Ignacio Miguel Rodríguez

Optimizing Security and Interoperability in Decentralized Wallets: A Multi-Platform Solution Using Blockchain and IPFS

Degree in Computer Science

Final Degree Project

Co-Directed by
Dr. Jordi Castellà Roca and Cristòfol Dauden Esmel

School of Engineering

UNIVERSITAT ROVIRA i VIRGILI

Tarragona, September 2024

Contents

Resum v

Resumen vi

Abstract vii

1 Introduction 1
1.1 Proposal . . . 2

2 State of art 3

3 Technologies 4
3.1 Blockchain . . . 4
3.1.1 Ganache . . . 6
3.1.2 Sepolia . . . 7
3.1.3 Etherscan . . . 8
3.2 Web3 library . . . 8
3.3 Bitcoin Improvement Proposal (BIP) . . . 9
3.3.1 BIP39 . . . 9
3.3.2 BIP32 . . . 10
3.4 Android Studio . . . 12
3.5 Plugin . . . 12
3.6 IPFS . . . 13

4 Requirements 15
4.1 Functional requirements . . . 15
4.2 Non-functional requirements . . . 15
4.2.1 Security and privacy requirements . . . 15
4.2.2 Design and implementation requirements . . . 15
4.2.3 Usability requirements . . . 15
4.2.4 Performance requirements . . . 15

5 Design and architecture 16
5.1 General structure . . . 16
5.2 Secure Key Management Smart Contract - SKM SC . . . 17
5.3 Use cases . . . 20
5.3.1 Smartphone App . . . 21
5.3.2 Browser Plugin . . . 28

6 Implementation 34
6.1 Android App . . . 34
6.1.1 QR scanner . . . 35
6.1.2 Android Secure Storage . . . 36
6.1.3 Network operations . . . 36
6.1.4 Shared preferences . . . 36
6.1.5 Showing keys . . . 37
6.1.6 Biometric access . . . 37

6.2 Cryptography . . . 37
6.2.1 Symmetric keys . . . 38
6.2.2 Asymmetric keys . . . 38
6.2.3 Encryption and decryption processes . . . 38
6.3 Blockchain . . . 40
6.4 Smart Contract . . . 42
6.5 Chrome Plugin . . . 43
6.5.1 Minified packages . . . 43
6.5.2 QR generator . . . 43
6.5.3 Background script . . . 43
6.5.4 BIP32 . . . 44
6.5.5 Utility functions . . . 44
6.6 IPFS . . . 44
6.6.1 Setting up IPFS . . . 44

7 Evaluation 46
7.1 App setup . . . 46
7.1.1 App installation . . . 46
7.1.2 App navigation . . . 46
7.1.3 User registration . . . 46
7.1.4 Signing In and Home Page . . . 48
7.2 Plugin setup . . . 49
7.2.1 Plugin installation . . . 49
7.2.2 Plugin configuration . . . 49
7.2.3 Adding devices . . . 49
7.3 Adding a new dApp . . . 51
7.4 Key management . . . 52
7.5 Key recovery . . . 54
7.6 Performance . . . 56
7.6.1 Environment and methodology . . . 57
7.6.2 Setting up the smartphone app . . . 58
7.6.3 Setting up the web browser plugin . . . 58
7.6.4 Adding a new dApp . . . 59
7.6.5 Key visualization . . . 60
7.6.6 Key recovery . . . 60

8 Conclusions 63
8.1 Future work . . . 63

References 65

Appendix A SKM SC in Solidity 69

List of Figures

1 Example of the PoW Sepolia Faucet. 6
2 Example of the Ganache GUI homepage. 7
3 Web3 libraries' interface architecture. 8
4 Plugin anatomy. 13
5 Client-server centralized model vs. IPFS decentralized model. 14
6 Architecture of the proposal. 16
7 App and plugin use cases. 21
8 Use case 0. SignUp. 22
9 Use case 1. Login. 23
10 Use case 2. VisualizeKeys. 24
11 Use case 3. VisualizeDappKeys. 25
12 Use case 4. ShowKeys. 25
13 Use case 5. HideKeys. 26
14 Use case 6. RecoverKeys. 26
15 Use case 7. Change2FAState. 28
16 Use case 8. AskBiometric. 29
17 Use case 9. SetUpPlugin. 30
18 Use case 10. RecoverQR. 30
19 Use case 11. NewDapp. 32
20 RSA encryption/decryption for master and browser keys. 39
21 DApp key generation and retrieval. Interaction between the app and the plugin via IPFS. 41
22 IPFS daemon configuration file. 45
23 IPFS daemon running. 45
24 App icon on an Android device. 46
25 App navigation. 47
26 Error handling on app sign up page. 47
27 Password fulfills the requirements for sign up. 48
28 Key generation during app setup. 48
29 Mnemonic list during app setup. 49
30 Newly-created SC address during app setup. 49
31 Error handling on app sign in page. 50
32 Biometric access. 50
33 Main page navigation. 51
34 Plugin icon in Google Chrome. 51
35 Plugin home page when not yet configured. 51
36 Plugin set up page when not yet configured. 52
37 Plugin set up. 52
38 QR code generation. 53
39 Invalid smart contract address. 53
40 QR scanning process and error handling. 54
41 Adding a new dApp. 54
42 Loading message when adding a new dApp. 54
43 Successful transaction when adding new dApp. 55
44 Master and Browser keys. 55
45 dApp keys. 56
46 Mnemonic words input page. 56

47 Dialog to select the number of devices to recover. 57
48 Execution time as the number of dApps increases. The dotted line indicates the fitted linear regression. 62

List of Tables

1 Size relation between ENT, CS and MS in BIP39. 9
2 "Setting up the Smartphone App" protocol step. 23
3 "Key Visualization" protocol step. 27
4 "Key Recovery" protocol step. 27
5 "Setting up the Browser Plugin" protocol step. 31
6 "Adding a new dApp" protocol step. 33
7 Performance metrics. 57
8 Setting up the Smartphone App performance. 58
9 Setting up the browser plugin performance. 59
10 Performance data for adding a new dApp. 59
11 Performance data for key visualization. 60
12 Performance data for key recovery when the number of devices is 1. . . 60
13 Performance data for key recovery when the number of devices is 2. . . 61
14 Performance data for key recovery when the number of devices is 5. . . 61
15 Summary statistics for key recovery. 61

List of Code Snippets

1 Contract example written in Solidity. 6
2 Traditional approach using resource IDs. 35
3 Using Binding to access GUI elements. 35
4 Generating and storing the RSA key in the Android KeyStore. 36
5 Web3 library initialization in JavaScript. 40
6 Web3 library initialization in Kotlin. 40
7 SKM SC contract in Solidity. 70

Resum

Recentment, l'ús d'aplicacions descentralitzades basades en *blockchain* (dApps) ha augmentat significativament. Per tal de gestionar les claus necessàries per connectar-se a aquestes dApps, els usuaris necessiten una cartera virtual. La gestió i sincronització segura de claus a través de diversos dispositius presenta un repte considerable. A més, la pèrdua o substitució de dispositius pot suposar un problema greu, ja que l'usuari perdria l'accés a la seva cartera virtual i als seus fons. Per abordar aquests problemes, proposem una solució que inclou una aplicació mòbil i un plugin per al navegador. L'aplicació mòbil constitueix l'element central que permet emmagatzemar i visualitzar les claus generades en qualsevol moment. Per altra banda, es preveu disposar d'un plugin instal·lable a les principals plataformes. En aquest treball s'ha desenvolupat per Google Chrome. Aquest plugin permet gestionar l'accés a les dApps i la creació de les claus associades de manera transparent. El sistema utilitza un contracte intel·ligent desplegat a la *blockchain* per facilitar la comunicació entre l'aplicació i els plugins, així com l'IPFS per emmagatzemar les claus de manera segura, facilitant-ne la recuperació en cas de pèrdua o canvi de dispositiu mòbil. El sistema implementat ha estat avaluat pel que fa a la seva usabilitat i rendiment. Proporciona un entorn segur i intuïtiu, amb temps d'execució ràpids en operacions crítiques com són l'afegiment de dApps o la recuperació de claus.

Paraules clau— Recuperació de Claus, Moneder Criptogràfic, Aplicacions Descentralitzades, Contracte Intel·ligent, Seguretat, Privadesa

Resumen

Recientemente, el uso de aplicaciones descentralizadas basadas en *blockchain* (dApps) ha aumentado significativamente. Con el fin de gestionar las claves necesarias para conectarse a estas dApps, los usuarios necesitan una cartera virtual. La gestión y sincronización segura de claves a través de diversos dispositivos supone un reto considerable. Además, la pérdida o sustitución de dispositivos puede suponer un problema grave, ya que el usuario perdería el acceso a su cartera virtual y a sus fondos. Para abordar estos problemas, se propone una solución que incluye una aplicación móvil y un plugin para el navegador. La aplicación móvil constituye el elemento central y permite almacenar y visualizar las claves generadas en cualquier momento. Por otro lado, se prevee disponer de un plugin instalable en las principales plataformas. En este trabajo se ha desarrollado para Google Chrome. El plugin permite gestionar el acceso a las dApps y la creación de las claves asociadas de manera transparente. El sistema utiliza un contrato inteligente desplegado en la *blockchain* para comunicar la aplicación y los plugins, además de IPFS para almacenar las claves de manera segura, facilitando su recuperación en caso de pérdida o cambio del dispositivo móvil. El sistema implementado ha sido evaluado en términos de usabilidad y rendimiento. Proporciona un entorno seguro e intuitivo, con tiempos de ejecución rápidos en operaciones críticas como son la inclusión de nuevas dApps o la recuperación de claves.

Palabras clave— Recuperación de Claves, Cartera Criptográfica, Aplicaciones Descentralizadas, Contratos Inteligentes, Seguridad, Privacidad

Abstract

Recently, the use of decentralized applications (dApps) based on blockchain has significantly increased. To manage the keys required to connect to these dApps, users need a virtual wallet. The secure management and synchronization of keys across multiple devices presents a considerable challenge. Additionally, the loss or replacement of devices could pose a serious problem, as it would result in the user losing access to their virtual wallet and funds. To address these issues, a solution is proposed that includes a mobile application and a browser plugin. The mobile application acts as the central component, allowing users to store and view the generated keys at any time. On the other hand, the plugin is expected to be available for installation on major platforms. In this work, it has been developed for Chrome. It transparently manages access to the dApps and the creation of associated keys. The system uses a smart contract deployed on the blockchain to facilitate communication between the application and the plugins, and it utilizes the IPFS to securely store the keys, enabling their recovery in case of device loss. The implemented system has been evaluated in terms of usability and performance. It provides a secure and intuitive environment with fast execution times for critical operations such as the addition of new dApps or key recovery.

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

1 Introduction

While the traditional client-server paradigm offers a straightforward and speedy approach for many applications, it comes with significant drawbacks compared to decentralized systems. Centralization presents a single point of failure, making the entire system more vulnerable to attacks and data leaks. Additionally, having to rely on a centralized authority can lead to privacy and trust issues. Centralized systems also tend to create bottlenecks, limiting the platform's scalability. In contrast, decentralization enhances system resilience, transparency, and security, making it an interesting choice over centralization.

The decentralized paradigm has numerous applications that enhance security and transparency, mainly by eliminating central authorities. Notable examples include blockchain and cryptocurrency (such as Ethereum, Bitcoin, and Solana), peer-to-peer networks (like BitTorrent, IPFS, and Gnutella), or decentralized finance (DeFi) platforms (such as Uniswap or Compound) [1]. These systems distribute control and data among various nodes, reducing the risk of a single point of failure and ensuring the availability of the network remains unaffected even if individual nodes fail.

The use of the blockchain technology offers several benefits, including resiliency, immutability, accessibility, and audibility. When a new block is added to the chain, the information it contains cannot be modified or removed. This data is replicated across all nodes that take part in the distributed network, enhancing failure resistance and enabling the easy tracking of attackers.

To interact with the blockchain, users must have a key pair (consisting of a private and public key). This key pair is a sensible piece of information that is critical for engaging with the blockchain, as it securely links the user to the network and demonstrates their ownership of assets. However, managing these sensitive keys requires a secure solution—this is where a wallet comes in. A wallet serves as a crucial tool that not only generates and stores these keys but also authenticates the user and manages their digital cryptocurrency, acting as a secure bridge between the user and the blockchain.

To secure access to the wallet, users must set a password. However, it is well known that people often choose passwords that are easy to remember, even if they compromise security. As a result, many online services now enforce composition policies that require users to create strong passwords, incorporating elements such as numbers, special characters, and both lowercase and uppercase letters [2].

Given the critical nature of the information stored in a wallet—particularly the key pair needed for blockchain interactions—using a strong password is crucial. Therefore, these composition policies are implemented in this project. Additionally, losing access to the wallet could mean losing access to the blockchain and any related smart contracts, with the most severe consequence being the potential loss of all assets.

Additionally, in the blockchain environment, users face heightened risks if they reuse the same cryptographic key pair across multiple decentralized apps (dApps). Attackers can exploit this pattern through record linkage attacks to gain access to user data. To reduce this risk, users should generate a unique key pair for each dApp they access.

One of the major challenges users face today is managing and recovering the many

keys needed to access different decentralized services. Additionally, keeping all this information synchronized across multiple devices adds another layer of complexity.

1.1 Proposal

To address these challenges, a fully decentralized wallet for secure key management is proposed. This wallet will create, store, and manage cryptographic keys transparently and automatically, ensuring their security while relieving users of the burden of manual management. Additionally, it will provide a robust recovery mechanism in case the user loses or changes their device.

The proposed system must meet the stringent requirements of modern devices, ensuring secure key management and synchronization across multiple platforms. The wallet will operate seamlessly within a multi-platform ecosystem while maintaining the advantages of a decentralized framework.

To achieve this description, we propose the usage of well-established security standards such as BIP39 and BIP32 to generate robust keys based on a user-provided password, also allowing their recovery if needed. To minimize dependence on centralized entities, the solution will utilize the decentralized nature of technologies such as the InterPlanetary File System (IPFS) and the blockchain, both of which are based on peer-to-peer networks.

The solution features a mobile application and a browser plugin. The plugin handles the seamless creation of keys for newly accessed dApps, while the mobile application is responsible for the secure management and visualization of these keys. The application deploys a smart contract on the blockchain to ensure system integrity and transparency. This setup not only guarantees the security of transactions but also facilitates smooth communication between the device and the browser.

2 State of art

The literature presents multiple approaches to designing functional and secure wallets. Depending on how the wallets create, store, recover, and utilize blockchain keys, they can be broadly categorized into three main types:

1. **Trusted Third-Party-Dependent Wallets**: The first category, often the most common, involves storing keys in one or more external storage locations. These repositories could be servers managed by a service provider or other third parties selected by the users. Some key contributions to this category are [3–10]. For example, [3] describes a two-server password-authenticated secret sharing (2PASS) approach. Initially, users distribute shares of their password and secret key to two servers. Users can later retrieve these shares by providing new secret shares of their password to the same servers, which then jointly verify whether the received secrets correspond to shares of the same password. Likewise, [7] uses a threshold secret-sharing algorithm to distribute shares of cryptographic keys among third parties for key backup and recovery. HasDPSS [11] advances this concept by incorporating a blockchain-based hierarchical access structure for decentralized key management. Eventually, recent works [8, 10] have enhanced these methods' security with biometric encryption of key shares. Despite these advancements, a significant limitation is the reliance on external entities for key protection.

2. **Multi-Signature Wallets**: This category includes solutions that focus on safeguarding cryptographic keys through multi-signature schemes. For instance, [12] presents a threshold-optimal signature algorithm where the secret key is distributed among n parties and that requires a subset of them to sign transactions, while [13] improves this proposal by using homomorphic encryption to decrease computational costs. The main drawbacks of these approaches are dependency on participant availability for key recovery and reliance on external entities.

3. **Seed-Derived Wallets**: These wallets generate cryptographic keys based on an initial mnemonic seed. Popular web systems like Metamask [1] use this approach, where the user must remember or securely store their mnemonic words to allow key recovery. Alternative methods have been explored to address the risks of relying solely on mnemonic phrases, such as using biometric data [14] or personal information [15, 16] for key recovery. For example, [14] employs biosensors for key recovery but faces limitations due to physiological changes (e.g., alterations in physiological conditions after an accident can drastically affect biosensor responses, making key recovery impossible). In [15], the "Partial Knowledge Recovery Scheme" (PKRS) is introduced. This scheme is used to recover an encrypted private key by asking personal security questions. Similarly, [16] explores image-based methods for key recovery rather than security questions. Lastly, [17] suggests using hardware wallets with Elliptic-Curve Diffie-Hellman (ECDH) protocols for secure backup, although this method may be impractical due to the substantial manual key management it requires.

[1] **Metamask:** https://metamask.io/

3 Technologies

In this section, we present a comprehensive explanation of the technologies used within the project.

3.1 Blockchain

The first approach to Blockchain technology was introduced in 1991 by Stuart Haber and W. Scott Stornetta [18]. Initially, they proposed an algorithm to time-stamp documents that makes it impossible for a user to back-date or forward-date her document. Later on, more work was developed in the field, with the contribution of the group under the pseudonym of Satoshi Nakamoto in 2008. The article not only conceptualized the blockchain theory but also introduced the concept of the bitcoin cryptocurrency [19].

The blockchain is a database of transactions that is continuously updated and shared across a decentralized network. It is constituted of blocks, each of which is formed by transactions. Various public blockchains enable users to easily add new blocks to the chain while preventing the removal of existing blocks. This means that any attempt by an attacker to alter any information would require modifying the data on the majority of the nodes in the network, thereby making it more secure [20].

In order to interact with the blockchain, an individual has to create an account and fund it with tokens, since transactions require digital money to be successful. These tokens can be acquired with real money or via token faucets. A token faucet provides an easy method for users to earn cryptocurrency without needing technical expertise or making any purchase of digital assets. They are typically developed to attract users to newly established networks, although such tools are becoming less common nowadays.

The common procedure of a faucet involves providing a small amount of tokens to a specified account, with usage generally restricted to once every 24 or 48 hours. Alternatively, some faucets utilize the computational power of your device for mining activities and reward you with tokens after a designated period based on your contribution.

In this project, the Sepolia PoW Faucet [2] has been used. This tool dispenses tokens used on the Ethereum testnet (Sepolia) in exchange for mining contributions, with a maximum allocation of 2.5 ETH per mining session. It is highly sought after by Web3 developers who wish to test their smart contracts and dApps on the testnet before deploying them on the main Ethereum network. Additionally, the faucet employs security policies to prevent bots from monopolizing the tokens, ensuring that other users have the opportunity to receive some funds [21]. An example of this tool can be seen in Figure 1.

The decentralized blockchain platform Ethereum was conceived in 2013 by Vitalik Buterin as an improvement of Bitcoin's technology. It would encompass a platform that extends beyond financial transactions to support a diverse range of applications [22], just as it is explained on their webpage, "While Bitcoin is only a payment network, Ethereum is more like a marketplace of financial services, games, social networks and other apps." [20]. Unlike Bitcoin, it is programmable and supports the deployment of smart contracts written in Solidity. Its native cryptocurrency, Ether (ETH), is often

[2] **Sepolia faucet:** https://sepolia-faucet.pk910.de/

mistakenly referred to by the same name as the blockchain, Ethereum.

Some concepts commonly used within the Ethereum blockchain technology will be introduced below:

Accounts Accounts are virtual wallets used to store cryptocurrency and allow user interaction with the blockchain. Two different kinds of accounts can be created in Ethereum: user accounts and contract accounts. Each account is uniquely identified by an address, which will be explained in the following section.

Addresses Ethereum addresses are generated by appending the hexadecimal prefix `"0x"` to the rightmost 20 bytes of the Keccak-256 hash applied to the ECDSA public key.

Example of public key:

`0x049d89ca59d7b54831c865f6d48cc864f5e8e3531f99687d4cabe14b9c9707b7e2`

`ef9cf8966d49f12c1b081b3cf9bf8eaf79599c7c8a21825ef61d90fe6f9d5a`

Keccak-256 hash:

`0x428ce0b3bfd786fe2c62348c80fc6725d928cc39a77dcfd7f386b634519f8d0a`

Address:

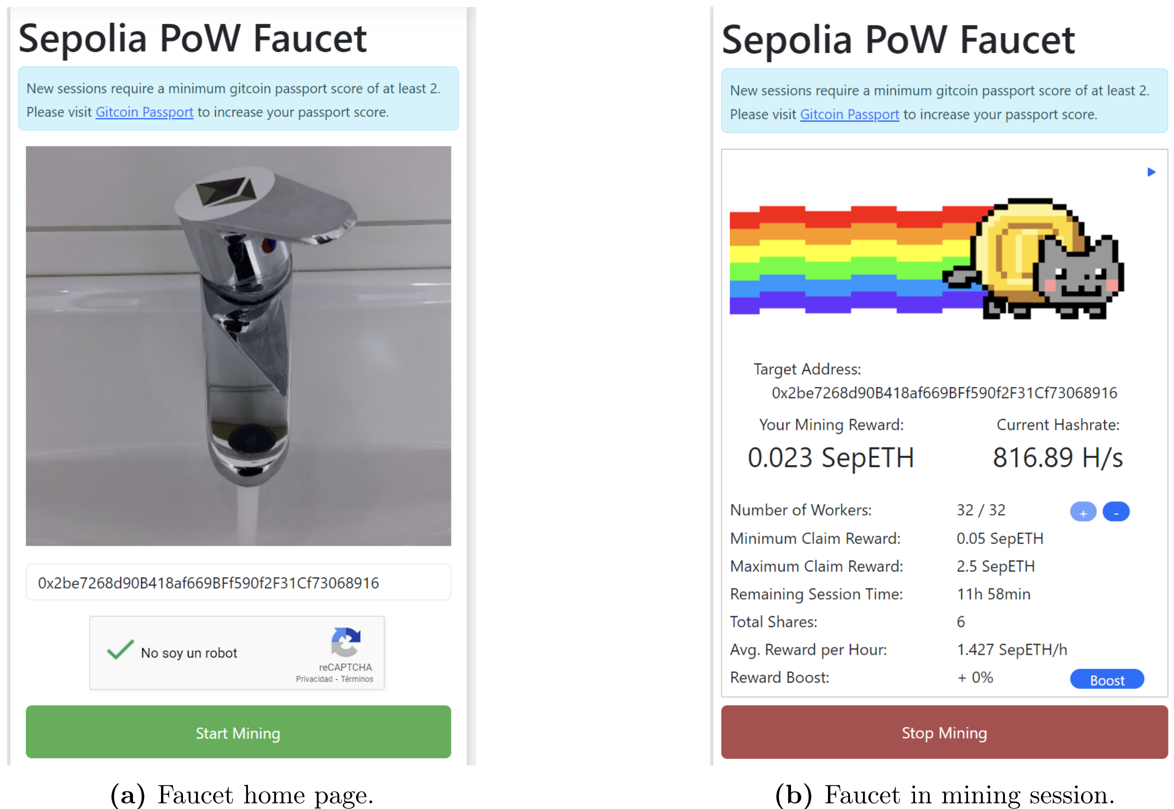
`0x80fc6725d928cc39a77dcfd7f386b634519f8d0a`

Gas On the Ethereum network, gas is the standard unit used to measure the computational effort required for a transaction (that is, the transaction fee). Miners charge fees in ETH to the sender so that their transaction is included in the blockchain. Think of it as paying for computing power to process the transaction. However, if the gas provided for a transaction is insufficient, validators may reject it, resulting in the loss of the offered funds and the transaction being reverted. To ensure a successful transaction, two gas-related requirements must be met: gas price and gas limit. The gas price fluctuates based on the network's current state (e.g. if the demand is high, the gas price will grow and vice-versa). The gas limit, on the other hand, depends on the type of transaction being committed. For instance, a simple fund transfer requires a gas limit of 21,000 units of gas. Nevertheless, if the transaction involves interacting with a smart contract, the gas limit increases significantly [23].

Contracts A smart contract is a program that runs on the Ethereum blockchain, encompassing both logic (functions) and data (state). These contracts are written in Solidity [3], a specific programming language designed for defining smart contracts on Ethereum. In Solidity, functions and variables can be specified as an interface that serves as the rules and agreements between different parties. Once deployed, the code becomes immutable, guaranteeing it cannot be altered. Additionally, all transactions and codes are publicly visible on the blockchain, ensuring transparency [24].

As an example, a smart contract featuring a simple data storage application is shown in Code 1.

[3] **Solidity:** https://soliditylang.org/



(a) Faucet home page. (b) Faucet in mining session.

Figure 1: Example of the PoW Sepolia Faucet.

Code 1 Contract example written in Solidity.

```
pragma solidity ^0.8.0;

contract SimpleStorage {
     uint256 public storedData;

     function set(uint256 x) public {
          storedData = x;
    }

    function get() public view returns (uint256) {
         return storedData;
    }
}
```

If somebody wants the contract to store funds (such as for an exchange between two parties), it is necessary to define a method using the payable modifier.

3.1.1 Ganache

Ganache is a tool that allows running a local blockchain for fast and secure Ethereum application development [25]. It is typically used to test dApps in a safe and controlled environment before moving to production.

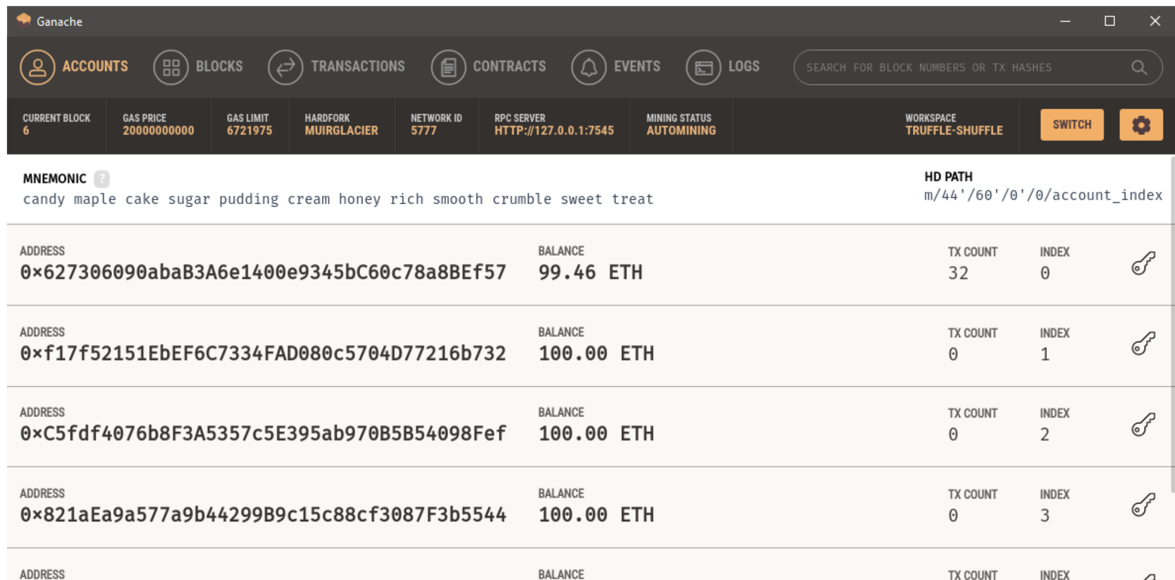


Figure 2: Example of the Ganache GUI homepage.

One significant advantage of Ganache is the virtually unlimited availability of funds, along with the instant mining of transactions, introducing minimal delay in the app operations. It provides several initial accounts with preloaded funding (e.g. 100 ETH), allowing straightaway interaction with the blockchain, deployment of smart contracts, and transaction execution. See an example of the Ganache UI in Figure 2.

Nevertheless, Ganache has proven not to be adequate for this project due to its restriction on dynamic account creation. Our solution aims to create new accounts based on keys generated using the BIP32 protocol. Furthermore, in September 2023, the blockchain software technology company Consensys [4] announced the discontinuation of Ganache [26], advising clients to transition to other ecosystems like Remix [5], Thirdweb [6] or Hardhat [7].

Consequently, Ethereum's Sepolia testnet has been used as an alternative. This testnet offers a production-like environment that closely resembles the Ethereum mainnet.

3.1.2 Sepolia

Sepolia was a proof-of-authority (PoA) testnet created in October 2021 by Ethereum developers and it has been maintained ever since, although it transitioned to a proof-of-stake (PoS) consensus mechanism.

Tesnets function as blockchain environments that replicate the conditions of the mainnet, but operate on separate registers. They provide a risk-free platform for developers to test their applications and smart contracts before deploying them on Ethereum's mainnet [27].

[4]**Consensys:** https://consensys.io/
[5]**Remix IDE:** https://remix.ethereum.org/
[6]**Thirdweb:** https://thirdweb.com/
[7]**Hardhat:** https://hardhat.org/

3.1.3 Etherscan

Etherscan [8] is a free Ethereum block explorer that provides search, API, and analytics capabilities. It supports both the Ethereum mainnet and the Sepolia testnet. This tool is invaluable for developers, offering detailed information on successful and reverted transactions, including the reasons for reversion, which can significantly simplify the debugging process. Users can search by account address, transaction hash, block hash, and more.

3.2 Web3 library

Web3 is a new concept of the internet based on decentralized blockchain services, as opposed to the current Web2, where centralized platforms like Google, Amazon, or Facebook dominate the network. Here, users will have more control over their data, online identity, and digital assets [28].

Some examples of Web3 applications are:

- **Decentralized Finance (DeFi):** Blockchain-based financial services that offer alternatives to traditional banking.
- **Non-Fungible Tokens (NFTs):** Unique digital assets representing ownership of virtual or physical items.
- **Decentralized Autonomous Organizations (DAOs):** Collectives governed by smart contracts, operating without traditional hierarchies.

In order to ease the transition to this new generation of the internet, several libraries have been implemented for different programming languages. For instance, web3j [9] is the library used for Java, whereas web3js [10] is used for JavaScript. A simple overview of the interaction with the blockchain using the aforementioned libraries can be seen in Figure 3.

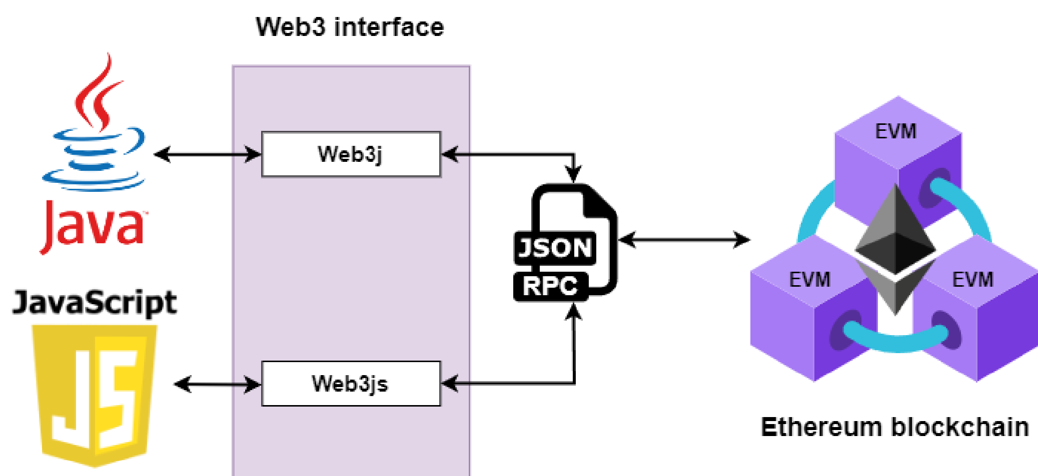


Figure 3: Web3 libraries' interface architecture.

[8] **Etherscan:** https://etherscan.io/
[9] **Web3j:** https://docs.web3j.io/
[10] **Web3js:** https://web3js.org/

3.3 Bitcoin Improvement Proposal (BIP)

Since Bitcoin is a software program, it needs periodic improvements and bug fixes. This is generally done through Bitcoin Improvement Proposals (BIP), which often enhance security and efficiency.

Anyone can suggest a BIP to the Bitcoin community. Since Bitcoin is not centrally managed, the network's users and developers collectively handle the approval and testing of new contributions and proposals [29].

3.3.1 BIP39

The BIP39, introduced by Palatinus in 2013, outlines the implementation of a mnemonic code or mnemonic phrase for generating deterministic wallets. This mnemonic sentence consists of a sequence of easy-to-remember words [30].

The process involves two main steps: generating the mnemonic and converting it into a binary seed, for further use in deterministic wallet creation, such as in BIP32.

First, a certain number of entropy bytes are generated (ranging from 128 to 256 bytes). The length of the entropy determines the number of words in the mnemonic sentence.

$$ENT = GenerateRandom(size)$$

Next, a SHA256 hash is performed on the entropy. The first ENT/32 bits are taken as the checksum (CS).

$$HASH = SHA_{256}(ENT)$$

$$CS = TakeFirst_{(|ENT|/32)}(HASH)$$

The checksum is then appended to the initial entropy. The concatenated bits are split into groups of 11 bits, with each group encoding a number from 0 to 2047. These numbers serve as indices into a predefined wordlist.

$$INDICES = divide_{11bits}(ENT|CS)$$

Finally, the indices are converted into their corresponding words to build up the mnemonic sentence (MS).

$$MS = parse(INDICES)$$

In Table 1, the relation between the initial entropy length |ENT|, the checksum length |CS|, and the length of the mnemonic sentence |MS| is described.

|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 1: Size relation between ENT, CS and MS in BIP39.

Once the mnemonic phrase is generated, it is converted into a seed using the PBKDF2 function. The UTF-8 encoding of the mnemonic sentence is used as the password, and the salt is formed by concatenating the string "mnemonic" with a passphrase (if no passphrase is provided, an empty string is used by default).

$$seed = PBKDF2(\text{password} = MS, \text{salt} = \texttt{"mnemonic"} + passphrase)$$

The iteration count is set to 2048, and HMAC-SHA512 is used as a pseudo-random function. The length of the derived key is 64 bytes.

3.3.2 BIP32

In early 2012, Pieter Wuille introduced the BIP32 protocol. This protocol revolutionized the management of cryptocurrency wallets by enabling the creation of hierarchical deterministic wallets (from now on HD Wallets) [31]. This protocol permits the generation of a tree of key pairs from a single seed, thereby allowing users to hold a unified wallet system across different clients, each capable of accessing all generated key pairs.

The BIP32 specification is divided into two main components:

1. **A method for generating key pairs**: this involves deriving a hierarchical tree of key pairs from an initial seed.
2. **Wallet structure construction**: how to build a wallet structure utilizing the aforementioned tree.

For the purposes of this protocol, we assume the use of elliptic curve cryptography based on the secp256k1 parameters defined by Brown [32].

The assumed standard conversion functions are:

- $point(p)$: returns the coordinate (x, y) resulting from Elliptic Curve (EC) point multiplication of the secp256k1 base point with the integer p.
- $ser_{32}(i)$: serializes a 32-bit unsigned integer i into a 4-byte array, with the most significant byte first.
- ser_{256}(p): serializes the integer p into a 32-byte array, with the most significant byte first.
- $ser_P(P)$: serializes the coordinate pair $P = (x, y)$ into a byte sequence using the compressed form specified in SEC1. It combines $0x02$ or $0x03$ with $ser_{256}(x)$, where the header byte is determined by the parity of the omitted y coordinate.
- $parse_{256}(p)$: translates a 32-byte array into a 256-bit number, with the most significant byte first.

The derivation protocol includes a method to obtain several child keys from a parent key. To ensure these do not depend solely on the parent key, both private and public keys are first extended with an additional 256 bits of entropy, known as the chain

code. The 32-byte chain code is identical for both keys. The extended key is written as (SK, c), with SK as the private key, $PK = point(SK)$ as the public key, and c as the chain code.

Each extended key can generate 2^{31} normal child keys and 2^{31} hardened child keys, each assigned a specific index. Normal child keys are indexed from 0 to $2^{31} - 1$, while hardened child keys uses indices from 2^{31} to $2^{32} - 1$.

The Child Key Derivation (CKD) functions take an extended parent key as input and return an extended child key. The algorithm varies whether the child key is hardened or unhardened, and it also differs based on whether the key is private or public. A notable vulnerability in BIP32 wallets was found. If an attacker has access to the master public key and any child private key, she can derive the master private key. Addressing this issue the protocol included hardened child keys, which do not compromise the master private key if exposed. These hardened keys also prevent the generation of public child keys from the master public key, avoiding brute force attacks. As a result, only the hardened child key derivation function will be explained.

$$CKD_{private}((SK_{parent}, c_{parent}), i) \rightarrow (SK_i, c_i) \quad (1)$$

Based on the parent private key, extended child private keys are derived as specified in equation 1, following the next steps:

1. Determine whether $i \geq 2^{31}$ to identify if the child is a hardened key.
   (a) For a hardened child key:

   $I = HMAC{-}SHA_{512}(\text{Key} = c_{parent}, \text{Data} = 0x00 \,||\, ser_{256}(SK_{parent}) \,||\, ser_{32}(i))$

   (Note: The prefix $0x00$ extends the private key to a length of 33 bytes.)

   (b) For a normal child key: $i+ = 2^{31}$ to attempt the derivation of a hardened child.
2. Divide I into two 32-byte arrays, denoted as I_L and I_R.
3. Compute the child key SK_i as follows:

   $$SK_i = parse_{256}(I_L) + SK_{parent} \mod n$$
4. Set the child chain code c_i to I_R.
5. If $parse_{256}(I_L) \geq n$ or $SK_i = 0$, the resulting key is invalid, and the process should move to the next value for i.
   (Note: The likelihood of this occurrence is less than 1 i 2^{127}).

For this derivation protocol to be started, an initial parent key, known as the master key, is required:

1. Randomly generate a seed byte array, denoted as S (of length ranging from 128 to 512, although 256 is recommended). In this project, the seed is produced using the BIP39 protocol, as described earlier.

2. Compute $I = HMAC - SHA_{512}(\text{Key} = \texttt{"Bitcoin seed"}, \text{Data} = S)$
3. Divide I into two 32-byte arrays, I_L and I_R.
4. The master private key, SK_0 will be $parse_{256}(I_L)$, and the master chain code c_0 will be I_R.

3.4 Android Studio

Android Studio [11] is the official IDE for Android app development. It is based on the IntelliJ IDEA code editor and tools, but it offers a wide variety of extra features to make Android programming smoother [33]. These tools encompass a Gradle-based build system, an Android emulator for testing all kinds of Android devices (smartphones, watches, TVs, cars...), GitHub version control integration, and much more.

Applications developed for Android OS are typically programmed using either Java or Kotlin. In 2017, Google declared Kotlin as the "Official language for Android Development" [34]. This announcement has led to discussions about which language to select. Both Java and Kotlin have their respective advantages and disadvantages. My preference for Kotlin over Java relies on several key benefits. Firstly, Kotlin is a relatively new and modern programming language, having been launched in 2016. Secondly, it provides null safety, allowing variables to be declared as either nullable or non-nullable, which enhances code reliability. In Java, developers often face issues with `NullPointerException` when object references have null values, leading to significant problems. In contrast, Kotlin ensures that all types are non-nullable by default. If a null value is assigned or returned, it will cause a compilation error. To handle null values, Kotlin requires variables to be explicitly marked as nullable using the `"?"` operator. Lastly, Kotlin reduces code verbosity compared to Java, resulting in more concise code. For instance, Kotlin's data classes simplify code by eliminating the need to manually define getters and setters for variables. Instead, these variables can be accessed directly.

All libraries natively programmed in Java can also be used in Kotlin. Therefore, the interoperability between these two languages makes it easier for the programmer to develop Android apps without further restrictions when moving to Kotlin.

3.5 Plugin

A plugin is a piece of code that can be integrated into a browser to enhance its functionality beyond what is originally supported [35]. Extensions are developed using web-based technologies such as HTML, CSS and JavaScript. Popular extensions include ad blockers, browser personalization tools, and writing aids like Grammarly [12]. These tools significantly enhance user experience by providing additional features and customization options.

The anatomy of an extension lies in the `manifest.json` file, which is compulsory for all extensions. This file contains essential metadata such as the extension's name, version, and permissions. Additionally, it includes references to other project files.

[11] **Android Studio:** https://developer.android.com/studio
[12] **Grammarly:** https://app.grammarly.com/

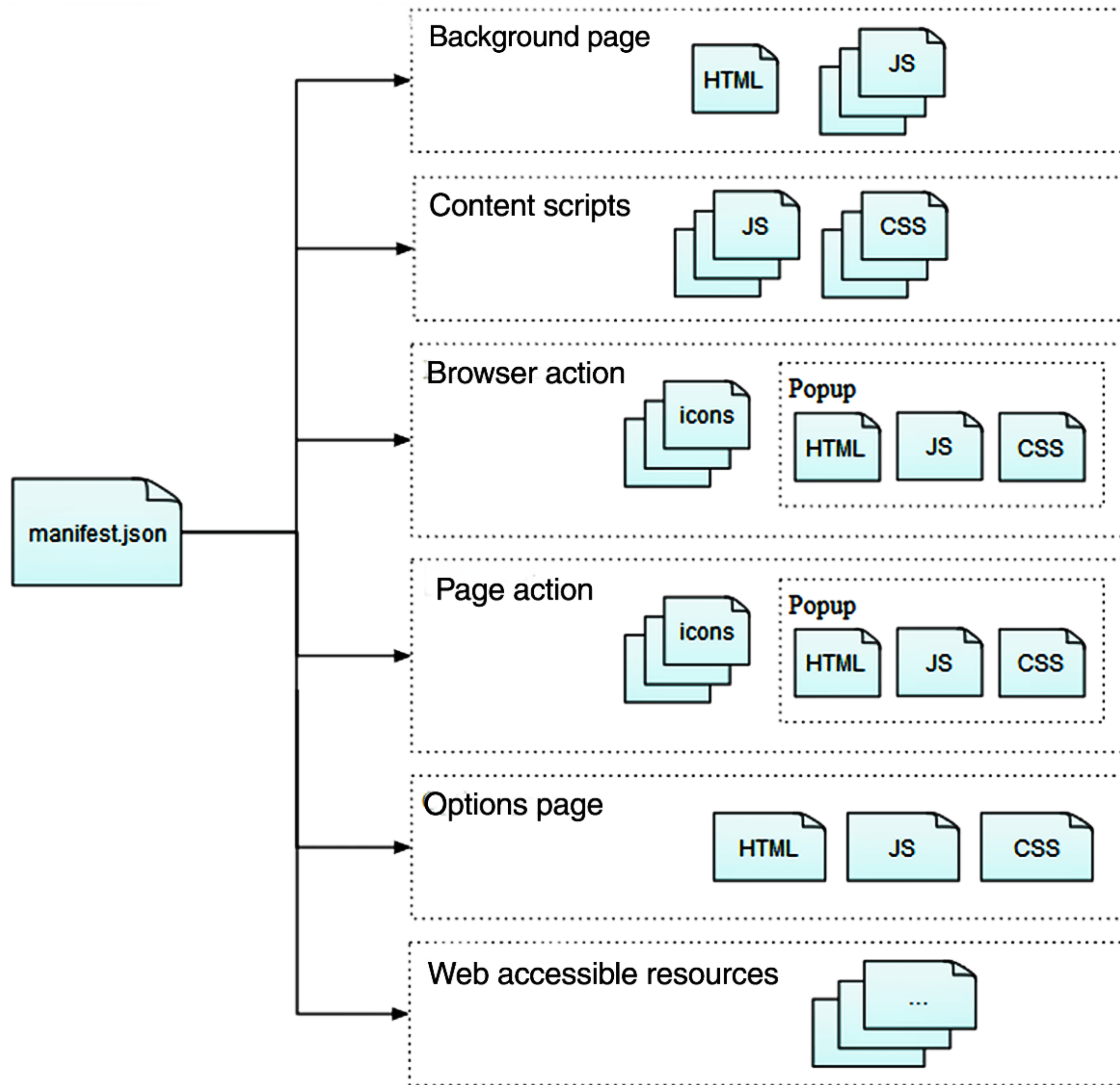


Figure 4: Plugin anatomy.

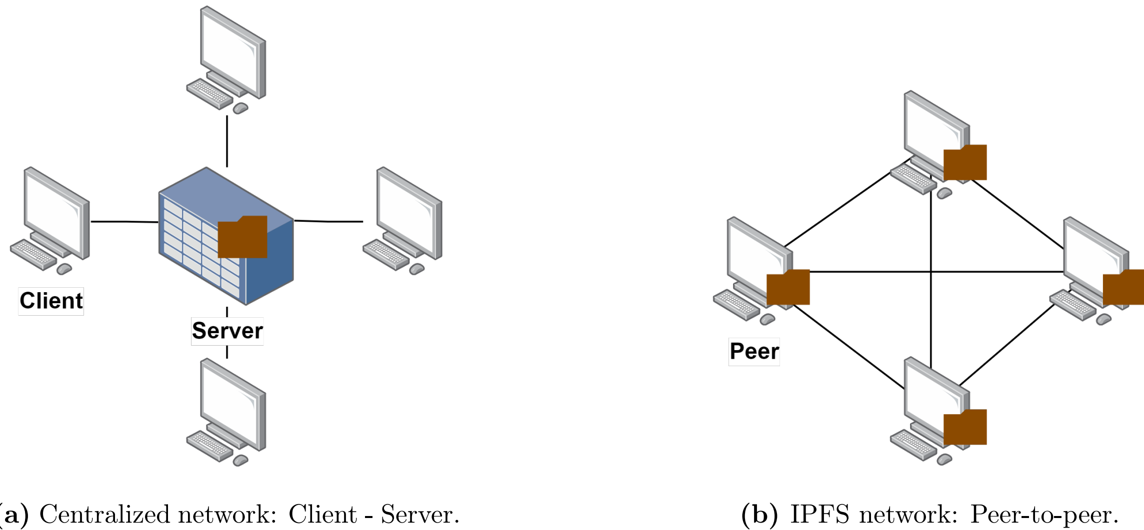
Such files include background scripts, icons, popups, content scripts, or web-accessible resources, as illustrated in Figure 4 [36].

3.6 IPFS

The InterPlanetary File System (IPFS), introduced by Juan Benet in 2014, is a distributed peer-to-peer file system designed to connect all the nodes within the same global namespace file system [37]. Files are uniquely identified through content-addressing, allowing users to store and retrieve files based on their content address from any node in the network using a distributed hash table (DHT).

Notably, IPFS is implemented in various technologies and languages, reflecting its versatility as a protocol. Much like BitTorrent, IPFS facilitates both the hosting and retrieval of content across the network.

As stated in previous sections, the key advantages of a decentralized architecture are resilience and reliability, security and privacy, and improved scalability. IPFS



(a) Centralized network: Client - Server.

(b) IPFS network: Peer-to-peer.

Figure 5: Client-server centralized model vs. IPFS decentralized model.

exemplifies these benefits through its distributed file storage approach. By dispersing data across multiple nodes, IPFS enhances resilience and reliability, ensuring that files remain accessible even if some nodes fail. The use of content-addressing and cryptographic hashing in IPFS reinforces security and privacy, as each file is uniquely identified and verified, reducing the risk of manipulation and unauthorized access. Furthermore, IPFS improves scalability by allowing the network to grow gradually; as new nodes join, the system's capacity to manage increased data and traffic also expands. In addition, IPFS reduces costs by utilizing the resources of individual nodes rather than relying on a centralized infrastructure, thus providing a more cost-effective solution.

In Figure 5, we show a comparison between the centralized client-server paradigm and the decentralized IPFS network architectures.

4 Requirements

After exploring the existing literature (Section 2), and analyzing the identified weaknesses, several key aspects that the system must address have been determined. These system requirements are categorized into functional and non-functional requirements and are introduced in the following sections.

4.1 Functional requirements

R1: Allow the creation of new accounts in the smartphone device.

R2: Enable the addition of new devices (plugins) to the app.

R3: Facilitate access to new dApps, ensuring transparent creation and management of the keys used within those dApps.

R4: Allow the visualization of all generated keys.

R5: Ensure secure synchronization of cryptographic keys and related data across multiple devices and platforms to provide seamless access for users.

R6: Include a robust recovery mechanism for all cryptographic keys to ensure users can regain access to their wallet in case of device loss or damage.

4.2 Non-functional requirements

4.2.1 Security and privacy requirements

R7: Passwords set by users must be secure enough to prevent common risks, such as dictionary attacks.

R8: The application must allow users to set two-factor authentication for enhanced security.

R9: Cryptographic keys must not be stored on any third-party system, avoiding reliance on external entities.

4.2.2 Design and implementation requirements

R10: The system must ensure high availability and fault tolerance in case of individual node failures.

4.2.3 Usability requirements

R11: The application and browser plugin must provide an intuitive, user-friendly interface that simplifies key management and recovery processes.

4.2.4 Performance requirements

R12: Ensure feasible execution times for key generation, management and recovery.

5 Design and architecture

In this section, we first outline the overall architecture of the system. Following that, a detailed specification of the smart contract is provided. Lastly, we examine the various use cases, accompanied by their respective sequence diagrams.

5.1 General structure

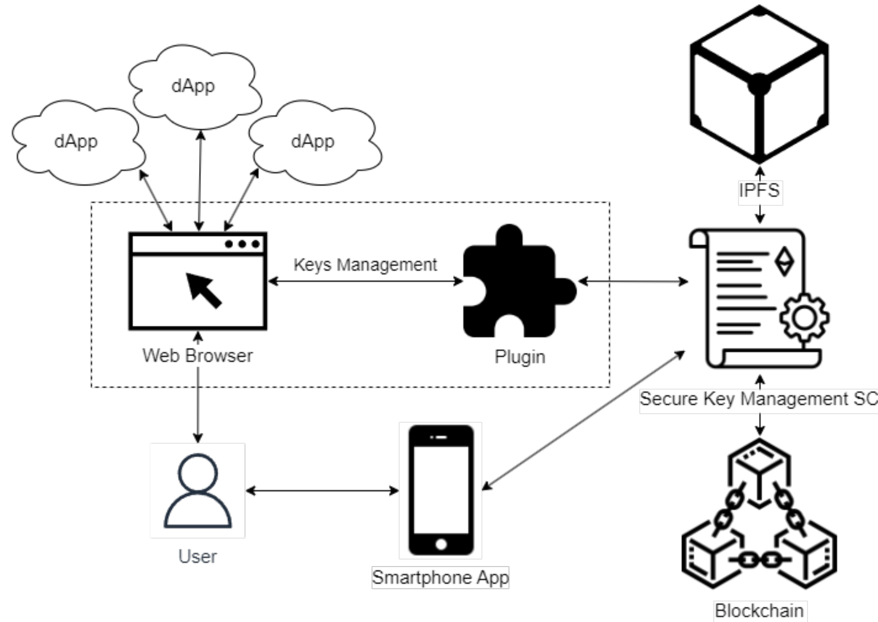


Figure 6: Architecture of the proposal.

The overall structure of the proposed system, illustrated in Figure 6, is composed of six main modules with which the user will interact via the smartphone app and browser plugins:

1. **Blockchain**: Enables the deployment of smart contracts for decentralized, secure key management and facilitates smooth communication between the app and the plugin.
2. **IPFS**: Offers decentralized and encrypted storage for the dApp keys, enabling secure communication between the smartphone and other devices.
3. **Secure Key Management Smart Contract**: Defines the rules and interfaces for interaction between the app and the plugins.
4. **Smartphone App**: Allows users to add newly installed plugins and monitor all associated keys through an intuitive interface. It also supports key recovery in case of device loss.
5. **Browser Plugin**: Installed on the user's web browser, this plugin creates and manages keys for various dApps, integrating with the smartphone app via the smart contract to maintain secure and consistent key usage.
6. **Dapps**: Decentralized or peer-to-peer applications running on a blockchain network. The plugin transparently manages the keys for dApps accessed by the user.

The system is designed to provide secure management and recovery of cryptographic keys for decentralized applications (dApps). Users start by installing the app on their smartphones. Next, they install a plugin on their web browsers—typically one plugin per device. Once the plugin is installed, it needs to be set up. During setup, the plugin generates a QR code, which the user scans with the smartphone app. This process registers the current browser as a trusted device and generates a new key pair specifically for that plugin.

After the setup is complete, users can browse the web, and whenever they access a new dApp, the plugin automatically generates a unique key pair based on the plugin's root key pair to authenticate the user in that dApp. All generated keys can be visualized and managed at any time through the smartphone app.

Additionally, if users change or lose their smartphone, they can easily recover all previously generated keys and restore their access to dApps, returning the system to its previous state. For more detailed information on each step, refer to Section 5.3.

5.2 Secure Key Management Smart Contract - SKM SC

The Secure Key Management SC not only facilitates the access to the cryptographic keys stored in the IPFS, but also acts as an intermediary, providing seamless communication between the smartphone app and the registered plugins during the setup of the system.

To achieve this behavior, the SKM SC includes a unique class that contains:

- **Smartphone app identifier:** This value, represented by the *smartphoneID* variable, uniquely identifies the smartphone app in the form of an Ethereum address. It is automatically assigned during the smart contract deployment, corresponding to the address of the transaction sender (the smartphone). This address is used to restrict certain functions to be callable only from the smartphone, thereby preventing unauthorized calls from the plugin.
- **Smartphone public key:** Represented by the *publicKey* variable, this master public key is generated when the user signs up and is stored in the smart contract during deployment. It is later used by the plugins to encrypt the dApp cryptographic keys.
- **Authorized plugins:** This list, represented by the *whiteList* mapping, stores the addresses of plugins authorized to interact with the smart contract. Each entry in the list is a *DeviceInfo* struct that includes a boolean *exists*, indicating whether the plugin has been added, and an *IPFSref*, which points to a file stored in IPFS.
- **References to cryptographic keys:** The *IPFSref* in each *DeviceInfo* struct within the *whiteList* mapping is a hash reference to a file in IPFS. This file contains links to other files that store the encrypted cryptographic keys.
- **Temporal field:** A temporary data field represented by the *temp* variable, used for exchanging information between the app and registered plugins.

Algorithm 1 SKM SC constructor.

Input: smartphoneAddress *addr*, smartphonePubKey PK_{sp}, digitalSignature ds_{sp}
Output: *res*
  $res \leftarrow error$
  if $verify(PK_{sp}, ds_{sp})$ **then**
    $smartphoneID \leftarrow addr$
    $publicKey \leftarrow PK_{sp}$
    $temp \leftarrow null$
    $whiteList \leftarrow < K, V >$
    $res \leftarrow success$
  end if
  return *res*

After successfully deploying the Secure Key Management Smart Contract (Algorithm 1), the user can interact with it through the following functions:

- `addDevice()`: This method, detailed in Algorithm 2, adds a new plugin address to the *whiteList*. It is exclusively callable from the smartphone app, which uses the smartphone's private key to sign the transaction.

Algorithm 2 Add device to the SC.

Input: deviceID *devAddr*, digitalSignature ds_{sp}
Output: *res*
  $res \leftarrow error$
  if $verify(smartphoneID, ds_{sp})$ **then**
    if $!whiteList[devAddr].exists$ **then**
      $whiteList[devAddr].exists = true$
      $res \leftarrow success$
    end if
  end if
  return *res*

- `removeDevice()`: In contrast to the `addDevice()` method, this function, described in Algorithm 3, removes a previously added plugin by setting the *exists* variable to false in the corresponding entry of the *whiteList*.

- `storeRef()`: This function (Algorithm 4) updates the *whiteList* mapping with a new value at the position corresponding to the address of the calling device. The value stored is a reference to the IPFS file where the dApp keys associated with a specific plugin are securely encrypted and stored. Only registered plugins or the smartphone app are authorized to invoke this function.

- `getRef()`: Under the same conditions as the `storeRef()` method, this function (shown in Algorithm 5) retrieves the value stored in the mapping for the calling device's address, returning the IPFS reference associated with the plugin.

- `modTemp()`: This function, detailed in Algorithm 6, updates the temporary value of the smart contract. It can only be invoked from the app.

Algorithm 3 Remove device from the SC.

Input: deviceID $devAddr$, digitalSignature ds_{sp}
Output: res
$res \leftarrow error$
if $verify(smartphoneID, ds_{sp})$ **then**
  if $whiteList[devAddr].exists$ **then**
    $whiteList[devAddr].exists = false$
    $res \leftarrow success$
  end if
end if
return res

Algorithm 4 Store the IPFS reference associated with a specific plugin.

Input: deviceID $devAddr$, IPFSreference ref, digitalSignature ds_{sp}
Output: res
$res \leftarrow error$
if $verify(smartphoneID, ds_{sp})$ OR $verify(devAddr, ds_{sp})$ **then**
  if $deviceAddr$ is in $whiteList$ **then**
    $whiteList[devAddr].IPFSref \leftarrow ref$
    $res \leftarrow success$
  end if
end if
return res

Algorithm 5 Get the IPFS reference associated with a specific plugin.

Input: deviceID $devAddr$, digitalSignature ds_{sp}
Output: res
$res \leftarrow error$
if $verify(smartphoneID, ds_{sp})$ OR $verify(devAddr, ds_{sp})$ **then**
  if $whiteList[devAddr].exists$ **then**
    $res \leftarrow whiteList[devAddr].IPFSref$
  end if
end if
return res

Algorithm 6 Modify the temporary value of the *temp* field of the SC.

Input: temporary value $newTemp$, digitalSignature ds_{sp}
Output: res
$res \leftarrow error$
if $verify(smartphoneID, ds_{sp})$ **then**
  $temp \leftarrow newTemp$
  $res \leftarrow success$
end if
return res

- `getTemp()`: This function, detailed in Algorithm 7, retrieves the current temporary value of the smart contract. It is exclusively callable from the plugins.

Algorithm 7 Get the temporary value of the *temp* field of the SC.

Input: deviceID $devAddr$, digitalSignature ds_{plugin}
Output: res
  $res \leftarrow error$
  if $verify(devAddr, ds_{sp})$ **then**
    $res \leftarrow temp$
  end if
  return res

- `getSmartphoneID()`: This method (Algorithm 8) retrieves the smartphone's account address.

Algorithm 8 Get the smartphone's account address (*smartphoneID* field of the SC).

Input: deviceID $devAddr$, digitalSignature ds_{plugin}
Output: res
  $res \leftarrow error$
  if $verify(smartphoneID, ds_{sp})$ OR $verify(devAddr, ds_{sp})$ **then**
    $res \leftarrow smartphoneID$
  end if
  return res

- `getPublicKey()`: This method, outlined in Algorithm 9, returns the smartphone's public key, which the plugin uses to encrypt a session key when generating new cryptographic keys for a dApp.

Algorithm 9 Get the smartphone's public key (*publicKey* field of the SC).

Input: deviceID $devAddr$, digitalSignature ds_{plugin}
Output: res
  $res \leftarrow error$
  if $verify(smartphoneID, ds_{sp})$ OR $verify(devAddr, ds_{sp})$ **then**
    $res \leftarrow publicKey$
  end if
  return res

5.3 Use cases

The complete protocol description is outlined in a forthcoming paper [38], which is currently under submission. A general overview of the actions that can be performed by the *User* in both the app and the plugin is shown in Figure 7. Each use case will be detailed in the following subsections. When applicable, the corresponding protocol steps will be presented in a tabular format. Note that the order followed to describe the protocol steps adheres to the original sequence, rather than the order defined in the use case diagram.

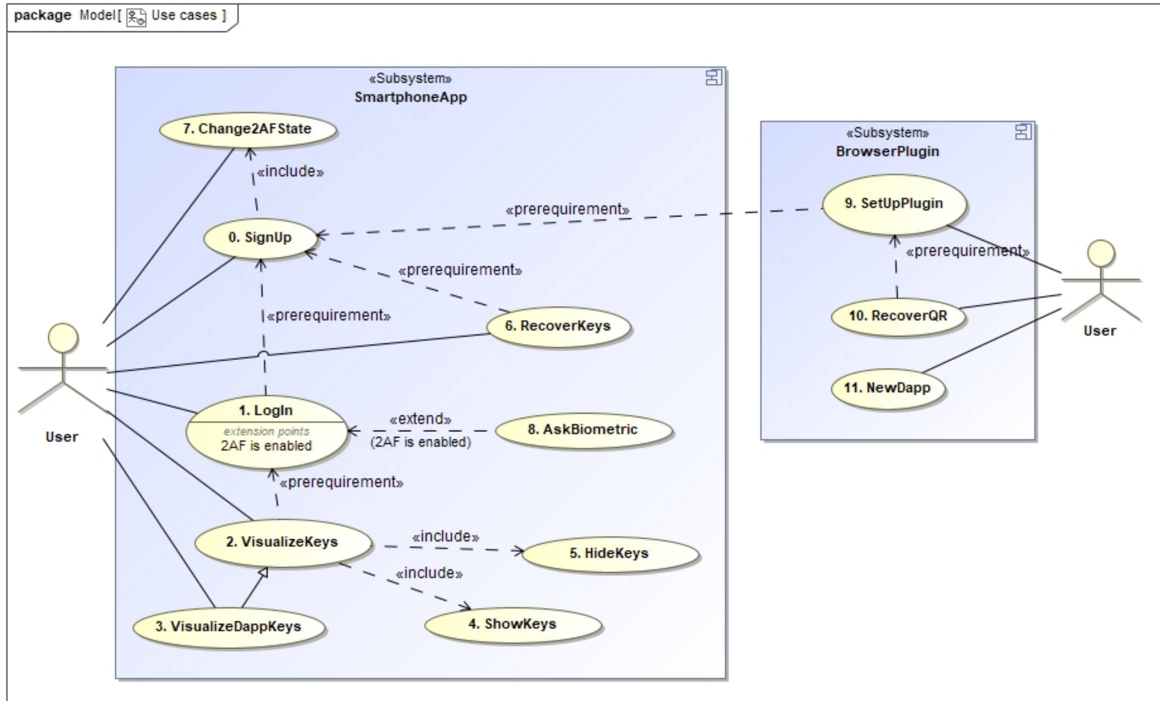


Figure 7: App and plugin use cases.

5.3.1 Smartphone App

The *User* can perform the following actions through the smartphone app:

0. SignUp This functionality (Figure 8) presents a sign-up page where *Users* must enter a valid email and password. The system continuously monitors both fields, updating the interface to indicate whether the email and password meet the necessary criteria. Once both fields are correctly filled out, the *User* can tap on the sign-up button, initiating a process that follows the steps shown in Table 2.

1. Login The *User* is presented with a login interface where they will enter their email and password. Upon attempting to log in, the provided credentials are compared with those stored in the app. If either field is incorrect, an error message is displayed. However, if the credentials are correct, the app will check if the *User* has enabled two-factor authentication and will prompt for biometric authentication accordingly. Upon successful authentication, the app will navigate to the home page. This is illustrated in Figure 9.

2. VisualizeKeys Allows the *User* to view cryptographic keys on the keys page. These keys are securely retrieved from the Android secure storage, where they have been encrypted and stored. Each key pair is displayed as a private key, public key, and chain code, with master keys also showing the associated SC address, although all keys are hidden by default for security reasons. The root keys linked to various plugins are also visible on this page. Additionally, *Users* can view keys associated with decentralized apps (dApps) using the 3. VisualizeDappKeys function and reveal hidden keys through the 4. ShowKeys function. The detailed process is available in Figure 10.

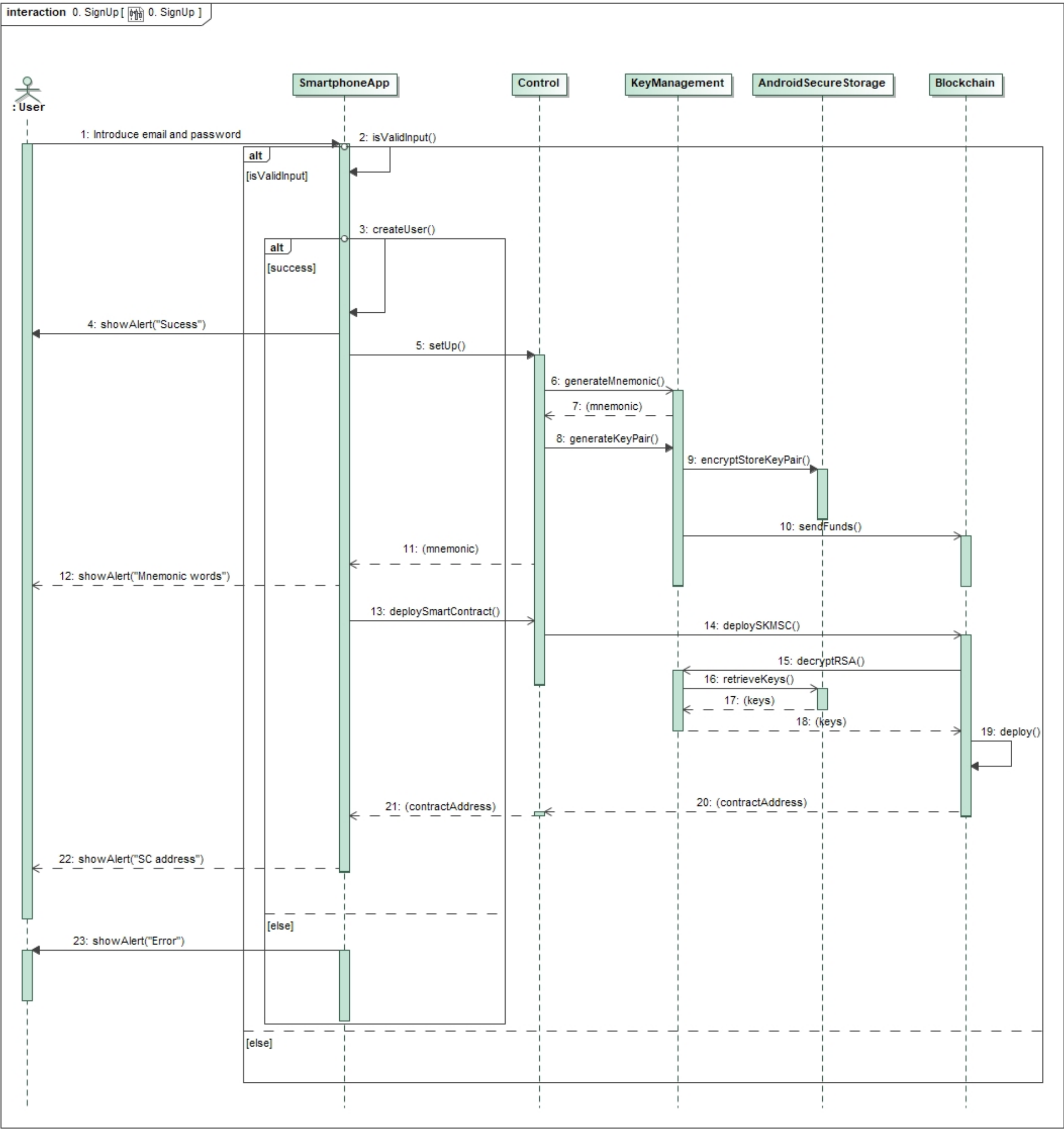


Figure 8: Use case 0. SignUp.

Step	Description
S1.1	The *User* installs the Smartphone App in her smartphone and creates a new secure login, by specifying a username and a password pw.
S1.2	The Smartphone App executes the BIP39 protocol using pw as input, generating 24 mnemonic words and a 512-bit seed, known as the BIP39 seed.
S1.3	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.
S1.4	The Smartphone App executes the BIP32 protocol using the BIP39 seed as input. This process generates the Root Secret Key SK_0 and the Root Chain Code C_0. The Smartphone App derives the Root Public Key PK_0 as follows: $PK_0 = SK_0 \times G$.
S1.5	The Smartphone App stores SK_0 in the smartphone's secure storage.
S1.6	The Smartphone App uses the Root Key pair (SK_0, PK_0) to deploy a new Secure Key Management Smart Contract (SKM SC) onto the blockchain, resulting in the generation of a hash hSC that uniquely identifies the published smart contract.

Table 2: "Setting up the Smartphone App" protocol step.

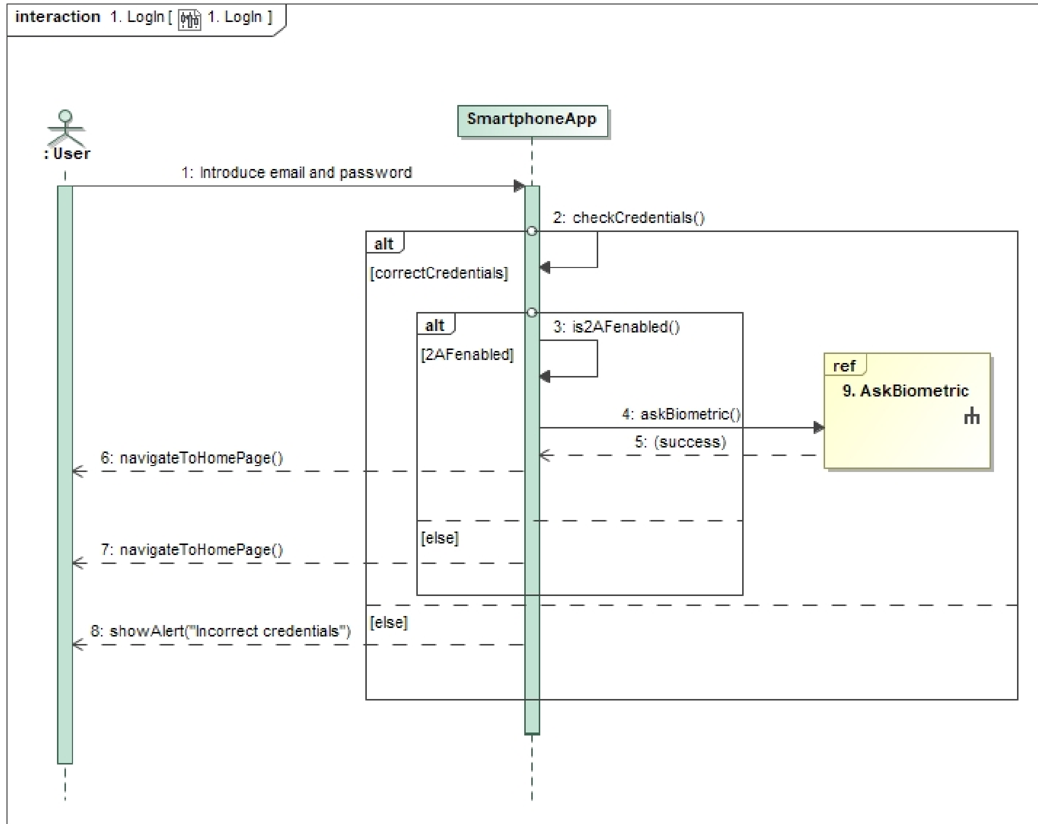


Figure 9: Use case 1. Login.

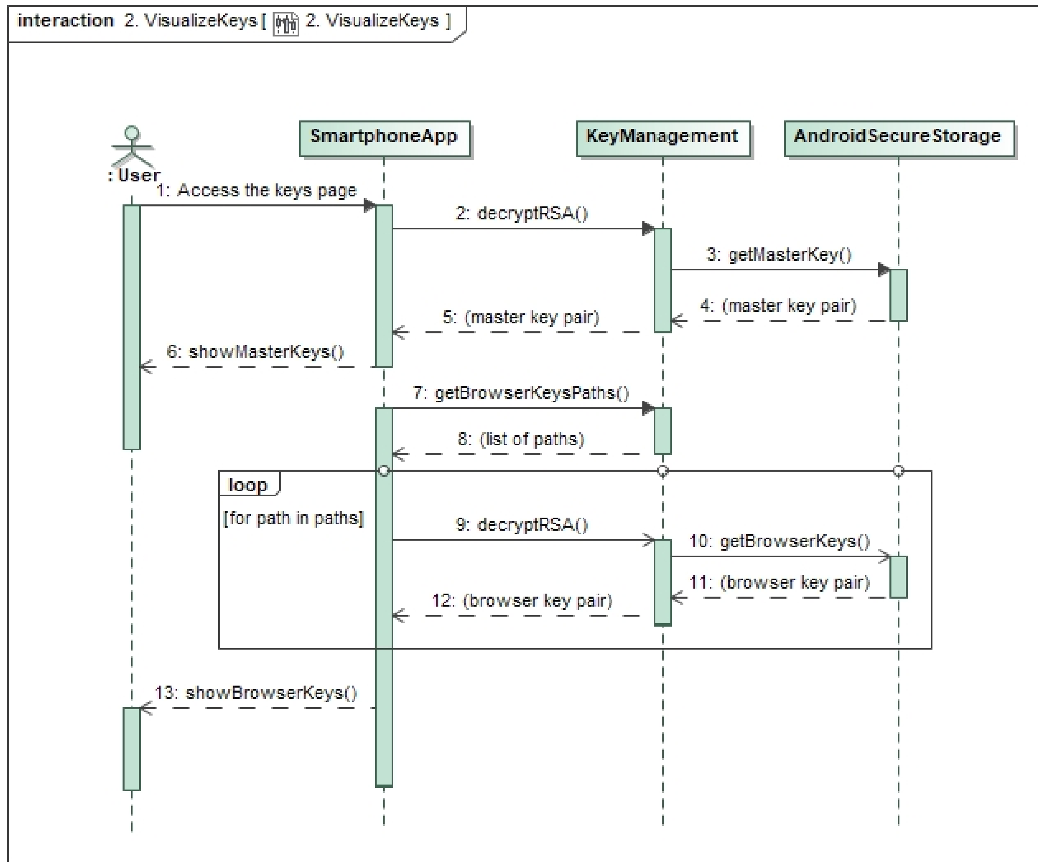


Figure 10: Use case 2. VisualizeKeys.

3. VisualizeDappKeys When the *User* swipes left on a browser key item, a new page displays all the dApp keys associated with that specific plugin. To achieve this, the app follows the steps specified in Table 3. The process can be seen in Figure 11.

4. ShowKeys By default, keys are hidden. To view them, the *User* can simply tap on a key item to expand it, revealing the private key, public key, and chain code. See the process in Figure 12.

5. HideKeys If the *User* has previously revealed the keys, they can return them to the hidden state by tapping on the key item again. This action will collapse the view, displaying only the device name for browser keys or the dApp URL for dApp keys. Figure 13 depicts this procedure.

6. RecoverKeys As shown in Figure 14, this process facilitates the recovery of keys in case the *User* loses access to them. This option appears as a "Lost device?" prompt on the login page. When selected, the *User* is directed to a page resembling the signup screen. The *User* must enter a valid email address and a secure password that meets all requirements, which must be the same as the one used during the initial signup. The app then navigates to a page where the *User* can input mnemonic words by selecting them from a provided list. After entering all 24 words, the *User* taps on the *Recover keys* button, prompting the app to request the number of previously registered devices. The user scans the corresponding number of QR codes. Once all QR codes are scanned, the recovery process begins with the steps shown in Table 4.

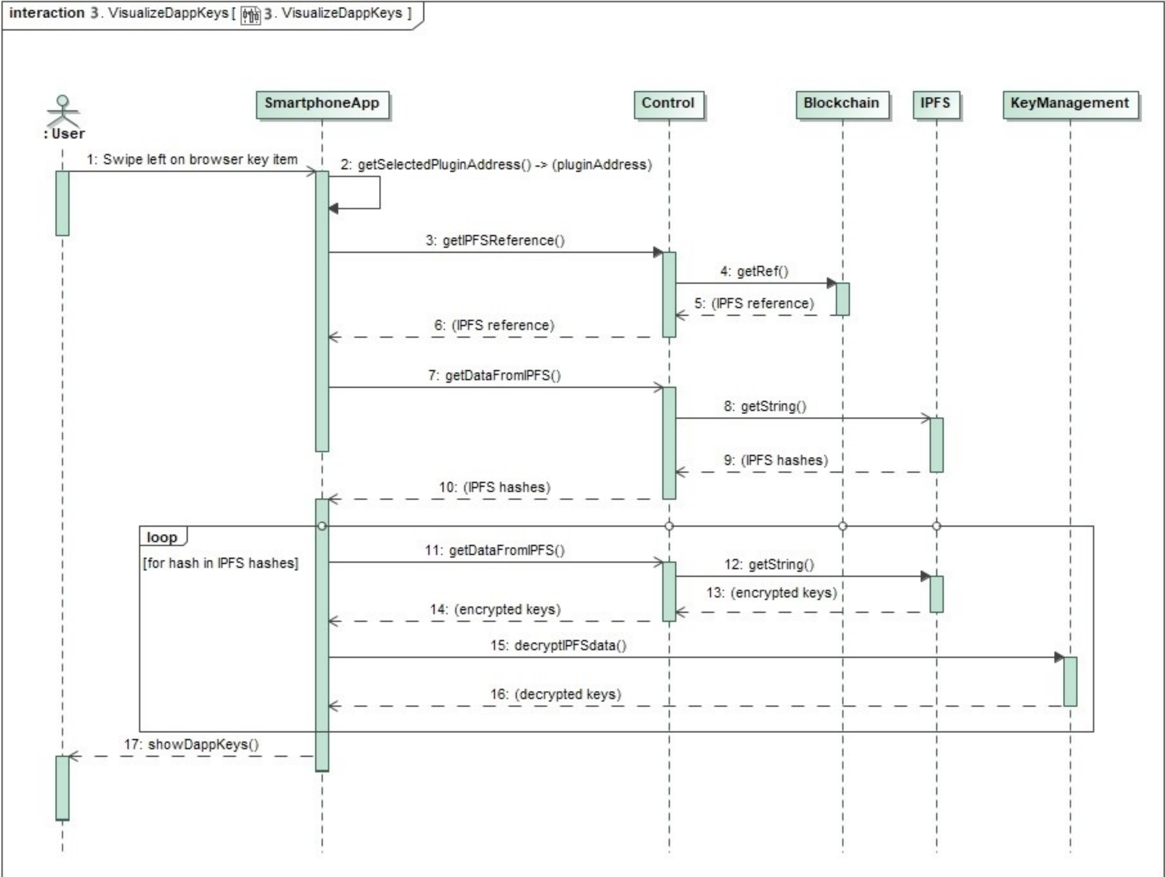


Figure 11: Use case 3. VisualizeDappKeys.

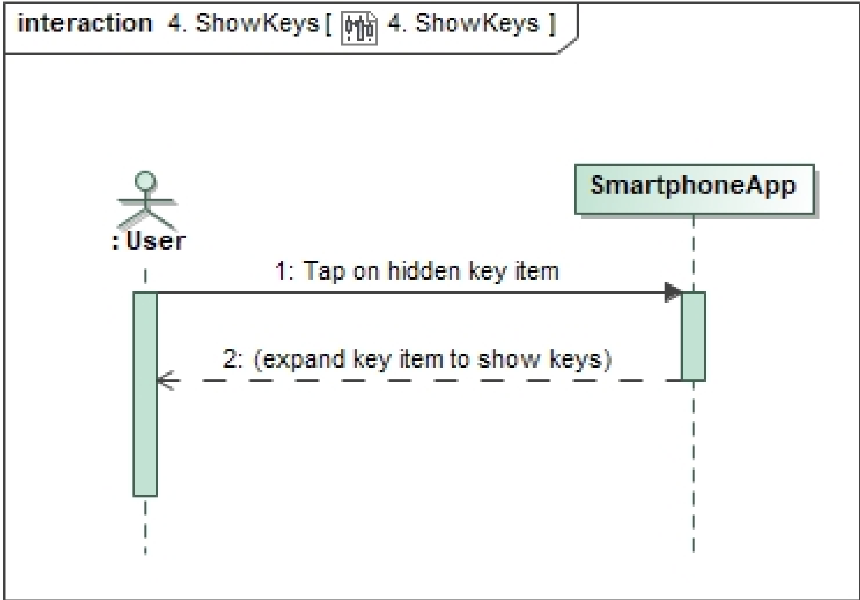


Figure 12: Use case 4. ShowKeys.

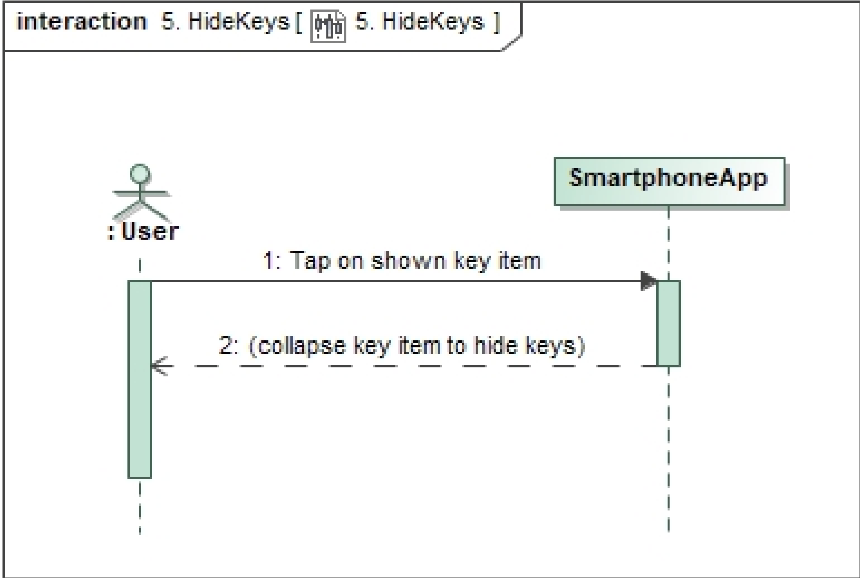


Figure 13: Use case 5. HideKeys.

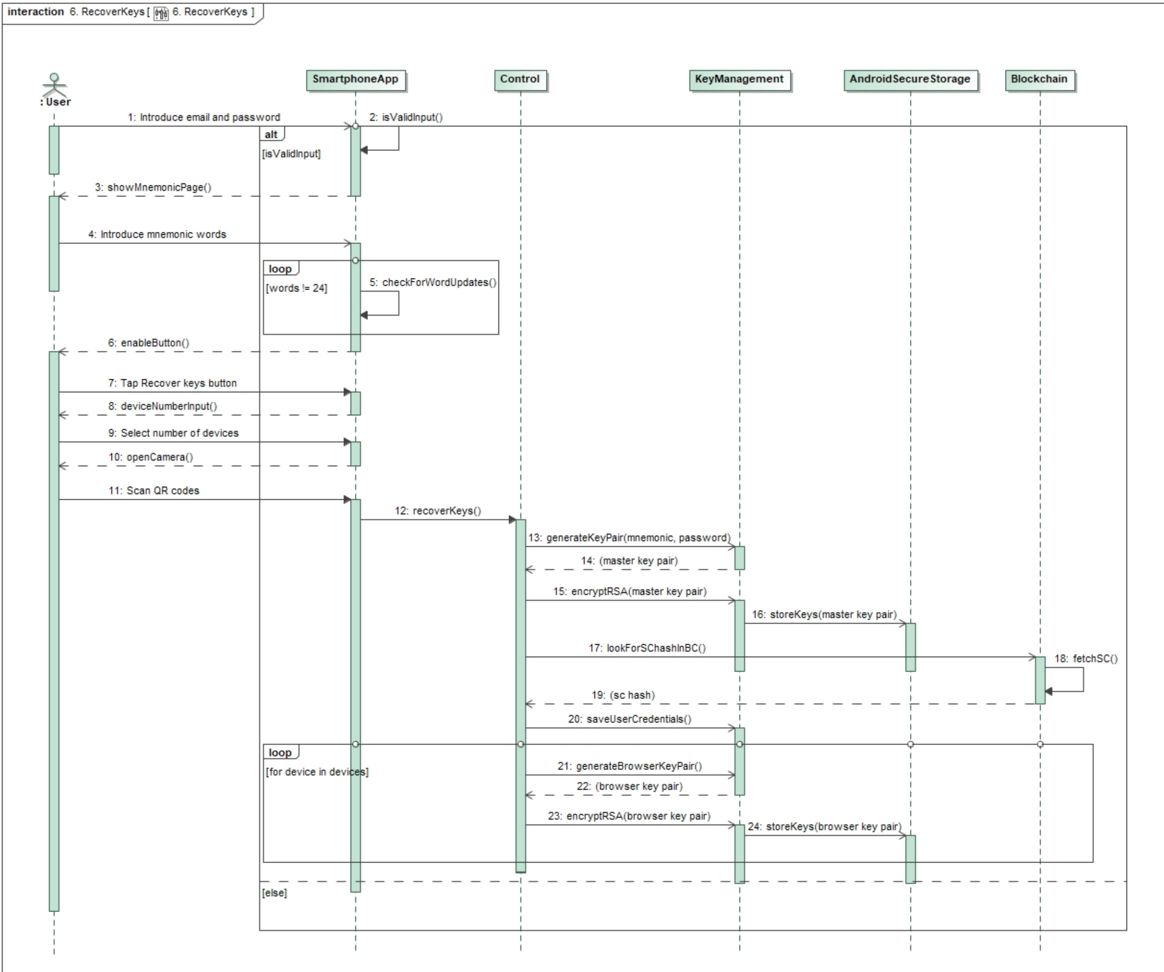


Figure 14: Use case 6. RecoverKeys.

Step	Description
S4.1	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.
S4.2	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. Otherwise, the protocol goes to the first step.
S4.3	The Smartphone App gets the encrypted keys linked to the updated $file_{h_i}$. This is, $EncK_{ij}(SK_{ij}, PK_{ij}, j), EncPK_0(K_{ij})$.
S4.4	The Smartphone App decrypts the session key K_{ij} using the Root Secret Key SK_0. This is, $DecSK_0(EncPK_0(K_{ij}))$.
S4.5	The Smartphone App decrypts the set $EncK_{ij}(SK_{ij}, PK_{ij}, j)$ using the session key K_{ij}. This is, $DecK_{ij}(EncK_{ij}(SK_{ij}, PK_{ij}, j))$. The decrypted set is stored in the Smartphone App's file system.
S4.6	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.

Table 3: "Key Visualization" protocol step.

Step	Description
S5.1	The *User* installs the Smartphone App on a new smartphone device and establishes a new secure login, by setting up her username and password pw.
S5.2	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.
S5.3	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.
S5.4	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. The Smartphone App derives the Root Public Key PK_0 by means of SK_0 as follows: $PK_0 = SK_0 \times G$.
S5.5	The Smartphone App stores SK_0 in the smartphone's secure storage
S5.6	The Smartphone App fetches the SKM SC published on the blockchain using the Root Key pair (SK_0, PK_0) and stores its hash hSC in the app's storage.
S5.7	The Smartphone App follows the procedure explained in Table 3 to retrieve all cryptographic key pairs previously generated.

Table 4: "Key Recovery" protocol step.

7. Change2FAState This use case can be initiated in various scenarios. During the sign-up process, *Users* have the option to activate or deactivate the two-factor authentication (2FA) by selecting or deselecting a checkbox. If the *User* decides to change this preference later, they can do so from the settings page by toggling a switch.

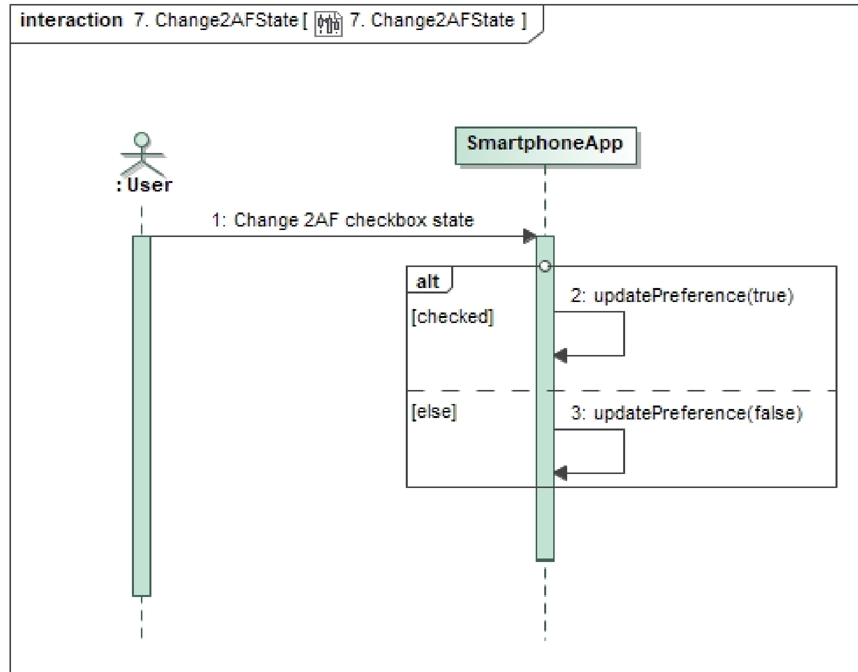


Figure 15: Use case 7. Change2FAState.

When 2FA is enabled, the preference is set to `true`; when it is disabled, the preference is set to `false`. These steps can be seen in Figure 15.

8. AskBiometric As shown in Figure 16, when 2FA is enabled, the system will request biometric verification. If the biometric sensor fails to recognize the *User*, they will be prompted to enter their device PIN or password as an alternative authentication method.

5.3.2 Browser Plugin

Through the browser plugin, the *User* can perform the next actions:

9. SetUpPlugin This step involves the initial configuration of the browser plugin as well as its integration with the smartphone app. The steps for the whole process are explained in Table 5. The sequence is illustrated in Figure 17.

10. RecoverQR If the *User* loses access to the app, they will need to recover all their keys. This process requires scanning a QR code for each device previously added. Unlike the QR code used for adding a device, the recovery QR code contains only the browser identifier. The recovery QR code can be generated from the plugin's settings page. Figure 18 shows the process.

11. NewDapp This process (Figure 19) facilitates the addition of the cryptographic keys of a new decentralized app (dApp) to the system through the plugin. To do so, the plugin performs the steps in Table 6.

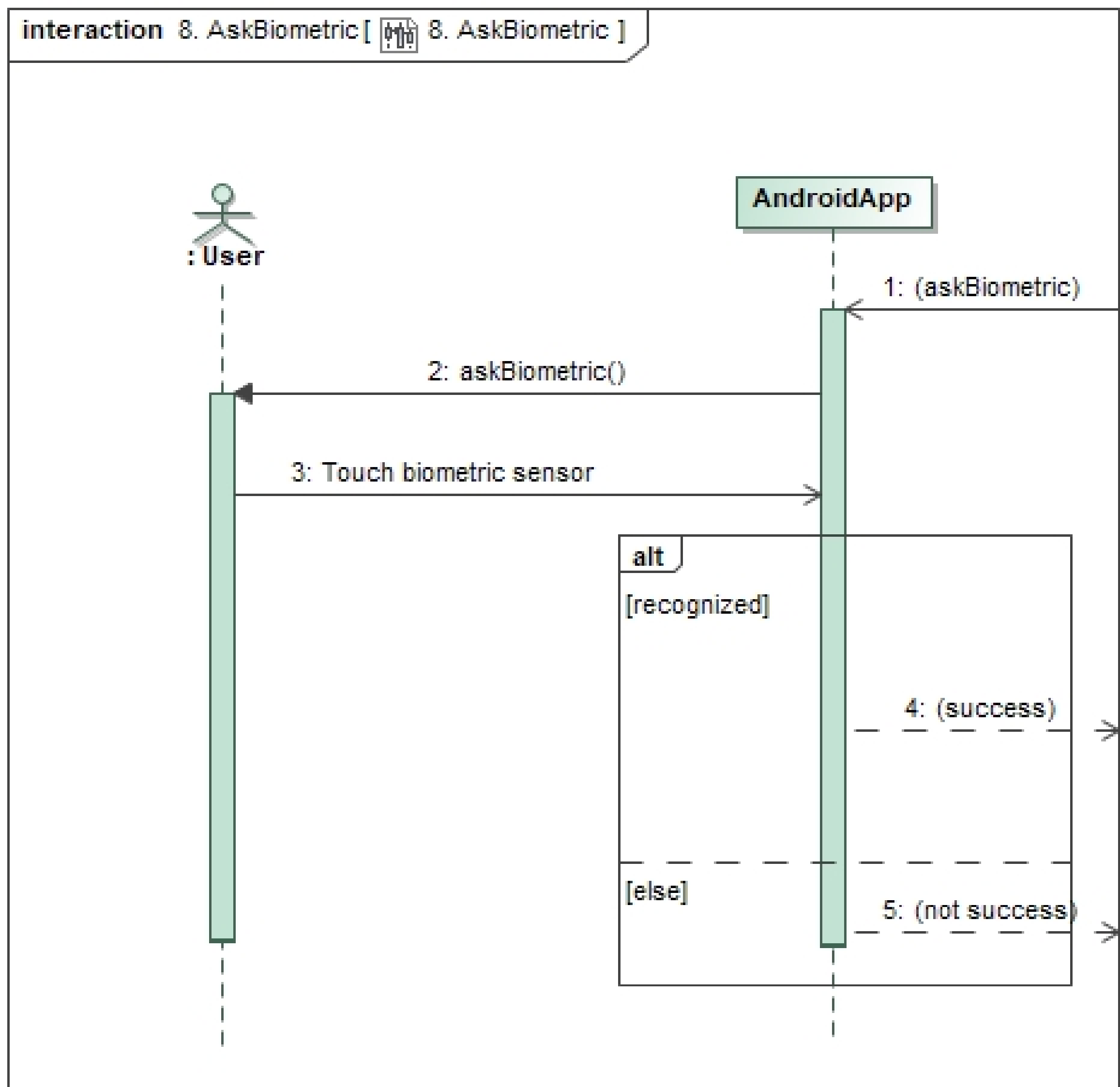


Figure 16: Use case 8. AskBiometric.

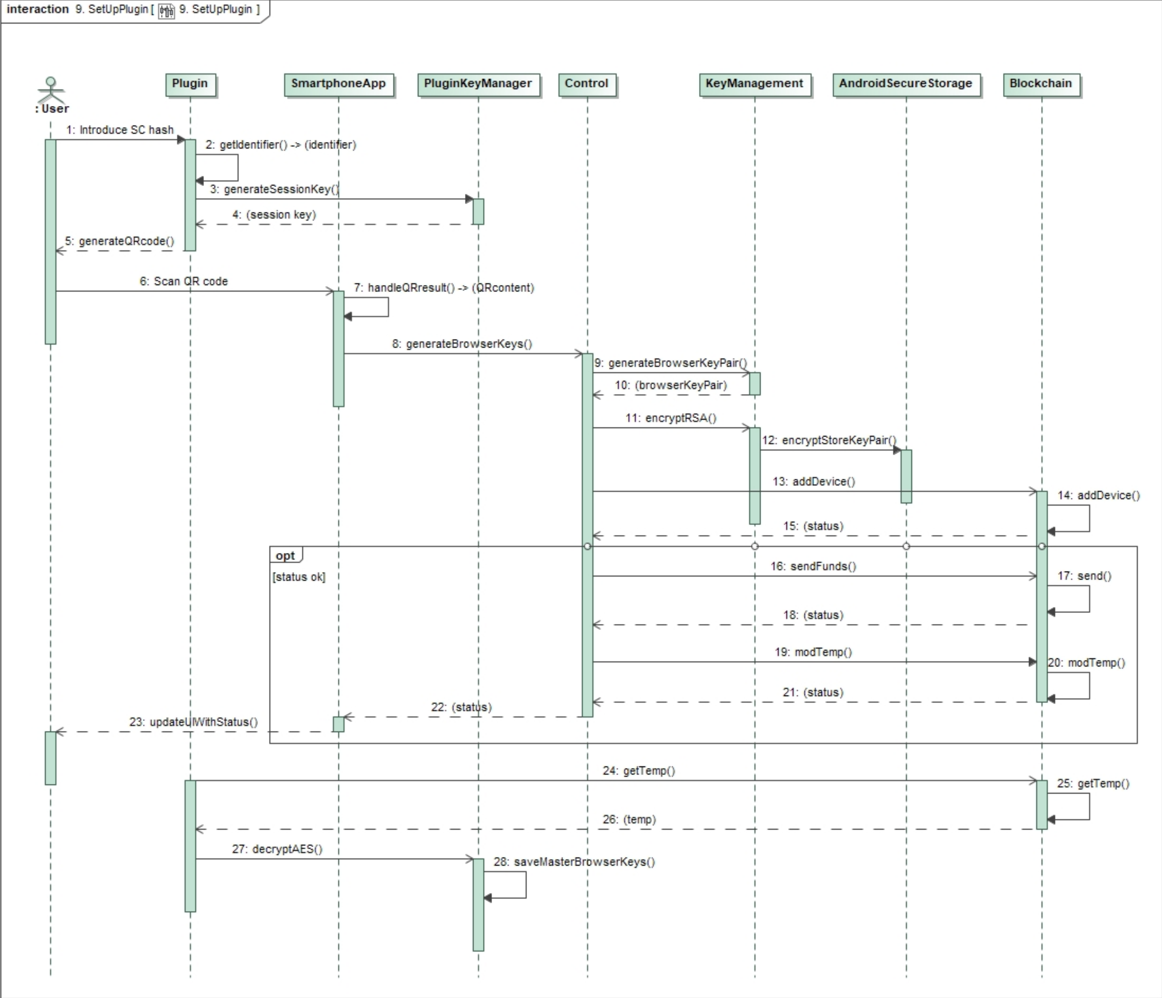


Figure 17: Use case 9. SetUpPlugin.

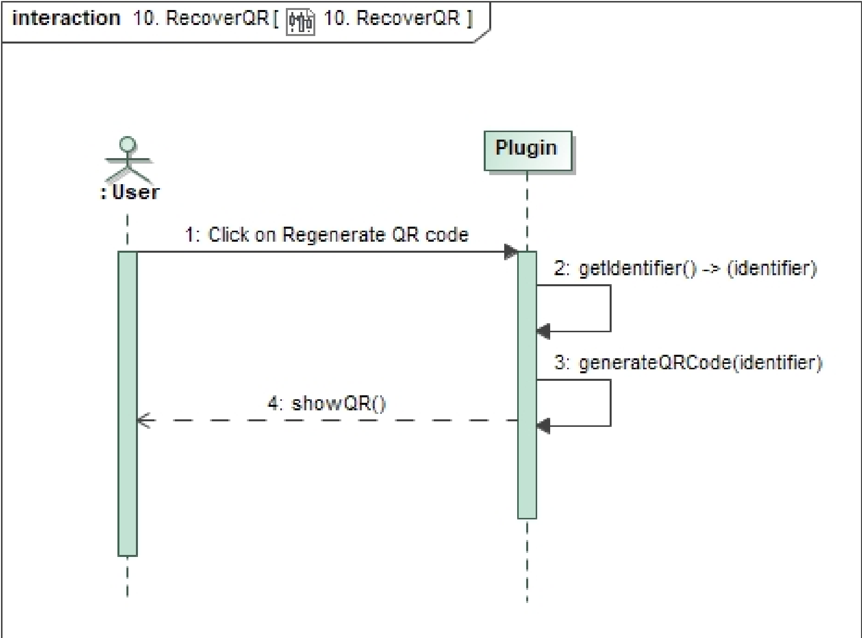


Figure 18: Use case 10. RecoverQR.

Step	Description
S2.1	The *User* installs the provided Plug-in in a browser of her choice.
S2.2	The *User* introduces the *hSC* (i.e., the SKM SC address) into the Plug-in.
S2.3	The Plug-in retrieves PK_0 from the smart contract.
S2.4	The Plug-in uses a secure symmetric cryptosystem (AES) to generate a new session key K_i.
S2.5	The Plug-in generates a QR image containing the computed session key K_i and the browser's identifier i.
S2.6	The Smartphone App scans the QR image shown in the web browser by the Plug-in and retrieves K_i and i.
S2.7	The Smartphone App employs SK_0, C_0, and the hashed browser's identifier, denoted as $SHA_{256}(i)$, as inputs to the BIP32 protocol. This process generates the Master Secret Key SK_i and the Master Chain Code C_i specific to that particular Plug-in. The Smartphone App derives the corresponding Master Public Key PK_i as follows: $PK_i = SK_i \times G$. This element unequivocally identifies that specific Plug-in component.
S2.8	The Smartphone App adds PK_i to the authorized Plug-in components list of the SKM SC (referred to as the *whiteList* argument) using the addDevice() method.
S2.9	The Smartphone App uses the session key K_i to encrypt the set (SK_i, PK_i, C_i). This is, $Enc_{K_i}(SK_i, PK_i, C_i)$.
S2.10	The Smartphone App stores the pair $(Enc_{K_i}(SK_i, PK_i, C_i), i)$ in the temporary data field of the SKM SC (referred to as the *temp* argument) using the modTemp() method.
S2.11	The Plug-in retrieves the pair $(Enc_{K_i}(SK_i, PK_i, C_i), i)$ from the *temp* argument of the SKM SC.
S2.12	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).

Table 5: "Setting up the Browser Plugin" protocol step.

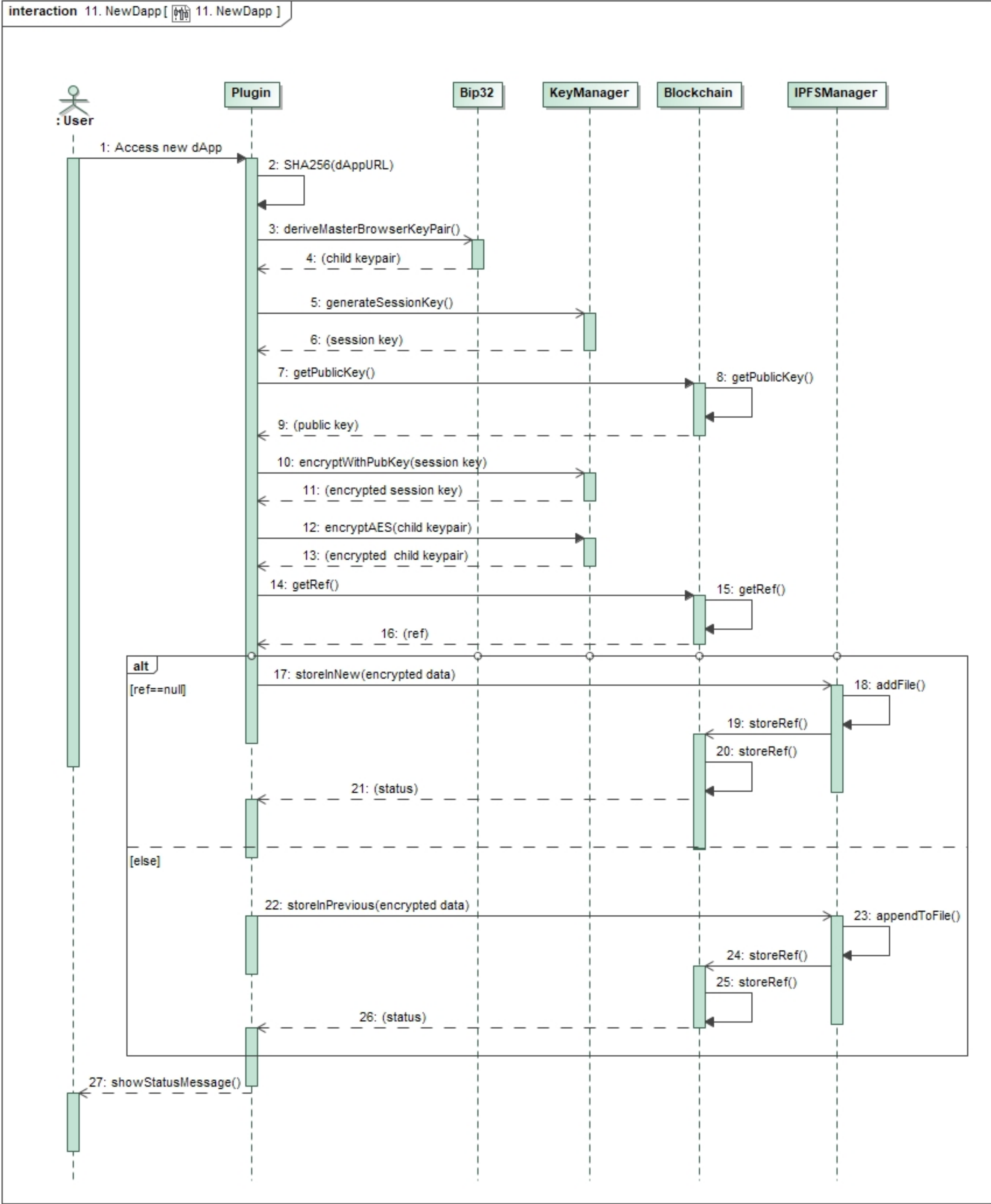


Figure 19: Use case 11. NewDapp.

Step	Description
S3.1	The *User* accesses a new blockchain-based dApp.
S3.2	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 $SHA_{256}(j)$.
S3.3	The Plug-in uses the browser's Master Private Key SK_i, Master Chain Code C_i, and the dApp identifier $SHA_{256}(j)$ 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: $PK_{ij} = SK_{ij} \times G$.
S3.4	The Plug-in uses a secure symmetric cryptosystem (e.g., AES) to generate a new session key K_{ij}.
S3.5	The Plug-in uses the session key K_{ij} to encrypt the set SK_{ij}, PK_{ij}, j. This is, $\text{Enc}K_{ij}(SK_{ij}, PK_{ij}, j)$.
S3.6	The Plug-in uses the Root Public Key PK_0 to encrypt the session key K_{ij}. This is, $\text{Enc}PK_0(K_{ij})$.
S3.7	The Plug-in stores the set $\text{Enc}K_{ij}(SK_{ij}, PK_{ij}, j), \text{Enc}PK_0(K_{ij})$ in the IPFS. This process entails a series of automated sub-steps that facilitate subsequent access to the cryptographic keys by the Smartphone App.
S3.7.1	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.
S3.7.2	The Plug-in gathers the file h_i from the IPFS.
S3.7.3	The Plug-in updates file h_i by appending the set $\text{Enc}K_{ij}(SK_{ij}, PK_{ij}, j), \text{Enc}PK_0(K_{ij})$. This process generates an updated reference h' for the stored file.
S3.7.4	The Plug-in uses the storeRef() method of its associated SKM SC to update the *refsList* mapping with the new reference.

Table 6: "Adding a new dApp" protocol step.

6 Implementation

The system's architecture comprises two main components: an Android Application and a Chrome Browser Plugin. Blockchain and IPFS technologies have been used to fulfill the necessity of building a decentralized system. In the following subsections, a detailed explanation of the implementation of each component will be provided.

6.1 Android App

The development of the Android application was carried out using the Android Studio IDE. Kotlin was chosen as the programming language instead of Java, as previously discussed in Section 3.4. The code can be accessed in `https://github.com/imiguelrodriguez/TFGwallet`.

The application was built following the MVC (Model-View-Controller) architecture. It is a common-spread paradigm in application development due to its numerous benefits. In this architecture, the Model represents the back-end where all the logic is implemented. The View handles the front-end or graphical user interface (GUI). Finally, the Controller acts as an intermediary, facilitating communication between the Model and the View. Employing the MVC paradigm enhances the application's scalability and maintainability. This concept was first introduced in the late 1970s by Trygve Reenskaug. The main advantage of this architecture is that the same logic (Model) can be displayed in multiple ways (such as mobile, desktop, or web applications) without the need to reimplement the entire back-end [39].

All the screens within the app are defined as activities. An Activity[13] is the basic class that represents a single screen with a user interface in the app. To navigate from one Activity to another, an Intent[14] is created and passed as an argument to the `startActivity()` function. Additionally, data can be transferred to the new Activity by using the `putExtra("key", value)` method. This project uses this mechanism to pass information inputted by the user on one screen to the next, for example when entering credentials in the key recovery section.

The `MainActivity` that defines the main app screen uses Android Fragments[15] (lighter versions of an Activity) to provide all the available screens shown in the lower navigation bar.

The graphical interface is defined using XML files. Traditionally, Android developers accessed UI components using the `R` resource ID system. This approach involves calling `findViewById()` to obtain references to the views in the layout, as shown in Code 2.

[13]**Activities:** `https://developer.android.com/guide/components/activities/intro-activities`

[14]**Intents:** `https://developer.android.com/reference/android/content/Intent`

[15]**Fragments:** `https://developer.android.com/guide/fragments`

Code 2 Traditional approach using resource IDs.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState
    setContentView(R.layout.my_layout)
}

val textView: TextView = findViewById(R.id.my_text_view)
textView.text = "Hello, World!"
```

However, View Binding[16] simplifies this process by generating a binding class for each XML layout file, providing direct references to the graphical elements without the need to call `findViewById()`, as shown in Code 3. The project adopts this alternative as it enhances code safety by removing null references, and also reduces boilerplate code.

Code 3 Using Binding to access GUI elements.

```
private lateinit var binding: ActivityMainBinding

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    val view = binding.root
    setContentView(view)
}

binding.myTextView.text = "Hello, World!"
```

6.1.1 QR scanner

To enable QR scanning functionality, the app includes the `zxing-android-embedded` library[17]. Camera access is required to scan QR codes, so the app requests this permission the first time it needs to use the camera. The `QRCodeScannerActivity` is utilized in two different scenarios:

1. **Adding a New Device**: In this case (identified with the `isBasicFunctionality` flag), the QR code is expected to contain the session key and browser ID. Once scanned, the application generates keys for the new plugin, deriving them from the device's master key pair.

2. **Key Recovery**: Here, the QR code only includes the browser ID. The process is slightly different as it focuses on recovering the existing keys rather than generating new ones.

[16]**View Binding:** https://developer.android.com/topic/libraries/view-binding
[17]**Zxing's GitHub:** https://github.com/journeyapps/zxing-android-embedded

6.1.2 Android Secure Storage

To ensure the secure storage of all cryptographic keys, the Android KeyStore module is employed. This module supports the storage of keys generated within its environment but does not directly support the storage of BIP32 keys created by the app. To work around this, an RSA key is generated to encrypt the BIP32 keys (Code 4). The RSA key itself is securely stored in the Android KeyStore, while the encrypted BIP32 keys are stored in a binary file within the app's private directory. Further details on the encryption and decryption process can be found in Section 6.2.

Code 4 Generating and storing the RSA key in the Android KeyStore.

```
1  private fun generateRSAkey(userId: String): KeyPair {
2          val keyPairGenerator =
3              KeyPairGenerator.getInstance(KeyProperties.KEY_ALGORITHM_RSA,
                ↪  "AndroidKeyStore")
4          keyPairGenerator.initialize(
5              KeyGenParameterSpec.Builder(
6                  "user_rsa_key_$userId",
7                  KeyProperties.PURPOSE_ENCRYPT or
                    ↪  KeyProperties.PURPOSE_DECRYPT
8              )
9                  .setKeySize(2048).
                    ↪  setEncryptionPaddings(KeyProperties.ENCRYPTION_PADDING_RSA_OAEP)
10                 .build()
11         )
12         return keyPairGenerator.generateKeyPair()
13     }
```

6.1.3 Network operations

Since Android restricts network operations on the main thread to avoid impacting user experience, Kotlin coroutines [18], which function similarly to threads, are utilized instead. When prolonged interactions are expected, a loading alert is shown. This alert has been implemented to provide feedback during operations that are not instantaneous. The message displayed in the dialog can be customized at the time of its creation. Some examples of this dialogue can be found in Section 7.

6.1.4 Shared preferences

The Android SharedPreferences API[19] is used to store key-value pairs. In this project, it serves several purposes:

- **Smart Contract Address Storage**: Stores the smart contract address for each registered user, enabling easy retrieval when interacting with the blockchain or when showing the master keys.

[18]**Kotlin Coroutines:** https://developer.android.com/kotlin/coroutines

[19]**Shared Preferences:** https://developer.android.com/training/data-storage/shared-preferences

- **Last Logged-In User**: Saves the email of the last logged-in user, so it is automatically filled in when the app is reopened, smoothing the login process.
- **Session Management**: Records a sign-in timestamp to implement a session timeout. The session expires after 15 minutes of inactivity, enhancing security.
- **Two-Factor Authentication State (2FA)**: Stores the status of the 2FA for each user. If enabled, the biometric prompt will be displayed; otherwise, the user is taken directly to the home page after login validation.

6.1.5 Showing keys

An Android RecyclerView[20] is utilized with a custom adapter to display cryptographic keys. Specifically, a `KeyItemAdapter` serves as the base adapter, and two specialized adapters are derived from it: `MasterKeyItemAdapter` for showing master keys and `BrowserKeyItemAdapter` for displaying browser and dApp keys. These adapters handle data classes like `KeyItem` and `MasterKeyItem`, with `MasterKeyItem` inheriting from `KeyItem` by composition.

The RecyclerView is populated differently based on the type of keys:

- **Master Keys:** Although represented as a dynamic list, this list is only populated once by retrieving the master keys encrypted in a binary file. The smart contract address is obtained from SharedPreferences, as detailed in Section 6.1.4.
- **Browser Keys:** Similarly, the list of keys associated with registered devices is dynamically populated.
- **dApp Keys:** For accessing keys used by dApps within a particular plugin, a swipe layout is implemented using a RecyclerView decorator[21]. This allows users to swipe left on a list item to reveal the corresponding dApp keys. The dApp keys are retrieved from the IPFS every time an item is swiped.

To enhance security, all list items are collapsed by default. Users can tap on any item to expand it and view the associated keys.

6.1.6 Biometric access

To enhance security, two-factor authentication can be enabled at any time. The app utilizes the Android Biometric Prompt to request user biometric data (such as fingerprint or facial recognition). If the biometric sensor fails to recognize the user—perhaps due to issues like a dirty sensor—the app provides an alternative by allowing PIN input. The implementation of this feature adheres to the official guidelines provided for Android developers[22].

6.2 Cryptography

This project involves the generation and usage of both symmetric and asymmetric cryptographic keys to ensure secure communication and data protection.

[20] **RecyclerView:** https://developer.android.com/develop/ui/views/layout/recyclerview
[21] **SwipeDecorator's GitHub:** https://github.com/xabaras/RecyclerViewSwipeDecorator
[22] **Biometric Authentication:** https://developer.android.com/identity/sign-in/biometric-auth

6.2.1 Symmetric keys

Symmetric keys are also referred to as session keys in this document. Within the project, 256-bit size symmetric keys are generated using the AES (Advanced Encryption Standard) algorithm in GCM (Galois/Counter Mode). The GCM mode offers several advantages:

- **Confidentiality and integrity**: The GCM mode not only provides confidentiality by encrypting the data but also ensures its integrity by generating an authentication tag. This tag is used to make sure the data has not been modified.
- **Parallelism**: Different blocks of data can be encrypted simultaneously, therefore enhancing performance.

An Initialization Vector (IV) is essential for AES encryption and decryption. More specifically, in GCM mode, the recommended length for the IV is 12 bytes. In this implementation, the IV is generated pseudo-randomly. When encrypting, both the app and the plugin prepend the IV to the encrypted data. Since the IV can be safely shared without compromising security, the receiving part can then extract it and use it to decrypt the data.

6.2.2 Asymmetric keys

Asymmetric cryptography relies on a pair of keys: a public key and a private key. The private key is kept secret, while the public key is distributed openly. Common algorithms used for asymmetric key generation in this project include RSA (Rivest-Shamir-Adleman) and ECDSA (Elliptic Curve Digital Signature Algorithm). Modules like android.security, java.security, javax.crypto, org.bitcoinj.crypto, org.web3.crypto, and org.bouncycastle.jce have been used in the smartphone application.

6.2.3 Encryption and decryption processes

The RSA algorithm with padding version PKCS2.2 (OAEP) [40] is utilized to encrypt the BIP32-generated master and browser keys, as introduced in Section 6.1.2. Figure 20, illustrates this process. During encryption, the byte lengths of the private and public keys are first placed in the initial two positions of a byte array. Following this, the byte array is populated with the concatenated bytes of the private key, public key, and chain code. The byte array is then encrypted with a unique RSA key securely stored in the Android KeyStore, with the RSA key being assigned to each user for the master key pair or to a specific plugin within the user's account for the browser key pairs. Once encrypted, the data is saved as a binary file within the app's private directory. Decryption follows the reverse procedure.

The process for retrieving dApp key pairs differs from that of master and browser keys. As detailed in Table 6 and illustrated in Figure 21, the steps involved are next explained. The ECIES Hybrid Encryption Scheme [23] is used by the plugin when calling the `encryptWithPubKey()` function, which performs the following steps:

[23] **ECIES**: `https://cryptobook.nakov.com/asymmetric-key-ciphers/ecies-public-key-encryption`

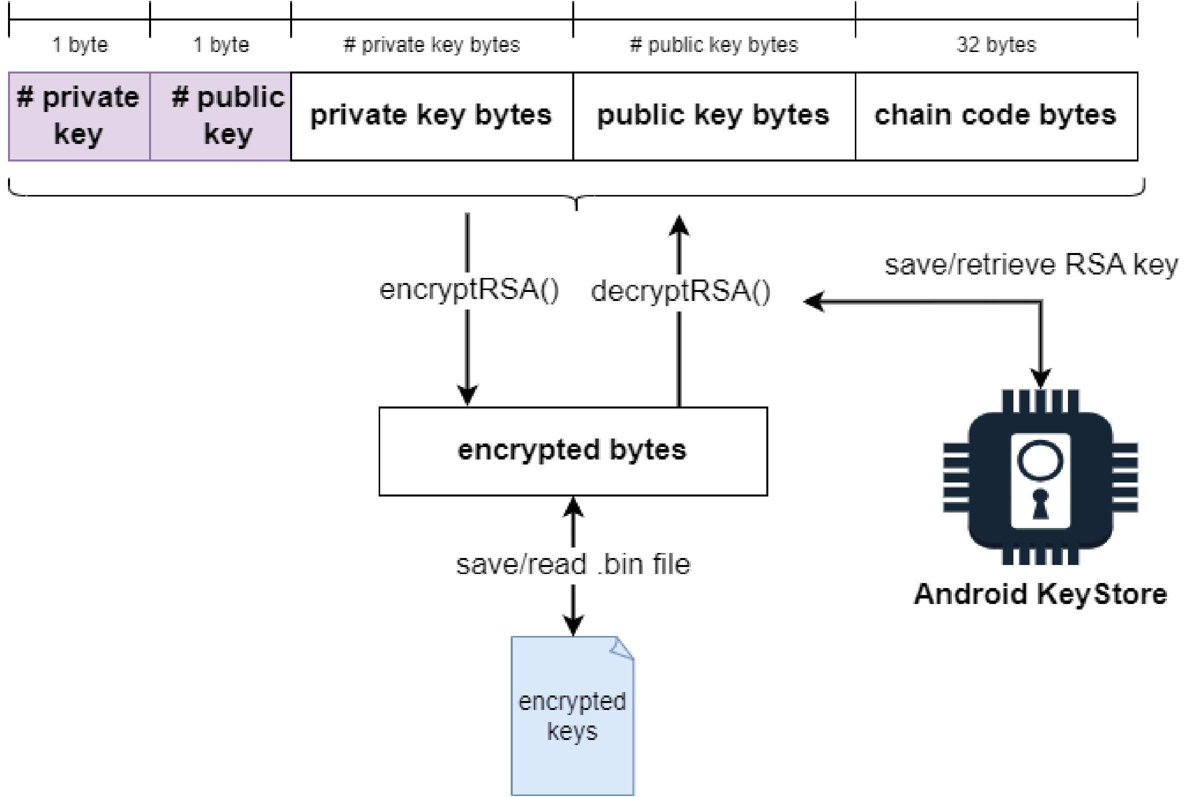


Figure 20: RSA encryption/decryption for master and browser keys.

1. **Public Key Conversion**: Convert the provided public key PK_0 into an elliptic curve key object, represented as $y_A = g^\alpha \mod p$, where g is the curve generator point. α is the smartphone's private key SK_0, and p is the curve's prime number.

2. **Generating Exponent** r: Generate a large exponent r such that $1 \leq r \leq p-1$.

3. **Session Key Calculation**: Compute the session key (distinct from K_{ij}) using $y_A^r \mod p = (g^\alpha)^r \mod p = x$. Concatenate the X and Y coordinates of the resulting elliptic curve point to form x. Hash this value to obtain the session key as $k = \text{SHA}_{256}(x)$.

4. **Message Encryption**: Encrypt the message with the AES session key k, producing the first cryptogram c_1.

5. **Generating the Second Cryptogram**: Create the second cryptogram c_2 using $g^r \mod p$. This cryptogram allows the app to retrieve the exponent r and decrypt c_1 to obtain the dApp session key K_{ij}.

6. **Concatenating Data**: Concatenate c_1, c_2, and the length of c_1 into a byte array.

This process results in $Enc_{PK_0}(K_{ij})$. The BIP32 new dApp key pair and the dApp ID (*dAppData*) are then encrypted with K_{ij} using the `encryptWithAES()` function, producing $Enc_{K_{ij}}(SK_{ij}, PK_{ij}, c_{ij}, j)$. All the encrypted data is stored on IPFS.

To retrieve the dApp key pair, the app first gets the IPFS file and calls the `decryptWithPrivKey()` function, which follows these steps:

1. **Deriving the Session Key from** c_2: Calculate the session key k by multiplying the elliptic curve point derived from c_2 with the smartphone's private key (SK_0), yielding $(g^r)^\alpha \mod p = x$.

2. **Generating the AES Session Key**: Perform a SHA-256 hash on x to get the AES session key $k = \text{SHA}_{256}(x)$.

3. **Decrypting the First Cryptogram** c_1: Use the AES session key k to decrypt c_1. This involves extracting the IV (the first 12 bytes) from the encrypted data and using the AES session key k to recover the original dApp AES session key K_{ij}.

Finally, the app uses K_{ij} to decrypt $Enc_{K_{ij}}(SK_{ij}, PK_{ij}, c_{ij}, j)$ and obtain the $dAppData$.

6.3 Blockchain

The Sepolia Network, Ethereum's testnet, was used as the blockchain for this project. This test network closely mirrors the Ethereum mainnet, enabling the system to operate in an environment that closely resembles real-world conditions. Additionally, alternatives like Ganache were not suitable for programmatically creating new accounts using self-generated key pairs, such as those derived from the BIP39 and BIP32 protocols, as stated in Section 3.1.1.

To interact with the testnet, web3 libraries tailored to the respective programming languages were used: web3j for the Android app and web3js for the Chrome plugin. These libraries provide the essential APIs for interacting with the specified blockchain.

Initialization of the library is done as shown in Code 5 and Code 6.

Code 5 Web3 library initialization in JavaScript.

```
1 const web3 = new Web3(BLOCKCHAIN-URL)
```

Code 6 Web3 library initialization in Kotlin.

```
1 var web3 = Web3j.build(HttpService(BLOCKCHAIN-URL))
```

Once the instance is initialized, methods to send funds, check account details, or interact with smart contracts can be called. To facilitate the deployment and interaction with smart contracts using the web3j library, a Java wrapper can be generated, as further explained in Section 6.4.

In the app, before any blockchain method is called, the initialization state of the `web3` variable is checked. If it hasn't been initialized yet, the class automatically invokes a `connect()` method to initialize it. This approach relieves the programmer

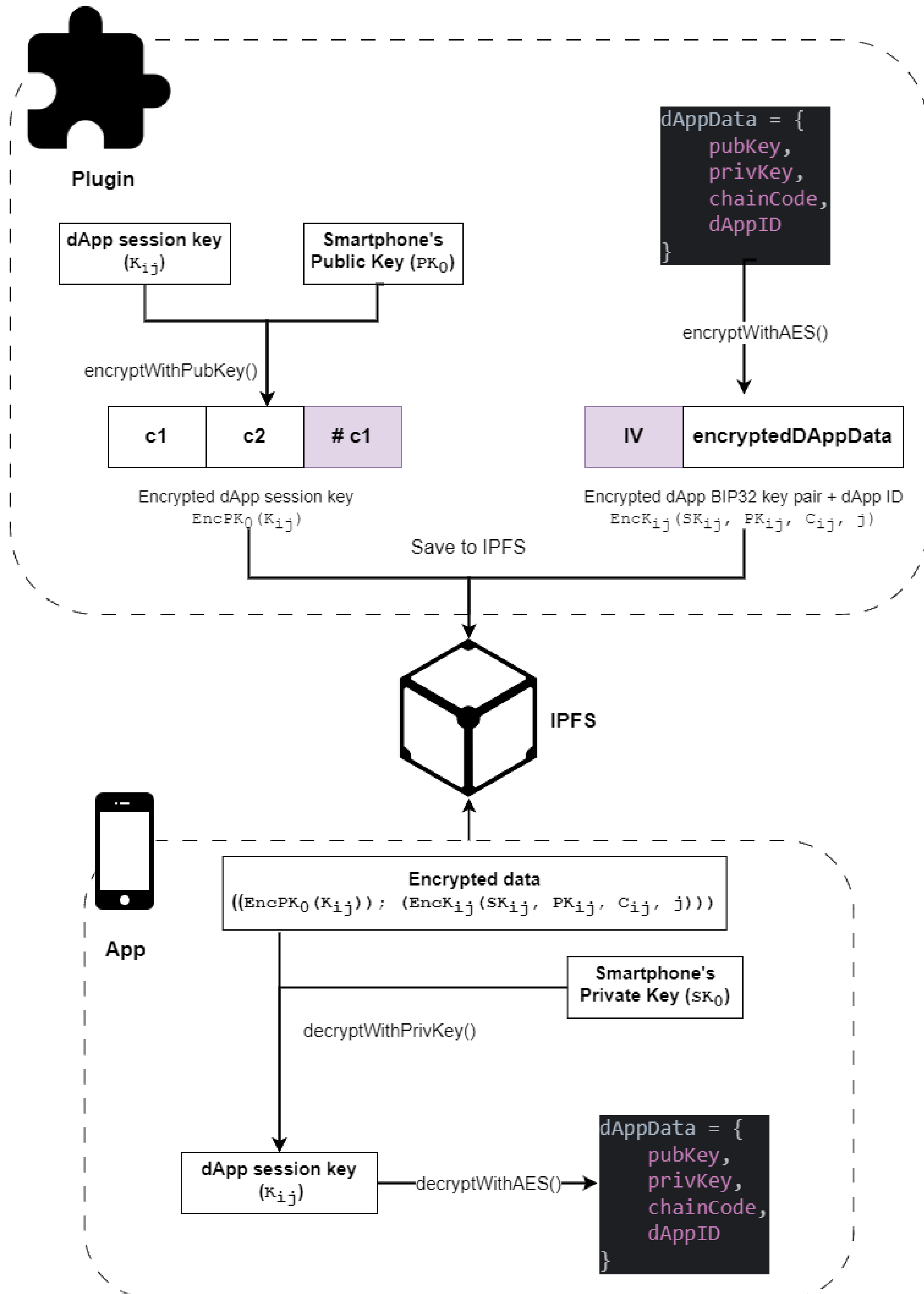


Figure 21: DApp key generation and retrieval. Interaction between the app and the plugin via IPFS.

from having to manually call the `connect()` method before other operations, as the initialization is transparently handled within the class.

During key recovery, the initial smart contract address needs to be retrieved from the blockchain. To accomplish this, the Etherscan API[24] is utilized. Transactions involving contract deployment are identifiable by an empty "to" field, meaning no recipient address is specified. The app examines the deploying account's transaction history until it locates a transaction with no receiver. From this transaction, the contract address can be extracted.

Similarly, the initialization state of the smart contract object is handled internally in both the app and the plugin, with the `initContract()` method being called automatically before executing any smart contract method.

To minimize waiting times for blockchain transactions, the gas cost for each transaction is estimated using the `ethEstimateGas()` method provided by the web3 libraries. The gas price is dynamically updated based on the network status through the Beaconcha.in API[25]. This API provides real-time gas prices depending on the current pending transactions in the execution pool, offering different values for various transaction speeds (rapid, fast, standard, slow). In this project, the gas price corresponding to the rapid transaction speed is consistently used to provide a more streamlined user experience.

6.4 Smart Contract

The SKM smart contract, used for smooth communication between the app and the plugin and that ensures secure key storage, was developed using the Solidity programming language. The code can be found in Appendix A.

To compile the code and generate its ABI (Application Binary Interface), the Truffle Suite, as mentioned in Section 6.3, or tools like Remix IDE can be utilized. The ABI is crucial for interacting with the smart contract through the web3js library, as no wrappers can be generated. Instead, a contract instance is initialized using the ABI in JSON format.

To interact with the contract in Kotlin, a Java wrapper[26] can be generated through the command-line tools provided by the developers of the library. The next steps must be followed to achieve this:

1. Write the smart contract in solidity.
2. Compile the smart contract to obtain its JSON format. `truffle compile`
3. Generate Java wrapper based on the JSON file. `web3j generate truffle --truffle-json /path/to/<smart-contract>.json --outputDir /path/to/src/main/java -p com.your.organisation.name`

These instructions assume that the Truffle library is installed. Alternatively, the smart contract's binary can be obtained, and the Java wrapper generated from it.

[24] **Etherscan's API:** https://docs.etherscan.io/api-endpoints/accounts
[25] **Beaconcha.in's API:** https://sepolia.beaconcha.in/api/v1/execution/gasnow
[26] **Java wrapper for smart contracts:** https://docs.web3j.io/4.11.0/getting_started/deploy_interact_smart_contracts/

6.5 Chrome Plugin

The browser plugin has been specifically developed for Google Chrome, utilizing technologies such as HTML, CSS, and JavaScript. Google Chrome was selected as the target platform due to its dominance, accounting for over 65% of global browser usage in the past year[27]. To enhance the plugin's design and layout, Bootstrap [28] was integrated, complemented by custom CSS to achieve a fully personalized appearance. The source code is available at `https://github.com/imiguelrodriguez/TFGwalletplugin`.

Typically, plugins are displayed as popups when opened. However, if the user clicks outside the popup, the current page's state is lost. To avoid this issue, this plugin opens in a new browser tab, functioning like a standard webpage. This behavior is handled by the `openTab.js` file, which utilizes the browser API to open the plugin's main page in a new tab.

6.5.1 Minified packages

Chrome imposes restrictions on the use of Content Delivery Networks (CDNs) within plugins. CDNs allow external libraries to be loaded via a URL, eliminating the need to include them directly in the project. Due to this limitation, all external libraries used in the plugin have been incorporated in their minified versions. A minified version is a compressed and optimized format of a library, typically stored in a single JavaScript file, which reduces file size and improves loading times. Specifically, the included libraries are web3[29], qrcode[30], elliptic[31], and bootstrap[32].

6.5.2 QR generator

QR codes are generated using the `qrcode.js` library, already mentioned in Section 6.5.1. The content of the QR code is provided as a hexadecimal string and varies based on the specific functionality required:

- **Setting up the Plugin**: For plugin setup, the QR code includes an AES session key and the browser identifier.
- **Device Recovery**: In the case of device recovery, the QR code contains only the browser identifier.

6.5.3 Background script

A background script, also known as a service worker, has been developed to handle functionalities that cannot be executed within the main script. For example, tasks like fetching content from a file must be managed by the service worker. Communication between the main script and the service worker occurs through message passing, allowing the retrieval of the IP address where the IPFS is running. This IP address is written to a file, thereby avoiding the need to hard-code it into the source code and facilitating future configuration changes.

[27] **Browsers' usage:** `https://gs.statcounter.com/browser-market-share`
[28] **Bootstrap:** `https://getbootstrap.com/`
[29] **Web3js library:** `https://web3js.readthedocs.io/en/v1.10.0/`
[30] **Qrcodejs library:** `https://davidshimjs.github.io/qrcodejs/`
[31] **Ellipticjs library:** `https://github.com/indutny/elliptic`
[32] **Bootstrap library:** `https://getbootstrap.com/docs/4.0/getting-started/download/`

6.5.4 BIP32

In contrast to the Android app, which utilized existing libraries for BIP32, the browser plugin implements these protocols using an object-oriented approach to enhance modularity and maintainability. Specifically, the BIP32 protocol was custom-developed for the plugin, since a suitable BIP32 library was only available for Node.js and not for the plugin environment.

The BIP32 class is responsible for generating key pairs for new dApps. It uses the plugin's root key pair to derive child key pairs, which are then assigned to each dApp as it is accessed. This implementation follows the specifications outlined in [31]. The secp256k1 elliptic curve's base point (G) is used for point multiplication, where the public key is derived by multiplying the private key by the base point G.

6.5.5 Utility functions

Some utility functions such as conversion between array buffers and hexadecimal strings, or communication between the main script and the service worker have been implemented in the `utils.js` file.

6.6 IPFS

Currently, running an IPFS node directly on an Android device is not a straightforward task, and implementing such a solution is beyond the scope of this project. Instead, we implemented a gateway to an IPFS daemon running on a computer within the same LAN as the smartphone hosting the app. This computer works as the IPFS node, enabling both the app and the plugin to interact with the decentralized network for file retrieval and upload.

To interact with IPFS, different approaches have been used for the app and the plugin:

- **App**: The `ipfs-api-kotlin` library[33] is used. This library offers an API for connecting the app to IPFS, allowing operations such as uploading or retrieving files via methods like `add()` and `get()`.
- **Plugin**: The RPC API[34] provided by the IPFS node is utilized. HTTP requests are used to invoke various RPC methods to add or retrieve content from files.

6.6.1 Setting up IPFS

This section explains the steps required to set up the IPFS daemon. Detailed instructions will ensure a smooth configuration process, enabling seamless interaction with the decentralized storage network. First, the IPFS daemon must be installed following the steps provided in the official webpage [35]. Then, the PC's IP must be set in the configuration file (Figure 22).

[33] **IPFS API for Kotlin:** https://github.com/komputing/ipfs-api-kotlin
[34] **IPFS' RPC API:** https://docs.ipfs.tech/reference/kubo/rpc/
[35] **Install IPFS:** https://docs.ipfs.tech/install/command-line/#install-official-binary-distributions

```
config
Archivo Editar Ver

{
  "API": {
    "HTTPHeaders": {
      "Access-Control-Allow-Methods": [
        "GET",
        "PUT",
        "POST"
      ],
      "Access-Control-Allow-Origin": [
        "*"
      ]
    }
  },
  "Addresses": {
    "API": "/ip4/192.168.1.60/tcp/5001",
    "Announce": [],
    "AppendAnnounce": [],
    "Gateway": "/ip4/192.168.1.60/tcp/8080",
    "NoAnnounce": [],
    "Swarm": [
      "/ip4/192.168.1.60/tcp/4001",
      "/ip6/::/tcp/4001",
      "/ip4/192.168.1.60/udp/4001/quic-v1",
      "/ip4/192.168.1.60/udp/4001/quic-v1/webtransport",
      "/ip6/::/udp/4001/quic-v1",
      "/ip6/::/udp/4001/quic-v1/webtransport"
    ]
  },
  "AutoNAT": {},
  "Bootstrap": [
    "/dnsaddr/bootstrap.libp2p.io/p2p/QmNnooDu7bfjPFoTZYxMNLWUQJyrVwtbZg5gBMjTezGAJN",
    "/dnsaddr/bootstrap.libp2p.io/p2p/QmQCU2EcMqAqQPR2i9bChDtGNJchTbq5TbXJJ16u19uLTa",
    "/dnsaddr/bootstrap.libp2p.io/p2p/QmbLHAnMoJPWSCR5Zhtx6BHJX9KiKNN6tpvbUcqanj75Nb",
    "/dnsaddr/bootstrap.libp2p.io/p2p/QmcZf59bWwK5XFi76CZX8cbJ4BhTzzA3gU1ZjYZcYW3dwt",
    "/ip4/104.131.131.82/tcp/4001/p2p/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ",
    "/ip4/104.131.131.82/udp/4001/quic-v1/p2p/QmaCpDMGvV2BGHeYERUEnRQAwe3N8SzbUtfsmvsqQLuvuJ"
  ],

Ln 24, Col 25 | 3.771 caracteres. | 100% | Unix (LF) | UTF-8
```

Figure 22: IPFS daemon configuration file.

After that, the IPFS daemon can be run in the command line by typing `ipfs daemon`, as shown in Figure 23.

```
PS C:\Users\User> ipfs daemon
Initializing daemon...
Kubo version: 0.29.0
Repo version: 15
System version: amd64/windows
Golang version: go1.22.4
Swarm listening on /ip4/192.168.1.60/tcp/4001
Swarm listening on /ip4/192.168.1.60/udp/4001/quic-v1
Swarm listening on /ip4/192.168.1.60/udp/4001/quic-v1/webtransport/certhash/uEiCAC95fJ-aKsl7At9aibTc-9AziDZBjAWxHb-
xGVSYCZw/certhash/uEiBz02zvCVW0g1VUu0DZ3FB0YB385sl8hrcsTEaLSgM93g
Swarm listening on /ip6/::1/tcp/4001
Swarm listening on /ip6/::1/udp/4001/quic-v1
Swarm listening on /ip6/::1/udp/4001/quic-v1/webtransport/certhash/uEiCAC95fJ-aKsl7At9aibTc-9AziDZBjAWxHb-xGVSYCZw/
certhash/uEiBz02zvCVW0g1VUu0DZ3FB0YB385sl8hrcsTEaLSgM93g
Swarm listening on /p2p-circuit
Swarm announcing /ip4/192.168.1.60/tcp/4001
Swarm announcing /ip4/192.168.1.60/udp/4001/quic-v1
Swarm announcing /ip4/192.168.1.60/udp/4001/quic-v1/webtransport/certhash/uEiCAC95fJ-aKsl7At9aibTc-9AziDZBjAWxHb-xG
VSYCZw/certhash/uEiBz02zvCVW0g1VUu0DZ3FB0YB385sl8hrcsTEaLSgM93g
Swarm announcing /ip6/::1/tcp/4001
Swarm announcing /ip6/::1/udp/4001/quic-v1
Swarm announcing /ip6/::1/udp/4001/quic-v1/webtransport/certhash/uEiCAC95fJ-aKsl7At9aibTc-9AziDZBjAWxHb-xGVSYCZw/ce
rthash/uEiBz02zvCVW0g1VUu0DZ3FB0YB385sl8hrcsTEaLSgM93g
RPC API server listening on /ip4/192.168.1.60/tcp/5001
WebUI: http://192.168.1.60:5001/webui
Gateway server listening on /ip4/192.168.1.60/tcp/8080
Daemon is ready
```

Figure 23: IPFS daemon running.

The utilized IP must also be changed in the `IPFS_IP.txt` file inside the plugin's directory, or the `IPFS_IP` string resource of the Android app.

7 Evaluation

In this section, we evaluate the functionality of the system to ensure that the aforementioned actions, such as adding a new device, adding a dApp, recovering keys, etc. can be effectively performed. Additionally, we assess the time required for these operations to verify that they can be completed within a feasible timeframe.

7.1 App setup

The procedure to set up the smartphone app is detailed in the next subsections.

7.1.1 App installation

Provisionally named *TFGwallet*, the app has been installed on an Android device and is ready for use. This section covers the app's setup, user registration, and login. The app icon, as shown in Figure 24, represents the *TFGwallet* application on an Android device.

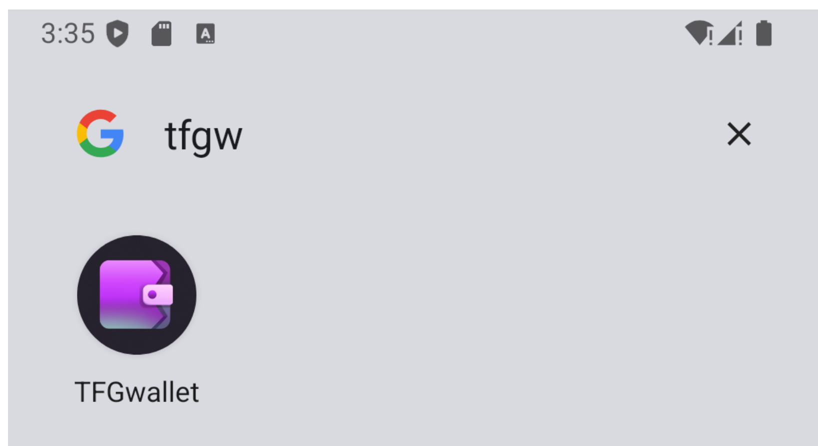


Figure 24: App icon on an Android device.

7.1.2 App navigation

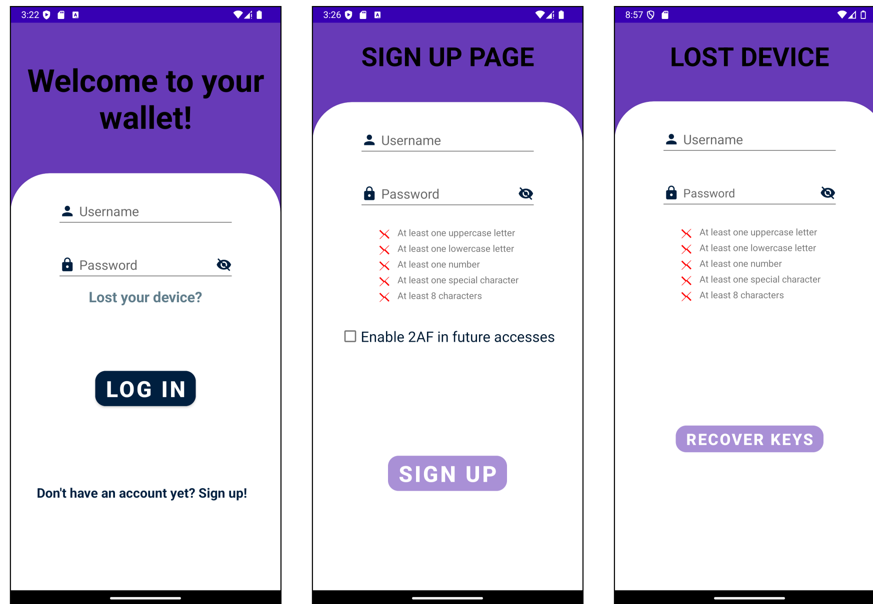
Upon opening the app, the user is presented with the login page (Figure 25a). Navigation options include the sign up page (Figure 25b) and the recovery page (Figure 25c).

7.1.3 User registration

To set up the app, the user must register by providing a valid email and a secure password. The sign up button is disabled until both fields are correctly filled to avoid errors.

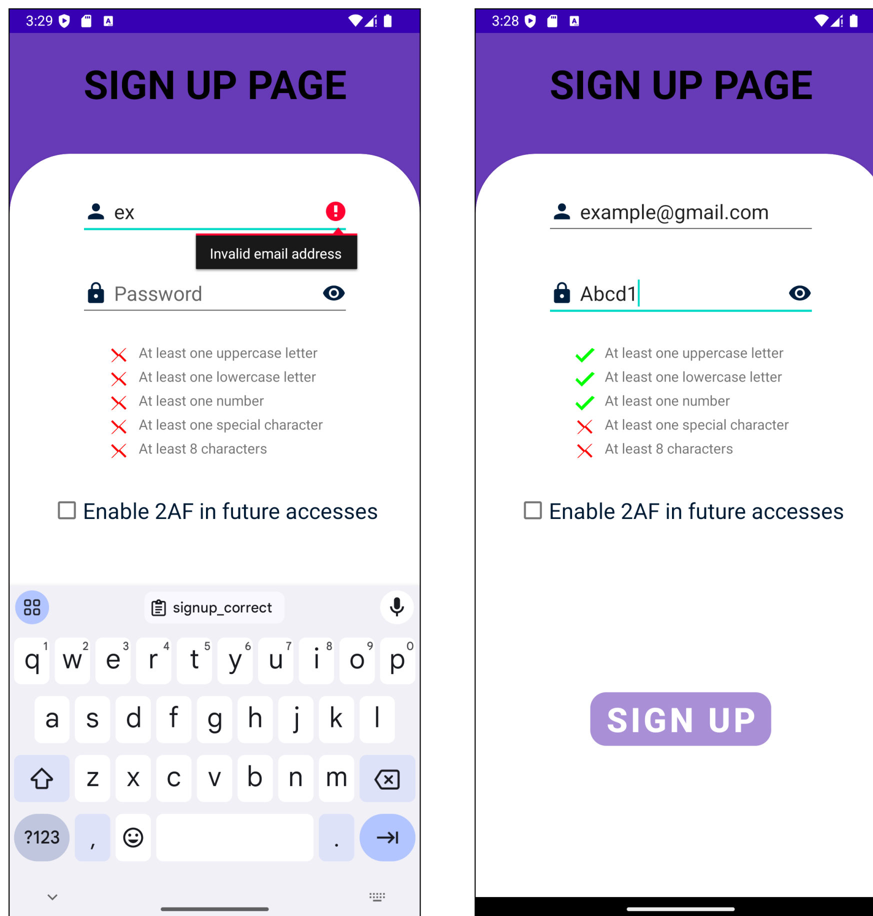
If the email is invalid, an error message is displayed (Figure 26a). Similarly, if the password does not meet the five required criteria (Figure 26b), the user cannot register. Red crosses indicate unmet requirements, while green ticks indicate fulfilled ones. All requirements must be met for the sign up button to be enabled, as shown in Figure 27. The user can also activate two-factor authentication by tapping on the checkbox.

Upon successful registration, a new account is created, keys are generated, and the



(a) Login page. **(b)** Sign up page. **(c)** Key recovery page.

Figure 25: App navigation.



(a) Invalid email during sign up. **(b)** Invalid password.

Figure 26: Error handling on app sign up page.

blockchain account is funded. The user must wait for this process to complete (Figure 28).

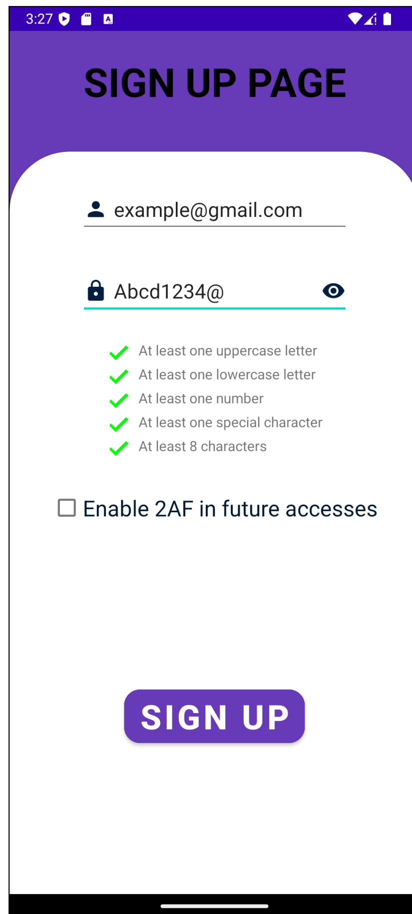


Figure 27: Password fulfills the requirements for sign up.

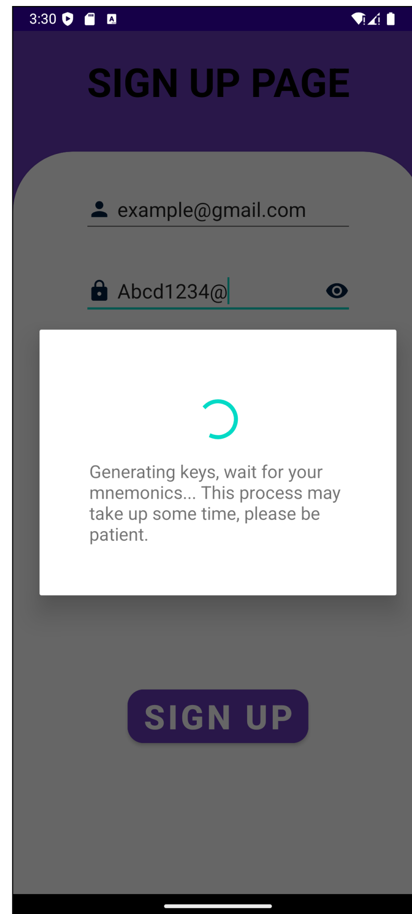


Figure 28: Key generation during app setup.

After key generation, the user is presented with their 24 mnemonic words (Figure 29), essential for key recovery. Finally, the smart contract is deployed, and its address (smart contract hash) is displayed (Figure 30).

7.1.4 Signing In and Home Page

After creating an account, the user can sign in to the app. Error handling for sign in has been simplified to display a generic message to avoid revealing sensitive information. For example, if the password is incorrect (Figure 31a), the error message does not specify whether the email or password is incorrect (Figure 31b).

If the user has enabled two-factor authentication during sign up, biometric verification will be prompted before navigating to the main page (Figure 32a). Upon successful recognition of the biometric factor, a green tick will indicate success (Figure 32c). However, if not recognized, a red cross will be displayed (Figure 32b). After several failed attempts (e.g., due to a dirty sensor), the app will offer alternative methods for authentication, such as using the device PIN or password (Figure 32d).

Once the user successfully signs in, the main page is displayed. This main page consists of four different sections: the home page (where QR codes are scanned, as shown in Figure 33a), the keys section (Figure 33b), the preferences section (Figure 33c), and the information page (Figure 33d).

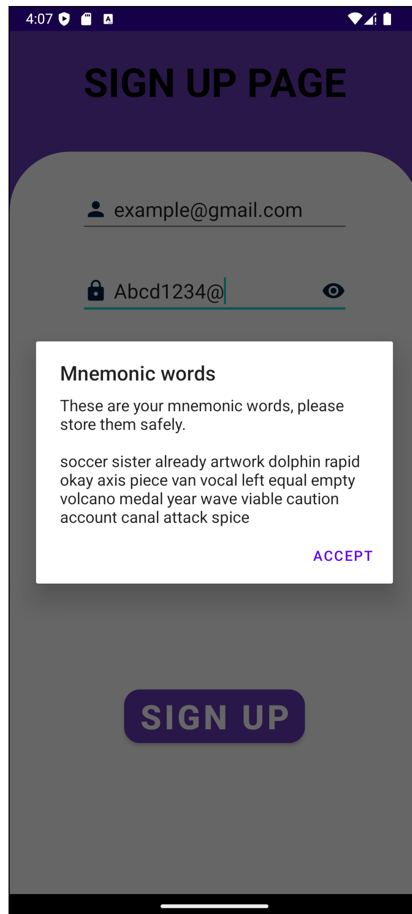


Figure 29: Mnemonic list during app setup.

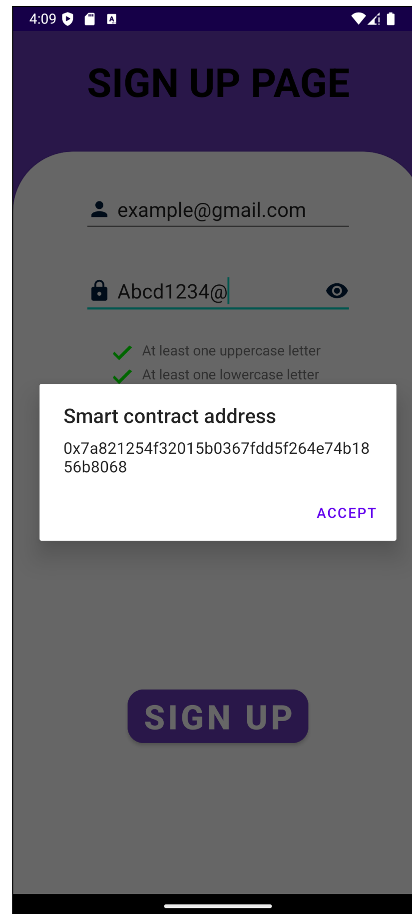


Figure 30: Newly-created SC address during app setup.

7.2 Plugin setup

In the next subsections, we explain how to set up the browser plugin.

7.2.1 Plugin installation

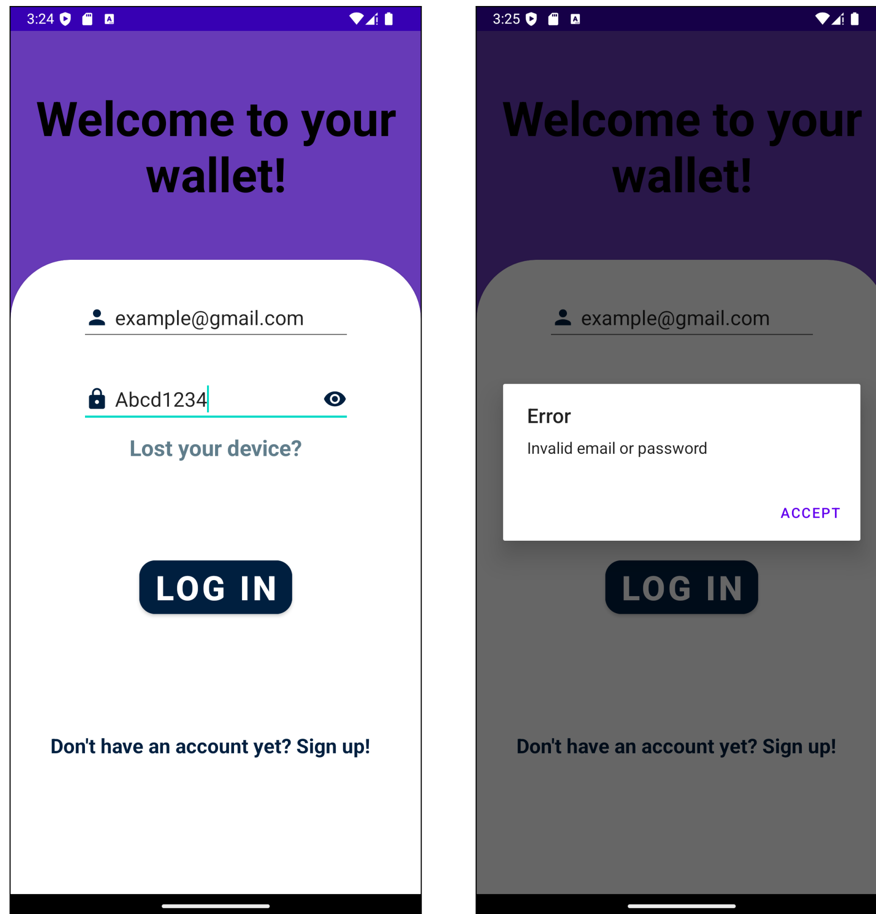
After installing the plugin, it will appear in the extensions section of Google Chrome (Figure 34). The first time opening the plugin, users will find a message informing them that it has not been configured yet (Figure 35). To proceed, users should follow the instructions and navigate to the setup page (Figure 36).

7.2.2 Plugin configuration

To configure the plugin, the user must introduce the smart contract address provided during app sign up (Figure 37). After entering the address, the QR code generator button will become active. Pressing this button will generate the QR code (Figure 38). Note that if the provided address is not valid, an error message will be prompted (Figure 39).

7.2.3 Adding devices

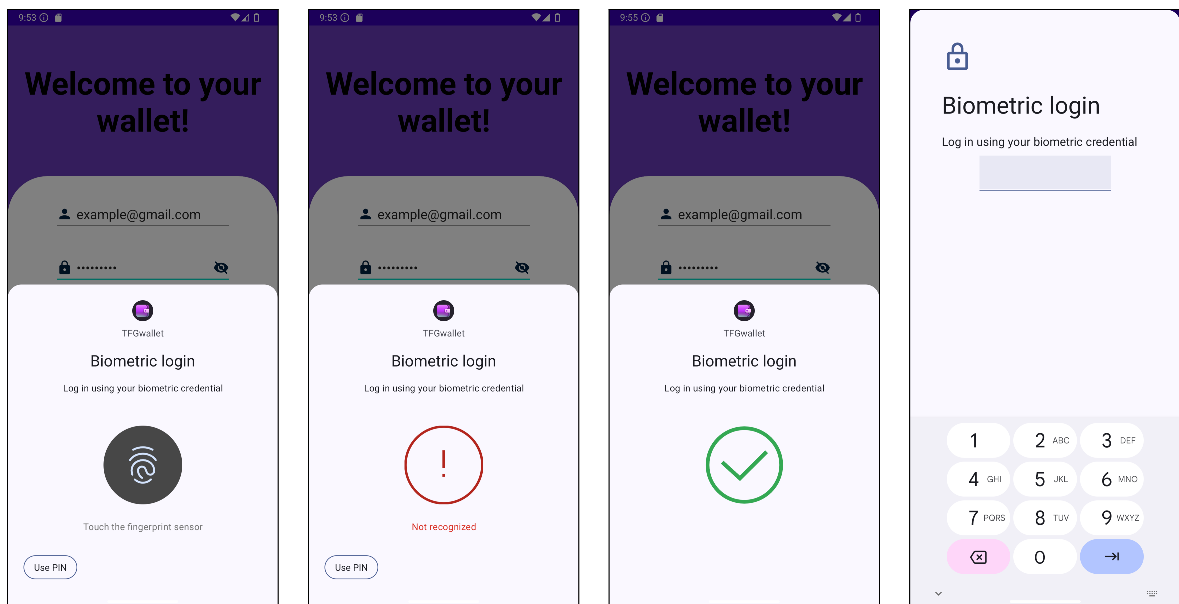
After successfully signing up and logging in to the smartphone app, the user can add new devices. To do this, assuming that the steps in Section 7.2.2 have been completed,



(a) Incomplete password during sign in.

(b) Generic error message on sign in page.

Figure 31: Error handling on app sign in page.



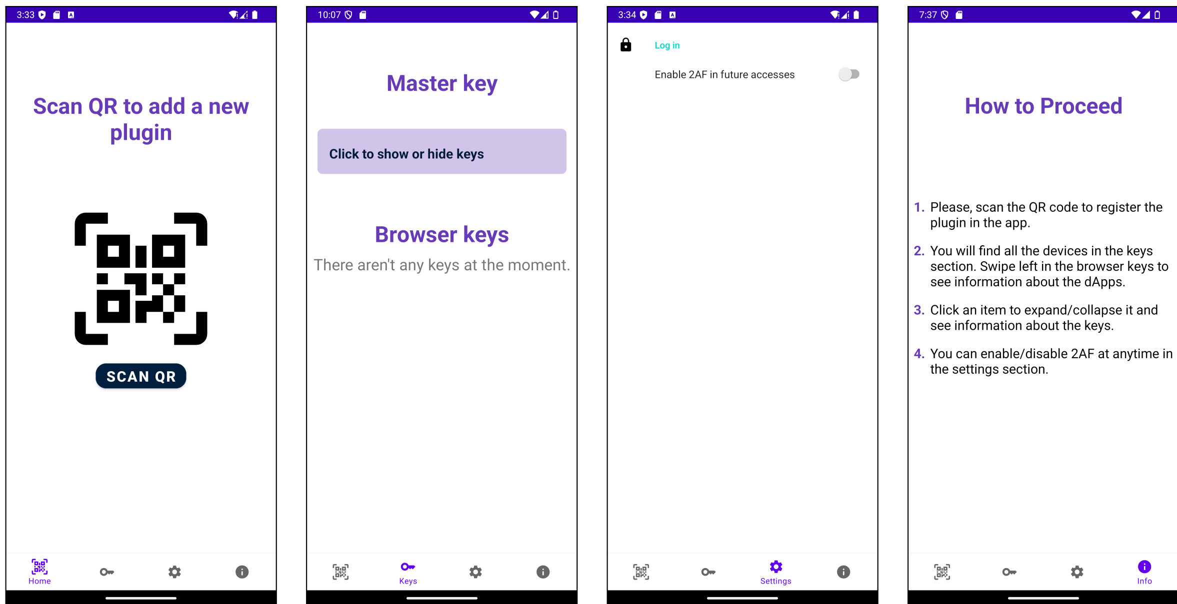
(a) Biometric prompt when 2FA is enabled.

(b) Biometric factor not recognized.

(c) Recognized biometric factor.

(d) Backup 2FA if biometrics fail.

Figure 32: Biometric access.



(a) QR code scan page. (b) Keys page. (c) Preferences page. (d) Information page.

Figure 33: Main page navigation.

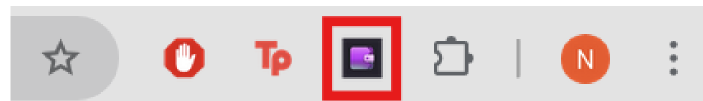


Figure 34: Plugin icon in Google Chrome.

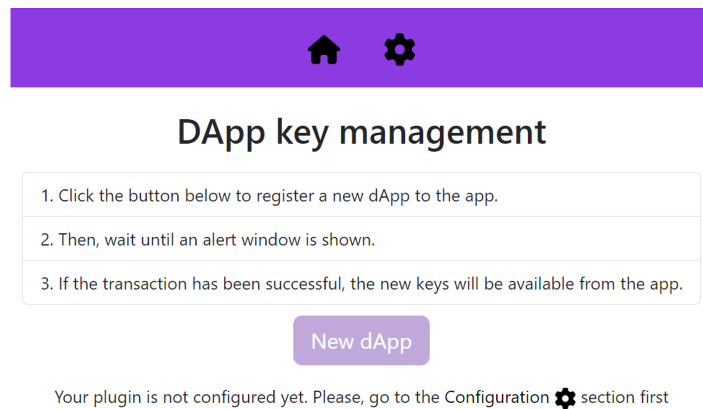


Figure 35: Plugin home page when not yet configured.

a QR code must be scanned from the scan page (33a). Upon clicking the scan button, the device's camera will open, allowing the user to scan the QR code (Figure 40a). Once the QR code is scanned, a loading message will be displayed while the app interacts with the blockchain (Figure 40b).

If the transaction is successful, a success message will be shown (Figure 40c). If the transaction fails, an error message will be displayed instead (Figure 40d).

7.3 Adding a new dApp

After generating the QR code and successfully adding the current device to the app, the user can access dApps by clicking the *New Dapp* button (Figure 41). Since

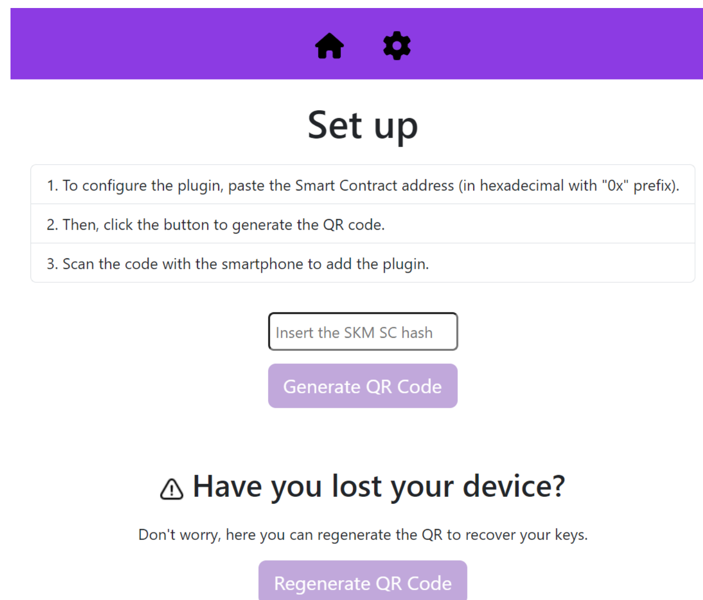


Figure 36: Plugin set up page when not yet configured.

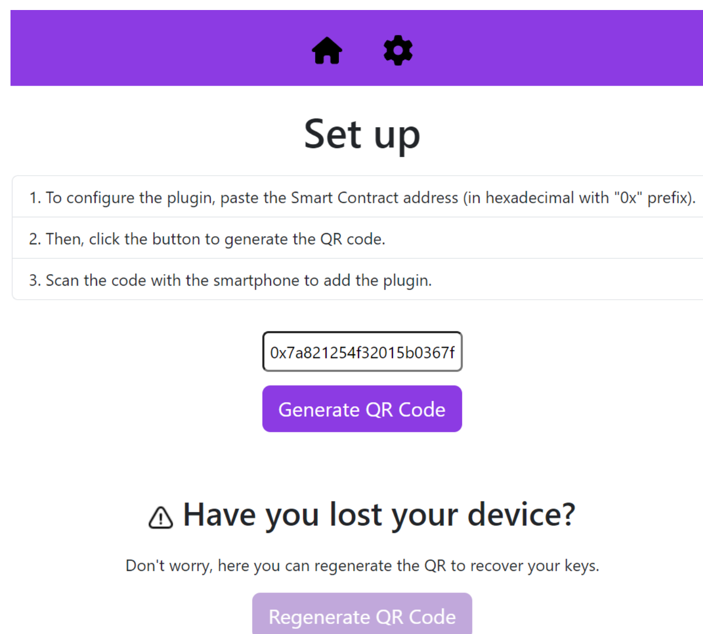


Figure 37: Plugin set up.

this process may take a few seconds, a loading message is displayed while waiting for a response (Figure 42). Finally, the user will receive a notification indicating whether the transaction was successful (Figure 43).

7.4 Key management

When devices are added, the keys section will list them by name only (Figure 44a). If the user wants to see the cryptographic keys associated with a specific device, they can simply tap on the device item, causing the listing to expand and reveal the keys (Figure 44c). This behavior applies to both browser keys and dApps keys. To enhance security, the master keys are not shown by default, preventing potential data leaks

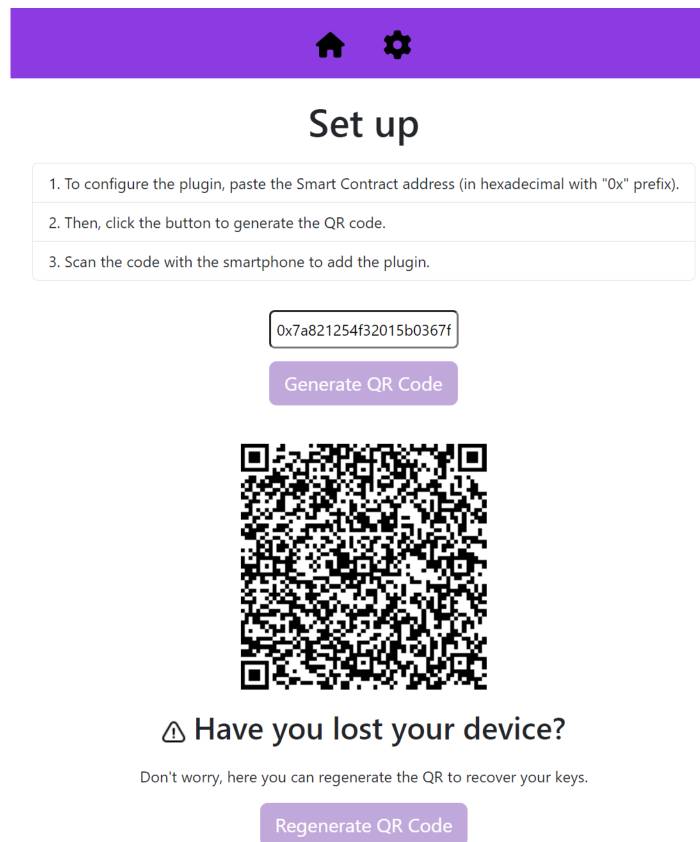


Figure 38: QR code generation.

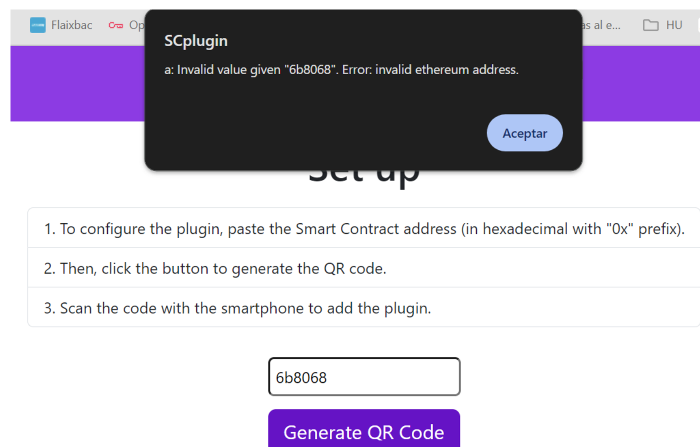
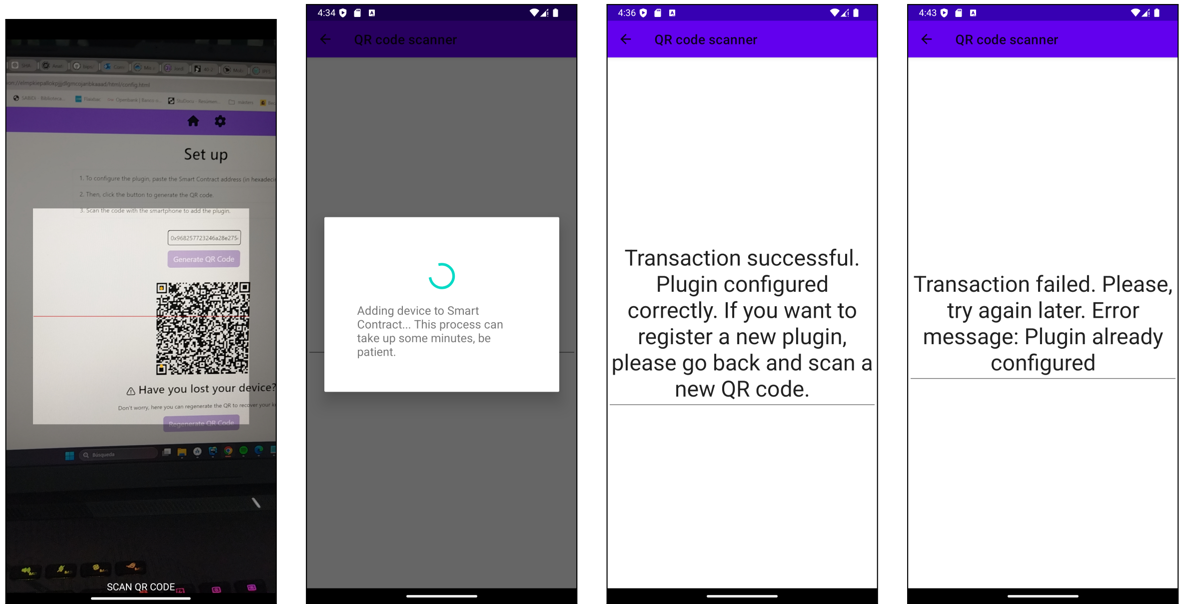


Figure 39: Invalid smart contract address.

(e.g., someone nearby taking a picture of the master keys, which could compromise the derived keys).

To access the dApps associated with a specific device, the user can swipe left (Figure 44d). This action will bring up a new screen displaying all accessed dApps (Figure 45a), and the associated keys can be viewed by tapping on the corresponding item (Figure 45b). If no dApps have been accessed from the device yet, a message is shown (Figure 45c).



(a) QR code scanning after plugin's set up.

(b) Loading message while adding device.

(c) Successful transaction.

(d) Unsuccessful transaction.

Figure 40: QR scanning process and error handling.

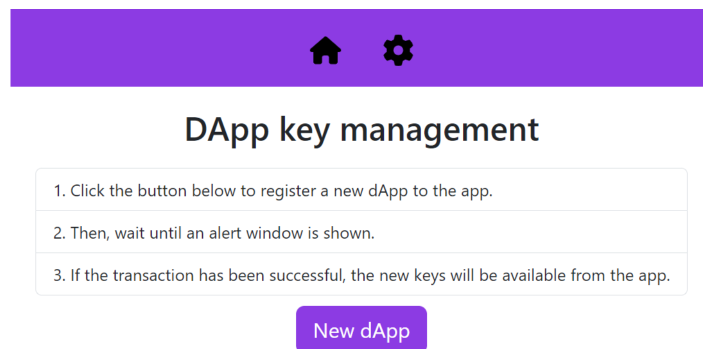


Figure 41: Adding a new dApp.

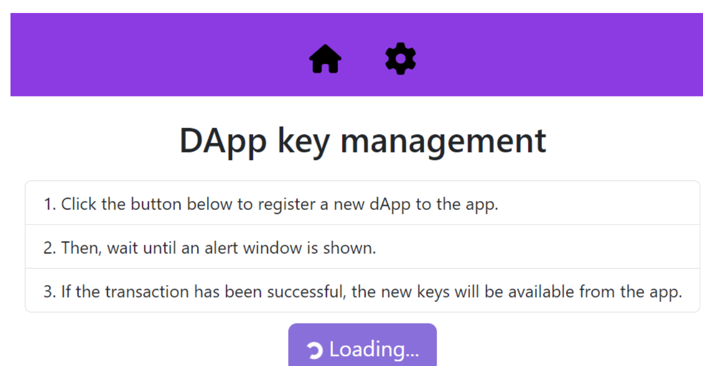


Figure 42: Loading message when adding a new dApp.

7.5 Key recovery

On the recovery page, similar to the sign up page, the system verifies that the provided email is valid and that the password meets the required criteria.

After validating both fields, the app navigates to a page where the mnemonic words

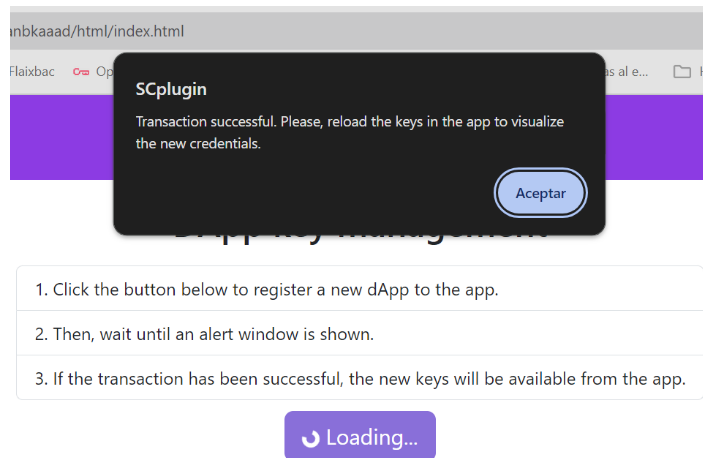
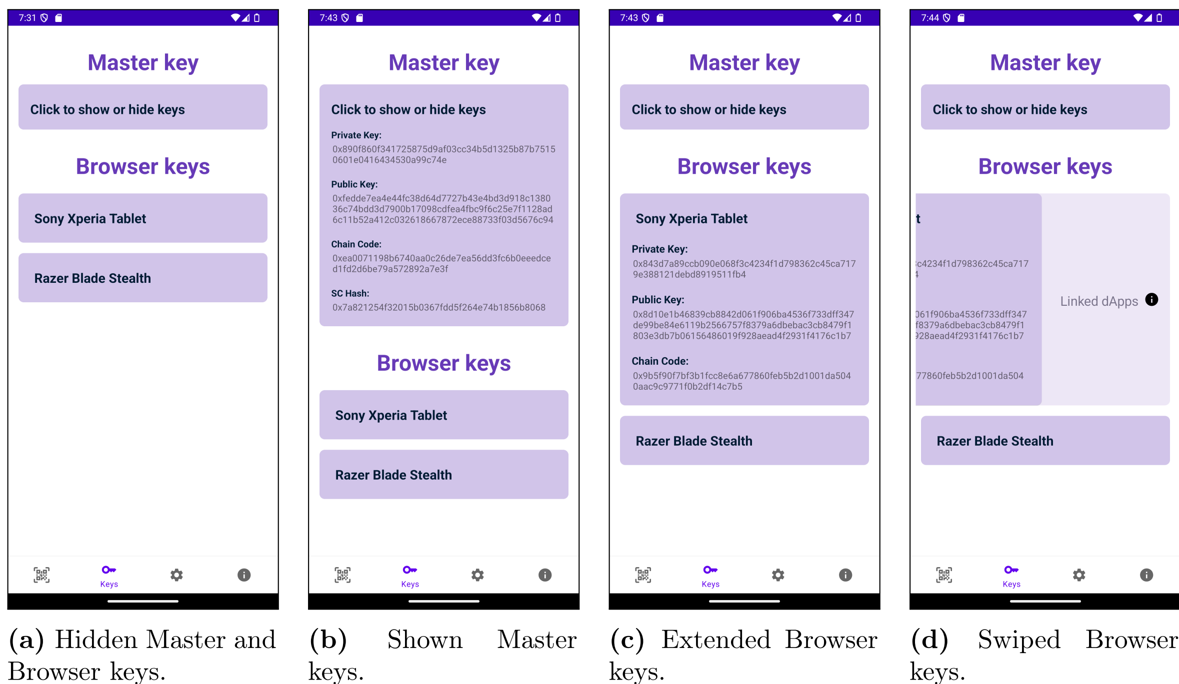


Figure 43: Successful transaction when adding new dApp.



(a) Hidden Master and Browser keys.

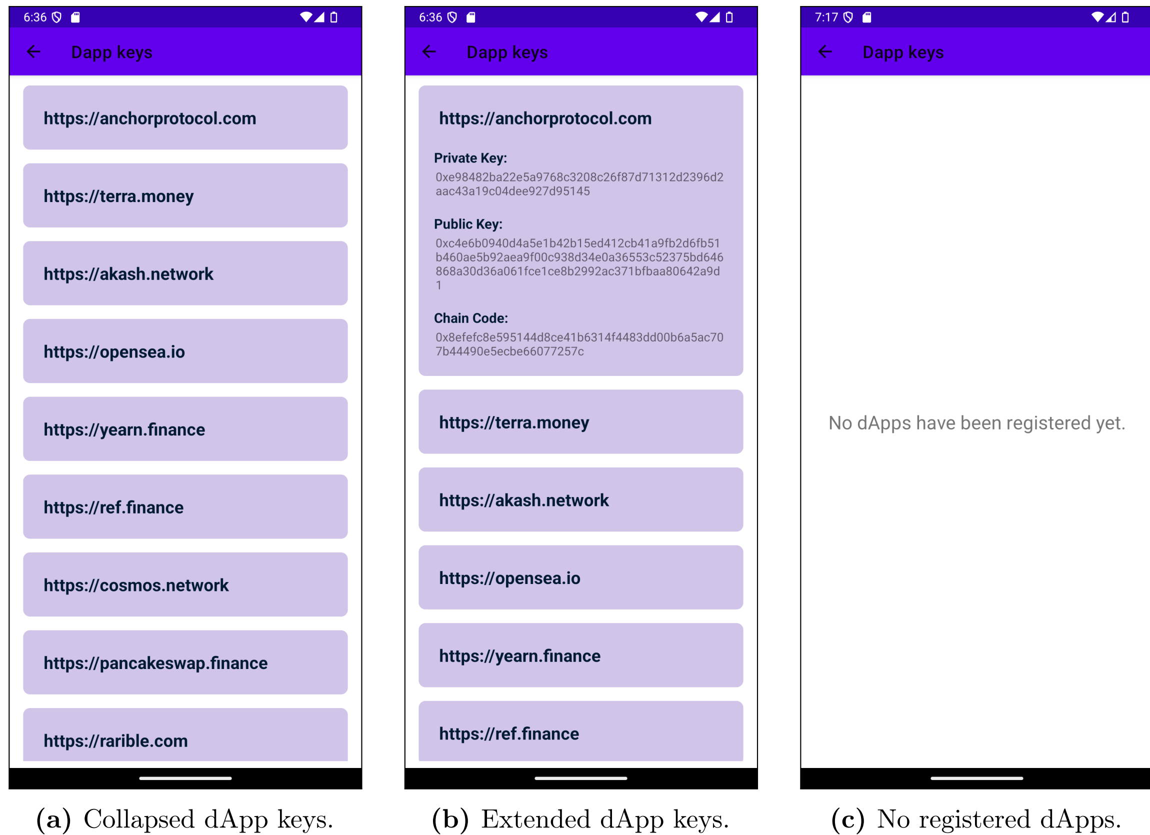
(b) Shown Master keys.

(c) Extended Browser keys.

(d) Swiped Browser keys.

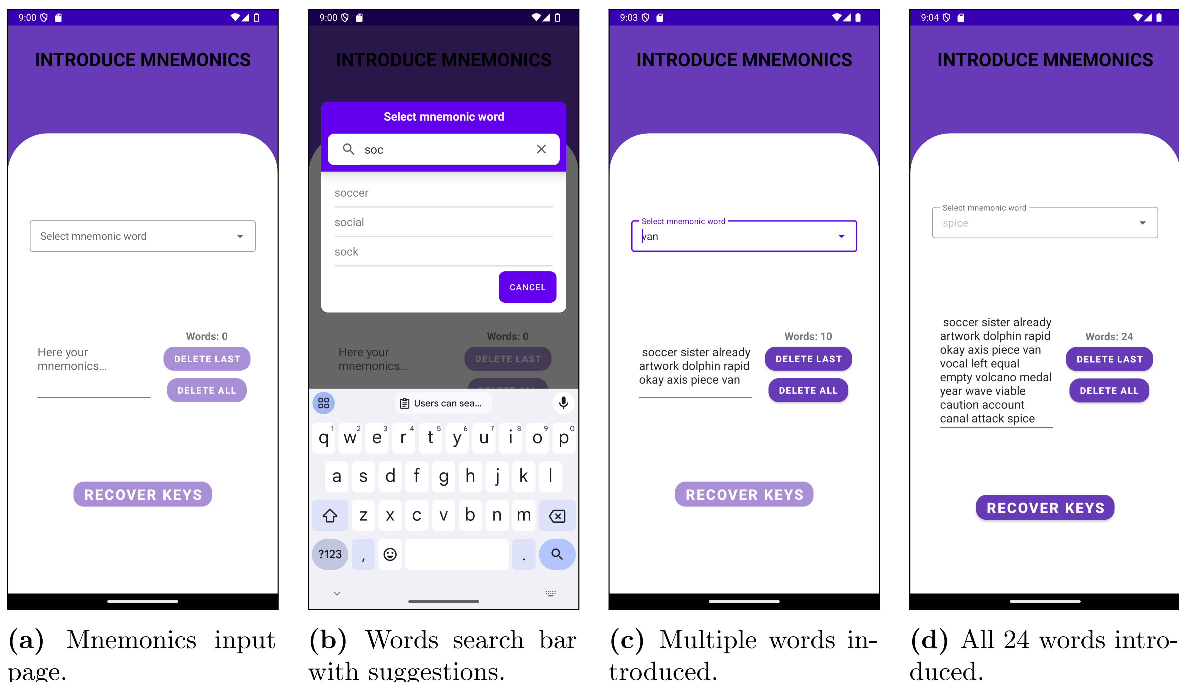
Figure 44: Master and Browser keys.

can be easily entered (Figure 46a). To minimize input errors, users can only select words from a predefined list of mnemonic words. Users can search for specific words, and the app will suggest matching options, as shown in Figure 46b. Selected words are displayed so that users can review and correct any mistakes (Figure 46c). Once all 24 mnemonic words are introduced, the search bar is disabled (Figure 46d) and the user can proceed with the recovery process. When the recover button is tapped on, the app prompts the user to enter the number of devices previously registered (Figure 47). Based on this number, the user will have to scan the corresponding number of QR codes. After scanning, the recovery process will take place and the user will be allowed to sign in as usual.



(a) Collapsed dApp keys. (b) Extended dApp keys. (c) No registered dApps.

Figure 45: dApp keys.

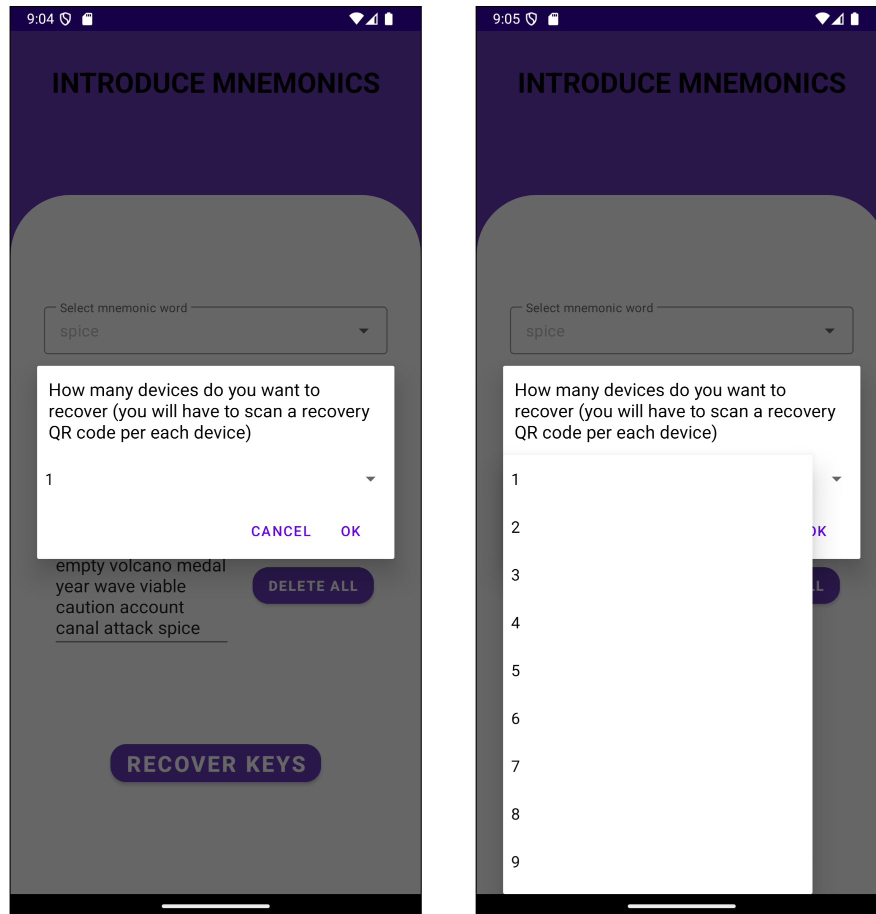


(a) Mnemonics input page. (b) Words search bar with suggestions. (c) Multiple words introduced. (d) All 24 words introduced.

Figure 46: Mnemonic words input page.

7.6 Performance

Since performance is a critical and defining factor in information systems, it is essential to guarantee that operations are executed promptly. Moreover, key operations must be completed with minimal delay. In the next subsections, we will present the measurement results for each component of the protocol, along with a comprehensive



(a) Prompt to pick the number of devices.

(b) Extended prompt showing possible picks.

Figure 47: Dialog to select the number of devices to recover.

analysis of the findings. A summary table highlighting overall execution times for each step is shown in Table 7.

Protocol step	Average time (ms)	Median (ms)	Standard deviation (ms)
Setting up the Smartphone App	26595	19074	10707
Setting up the web browser Plugin	44953	42410	6788
Session key pair generation	18816	17193	8118
Key visualization	182	71	272
Key recovery	690	695	95

Table 7: Performance metrics.

7.6.1 Environment and methodology

Performance monitoring tools were integrated into the development environment to measure the execution time of the different protocol components. This data, measured in milliseconds, will help assess the wallet's performance and suitability for real-world

use cases. The testing device was an Android device running Android version 13.

To interact with the blockchain, Sepolia ETH cryptocurrency is required. However, as discussed in Section 3.1, funding the account is challenging due to limited mining operations, which restricts the number of tests. Despite this, each procedure was repeated 10 times to ensure data reliability. It is important to note that operations heavily influenced by user interaction, such as password entry or mnemonic word selection, were not measured (although displayed), as they might introduce a huge variability in the results.

Actions done by the *User* are marked with the (U) tag, whereas steps involving interaction with the blockchain are marked with the (BC) tag.

7.6.2 Setting up the smartphone app

The time measurements presented in Table 8 highlight the time execution of the smartphone app's setup. It is worth noting that the variability in the overall times is attributed to fluctuations in the Sepolia Test network, where factors such as changes in gas prices and the number of pending transactions can significantly impact execution time. Another notable observation is the initial measurement of the BIP39 seed generation, which is nearly ten times greater than the subsequent measurements. This discrepancy is due to the initialization of the cryptographic libraries used in the process. Overall, and in standard network conditions, the app setup takes about 19 seconds. All the previous analysis is valid for the following subsections.

Step	**Iterations (time measured in ms)**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
S1.1 (U)	-	-	-	-	-	-	-	-	-	-
S1.2	106	16	15	16	19	18	16	17	17	18
S1.3 (U)	-	-	-	-	-	-	-	-	-	-
S1.4	84	18	18	18	20	19	18	18	19	19
S1.5	42	59	68	32	49	74	49	77	49	66
S1.6 (BC)	18495	34009	18972	19006	34150	18963	34293	18937	49111	18939
Subtotals	**18727**	**34102**	**19073**	**19072**	**34238**	**19074**	**34376**	**19049**	**49196**	**19042**

Table 8: Setting up the Smartphone App performance.

7.6.3 Setting up the web browser plugin

Table 9 presents the execution times of this step. The time taken by cryptographic operations is almost negligible compared to blockchain operations and the overall execution time. Within the blockchain operations, there is a significant difference between getter and setter operations. Steps involving getters (S2.3, S2.11) are completed very quickly since they do not consume gas. In contrast, operations that require gas (S2.8, S2.10) account for nearly the entire time spent on the current step. Configuring the plugin takes about 42 seconds to complete. Given the infrequency of setup operations for both the app and plugin, the resulting times are not a significant concern.

Step	Iterations (time measured in ms)									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
S2.1 (U)	-	-	-	-	-	-	-	-	-	-
S2.2 (U)	-	-	-	-	-	-	-	-	-	-
S2.3 (BC)	998	1104	995	991	1000	997	993	1003	1004	997
S2.4	1.29	1.79	0.67	0.57	0.69	0.93	0.66	0.64	0.54	0.82
S2.5	17	30	12	12	12	12	12	12	12	12
S2.6 (U)	-	-	-	-	-	-	-	-	-	-
S2.7	13	17	2	2	4	2	1	5	7	3
S2.8 (BC)	18336	20790	35211	20160	20212	35531	20241	19107	20180	20107
S2.9	2	0	0	0	0	0	0	0	0	0
S2.10 (BC)	21069	21229	21230	21041	21075	21051	21056	19087	20098	21050
S2.11 (BC)	127	213	126	179	128	132	129	142	87	110
S2.12	1.63	1.08	0.63	0.63	0.58	0.57	0.82	0.36	0.79	0.75
Subtotals	**40565**	**43387**	**57577**	**42387**	**42432**	**57725**	**42433**	**39358**	**41389**	**42281**

Table 9: Setting up the browser plugin performance.

7.6.4 Adding a new dApp

This is the most critical and frequently executed step, as it occurs each time a user accesses a new dApp via their browser. The execution times for this step are detailed in Table 10. While the entire process takes approximately 17 seconds, which is considerable given its importance, the user only experiences a brief wait of around 270 milliseconds. This is because the longest operation (S3.7) runs in the background. Moreover, this operation only occurs the first time the user connects to a specific dApp. Overall, the time required for this step is considered acceptable given its nature.

Step	Iterations (time measured in ms)									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
S3.1 (U)	-	-	-	-	-	-	-	-	-	-
S3.2	0.65	0.16	0.71	0.61	0.57	0.40	0.11	1.13	0.12	0.42
S3.3	2.51	0.74	2.10	2.45	0.83	0.59	0.60	0.69	0.67	0.56
S3.4	0.13	0.14	0.20	0.10	0.41	0.09	0.08	0.06	0.16	0.07
S3.5	0.20	0.71	0.22	0.17	0.27	0.33	0.20	0.15	0.50	0.14
S3.6	11	4.47	10	11	4.26	4.33	3.59	3.24	3.93	3.16
S3.7	18154	18205	16405	35598	30913	15068	11225	11950	17962	12596
S3.7.1 (BC)	116	117	300	584	389	49	177	362	305	121
S3.7.2	7	5	5	4	6	3	5	4	5	4
S3.7.3	14	12	18	16	16	16	18	17	15	14
S3.7.4 (BC)	18018	18071	16082	34994	30502	15000	11025	11568	17637	12456
Subtotals	**18169**	**18211**	**16419**	**35612**	**30919**	**15074**	**11230**	**11956**	**17967**	**12601**

Table 10: Performance data for adding a new dApp.

7.6.5 Key visualization

The performance of this step is shown in Table 11. It is the fastest of all, taking approximately 71 milliseconds to execute, considering standard blockchain conditions.

Step	Iterations (time measured in ms)									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
S4.1 (BC)	877	67	314	58	52	52	57	56	61	49
S4.2	28	7	12	5	6	6	6	6	6	5
S4.3	5	6	5	4	7	5	5	4	4	6
S4.4	7	6	4	3	3	2	3	2	3	2
S4.5	1	0	0	1	0	1	0	0	0	0
S4.6 (U)	-	-	-	-	-	-	-	-	-	-
Subtotals	**918**	**86**	**335**	**71**	**68**	**66**	**71**	**68**	**74**	**62**

Table 11: Performance data for key visualization.

7.6.6 Key recovery

The tests conducted in this step involved key recovery when the number of added devices is 1, 2, and 5. The results of these tests are presented in Tables 12, 13 and 14. For an overall comparison, a summary of the metrics is provided in Table 15. Although an increase in recovery time was expected as the number of devices increased, the results do not show this trend. This is likely because most of the time is consumed by blockchain interactions. However, a closer look at step S5.7 reveals a slight increase in time corresponding to the number of recovered devices.

Step	Iterations for 1 Device (time measured in ms)									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
S5.1 (U)	-	-	-	-	-	-	-	-	-	-
S5.2 (U)	-	-	-	-	-	-	-	-	-	-
S5.3	35	49	74	33	39	41	61	40	34	49
S5.4	93	101	98	32	77	94	120	30	60	83
S5.5	39	90	105	111	76	153	115	145	92	94
S5.6 (BC)	516	209	377	518	474	395	531	462	570	431
S5.7	14	15	20	14	15	14	15	16	14	14
Subtotals	**697**	**464**	**674**	**708**	**681**	**697**	**842**	**693**	**770**	**671**

Table 12: Performance data for key recovery when the number of devices is 1.

Additionally, to assess how recovery time varies with the number of dApps on a single device, tests were carried out for the recovery of 5, 10, 15, 20, 25, and 30 different dApp keys. As shown in Figure 48, there is a direct correlation between the number of dApps and the time required to recover their keys. On average, each additional 5 dApps increases the recovery time by 59 milliseconds. This time increase is generally imperceptible to users.

Based on Nielsen's findings in [41], users can maintain their attention for up to 10 seconds before losing focus. The optimal response time, however, falls between 0.1

Step	Iterations for 2 Devices (time measured in ms)									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
S5.1 (U)	-	-	-	-	-	-	-	-	-	-
S5.2 (U)	-	-	-	-	-	-	-	-	-	-
S5.3	63	51	42	38	45	77	41	44	37	35
S5.4	87	94	83	82	71	71	70	76	90	72
S5.5	102	136	98	140	89	53	52	94	117	85
S5.6 (BC)	564	412	374	435	331	271	287	404	265	276
S5.7	21	27	31	30	25	18	20	22	20	21
Subtotals	**837**	**720**	**628**	**725**	**561**	**490**	**470**	**640**	**529**	**489**

Table 13: Performance data for key recovery when the number of devices is 2.

Step	Iterations for 5 Devices (time measured in ms)									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
S5.1 (U)	-	-	-	-	-	-	-	-	-	-
S5.2 (U)	-	-	-	-	-	-	-	-	-	-
S5.3	37	44	32	50	39	47	49	37	33	31
S5.4	53	64	58	61	62	61	52	52	59	49
S5.5	83	53	36	114	46	145	119	68	87	78
S5.6 (BC)	257	381	523	340	568	314	446	292	293	355
S5.7	57	35	43	41	47	37	35	36	36	36
Subtotals	**487**	**577**	**692**	**606**	**762**	**604**	**701**	**485**	**508**	**549**

Table 14: Performance data for key recovery when the number of devices is 5.

Number of devices	Average time (ms)	Median (ms)	Standard deviation (ms)
1	690	695	95
2	609	595	123
5	597	591	96

Table 15: Summary statistics for key recovery.

and 1 second. To estimate the number of dApps needed to create a noticeable delay during key recovery, we applied a linear regression to the data points, resulting in the following equation:

$$y = 11.63x + 114.07 \tag{2}$$

Using this equation, we can determine the number of dApps that would cause a delay sufficient to distract the user (over 10 seconds):

$$x = \frac{10000 - 114.07}{11.63} \approx 850 \text{ dApps}$$

Therefore, a delay exceeding 10 seconds would only occur if the number of dApp keys to be recovered surpasses approximately 850, a scenario that appears improbable.

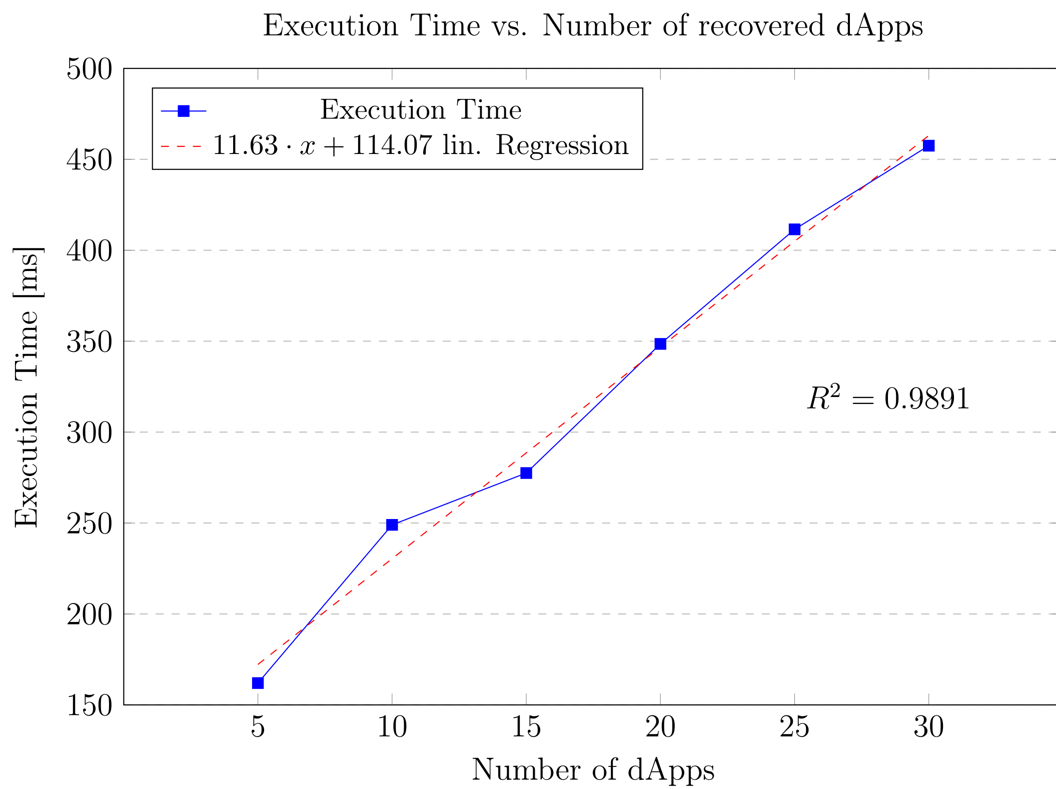


Figure 48: Execution time as the number of dApps increases. The dotted line indicates the fitted linear regression.

8 Conclusions

In this project, a secure, multi-platform decentralized wallet has been developed, comprising a browser plugin that transparently generates keys for newly accessed dApps and a smartphone app that manages and provides access to all the generated keys.

Decentralized technologies like blockchain and IPFS were leveraged, thus eliminating reliance on central entities. However, the development process encountered several challenges. Initially, the use of the Ganache tool proved inadequate for creating new blockchain accounts based on BIP32-generated keys, necessitating a shift to Sepolia. While Sepolia offers an environment closer to the real Ethereum network, it is more complex to work with. This complexity required the integration of a tool to estimate gas fees, as gas prices fluctuate with the number of pending transactions, making debugging a more tedious and time-consuming process.

Additionally, running IPFS directly on the Android device was not feasible due to the lack of suitable tools or libraries, and implementing such functionality was beyond the scope of this project. As an alternative, a gateway was implemented on a computer to access the IPFS network. Despite these challenges, the project successfully achieved its primary goal of creating a decentralized wallet that securely manages cryptographic keys across multiple platforms.

After thoroughly evaluating the execution times of the system processes and analyzing the graphical user interface, the application has demonstrated strong usability with practical waiting times for users. This is particularly evident in critical operations, such as adding a new dApp, which is one of the most frequently performed tasks. The application consistently delivers prompt responses in these essential operations, ensuring a smooth user experience. These results affirm that the application not only meets functional requirements but also provides a responsive and intuitive interface, making it well-suited for everyday use.

Finally, I want to express that this project has profoundly contributed to my personal and academic growth. It has provided me with a wealth of new knowledge and significantly enhanced my decision-making skills. While there were challenging moments, I was able to overcome these obstacles through perseverance and problem-solving. The support and guidance from my tutors were invaluable in navigating these difficulties. Their assistance played a crucial role in helping me successfully complete this project and achieve my goals.

8.1 Future work

In future iterations of the system, we plan to integrate the IPFS daemon directly into the mobile app, removing the requirement for users to connect to the gateway running on a PC.

We also aim to expand the system's functionalities, by adding features such as key exportation in PKCS#12 format[36]. This will allow users to send keys via email or transfer them to external devices.

To enhance interoperability, we intend to develop an iOS version of the app, along

[36] **PKCS#12:** https://datatracker.ietf.org/doc/rfc7292/

with plugin versions for other platforms like Mozilla Firefox and Microsoft Edge.

The business model for this prototype also requires careful consideration. Currently, the prototype relies on a primary account funded through mining to create new accounts. For a production version, alternative funding strategies should be explored. These could include incorporating advertisements within the app, utilizing in-app mining to partially fund accounts, or requiring users to fund their own accounts. Additionally, to improve security and protect against tracking attacks, implementing a mixer[37] for blockchain transactions should be also considered.

[37]**Mixers:** https://www.scorechain.com/resources/crypto-glossary/mixers

References

[1] Parikshit Hooda. Centralized vs. decentralized vs. distributed systems, July 2024. URL https://www.geeksforgeeks.org/comparison-centralized-decentralized-and-distributed-systems/#what-are-decentralized-systems. Accessed: 2024-08-01.

[2] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and Xiaofeng Wang. The tangled web of password reuse. 01 2014. ISBN 1-891562-35-5. doi: 10.14722/ndss.2014.23357.

[3] Jan Camenisch, Anna Lysyanskaya, and Gregory 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, page 525–536, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316514. doi: 10.1145/2382196.2382252. URL https://doi.org/10.1145/2382196.2382252.

[4] Shuangyu He, Qianhong Wu, Xizhao Luo, Zhi Liang, Dawei Li, Hanwen Feng, Haibin Zheng, and Yanan Li. A social-network-based cryptocurrency wallet-management scheme. *IEEE Access*, 6:7654–7663, 2018. doi: 10.1109/ACCESS.2018.2799385.

[5] Qianwen Wei, Shujun Li, Wei Li, Hong Li, and Mingsheng 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*, volume 14 of *WASA*, pages 358–369. Springer, 2019.

[6] Niko Lehto, Kimmo Halunen, Outi Marja Latvala, Anni Karinsalo, and Jarno Salonen. Cryptovault-a secure hardware wallet for decentralized key management. In *2021 IEEE International Conference on Omni-Layer Intelligent Systems, COINS 2021*, United States, August 2021. IEEE Institute of Electrical and Electronic Engineers. ISBN 978-1-6654-3157-6. doi: 10.1109/COINS51742.2021.9524133.

[7] Reza Soltani, Uyen Trang Nguyen, and Aijun 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)*, pages 320–325, 2019. doi: 10.1109/DASC/PiCom/CBDCom/CyberSciTech.2019.00066.

[8] Serkan Ayvaz Mehmet Aydar, Salih Cemil Cetin and Betul Aygun. Private key encryption and recovery in blockchain, 2019. URL https://arxiv.org/abs/1907.04156v2. arXiv:1907.04156v2 [cs.CR].

[9] Xiaojian He, Jinfu Lin, Kangzi Li, and Ximeng Chen. A novel cryptocurrency wallet management scheme based on decentralized multi-constrained derangement. *IEEE Access*, 7:185250–185263, 2019. doi: 10.1109/ACCESS.2019.2961183.

[10] Guojia Li and Lin You. A consortium blockchain wallet scheme based on dual-threshold key sharing. *Symmetry*, 13(8), 2021. ISSN 2073-8994. doi: 10.3390/sym13081444. URL https://www.mdpi.com/2073-8994/13/8/1444.

[11] Zhang Yifang, Wang Mingyue, Guo Yu, and Guo Fangda. 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*, CIKM '23, page 3371–3380, New York, USA, 2023. Association for Computing Machinery. ISBN 9798400701245. doi: 10.1145/3583780.3615090. URL https://doi.org/10.1145/3583780.3615090.

[12] Rosario Gennaro, Steven Goldfeder, and Arvind Narayanan. Threshold-optimal dsa/ecdsa signatures and an application to bitcoin wallet security. In Mark Manulis, Steve Schneider, and Ahmad-Reza Sadeghi, editors, *Applied Cryptography and Network Security - 14th International Conference, ACNS 2016, Proceedings*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 156–174, Germany, 2016. Springer Verlag. ISBN 9783319395548. doi: 10.1007/978-3-319-39555-5_9.

[13] Dan Boneh, Rosario Gennaro, and Steven Goldfeder. Using level-1 homomorphic encryption to improve threshold dsa signatures for bitcoin wallet security. In Tanja Lange and Orr Dunkelman, editors, *Progress in Cryptology – LATINCRYPT 2017*, pages 352–377, Cham, 2019. Springer International Publishing. ISBN 978-3-030-25283-0.

[14] Huawei Zhao, Yong Zhang, Yun Peng, and Ruzhi Xu. Lightweight backup and efficient recovery scheme for health blockchain keys. In *2017 IEEE 13th International Symposium on Autonomous Decentralized System (ISADS)*, pages 229–234, 2017. doi: 10.1109/ISADS.2017.22.

[15] Har Preet Singh, Kyriakos Stefanidis, and Fabian Kirstein. A private key recovery scheme using partial knowledge. In *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2021. doi: 10.1109/NTMS49979.2021.9432642.

[16] Jungwon Seo, Deokyoon Ko, Suntae Kim, Vijayan Sugumaran, and Sooyong Park. Reminisce: Blockchain private key generation and recovery using distinctive pictures-based personal memory. *Mathematics*, 10(12), 2022. ISSN 2227-7390. doi: 10.3390/math10122047. URL https://www.mdpi.com/2227-7390/10/12/2047.

[17] Hossein Rezaeighaleh and Cliff C. Zou. New secure approach to backup cryptocurrency wallets. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2019. doi: 10.1109/GLOBECOM38437.2019.9014007.

[18] S. Haber and W. S. Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3:99–111, 1991. doi: 10.1007/BF00196791. URL https://doi.org/10.1007/BF00196791.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL https://www.bitcoin.org/bitcoin.pdf. Email: satoshin@gmx.com.

[20] Ethereum. What is ethereum?, 2024. URL https://ethereum.org/en/what-is-ethereum/. Accessed: [July 2024].

[21] pk910. Pow faucet, 2024. URL https://github.com/pk910/PoWFaucet/wiki. Accessed: 2024-07-21.

[22] D. Tapscott and A. Tapscott. *Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World*. Penguin Publishing Group, 2016. ISBN 9781101980156. URL https://books.google.es/books?id=NqBiCgAAQBAJ.

[23] Ethereum. Gas and fees, March 2024. URL https://ethereum.org/en/developers/docs/gas/. Edited by Riley Annon.

[24] Ethereum. Introduction to smart contracts, April 2024. URL https://ethereum.org/en/developers/docs/smart-contracts/. Edited by Paul Wackerow.

[25] Truffle Suite. Ganache documentation, 2024. URL https://archive.trufflesuite.com/docs/ganache/. Accessed: 2024-07-21.

[26] Kingsley Okonkwo. Consensys announces the sunset of truffle and ganache and new hardhat partnership, September 2023. URL https://www.consensys.net/blog/news/consensys-announces-the-sunset-of-truffle-and-ganache-and-new-hardhat-partnership/. News.

[27] Alchemy. What is the sepolia testnet?, March 2023. URL https://alchemy.com. Reviewed by Brady Werkheiser.

[28] Dock. Web3 identity: Beginner's guide, July 2024. URL https://www.dock.io/post/web3-identity. Published July 24, 2024.

[29] River Financial. What is a bitcoin improvement proposal (bip)?, 2024. URL https://river.com/learn/what-is-a-bitcoin-improvement-proposal-bip. Accessed: 2024-07-27.

[30] Marek Palatinus, Pavol Rusnak, Aaron Voisine, and Sean Bowe. Mnemonic code for generating deterministic keys. Standards track, Bitcoin Improvement Proposals, September 2013. URL https://github.com/bitcoin/bips/wiki/Comments:BIP-0039. Unanimously Discouraged for implementation.

[31] Pieter Wuille. Bip: 32 - hierarchical deterministic wallets, February 2012. URL https://github.com/bitcoin/bips/wiki/Comments:BIP-0032.

[32] Daniel R. L. Brown. Standards for efficient cryptography: Sec 2 - recommended elliptic curve domain parameters. Technical Report Version 2.0, Certicom Research, January 2010. URL http://www.certicom.com.

[33] Android Developers. Meet android studio, 2024. URL https://developer.android.com/studio. Last update: 2024-07-25.

[34] Yuvraj10. Kotlin vs java – which one should i choose for android development. *GeeksforGeeks*, 2024. URL https://www.geeksforgeeks.org/kotlin-vs-java/. Last updated: 26 Jul, 2024.

[35] Mozilla Developer Network. What are extensions?, 2024. URL https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/What_are_WebExtensions. Accessed: 2024-07-27.

[36] Mozilla Developer Network. Anatomy of an extension, 2024. URL https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Anatomy_of_a_WebExtension. Accessed: 2024-07-27.

[37] Juan Benet. Ipfs - content addressed, versioned, p2p file system, 2014. URL https://arxiv.org/abs/1407.3561.

[38] C. Daudén-Esmel, J. Castellà-Roca, and A. Viejo. Multi-platform wallet for privacy protection and key recovery in decentralized applications. *Blockchain: Research and Applications*, 2024. ISSN 2096-7209. Submitted.

[39] Kaalel. MVC Framework Introduction, 2024. URL https://www.geeksforgeeks.org/mvc-framework-introduction/. Last Updated: 08 Jul, 2024.

[40] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. Pkcs #1: Rsa cryptography specifications version 2.2. Request for Comments 8017, November 2016. URL https://datatracker.ietf.org/doc/html/rfc8017. Obsoletes: RFC 3447. Category: Informational.

[41] Jakob Nielsen. *Usability Engineering*. Academic Press, 1993. Excerpt from Chapter 5, published January 1, 1993. Discusses time limits based on human perceptual abilities for optimizing web and application performance.

Appendix A: SKM SC in Solidity

```
// SPDX-License-Identifier: GPL-3.0-or-later

pragma solidity >=0.4.16 <0.9.0;

/**
 * @title Secure Key Management Smart Contract
 * @dev The Secure Key Management SC allows the user access to the cryptographic
  keys stored in the IPFS, but also to exchange some information between the
  plug-in installed in a web browser and the smartphone during the set-up of the system.
 */

contract SKM_SC {

    // Identities of the actors
    address private smartphoneID;
    bytes private publicKey;

    // Struct to store all plug-ins info.
    struct DeviceInfo {
        bool exists;
        string IPFSref;
    }
    mapping(address => DeviceInfo) private whiteList;

    bytes private temp;

    /** Constructor
     * @dev Create a new Secure Key Management Smart Contract with the given public key.
     * @param _publicKey The public key to be stored in the contract
     */
    constructor(bytes memory _publicKey) {
        smartphoneID = msg.sender;
        publicKey = _publicKey;
    }

    /**
     * @dev Adds a new public key to the whiteList list
     * @param deviceID public key of the new device
     */
    function addDevice(address deviceID) external onlySmartphone {
        require(!whiteList[deviceID].exists, "This plug-in has already been configured.");
        whiteList[deviceID].exists = true;
    }

    /**
     * @dev Removes an existing public key from the whiteList list
     * @param deviceID public key of an existing device
     */
    function removeDevice(address deviceID) external onlySmartphone {
        require(whiteList[deviceID].exists, "This plug-in is not configured.");
        whiteList[deviceID].exists = false;
    }

    /**
     * @dev Modifies the reference to the IPFS in the mapping position of the device
      that has used the method
     * @param IPFSref new IPFS reference
     */
    function storeRef(string memory IPFSref) external {
        require(whiteList[msg.sender].exists, "This plug-in is not configured.");
        whiteList[msg.sender].IPFSref = IPFSref;
    }

    /**
     * @dev Returns IPFS reference
     */
```

```
67        function getRef() external view returns (string memory) {
68            require(whiteList[msg.sender].exists, "This plug-in is not configured.");
69            return whiteList[msg.sender].IPFSref;
70        }
71
72        /**
73         * @dev Modifies the reference to the IPFS in the mapping position of the device
74          that has used the method
75         * @param IPFSref new IPFS reference
76         */
77        function storeRef(address deviceID, string memory IPFSref) external onlySmartphone {
78            require(whiteList[deviceID].exists, "This plug-in is not configured.");
79            whiteList[deviceID].IPFSref = IPFSref;
80        }
81
82        /**
83         * @dev Returns IPFS reference
84         */
85        function getRef(address deviceID) external view onlySmartphone returns (string memory) {
86            require(whiteList[deviceID].exists, "This plug-in is not configured.");
87            return whiteList[deviceID].IPFSref;
88        }
89
90        /**
91         * @dev Modifies the data contained in the temp field.
92         * @param newTemp new temporal value
93         */
94        function modTemp(bytes calldata newTemp) external onlySmartphone {
95            temp = newTemp;
96        }
97
98        /**
99         * @dev Returns temporal value
100        */
101       function getTemp() external view returns (bytes memory) {
102           return temp;
103       }
104
105       /**
106        * @dev Returns the smartphone's account address.
107        */
108       function getSmartphoneID() external view returns (address) {
109           return smartphoneID;
110       }
111
112       /**
113        * @dev Returns the stored public key
114        */
115       function getPublicKey() external view returns (bytes memory) {
116           return publicKey;
117       }
118
119       modifier onlySmartphone() {
120           require(msg.sender == smartphoneID, "Only the smartphone "
121           "is allowed to do this action.");
122           _;
123       }
124   }
```

Code 7: SKM SC contract in Solidity.