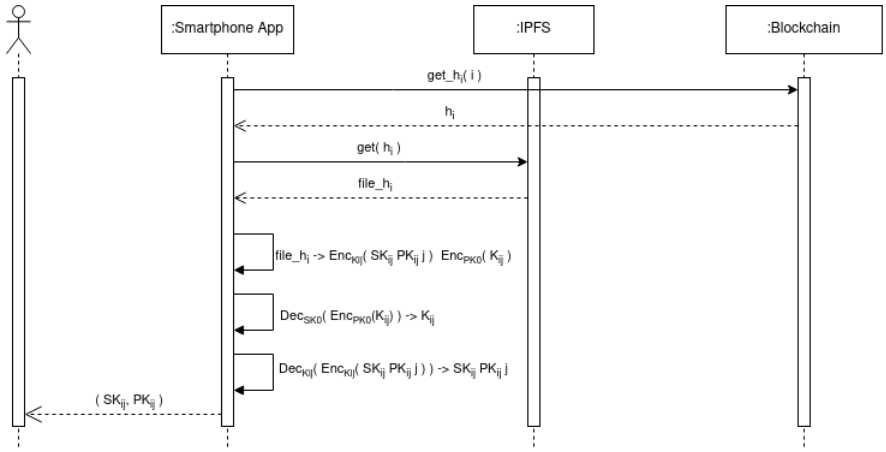


Figure 5: Key Management

4.4. Key recovery

The *User* should be capable of recovering her/his cryptographic keys if her/his device is compromised, lost, or damaged. Accordingly, this section details the recovery process of the proposed new wallet covering three scenarios: i) the *User*'s smartphone is no longer accessible (e.g. stolen, lost, or broken); ii) the IPFS files that store the keys generated by the web browser plug-ins are deleted; and iii) both situations occur simultaneously.

4.4.1. Smartphone App is no longer accessible

When a *User*'s smartphone is compromised, lost, or damaged, access to her/his *Smartphone App* is forfeited. Consequently, she also loses access to the Root Key pair (SK_0, PK_0) necessary for decrypting and accessing all cryptographic keys generated by the wallet. In such an event, the recovery key process entails regenerating this Root Key pair through the following steps (see Figure 6):

1. The *User* installs the *Smartphone App* on a new smartphone device and establishes a new secure login, by setting up her/his username and password *pw*.
2. The *User* inputs the 24 mnemonic words generated during the *Set-up protocol* into the *Smartphone App*, along with the initial password *pw* used during the wallet's initial configuration
3. The *Smartphone App* runs the BIP-39 protocol using the provided 24 mnemonic words and the initial password *pw*, thereby generating the original 512-bit seed known as the BIP-39 seed.
4. The *Smartphone App* executes the BIP-32 protocol using the BIP-39 seed as input. This process generates the original Root Secret Key SK_0 and Root Chain code C_0.
5. The *Smartphone App* stores SK_0 in the smartphone's secure storage.
6. The *Smartphone App* derives the Root Public Key PK_0 by means of SK_0 as follows: $PK_0 = SK_0 \times G$ (see Section 2.1).
7. The *Smartphone App* fetches the *SKM SC* published on the blockchain using the Root Key pair (SK_0, PK_0) and stores its hash h_{SC} in the app's storage.
8. The *Smartphone App* follows the procedure explained in Section 4.3 to retrieve all cryptographic key pairs previously generated.

4.4.2. IPFS file is unavailable

In the event that the IPFS nodes hosting the key pairs file generated from a specific browser *i* become inaccessible or delete the file, the *User* directs the *Smartphone App* to execute the following steps to regenerate the file (see Figure 7):

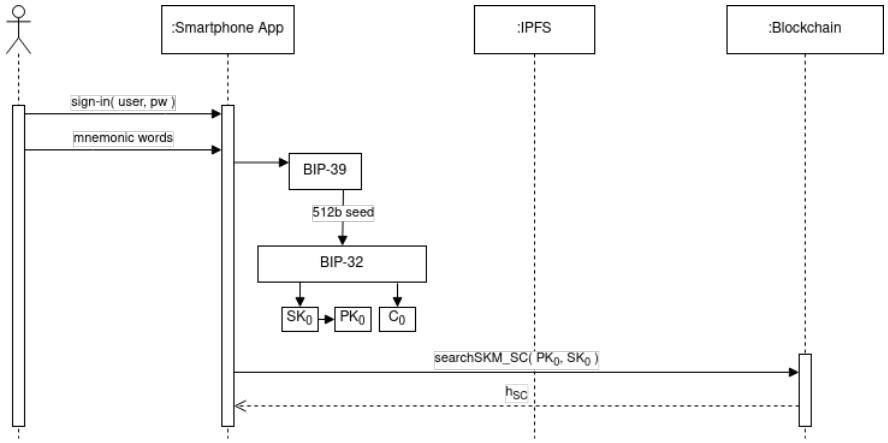


Figure 6: Key Recovery when Smartphone App is no longer accessible

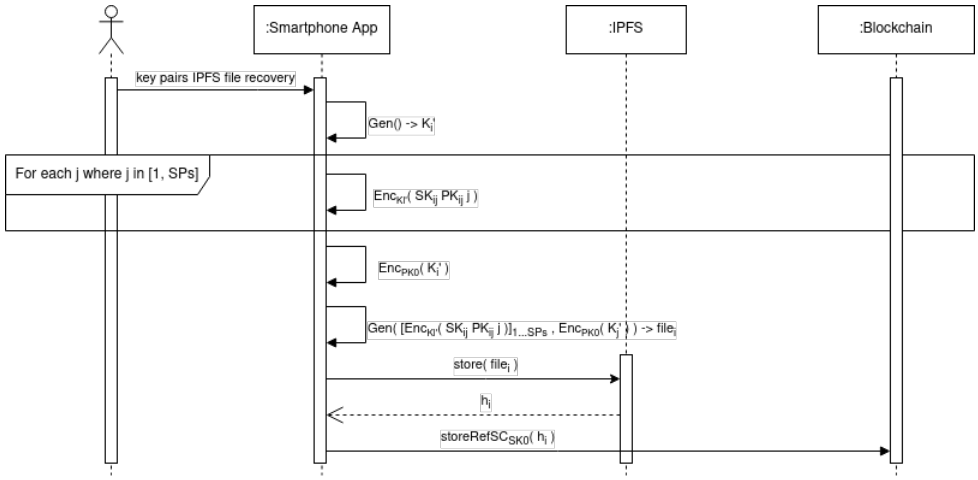


Figure 7: Key Recovery when IPFS file is unavailable

1. The *User* activates the process to regenerate the session key pairs stored in the IPFS through the *Smartphone App*.
2. The *Smartphone App* generates a new session key K'_i.
3. The *Smartphone App* retrieves from its file system all the stored key pairs that the *Plug-in i* generated to interact with each distinct *dApp j*.
4. The *Smartphone App* uses the new session key K'_i to encrypt all the key pairs retrieved in the former step. This is, $Enc_{K'_i}(SK_{ij}, PK_{ij}, j)$
5. The *Smartphone App* encrypts K'_i using the Root Public Key PK_0. This is, $Enc_{PK_0}(K'_i)$.
6. The *Smartphone App* adds all the newly re-encrypted key pairs and the session key into a new file, denoted as $file'_i$.
7. The *Smartphone App* stores $file'_i$ in the IPFS. This process returns the reference h'_i to the stored file.
8. The *Smartphone App* utilizes the *storeRef()* method within the SKM smart contract to update the "RefsList" field linked to the *Plug-in i* with the fresh reference h'_i. This method is run through a transaction signed by the Root Secret Key SK_0, with the *Plug-in*'s public key PK_i provided as an argument.

4.4.3. Lost access to Smartphone's app and IPFS file is unavailable

In the event of the *User* losing access to the *Smartphone App* and simultaneous unavailability or removal of the key pairs file stored by a specific *Plug-in i* on IPFS nodes, the recovery process for each key pair generated for each *dApp j* proceeds as follows (see Figure 8):

1. The *User* initiates the *Key recovery* process detailed in Section 4.4.1 (i.e., "*Smartphone App* is no longer accessible") to reinstall the *Smartphone App* and regain access to the Root key pair (SK_0, PK_0).
2. The *Smartphone App* retrieves the identifier *i* associated with the *Plug-in i* whose IPFS key pairs file is unavailable from the *SKM SC*.
3. The *Smartphone App* retrieves the Master Secret Key SK_i and Master Chain code C_i for the specific *Plug-in i* via the BIP-32 protocol. This procedure uses the Root Private Key SK_0, the Root Chain code C_0, and the hash of the *Plug-in* identifier $SHA256(i)$ as input parameters.
4. The *Smartphone App* uses SK_i to derive the *Plug-in*'s Master Public Key PK_i as follows: $PK_0 = SK_0 \times G$ (see Section 2.1).
5. The *Smartphone App* generates a new Secret Key SK_{ij} and a new Chain code C_{ij} for each *dApp j* via the BIP-32 protocol. This procedure uses the *Plug-in*'s Master Private Key SK_i, the *Plug-in*'s Master Chain code C_i, and the hash of the *dApp* identifier $SHA256(j)$ as input parameters.
6. The *Smartphone App* uses SK_{ij} to derive the corresponding Public Key PK_{ij} as follows: $PK_0 = SK_0 \times G$ (see Section 2.1).
7. The *Smartphone App* shows to the *User* each generated key pair (SK_{ij}, PK_{ij}) associated with a particular *Plug-in i* and a specific *dApp j*. Additionally, the system provides options for exporting a selected key pair in PKCS#12 format, sending it via email, or transferring it to an external device.

5. Functional analysis

This section examines the functional requirements outlined in Section 3.1.1. The following discussion comprises a series of propositions, each addressing a specific requirement, with multiple claims provided to support its fulfillment.

5.1. Proposition-1: Users must be capable of using different devices to generate their cryptographic keys

This proposition addresses requirement R1.1, which states that *Users* should have the capability to generate keys from various devices utilizing the proposed wallet. The following two claims support this assertion.

Claim 1. *The web browser Plug-ins are the components that generate the cryptographic keys needed to interact with the different dApps*

Proof. A *User* can install and configure a web browser *Plug-in i* on one of her/his devices by following the steps specified in 4.1.2. As a result of this process, *Plug-in i* acquires a Master key set (SK_i, PK_i, C_i). Next, when the *User* accesses a *dApp j*, *Plug-in i* uses SK_i, C_i and the hash of the *dApp*'s identifier $SHA256(j)$ as inputs in the BIP-32 protocol to compute a new key pair (SK_{ij}, PK_{ij}). This key pair is then used to interact with the specific dApp *j*. □

Claim 2. *Users have the flexibility to install and set-up a Plug-in within the web browser of each of their capable devices, allowing for multiple installations across different devices.*

Proof. As specified in the "Setup" protocol (see Section 4.1), each *User* owns a single *Smartphone App* operating within her/his smartphone. The *Smartphone App* is responsible for generating the 24 mnemonic words, the BIP-39 seed, the Root Keys (SK_0, PK_0), and the Root Chain Code C_0. Next, this cryptographic material is used during the installation of the distinct *Plug-ins* to compute the corresponding Master key sets (SK_i, PK_i, C_i) mentioned in the previous claim.

As shown in "Key inventory" protocol (see Section 4.3), the cryptographic keys generated by all *Plug-ins* linked to a particular *Smartphone App* are directly managed by the *User* through the *Smartphone App* itself. □

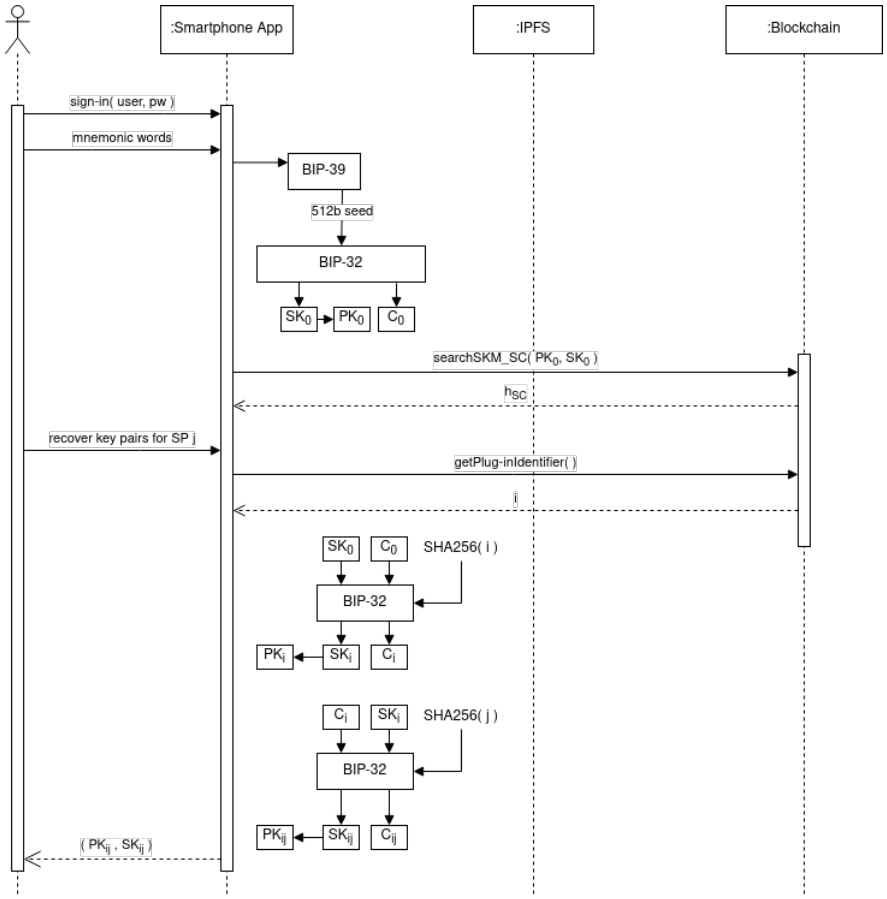


Figure 8: Key Recovery when Smartphone App is no longer accessible and IPFS file is unavailable

5.2. Proposition-2: Users must have the capability to access their cryptographic keys at any given moment and utilize them on alternative platforms

This proposition addresses requirement R1.2, which states that key pairs must be available for *Users* at any time, also in the case they want to export them from the wallet and import them in another application. The following three claims support this assertion.

Claim 3. *All key pairs (SK_{ij}, PK_{ij}) generated by a User through her/his installed Plug-ins to interact with dApps are stored in the IPFS, with corresponding references stored and accessible via the SKM smart contract.*

Proof. As specified in the "Session key pair generation" protocol (see Section 4.2), each *Plug-in i* stores distinct key pairs (SK_{ij}, PK_{ij}) generated for interaction with various dApps j in a file $file_{h_i}$ on IPFS. Each key pair is stored in the format of the set ($Enc_{K_{ij}}(SK_{ij}, PK_{ij}, j)$, $Enc_{PK_0}(K_{ij})$). The reference h_i of $file_{h_i}$ in IPFS is stored in the "refsList" field of the publicly accessible SKM smart contract. □

Claim 4. *A User can access the key pairs generated by the different Plug-ins using her/his Root Key pair (SK_0, PK_0).*

Proof. As specified in the "Key inventory" protocol (see Section 4.3), a *User* can use her/his *Smartphone App* to access the "refsList" field of the deployed SKM smart contract. From there, she can retrieve the $file_{h_i}$ file stored in IPFS, obtain the encrypted key pairs ($Enc_{K_{ij}}(SK_{ij}, PK_{ij}, j)$, $Enc_{PK_0}(K_{ij})$), and decrypt them using the Root Secret key SK_0 which is only known by the *Smartphone App*. □

Claim 5. *A User can export a selected key pair in PKCS#12 format using her/his Smartphone App, with options to send it via email or transfer it to an external device.*

Proof. As seen in the previous claim, a *User* can retrieve the key pairs generated by the different *Plug-ins* through her/his *Smartphone App*. Furthermore, this tool provides options for exporting a selected key pair in PKCS#12 format, sending it via email, or transferring it to an external device (refer to the "Key inventory" protocol in Section 4.3). □

5.3. Proposition-3: Users must have the capability to recover their cryptographic keys in the event of loss of access to the wallet

This proposition addresses requirement R1.3, stating that the proposed system should enable *Users* to recover all cryptographic keys generated by the various *Plug-ins* in the event of smartphone loss or change, resulting in the loss of access to the *Smartphone App*. The following two claims supports this assertion.

Claim 6. *A User can recover her/his Root Key pair (SK_0, PK_0) in the event of smartphone loss or damage.*

Proof. According to the "Key recovery" protocol (see Section 4.4), a *User* who has lost access to her/his *Smartphone App* (for instance, due to smartphone loss or damage) can install a new instance of the *Smartphone App* on a new device. Then, by inputting the 24 mnemonic words and the password *pw*, she can re-generate her/his Root Key pair (SK_0, PK_0) using the BIP-39 and BIP-32 protocols. □

Claim 7. *A User can recover all the key pairs generated by the different Plug-ins using her/his Root Key pair (SK_0, PK_0).*

Proof. After regenerating the Root Key pair, the *User* can utilize it to recover all key pairs generated by the various Plug-ins, as depicted in the previous Claim 4 □

5.4. Proposition-4: The new system must be economically viable for users

This proposition addresses requirement R1.4, stating that the proposed system should incur low costs in terms of blockchain gas consumption, ultimately leading to reduced economic costs. This factor is crucial, as it significantly impacts the feasibility of deploying the system in real-world scenarios.

To evaluate these costs, a realistic environment for the new wallet has been established. This setting is based on the following points:

- The personal blockchain *Ganache*[10], a widely adopted platform for Ethereum distributed application development, has been employed. Ganache empowers developers to deploy Smart Contracts, execute commands, and inspect data state, all while controlling chain operations and associated costs.
- The SKM smart contract, coded in Solidity[11], has been compiled using the Truffle suite[12], a well-known ecosystem for Web3[13] development. This suite offers a comprehensive development environment, asset pipeline, and testing framework for Smart Contracts using the Ethereum Virtual Machine (EVM).
- The new proposal has been deployed on the *Polygon Chain*[14] network, chosen after an analysis of some of the most relevant EVM-compatible networks concerning latency, throughput, and gas costs. Note that every interaction involving the deployment of a smart contract function requires a fee to compensate the mining node for processing and recording the transaction on the blockchain. Gas serves as the unit representing this fee in EVM-compatible networks. Users obtain gas from mining nodes by exchanging it for the network token (such as *Ether* in the Ethereum network). It is important to differentiate between gas and the network token: gas denotes a fixed cost for actions on the blockchain, while the network token is a fluctuating virtual currency used to acquire network resources. Table 3 provides a summary of the aforementioned analysis, highlighting the Polygon Chain as the most efficient network.

Network	Consensus Protocols	Token	Mining Block Time	Transaction Throughput	Gas Cost	Token Price	Tx Base Fee Cost
Ethereum Mainnet [15]	PoS	ETH	13s	13.1 tps	48 Gwei	2,637.81 $	2.66 $
Binance Smart Chain [16]	DPoS	BNB	3s	51.6 tps	3 Gwei	322.49 $	0.02 $
Polygon Chain	PoS	MATIC	2s	45.7 tps	136.9 Gwei	0.86 $	0.002 $
Fantom Opera Chain [17]	Lachesis	FTM	1s	2.5 tps	9.97 Gwei	0.39 $	0.01 $
BitTorrent Chain [18]	PoS	BTT	2s	0.04 tps	304015 Gwei	$9.7x10^{-7}$$	$6.19x10^{-6}$$
Avalanche Chain [19]	Snowman	AVAX	2s	17.3 tps	25 nAVAX	39.39 $	0.021 $
HECO Chain [20]	HPoS	HT	3s	5.32 tps	2.3 Gwei	1.44 $	$6.96x10^{-5}$$
Cardano [21]	PoS	ADA	20s	7 tps	0.17 ADA (per Tx)	0.53 $	0.09 $

Table 3: Comparison of EVM compatible blockchains (data gathered in February 2024)

The following claim supports this proposition.

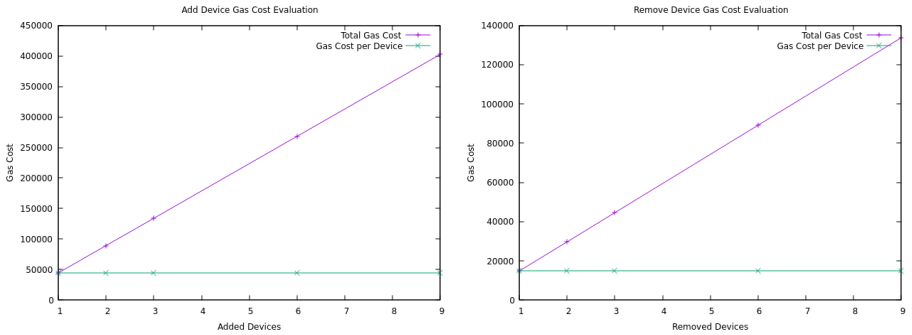
Claim 8. *When deployed on the Polygon Chain network, the new system demonstrates reduced costs in terms of gas, tokens, and dollars.*

Proof. The cost analysis of the functions provided by the SKM smart contract considers *Users* with 1 to 9 installed *Plug-ins* and 1 to 20 generated keys per *Plug-in*.

Table 4 illustrates that deploying the SKM smart contract requires 768490 gas, equivalent to 0.09 USD when utilizing the *Polygon Chain* network. Concerning the operational costs of the SKM smart contract, Figures 9(a) and 9(b) demonstrate that the *addDevice* and *removeDevice* functions entail a constant gas cost of 44713 and 14856, respectively. Similarly, Figure 10 indicates that the *modTemp* function carries a constant cost of 29426 gas. Finally, Figure 11 reveals that the *storeRef* function incurs an initial cost of 43049 gas due to the generation of the data structure for references. From that point, the cost remains constant at 28058 gas, notwithstanding of the number of loaded devices or stored key pairs in the system.

As a concluding remark, focusing on the cost in USD of running the smart contract functions as a measure of the economic feasibility of the new system, our findings on the *Polygon Chain* network indicate that all operations are economically viable. □

[10] Ganache: `Ganache:https://trufflesuite.com/ganache/`
[11] Solidity: `https://soliditylang.org/`
[12] Truffle suite: `https://trufflesuite.com/truffle/`
[13] web3j: `https://docs.web3j.io/`
[14] Polygon Chain: `https://docs.polygon.technology/`
[15] Ethereum Mainnet: `https://etherscan.io`
[16] Binance Smart Chain: `https://bscscan.com`
[17] Fantom Opera Chain: `https://ftmscan.com`
[18] BitTorrent Chain: `https://bttcscan.com`
[19] Avalanche Chain: `https://snowtrace.io`
[20] HECO Chain: `https://hecoinfo.com`
[21] Cardano: `https://cardano.org/`



(a) Add Device Operation Evaluation (b) Remove Device Operation Evaluation

Figure 9: Add and Remove Device Operation Evaluation

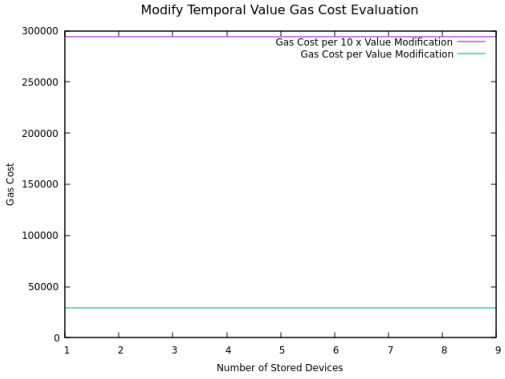


Figure 10: Modify Temporal Value Operation Evaluation

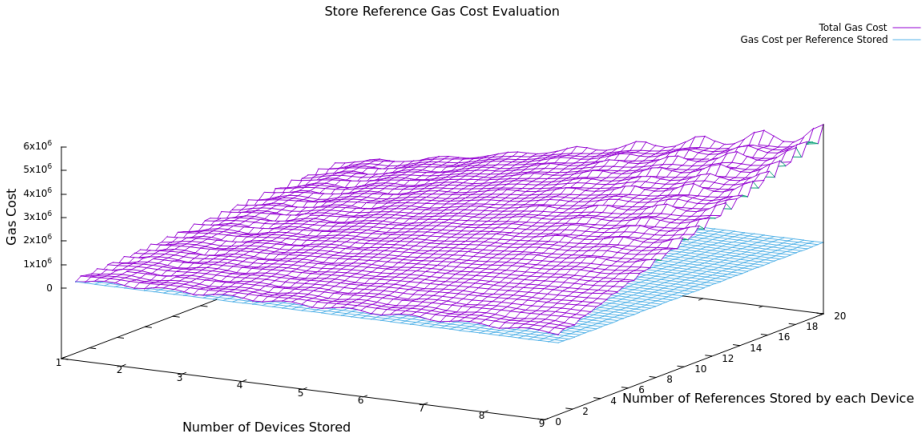


Figure 11: Store the Reference of a new Key Pair Operation Evaluation

Operation	Gas used	Cost (Token)	Cost (USD $)
newSKMSmartContract	768490	0.1052	0.0905
addDevice	44713	0.0061	0.0053
removeDevice	14856	0.0020	0.0017
modTemp	29426	0.0040	0.0035
storeRef	43049 - 28058	0.0059 - 0.0038	0.0051 - 0.0033

Table 4: Cost of the different functions of the *SKM SC*

Attack target	Attack description and coverage	
Key confidentiality	Attacker physically steals the smartphone and attempts to access all keys.	Addressed by the multi-factor authentication system in use (as stated in the attacker model).
	Attacker obtains the mnemonic words and tries to reconstruct all keys.	Covered in Proposition 5, Claim 9.
	Attacker attempts to steal the *Plug-in*'s master key pair during its transmission between the *Smartphone App* and the *Plug-in* or while stored in IPFS.	Covered in Proposition 5, Claims 10 and 11.
	Attacker tries to steal the *Plug-in*'s master key pair while stored on the smartphone or the *Plug-in* device.	Addressed by the native protection measures implemented by their respective operating systems (as stated in the attacker model).
	Attacker acts as a trusted third party and attempts to exploit this trust to access all keys.	Covered in Proposition 7, Claim 16.
Key integrity	Attacker tries to tamper with the *Smartphone App*'s keys stored on the smartphone.	Addressed by the native protection measures implemented by their respective operating systems (as stated in the attacker model).
	Attacker attempts to tamper with the *Plug-in*'s keys while stored on the smartphone or the *Plug-in* device.	Addressed by the native protection measures implemented by their respective operating systems (as stated in the attacker model).
	Attacker tries to tamper with the *Plug-in*'s keys during their transmission between the *Smartphone App* and the *Plug-in*.	Covered in Proposition 6, Claim 12.
	Attacker attempts to tamper with the reference to the *Plug-in*'s keys stored in IPFS or the uploaded file itself.	Covered in Proposition 6, Claims 13 and 14.
User privacy	Attacker tries to compromise the privacy of the *Users* by linking different keys or associating a specific key with a particular *User*.	Covered in Proposition 8, Claim 16.

Table 5: Types of attacks and their coverage

6. Security and privacy analysis

This section focuses on studying the security and privacy requirements of the provided system (see Section 3.1.2). The discussion is organized as a set of propositions, where each proposition may have several claims to support its fulfillment.

To provide context for this analysis, Table 5 maps the various anticipated attack types to the propositions and

claims that form the basis of this security evaluation. Additionally, the attacker model is based on the following assumptions:

- The computational power available to the attacker is insufficient to compromise modern computationally secure cryptosystems.
- An attacker who gains physical access to the *User*'s devices running the *Smartphone App* and the distinct web browser *Plug-ins* cannot log into the compromised devices due to the native protection measures implemented by their respective operating systems. These measures prevent adversaries from retrieving or altering sensitive data or modifying the application code in any way.
- All the underlying algorithms are implemented correctly, and the parameters meet the necessary security requirements.
- The digital wallet, installed on the smartphone, effectively authenticates the user using a multi-factor authentication system comprising a user ID, password, and biometric third factor. It stands as the sole trusted entity empowered to generate, backup, recover, and store all cryptographic keys. During the setup process, users are prompted to establish a secure passphrase in compliance with NIST standards [42].

6.1. Proposition-5: Keys must be kept confidential from adversaries

This proposition addresses requirement R2.1, which focuses on protecting the confidentiality of the generated cryptographic keys. This assertion is based on the premise that the personal devices (i.e., smartphones or laptops) running the software components of the proposed system are natively secure, as stated in the attacker model. It is supported by the following three claims, demonstrating that even if potential adversaries gain access to the mnemonic words used to generate the Root secret key, the SKM smart contract, or the files stored in IPFS, they are unable to retrieve the distinct computed secret keys.

Claim 9. *Adversaries cannot deduce the Root secret key solely from knowledge of the mnemonic words used during its generation.*

Proof. The Root secret key SK_0 is derived through the use of the BIP-32 protocol alongside a *BIP-39 seed* as an input parameter. This seed, in turn, is generated by employing the BIP-39 protocol, with input parameters consisting of a *User*-provided password *pw* and the mnemonic words. Consequently, the Root secret key SK_0 remains secure as long as the password *pw* is protected against adversaries.

Note that, with knowledge of the mnemonic words, a computationally capable attacker can initiate a brute-force attack to deduce *pw*. The success of such an attempt is directly correlated with the complexity of *pw*. Therefore, to mitigate these risks, *Users* must set strong *pw* according to the NIST standards [42]. □

Claim 10. *The confidentiality of the Plug-in's Master key pair during its exchange between the Smartphone App and the Plug-in is securely protected against any adversary.*

Proof. The exchange of information between the *Smartphone App* and the *Plug-in* occurs through the designated "temp" field of the *SKM SC*. All data transmitted within this field is encrypted using a secure symmetric cryptosystem and a unique session key K_i specific to *Plug-in i*. Consequently, during the exchange of the *Plug-in*'s Master key pair, it is encrypted using the corresponding K_i. This session key in use is encoded into a QR code, requiring physical access for reading it. By ensuring the concealment of the QR code, the session key remains protected, preventing any attempts by adversaries to eavesdrop it and, hence, decrypt the *Plug-in*'s Master key pair. □

Claim 11. *The key pairs (SK_{ij}, PK_{ij}) generated by Plug-in i to interact with distinct dApps j are publicly stored in IPFS. Nevertheless, their confidentiality against any potential adversary is securely protected.*

Proof. In the proposed system, all key pairs (SK_{ij}, PK_{ij}) are stored in a file published on IPFS, with its reference stored in the *SKM SC*'s "refsList" field, thus making it publicly accessible. In order to protect against unauthorized access by attackers who might obtain this file, the key pairs (SK_{ij}, PK_{ij}) are encrypted using a secure symmetric encryption scheme and a session key K_{ij} before being stored on the IPFS. Then, the session key is further encrypted using the Root public key PK_0, ensuring that only the holder of the corresponding Root secret key SK_0 possesses the ability to decrypt the aforementioned key pairs. □

6.2. Proposition-6: Keys must remain intact against tampering attempts by attackers

This proposition addresses requirement R2.2, which focuses on protecting the integrity of the *Plug-in*'s Master key pair (SK_i, PK_i) and Chain code C_i during transfer, as well as the end key pairs (SK_{ij}, PK_{ij}) stored in IPFS. This assertion is based on the premise that the personal devices (i.e., smartphones or laptops) running the software components of the proposed system are natively secure, as stated in the attacker model. It is supported by the following three claims.

Claim 12. *The proposed system ensures the integrity of the Plug-in's Master key pair (SK_i, PK_i) and Chain code C_i against any adversary during its transfer from the Smartphone App to the Plug-in.*

Proof. As mentioned in the "Setting-up the web browser Plug-in" protocol (see Section 4.1.2), the Master key pair (SK_i, PK_i) and the Master Chain code C_i are transmitted from the *Smartphone App* to each *Plug-in* via the "temp" field of the SKM smart contract.

To manipulate this field, an entity must execute a digitally signed transaction on the blockchain, invoking the *modTemp()* function of the SKM smart contract. Each function within the SKM smart contract is tied to a designated set of authorized keys, dictating which entities can execute them. In the case of the *modTemp()* function, when the *Smartphone App* creates and deploys this smart contract, it sets its Root Public key PK_0 as the only authorized party that can perform this operation.

As a result, only the *Smartphone App*, as the sole owner of the corresponding Root Secret key SK_0, retains the capability to use this function and alter the contents of the "temp" field. This strict control ensures that while the Root Secret key SK_0 remains secure, the integrity of the cryptographic keys stored in the "temp" field remains secure throughout their transit from the *Smartphone App* to each *Plug-in*. □

Claim 13. *The proposed system ensures the integrity of the reference to the key pairs (SK_{ij}, PK_{ij}) stored in IPFS, protecting it against unauthorized modifications.*

Proof. As mentioned in the "Session key pair generation" protocol (see Section 4.2), key pairs (SK_{ij}, PK_{ij}) are stored in the IPFS and the corresponding reference to their location is recorded in the "refsList" field of the SKM smart contract.

To manipulate this field, an entity must execute a digitally signed transaction on the blockchain, invoking the *storeRef()* function of the SKM smart contract. Only entities possessing the Master Secret keys SK_i linked to the Master Public keys PK_i, specified in the "whitelist" field of the SKM smart contract, are permitted to use the *storeRef()* function. Moreover, the smart contract only allows an entity that knows a certain Master Secret key SK_i to modify its particular position in the "refsList" field. In order to add or remove values from the "whileList" field, an entity must send a digitally signed transaction invoking the *addDevice()*/*removeDevice()* methods. These operations, in turn, require the entity to possess the Root Secret key SK_0, which is only know by the *Smartphone App*.

As a result, as long as the *Smartphone App*'s Root Secret key SK_0 and the *Plug-ins*' Master Secret keys SK_i are kept confidential, no adversary can tamper with the references to the IPFS. Also, if an adversary can disclose the Master Secret key of a certain *Plug-in*, she will then be able to modify only the IPFS reference of this specific *Plug-in*. □

Claim 14. *The proposed system ensures the integrity of the key pairs (SK_{ij}, PK_{ij}) stored in IPFS, protecting it against unauthorized modifications.*

Proof. Regarding the integrity of the files uploaded to the IPFS, it is noteworthy to mention the *immutability property* [22] inherent in IPFS technology. This property establishes that once a file is added to the IPFS network, its content remains unalterable without modifying the corresponding reference. Consequently, any attempt by an adversary to manipulate the key pairs (SK_{ij}, PK_{ij}) stored within IPFS would require altering their original references. □

[22] IPFS Immutability: `https://docs.ipfs.tech/concepts/immutability/`

6.3. Proposition-7: Keys must not be held by third parties

This proposition addresses requirement R2.3, which focuses on offering a fully decentralized multi-platform wallet that does not rely on external trust to provide its functionalities. The following claim supports this assertion.

Claim 15. *The proposed system does not depend on any trusted third party to store the generated cryptographic keys.*

Proof. The proposed system leverages the fully distributed technologies blockchain and IPFS to facilitate communication between its various components and securely store generated cryptographic keys. Specifically, the SKM smart contract deployed on the blockchain orchestrates communication between the *Smartphone App* and the *Plug-ins* while managing the keys, which are securely stored in the IPFS instead of relying on any third-party storage system. □

6.4. Proposition-8: Keys must remain unlinked between them and unassociated with individuals

A blockchain wallet is considered anonymous when it is impossible to connect addresses (key pairs) with their owners' identities. In this context, this proposal addresses requirement R2.4, aiming to ensure that a derived public key remains indistinguishable from another unless additional information is leaked.

The following claim supports this assertion, showing how the proposed system implements the BIP-32 protocol to generate unlinkable cryptographic keys.

Claim 16. *The proposed system generates key pairs that are unlikable between them.*

Proof. In the proposed system, the BIP-32 protocol is used to generate all the cryptographic keys. As explained in Section 2.2.2, this method contains a vulnerability wherein an attacker can deduce a parent private key SK_{par} given the parent public key PK_{par} and a child private key SK_i. This is possible because parent public keys are used in the key derivation function $CKD_{pub}((PK_{par}, c_{par}), i)$. In order to avoid this weakness, our implementation of the BIP-32 protocol exclusively employs the key derivation function $CKD_{priv}((SK_{par}, c_{par}), i)$ to generate‘ ‘hardened" child private keys using the parent private key as input. In this way, even in the event of an attacker knowing a child private key, the recovery of the parent private key SK_{par} becomes infeasible. □

7. Comparative analysis

This section compares the new proposal with both the well-established mainstream schemes introduced in Section 2.2 and the state-of-the-art solutions present in the literature and detailed in Section 1.1.

For this analysis, we have mapped each of the considered requirements to specific properties that a wallet may or may not fulfill. This correspondence is illustrated in Table 6. Note that we have divided requirement 1.2 into two properties: one addressing direct access to the user's generated keys, and the other focusing on the ability to use those keys across different platforms or devices.

Table 7 summarizes the comparison between the new proposed solution and existing mainstream schemes based on the aforementioned properties (taking into account their implementation specification and the study of [33]). The table indicates whether a wallet fully satisfies a property (✓), partially satisfies it (+/-), does not satisfy it (X), or we lack sufficient information to reach a conclusion (EMPTY SPACE).

As illustrated in Table 7, the new proposal particularly excels in terms of interoperability (immediate access to keys and/or cross-device portability), key unlinkability (i.e., privacy), key recovery, and resistance to theft. This superiority is anticipated, as these features were fundamental to the design of the new proposal, making the difference with respect to mainstream schemes.

Regarding the state-of-the-art schemes in the literature, Table 8 reveals that multi-signature wallets (i.e., [21, 22, 24, 25]) fall behind in terms of immediate Access to keys, as their primary focus is on safeguarding cryptographic keys against misuse rather than ensuring their availability. These schemes are assumed to offer some degree of key recovery through the distribution of keys or key shares among a group of participants. However, they do not account for scenarios where participants lose their shares or become malicious.

The *seed-derived wallet* presented in [26] employs biosensors for key generation. Consequently, the number of keys that can be generated is limited by the range of signals these biosensors can measure, and the system's interoperability is confined to its designed purpose. Additionally, significant changes in a user's physiological signals, such as

Requirement	Property
R1.1. Users should have the capability to generate keys from various devices utilizing the provided wallet.	Multiple Key Pairs Generation
R1.2. Key pairs should be accessible to users at all times, including the option to export them from the wallet for use in other applications if desired.	Immediate Access to Keys
	Cross-Device Portability
R1.3. Users should have the capability to recover all cryptographic keys in the event of loss of access to the wallet.	Key Recovery
R1.4. The solution must be economically viable for users.	Economically available
R2.1. Keys must be kept confidential from adversaries.	Resistant to Theft
R2.2. Keys should be safe against tampering by attackers.	Key Integrity and Authenticity
R2.3. Keys should not be kept by a third party, thus ensuring that the system does not rely on external trust.	No Trusted Third Parties
R2.4. Keys should remain unlinked to each other or to a specific user, ensuring that all generated keys are independent.	Key Unlikablility

Table 6: Proposal comparison with existing solutions

those caused by accidents or illnesses, may impede key recovery. Furthermore, there is a potential vulnerability: an adversary who intercepts the physiological signals used in seed generation could gain access to the user's keys

The proposals by Singh et al. [27] and Seo et al. [28] utilize the individuals' memory for key protection. Specifically, they employ answers to questions and specified picture locations to encrypt fragments of the private key for future recovery. While these schemes do not require an external entity during the recovery process, they do rely on a third party for storing the secured private key. Furthermore, an adversary might exploit social engineering techniques to gather user information, deduce the answers to security questions, or identify the images and their locations.

Rezaeighaleh et al.'s proposal [29] utilizes hardware wallets to safeguard cryptographic material, wherein users possess two hardware wallets with identical private keys, one acting as the primary wallet and the other as a backup. However, should users wish to generate multiple key pairs, this scheme imposes a significant overhead. Additionally, using these keys across different devices or services entails further inconvenience, requiring users to carry and connect the device each time they are used. Moreover, storing keys on these devices exposes them to the risk of physical theft. If both devices are lost or one is damaged unexpectedly, access to the cryptographic keys could be irretrievably lost.

In conclusion, this comparison highlights the superiority of the new wallet over existing literature, particularly in terms of key recovery, and it also demonstrates notable strengths in properties such as immediate Access to keys, cross-device portability and key unlinkability. On top of that, it is important to note that none of the previously mentioned solutions consider the economic costs they entail, a crucial factor in assessing the feasibility of these schemes.

8. Preliminary prototype implementation

To evaluate the viability of the new proposal, we developed a preliminary prototype of the wallet. This section begins with a discussion of the technical aspects of the implementation, followed by an overview of the user interface, and concludes with an analysis of the runtime performance of the various protocols.

8.1. Implementation details

The implemented prototype consists of three essential components: the *Smartphone App*, the *Secure Key Management Smart Contract*, and the web browser *Plug-in*. The source code for the *Smartphone App* and the *Secure Key Management Smart Contract* can be found at `https://github.com/imiguelrodriguez/TFGwallet`, while the

Wallet	*Multiple Key Pairs Generation*	*Immediate Access to Keys*	*Cross-Device Portability*	*Key Recovery*	*Economically Available*	*Resistant to Theft*	*Key Integrity and Authenticity*	*No Trusted Third Parties*	*Key Unlikablility*
Our Proposal	✓	✓	✓	✓	+/-	✓	✓	✓	✓
Keys in Local Storage (e.g., Bitcoin Core[23], Android Bitcoin Wallet[24])	✓	✓	X	X	✓	X		✓	
Password-protected (Encrypted) Wallets (e.g., MultiBit[25])	✓	✓	X	X	✓	X		✓	
Offline Storage of Keys (e.g., Bitaddress[26], Bitcoin Paper Wallet[27])	✓	X	✓	X	✓	X		✓	
Air-gapped Key Storage (e.g., Bitcoin Armory[28], Trezor[29]).	✓	X	X	X	+/-	X		✓	
Password-derived Keys (e.g., Brainwallet[30])	✓	✓	✓	✓				✓	
Hosted Wallet (e.g., Coinbase[31], Binance [32])		+/-	+/-	✓	+/-	+/-	✓	X	

Table 7: Proposal comparison with existing mainstream schemes

code for the *Plug-in* is accessible at `https://github.com/imiguelrodriguez/TFGwalletPlugIn`. Below, we provide an overview of the most significant aspects of their development:

- The *Smartphone App* was designed to run on an Android device, while the *Plug-in* was developed to operate on the Google Chrome web browser.
- The *Smartphone App* was developed using the Android Studio IDE, with the code written in Kotlin and structured according to the Model-View-Controller (MVC) architecture.
- The *Plug-in* was developed using web-based technologies, including HTML, CSS, and JavaScript.
- Protocols BIP32 and BIP39, discussed in Section 2, were implemented with an object-oriented approach to

Wallet	*Multiple Key Pairs Generation*	*Immediate Access to Keys*	*Cross-Device Portability*	*Key Recovery*	*Economically Available*	*Resistant to Theft*	*Key Integrity and Authenticity*	*No Trusted Third Parties*	*Key Unlikability*
Our Proposal	✓	✓	✓	✓	+/-	✓	✓	✓	✓
[21]		X		+/-		✓	✓	X	
[22]		X		+/-		✓	✓		
[24]		X		+/-		✓	✓	✓	✓
[25]		X		+/-		✓	✓	✓	
[26]	+/-	✓	X	+/-		+/-	✓	✓	✓
[27]	✓	✓	✓	+/-		+/-	✓	X	
[28]	✓	✓	+/-	+/-		+/-	✓	X	
[29]	+/-	+/-	✓	+/-		+/-	✓	✓	

Table 8: Proposal comparison with state-of-the-art schemes present in the literature

ensure modularity in both Kotlin and JavaScript, facilitating their integration into the *Smartphone App* and *Plug-in*, respectively.

- To deploy and interact with the *Secure Key Management Smart Contract* from both the *Smartphone App* and the *Plug-in*, the web3j[33] and web3js[34] libraries were utilized, respectively. Additionally, to streamline interaction with smart contracts via the web3j library, a Java wrapper was generated using the Truffle suite[35].
- The *Secure Key Management Smart Contract* that manages the wallet was implemented in Solidity[36], an object-oriented programming language specifically designed for the Ethereum Virtual Machine (EVM). Next, it was deployed on the Sepolia Test Network[37], a blockchain environment tailored for testing, enabling the evaluation of blockchain applications in a realistic setting.
- To connect to the IPFS network in our test environment, we downloaded and executed the IPFS daemon from the official website[38]. Running the daemon integrates the designated computer into the IPFS network, facilitating the publication and retrieval of files within this decentralized system. For deploying the prototype in a real-world setting, utilizing a public IPFS gateway[39] would be a more practical alternative.
- The ECDSA cryptosystem, with a 256-bit key size, was utilized for generating signatures. For encryption, the Elliptic Curve Integrated Encryption Scheme (ECIES) [43] was employed, using AES in GCM mode.
- To perform cryptographic operations within the digital wallets, the EVM key specifications were followed. Specifically, ECDSA was employed with a 256-bit private key and a point on the secp256k1 ECDSA curve,

[33] web3j: `https://docs.web3j.io/`
[34] web3js: `https://docs.web3js.org/`
[35] Truffle suite: `https://archive.trufflesuite.com/`
[36] Solidity Programming Language: `https://soliditylang.org/`
[37] Ethereum Sepolia Faucet: `https://sepoliafaucet.com/`
[38] InterPlanetary File System (IPFS): `https://ipfs.tech/`
[39] IPFS gateway: `https://docs.ipfs.tech/concepts/ipfs-gateway/`

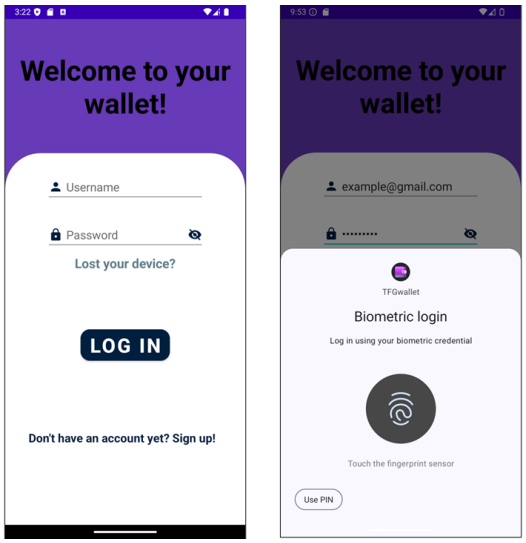


Figure 12: Login

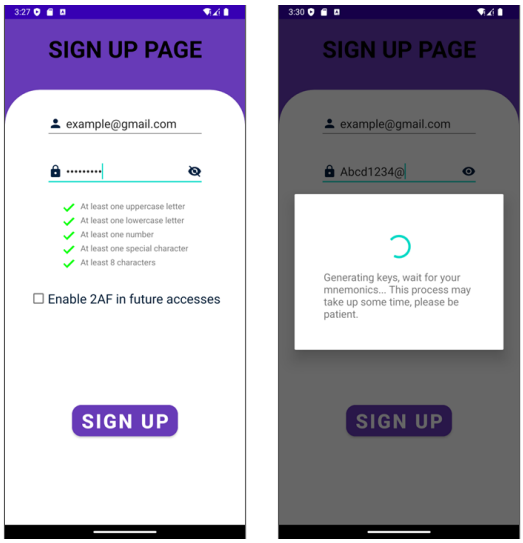


Figure 13: Sign-up

represented as an (x, y) point, for the corresponding public key. The public Ethereum addresses were generated by taking the lower 160 bits of the Keccak-256 (also known as SHA-3) digest of the public key.

8.2. *Interface overview*

Upon launching the *Smartphone App*, the *User* is presented with a login screen, which can be configured to utilize *Biometric Two-Factor Authentication*, as illustrated in Figure 12.

If the *User* does not have an account, one must be created by setting up the *Smartphone App*. This involves registering with a valid email address and a secure password. Upon successful registration, a new local account is established, keys are generated, and the blockchain account is funded. The *User* must wait for this process to be completed, as shown in Figure 13.

Following key generation, the *Smartphone App* presents the *User* with her 24 mnemonic words, which are crucial for key recovery. Once this step is completed, the smart contract is deployed, and its address (i.e., the smart contract's hash) is displayed. Figure 14 depicts this process.

After successfully logging into the *Smartphone App*, the *User* can add new *Plug-ins* by scanning a QR code. By clicking the scan button on the designated screen, the device's camera will activate, enabling the user to capture the QR code. Once scanned, a loading message will appear as the *Smartphone App* communicates with the blockchain. Figure 15 illustrates this procedure.

Figure 16 depicts the screen where the *User* can view the *Smartphone App*'s master key and the list of added *Plug-ins* along with their corresponding cryptographic keys. Initially, the *Smartphone App* conceals the master key details, which the *User* can reveal by tapping on the corresponding item.

As it can be seen in Figure 17, the same approach applies to the list of registered *Plug-ins*. Initially, only the names of the devices where the *Plug-ins* are installed are displayed. By tapping on a specific *Plug-in*, the app reveals its cryptographic keys. Additionally, by swiping left, the *User* can access a new screen displaying all the accessed *dApps* associated with that *Plug-in*, as shown. This procedure is shown in Figure 18.

Finally, Figure 19 illustrates the key recovery process. The *Smartphone App* first displays a recovery screen, resembling the sign-up interface, where the system verifies that the provided email is valid, and that the password meets the required criteria. Once both fields are validated, the app transitions to a screen where the mnemonic words can be entered. Upon tapping the recovery button, the user is prompted to enter the number of previously registered devices. The user must then scan the corresponding number of QR codes. After scanning, the recovery process is initiated, allowing the user to sign in as usual.

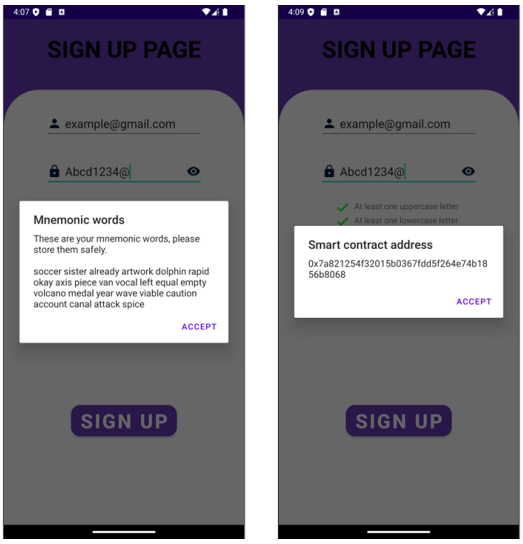


Figure 14: Mnemonic words and smart contract hash

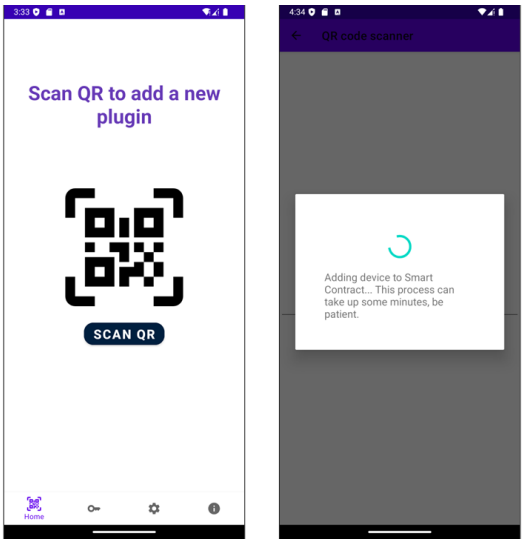


Figure 15: Adding a Plug-in

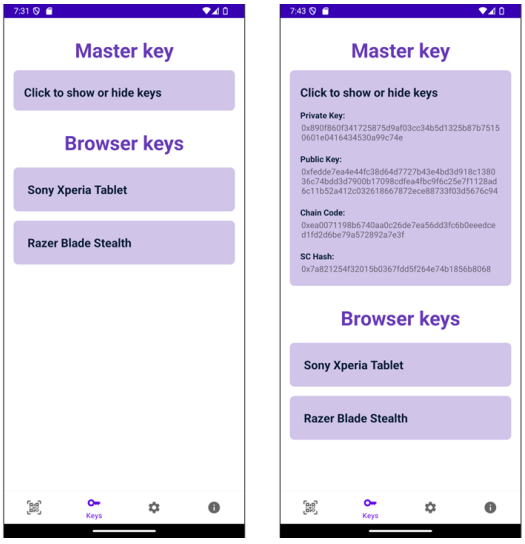


Figure 16: Smartphone App's cryptographic keys

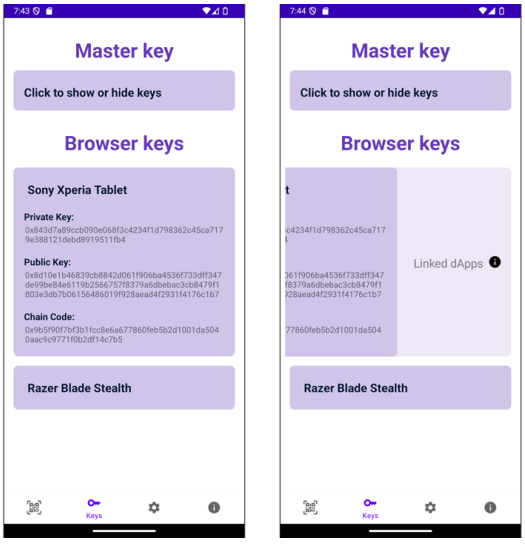


Figure 17: Plug-ins' cryptographic keys

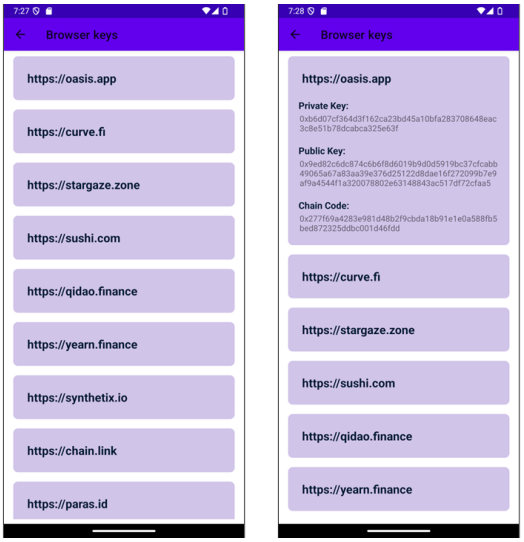


Figure 18: dApps' cryptographic keys

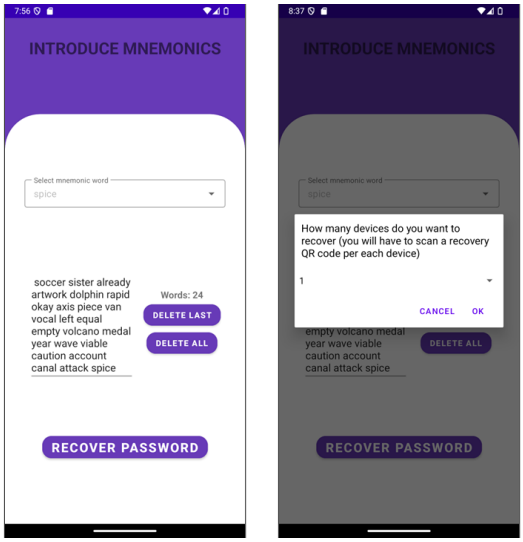


Figure 19: Key recovery

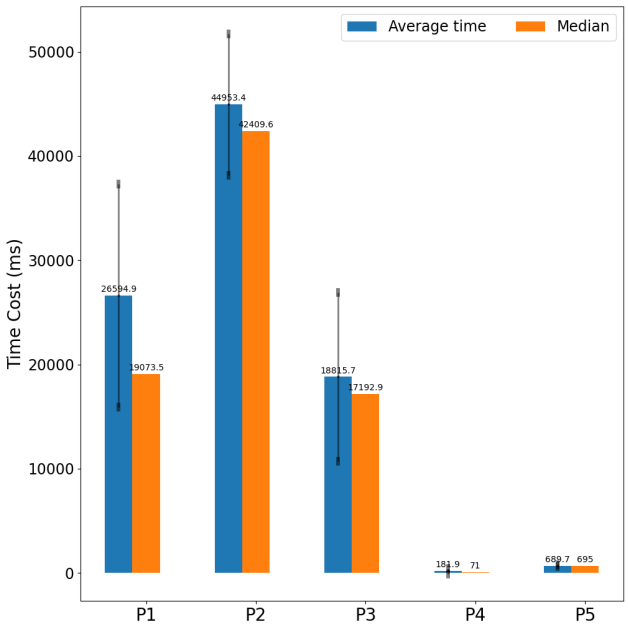


Figure 20: Cost in milliseconds of the designed protocols

Protocol	Average time (ms)	Median (ms)	Deviation (ms)
P1 - Setting-up the *Smartphone App*	26594.90	19073.50	10706.98
P2 - Setting-up the web browser *Plug-in*	44953.40	42409.57	6787.60
P3 - Session key pair generation	18815.67	17192.9	8117.89
P4 - Key inventory	181.90	71	271.71
P5 - Key recovery at the *Smartphone App*	689.70	695	95.28

Table 9: Cost in milliseconds of the designed protocols

8.3. Runtime performance analysis

Leveraging the implemented prototype, this section examines the execution time of each designed protocol, measured in milliseconds, to assess the performance of the new wallet and its practical feasibility in a realistic environment. In the case of the "Key recovery" procedure, we tested the case in which the *Smartphone App* is no longer accessible. Each protocol was executed 10 times for this analysis.

As outlined in Section 8.1, these tests were conducted using the Sepolia Test Network due to its close resemblance to a realistic environment. This choice is significant for two reasons. First, all interactions with the blockchain require Sepolia ETH tokens, which are limited and earned through network mining, thereby constraining the number of tests that can be performed. Second, the status and current traffic of the Sepolia Test Network can fluctuate, directly influencing the execution times of operations involving blockchain interactions.

Table 9 presents a summary of the execution times, measured in milliseconds, for the evaluated protocols. Figure 20 provides a graphical representation of these values for clearer assessment. Key findings from this analysis include the following points:

- Setting up the *Smartphone App* under standard blockchain load takes approximately 19 seconds, while configuring a *Plug-in* requires about 42 seconds. Since these operations are infrequent, the times involved seem to be manageable.

- The "Session key pair generation" protocol is crucial, as it is executed when a user first interacts with a new *dApp* while browsing the Internet. Under typical blockchain conditions, this process takes approximately 17 seconds, which is a significant cost in this context. However, it is important to emphasize that this duration represents the total time for the entire protocol. Notably, the most time-consuming part of this execution is the Step 8d, an automated process transparent to the user, in which the *Plug-in* utilizes the *storeRef()* method of its associated *SKM SC* to update the "refsList" field with the new reference. During this step, the user is already engaged with the *dApp*, so it does not impact the user experience. Consequently, excluding this step, the user-perceived waiting time for this protocol is approximately 270 milliseconds. It is also worth noting that this operation only occurs the first time the *Plug-in* interacts with a particular *dApp*. In summary, we consider this cost acceptable for such a critical procedure.

- Interactions with the blockchain are by far the most time-consuming, accounting for 95% to 99% of the total execution time in the first four protocols. In the "Key recovery" protocol, these interactions represent approximately 75% of the total time. This indicates that using a faster blockchain could significantly reduce the times reported in this study.

- The "Key inventory" and "Key recovery" protocols require negligible time overall.

- The time listed in Table 9 for the "Key recovery" protocol refers exclusively to the steps executed by the *Smartphone App*. An additional step is carried out by each involved *Plug-in*, which must download a file containing keys from the IPFS. The size of this file is directly proportional to the number of dApps linked to the *Plug-in*, with each *dApp* corresponding to one key pair. In the tests conducted, downloading a file with 10 key pairs from the IPFS took an average of 245.4 milliseconds, with a deviation of 25.73 milliseconds. For a file with 20 key pairs, the download time averaged 361.8 milliseconds, with a deviation of 64.32 milliseconds. All those reported times seem to be affordable.

9. Concluding remarks

In recent times, there has been growing attention on blockchain-powered dApps serving as Serverless REST APIs, offering an inherent solution to reducing dependence on conventional centralized architectures. However, to engage with dApps, users necessitate a blockchain wallet. This tool facilitates the generation and secure storage of a user's private key, verifies users' identity by confirming possession of their private cryptographic material, and affords users access to their individual digital assets.

Despite their critical role, blockchain wallets also present notable challenges: i) reliance on trusted third parties to safeguard users' cryptographic keys and digital assets; ii) vulnerability to adversaries who may observe and potentially link user interactions, thereby compromising sensitive information; iii) the complexity of integrating key recovery mechanisms within a fully decentralized framework; and iv) the need to ensure the secure synchronization of cryptographic key pairs across multiple devices.

To overcome these challenges, this paper introduces a fully decentralized multi-platform wallet that leverages blockchain and IPFS technologies for managing asymmetric keys and enabling key recovery. This novel approach empowers users to interact with dApps built on smart contracts deployed on the blockchain, while preserving their privacy from potential attackers aiming to access sensitive information. Furthermore, our proposed system ensures seamless key recovery in case of device theft or damage, and it does not rely on any trusted third party.

The new proposal has been designed considering that the solution must be economically viable for the users. In this way, the smart contract that runs the core methods of the new scheme has been implemented and the costs associated to its use have been studied in depth. In terms of its resilience against security and privacy attacks, a detailed analysis reveals that: i) the scheme remains robust against adversaries attempting unauthorized access to cryptographic keys; ii) cryptographic keys generated remain secure against tampering; and iii) there is no correlation between generated cryptographic keys, preventing them from being linked to individual users.

The paper also provides a comparative analysis between the proposed system, established mainstream schemes, and other state-of-the-art solutions in the literature, highlighting the distinct advantages of this new contribution.

Finally, to assess the overall feasibility of the new wallet, a preliminary prototype has been developed. This includes a detailed discussion of the technical implementation, an overview of the user interface, and an analysis of the runtime performance of the designed protocols.

Acknowledgments and disclaimer

This research is supported by Project HERMES funded by the European Union NextGenerationEU/PRTR via INCIBE, by the project PID2021-125962OB-C32 "SECURING/DATA" funded by MCIN/AEI/10.13039/501100011033/FEDER, UE, and by the grant 2021SGR 00115 from the Government of Catalonia. The first author is also supported by the Spanish Government under an FPU grant (ref. FPU20/03254).

References

[1] K. Wu, Y. Ma, G. Huang, X. Liu, A first look at blockchain-based decentralized applications, Software: Practice and Experience 51 (10) (2021) 2033–2050. doi:10.1002/spe.2751.

[2] P. Zheng, Z. Jiang, J. Wu, Z. Zheng, Blockchain-based decentralized application: A survey, IEEE Open Journal of the Computer Society 4 (2023) 121–133. doi:10.1109/OJCS.2023.3251854.

[3] M. Shuaib, N. H. Hassan, S. Usman, S. Alam, S. Bhatia, A. Mashat, A. Kumar, M. Kumar, Self-sovereign identity solution for blockchain-based land registry system: A comparison, Mobile Information Systems 2022 (1) (2022) 8930472. doi:10.1155/2022/8930472.

[4] M. Shuaib, S. Alam, M. S. Alam, M. S. Nasir, Self-sovereign identity for healthcare using blockchain, Materials Today: Proceedings 81 (2023) 203–207. doi:10.1016/j.matpr.2021.03.083.

[5] N. B. Truong, K. Sun, G. M. Lee, Y. Guo, Gdpr-compliant personal data management: A blockchain-based solution, IEEE Transactions on Information Forensics and Security 15 (2020) 1746–1761. doi:10.1109/TIFS.2019.2948287.

[6] C. Daudén-Esmel, J. Castellà-Roca, A. Viejo, Blockchain-based access control system for efficient and gdpr-compliant personal data management, Computer Communications 214 (2024) 67–87. doi:https://doi.org/10.1016/j.comcom.2023.11.017.

[7] N. Duong-Trung, H. X. Son, H. T. Le, T. T. Phan, On components of a patient-centered healthcare system using smart contract, in: Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy, ICCSP 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 31–35. doi:10.1145/3377644.3377668.

[8] N. Duong-Trung, H. X. Son, H. T. Le, T. T. Phan, Smart care: Integrating blockchain technology into the design of patient-centered healthcare systems, in: Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy, ICCSP 2020, Association for Computing Machinery, New York, NY, USA, 2020, p. 105–109. doi:10.1145/3377644.3377667.

[9] C. Anglés-Tafalla, A. Viejo, J. Castellà-Roca, M. Mut-Puigserver, M. M. Payeras-Capellà, Security and privacy in a blockchain-powered access control system for low emission zones, IEEE Transactions on Intelligent Transportation Systems 24 (1) (2023) 580–595. doi:10.1109/TITS.2022.3211659.

[10] D. Das, K. Dasgupta, U. Biswas, A secure blockchain-enabled vehicle identity management framework for intelligent transportation systems, Computers and Electrical Engineering 105 (2023) 108535. doi:10.1016/j.compeleceng.2022.108535.

[11] D. Das, S. Banerjee, P. Chatterjee, U. Ghosh, W. Mansoor, U. Biswas, Design of a trust-based authentication scheme for blockchain-enabled iov system, in: 2023 Advances in Science and Engineering Technology International Conferences (ASET), 2023, pp. 1–6. doi:10.1109/ASET56582.2023.10180814.

[12] J. Camenisch, A. Lysyanskaya, G. Neven, Practical yet universally composable two-server password-authenticated secret sharing, in: Proceedings of the 2012 ACM Conference on Computer and Communications Security, CCS '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 525–536. doi:10.1145/2382196.2382252.

[13] N. Lehto, K. Halunen, O.-M. Latvala, A. Karinsalo, J. Salonen, Cryptovault - a secure hardware wallet for decentralized key management, in: 2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS), 2021, pp. 1–4. doi:10.1109/COINS51742.2021.9524133.

[14] Q. Wei, S. Li, W. Li, H. Li, M. Wang, Decentralized hierarchical authorized payment with online wallet for blockchain, in: Wireless Algorithms, Systems, and Applications: 14th International Conference, WASA 2019, Honolulu, HI, USA, June 24–26, 2019, Proceedings 14, Springer, 2019, pp. 358–369. doi:10.1007/978-3-030-23597-0_29.

[15] S. He, Q. Wu, X. Luo, Z. Liang, D. Li, H. Feng, H. Zheng, Y. Li, A social-network-based cryptocurrency wallet-management scheme, IEEE Access 6 (2018) 7654–7663. doi:10.1109/ACCESS.2018.2799385.

[16] R. Soltani, U. T. Nguyen, A. An, Practical key recovery model for self-sovereign identity based digital wallets, in: 2019 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), IEEE, 2019, pp. 320–325. doi:10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00066.

[17] X. He, J. Lin, K. Li, X. Chen, A novel cryptocurrency wallet management scheme based on decentralized multi-constrained derangement, IEEE Access 7 (2019) 185250–185263. doi:10.1109/ACCESS.2019.2961183.

[18] Y. Zhang, M. Wang, Y. Guo, F. Guo, Towards dynamic and reliable private key management for hierarchical access structure in decentralized storage, in: Proceedings of the 32nd ACM International Conference on Information and Knowledge Management, 2023, pp. 3371–3380. doi:doi.org/10.1145/3583780.361509.

[19] M. Aydar, S. C. Cetin, S. Ayvaz, B. Aygun, Private key encryption and recovery in blockchain, arXiv preprint arXiv:1907.04156 (2019). doi:10.48550/arXiv.1907.04156.

[20] G. Li, L. You, A consortium blockchain wallet scheme based on dual-threshold key sharing, Symmetry 13 (8) (2021) 1444. doi:10.3390/sym13081444.

[21] F. Zhu, W. Chen, Y. Wang, P. Lin, T. Li, X. Cao, L. Yuan, Trust your wallet: A new online wallet architecture for bitcoin, in: 2017 International Conference on Progress in Informatics and Computing (PIC), IEEE, 2017, pp. 307–311. doi:10.1109/PIC.2017.8359562.

[22] R. Gennaro, S. Goldfeder, A. Narayanan, Threshold-optimal dsa/ecdsa signatures and an application to bitcoin wallet security, in: Applied Cryptography and Network Security: 14th International Conference, ACNS 2016, Guildford, UK, June 19-22, 2016. Proceedings 14, Springer, 2016, pp. 156–174. doi:10.1007/978-3-319-39555-5_9.

[23] D. Boneh, R. Gennaro, S. Goldfeder, Using level-1 homomorphic encryption to improve threshold dsa signatures for bitcoin wallet security, in: Progress in Cryptology–LATINCRYPT 2017: 5th International Conference on Cryptology and Information Security in Latin America, Havana, Cuba, September 20–22, 2017, Revised Selected Papers, Springer, 2019, pp. 352–377. doi:10.1007/978-3-030-25283-0_19.

[24] Z. Jian, Q. Ran, S. Liyan, Securing blockchain wallets efficiently based on threshold ecdsa scheme without trusted center, in: 2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS), IEEE, 2021, pp. 47–51. doi:10.1109/ACCTCS52002.2021.0001.

[25] P. Dikshit, K. Singh, Efficient weighted threshold ecdsa for securing bitcoin wallet, in: 2017 ISEA Asia Security and Privacy (ISEASP), IEEE, 2017, pp. 1–9. doi:10.1109/ISEASP.2017.7976994.

[26] H. Zhao, Y. Zhang, Y. Peng, R. Xu, Lightweight backup and efficient recovery scheme for health blockchain keys, in: 2017 IEEE 13th international symposium on autonomous decentralized system (ISADS), IEEE, 2017, pp. 229–234. doi:10.1109/ISADS.2017.22.

[27] H. P. Singh, K. Stefanidis, F. Kirstein, A private key recovery scheme using partial knowledge, in: 2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS), IEEE, 2021, pp. 1–5. doi:10.1109/NTMS49979.2021.9432642.

[28] J. Seo, D. Ko, S. Kim, V. Sugumaran, S. Park, Reminisce: Blockchain private key generation and recovery using distinctive pictures-based personal memory, Mathematics 10 (12) (2022) 2047. doi:10.3390/math10122047.

[29] H. Rezaeighaleh, C. C. Zou, New secure approach to backup cryptocurrency wallets, in: 2019 IEEE Global Communications Conference (GLOBECOM), IEEE, 2019, pp. 1–6. doi:10.1109/GLOBECOM38437.2019.9014007.

[30] D. Johnson, A. Menezes, S. Vanstone, The elliptic curve digital signature algorithm (ecdsa), International Journal of Information Security 1 (2014) 36–63. doi:10.1007/s102070100002.

[31] S. Suratkar, M. Shirole, S. Bhirud, Cryptocurrency wallet: A review, in: 2020 4th international conference on computer, communication and signal processing (ICCCSP), 2020, pp. 1–7. doi:10.1109/ICCCSP49186.2020.9315193.

[32] O. Pal, B. Alam, V. Thakur, S. Singh, Key management for blockchain technology, ICT Express 7 (1) (2021) 76–80. doi:10.1016/j.icte.2019.08.002.

[33] S. Eskandari, D. Barrera, E. Stobert, J. Clark, A first look at the usability of bitcoin key management, in: Proceedings 2015 Workshop on Usable Security, USEC 2015, Internet Society, 2015. doi:10.14722/usec.2015.23015.

[34] P. Wuille, Bip32: Hierarchical deterministic wallets, `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki` (2012).

[35] M. Palatinus, P. Rusnak, A. Voisine, S. Bowe, Bip39: Mnemonic code for generating deterministic keys, `https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki` (2013).

[36] D. R. Brown, Sec 2: Recommended elliptic curve domain parameters, Standars for Efficient Cryptography (2010).

URL https://cir.nii.ac.jp/crid/1572824501046106880

[37] M. Nystrom, Identifiers and test vectors for hmac-sha-224, hmac-sha-256, hmac-sha-384, and hmac-sha-512, RFC 4231 (Dec. 2005). doi:10.17487/RFC4231.

[38] J. Benet, Ipfs - content addressed, versioned, p2p file system, arXiv preprint arXiv:1407.3561 (2014). doi:10.48550/ARXIV.1407.3561.

[39] R. Kumar, R. Tripathi, Implementation of distributed file storage and access framework using ipfs and blockchain, in: 2019 Fifth International Conference on Image Information Processing (ICIIP), 2019, pp. 246–251. doi:10.1109/ICIIP47207.2019.8985677.

[40] L.-Y. Yeh, C.-Y. Shen, W.-C. Huang, W.-H. Hsu, H.-C. Wu, Gdpr-aware revocable p2p file-sharing system over consortium blockchain, IEEE Systems Journal 16 (4) (2022) 5234–5245. doi:10.1109/JSYST.2021.3139319.

[41] M. J. Dworkin, E. B. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, J. F. Dray Jr, Advanced encryption standard (AES), Federal Inf. Process. Stds. (NIST FIPS) 197 (2001).

[42] P. A. Grassi, J. L. Fenton, E. M. Newton, R. Perlner, A. Regenscheid, W. E. Burr, J. P. Richer, N. Lefkovitz, J. M. Danker, Y.-Y. Choong, et al., Digital identity guidelines: Authentication and lifecycle management [includes updates as of 03-02-2020], National Institute of Standards and Technology Special Publication 800-63B (2020). doi:10.6028/NIST.SP.800-63b.

[43] V. Gayoso Martínez, L. Hernández Encinas, A. Queiruga Dios, Security and practical considerations when implementing the elliptic curve integrated encryption scheme, Cryptologia 39 (3) (2015) 244–269. doi:10.1080/01611194.2014.988363.