Patterns for Anonymity Enhancing Cryptocurrencies Non-Custodian Mobile Wallets

Francisco Gindre, LIFIA, Universidad Nacional de La Plata
Matias Urbieta, LIFIA, Universidad Nacional de La Plata
Gustavo Rossi, LIFIA, Universidad Nacional de La Plata

Since their appearance in 2009, the use of cryptocurrencies has been growing constantly in terms of market cap and adoption. This boom is publicly visible as well as the grand majority of the decentralized finance transactions. Despite the use of advanced cryptography, privacy in the "crypto world" is relatively low, with certain exceptions: Privacy Coins (or Anonymity Enhanced Coins, AEC). Studies show that adoption is growing steadily on younger generation of users mostly through mobile devices and applications. This work focuses on patterns for developing mobile wallets for AECs, analyzing the cryptocurrencies Monero and primarily Zcash, taking the latter as study case. Its contributions are four design patterns that capture functional and non-functional requirements to develop a non-custodian privacy coin mobile wallet and a reference architecture that addresses these requirements in an abstract manner.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**] Software Architectures–Patterns

General Terms: Cryptocurrency Non-Custodian Mobile Wallet

Additional Key Words and Phrases: Anonymity, Privacy, privacy coin, mobile, wallet, architecture, cryptocurrency

ACM Reference Format:

Gindre, Francisco and Urbieta M. and Rossi, G., 2022. Patterns for Anonymity Enhancing Cryptocurrencies Non-Custodian Wallets. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 22 (October 2022), 29 pages.

1. INTRODUCTION

The paper published under the pseudonym "Satoshi Nakamoto" [Nakamoto 2009] is considered the origin of Bitcoin. Since then, many other cryptocurrency protocols were created and deployed with their corresponding decentralized peer-to-peer (p2p) network of validators, relayers and miners. Many years passed until the privacy question was discussed in the public "crypto" debate. The matter of preserving users' privacy presents challenges at every level of the crypto ecosystem and mobile wallets are not exempt from them. This paper will discuss the topic of privacy and anonymity enhancing cryptocurrencies and present four design patterns related to implementing non-custodian mobile wallet applications for privacy coins. It is assumed that readers have a basic degree of knowledge of cryptocurrencies, however there is an appendix offering a level setting discussion on the matter.

Privacy and Cryptocurrencies

Although all cryptocurrencies can offer some degree of privacy and anonymity, there are certain characteristics that indicate that some provide higher privacy and anonymity than others. The advantages of cryptocurrencies

Authors' contacts: Francisco Gindre, email: fgindre@lifia.info.unlp.edu.ar; Matías Urbieta, email: murbieta@lifia.info.unlp.edu.ar; Gustavo Rossi, email: gustavo@lifia.info.unlp.edu.ar.

that potentially provide transaction privacy and anonymity are succinctly described in “CryptoNote v2.0” from Monero Labs [Van Saberhagen 2013]. It also underlines that a “Privacy Coin”, has two fundamental properties: untraceability and unlinkability. The former meaning that given an incoming transaction all possible senders are “equiprobable”. The latter implies that for any two outgoing transactions it is impossible to prove they were sent to the same person. In a more general way, privacy in distributed ledger currencies implies that only the involved parties can learn about the graph that a set of transactions create. We consider that these “involved parties” are entities who are in possession of keys involved either in the sending or receiving side of a transaction. When privacy holds, from a receiver point of view, only someone who possesses viewing keys for incoming transactions can decrypt those transaction’s outputs and learn about funds sent to addresses that belong to those keys. That also has to stand for the sending side, where only those who posses keys that can detect outgoing payments can decrypt them, being the resulting graph unknown to eavesdroppers. Privacy Coins are also defined as Anonymity Enhancing Cryptocurrency or AEC. We will use these two terms indistinctly.

In a transparent blockchain like Bitcoin’s, an incoming transaction can be identified by deriving addresses from the user’s keys and linearly comparing those values with the fields present in the transactions committed to the blockchain [Antonopoulos 2017]. Blockchain explorers are websites dedicated to index and accumulate blockchain information to make it publicly available and efficiently consulted with queries. A good exercise to learn about information visibility on different protocols is to browse transactions on these explorers and reflect on what is publicly available and what can be learned from transaction details.

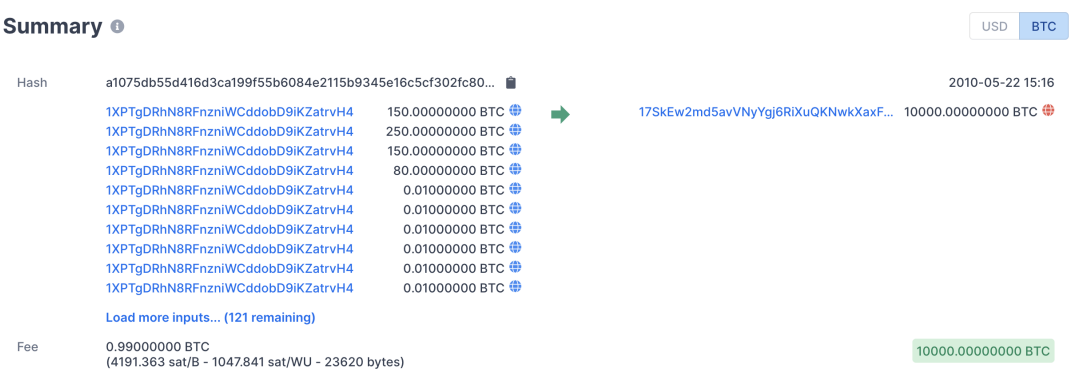

Fig. 1: Bitcoin Pizza Transaction

Figure 1 shows a screenshot of a Bitcoin block explorer site for the transaction detail of the most famous Bitcoin transaction: The “Bitcoin Pizza” [Hanyecz 2010]. Leaving the story behind it aside, this bitcoin transaction offers a great insight on how much information can be learned from a transaction in an open ledger blockchain. The fact that it can be pinpointed proves that the transaction is not “unlinkable”, moreover anyone can browse the addresses involved an learn other transactions performed before or after it at plain sight, ruling out the “untraceability” property as well. Achieving these two properties is a complex problem which, admittedly by Bitcoin’s creator itself, had no apparent solution back in 2010. Figure 2 captures a forum post from Satoshi itself where the problem, its challenges and possible solution to it are explained succinctly.

In 2013 The “Zerocoin” paper [Miers et al. 2013] proposes the possibility of making a Bitcoin transaction private and its extension Zerocash [Sasson et al. 2014] approaches that with a cryptocurrency protocol based on the privacy of transactions between peers for all parties involved. Zcash is a Bitcoin Fork that achieves transaction *unlinkability* and *untraceability* thanks to the use of Zero-Knowledge Proofs that allow its users to transact without

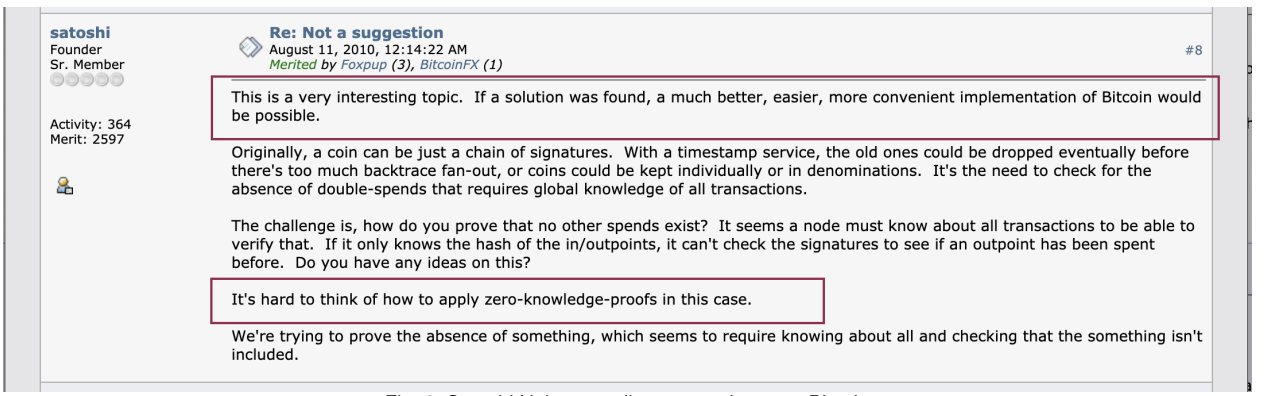

Fig. 2: Satoshi Nakamoto discusses privacy on Bitcoin

revealing information about inputs and outputs if they decide they want to make those private. It is considered an inflection point in the privacy space since Zcash is the first production-ready application and deployment of such ZK-Proofs at a large scale.

On AECs the blockchain remains being public and it is as well stored completely on full nodes of the p2p network. It remains being fully verifiable, but unlike Bitcoin or Ethereum [Wood 2014], thanks to *untraceability* and *unlinkability*, they provide a higher grade of privacy on transactions and the graph that these conform.

In privacy coins like Monero [Monero Labs 2022] or Zcash [Electric Coin Company 2022] transaction information is, at most, partially available or totally unavailable to the public. But what does it mean in terms of their users? The following sections will describe the subject of cryptocurrency wallet application, their responsibilities and effects that *unlinkability* and *untraceability* properties of a blockchain have on them.

What is a wallet?

A “Crypto Wallet” can be described as a representation of the blockchain from the point of view of a set of public and private keys. This means that a wallet is composed of two main things: cryptographic keys and the blockchain data related to those keys. The main objective of a wallet application is to stay up-to-date with the blockchain and keep track of information related to keys associated to it, and consolidate it for the user in a human-friendly fashion. That is a broad specification that defines every cryptocurrency wallet and it can be broken down further.

Types of wallets

When we defined what a wallet is, we did so in terms of two specific things: Keys and blockchain information. In those terms, wallets can be classified by answering two basic questions: Who is in custody of the keys, the user or a third party? How is the blockchain accessed, does the wallet download the whole blockchain or does it rely on a third party (client-server)? Table I shows how these two questions lay down different wallet variants.

Wallets and custody of the keys

A custodian is an entity that takes custody of a certain asset on behalf of another person or entity. This role is not new and it can be easily related to organizations offering safe vault facilities to their customers. While anyone could just have a safe vault at their home, the custodian is delegated the infrastructural burden of what the custody entails in exchange for a service charge. Those who delegate the custody of an asset do it at the risk of knowing that the custodian might deny the access to it arbitrarily.

The counterpart would be a non-custodian scheme where the owner of the asset makes sovereign custody of it. This eliminates the risk of the custodian taking control of the asset at the cost of being the owner responsible to

		Who is in custody of the keys?	
		User	3rd Party
Blockchain access	Access as peer	**Non-Custodian Full-node wallet** User hosts its own node and uses its wallet implementation.	User has a **custodian service** to access keys through a multi-signature scheme while retains the capability of accessing the blockchain
	Through 3rd Party or Server	**Non-custodian light client.** User uses a wallet application that holds custody of the keys while it delegates access to the blockchain to a server or a node hosted by a third party or the user itself. **This is the subject of this paper**	**Custodian light client** User delegates custody of keys and blockchain to a third party that gives privileged access once it has proved the identity of the user. **This is what centralized exchange entities are.**

Table I. : Wallet types taxonomy based on Custody of Keys and access to the blockchain

guarantee that the asset is safe and only accessible when needed. For example, a safe vault behind a disguising furniture at the owners' home.

The same analogy applies to the keys that have spend authority over cryptocurrencies. Users can delegate the custody to organizations like centralized cryptocurrency exchanges or cryptocurrency custodian firms. They can also choose to hold the keys themselves following the "Not your keys not your coins" philosophy. To be able to do that, the wallet application needs to provide a set of features that allows the user to hold its keys in a secure container that encrypts them and guarantees that no unauthorized access to them occurs.

How the wallet accesses the blockchain data

Cryptocurrencies are peer to peer networks that run over a consensus algorithm. Each peer is a server called "Node" which holds an entire copy of the blockchain at the same time that verifies and relays new blocks generated by miner nodes (for the case of Proof-of-Work consensus). If a node additionally holds a set of keys and runs a wallet application, that wallet is known as a Full-Node wallet. This kind of wallet has considerably big hardware requirements since it needs to be able to run its own node locally. Wallet applications that don't run their own nodes and rely on an intermediary server to access blockchain data, usually do so to constrain hardware requirements at the expense of trusting a server that carries that burden. This kind of wallets are called **light clients**. Light clients can be desktop, mobile or browser based applications. This paper studies non-custodian light clients supporting privacy coins.

Responsibilities of Non-custodian AEC mobile wallets

The base requirement of a wallet of any kind for a given cryptocurrency is to comply with the corresponding protocol specifications. Our analysis covered wallets that were officially advertised in both Zcash and Monero websites. They are all aesthetically different and have their own look and spirit, but with the core commonality of having to abide by the Zcash [Hopwood et al. 2020] or Monero [SerHack 2018] protocols. The wallets reviewed were ZecWallet Lite [Kulkarni 2020b], ECC Wallet iOS and Android Company [2020; Electric Coin Company and Gorham [2020], Unstoppable Wallet Horizontal Systems [2021d; Horizontal Systems [2021c], Zcash Mobile SDKs Electric Coin Company [2019a; Electric Coin Company [2019c] for Zcash, and Cake Wallet Cake Technologies [2020; Cake Technologies [2018] and Monerujo [Monerujo Team 2020] for Monero. The analysis covered their source code, documentation, the present use cases on released application builds and the available user stories on the repositories. Also a manual side by side comparison of each one of them was performed to compare the use cases and usability of each one of them. We could distinguish some themes around the user stories that

conform these wallets which are management and handling of users' private and public keys; wallet operations and state of the wallet. In terms of keys, the wallet is responsible of generating new keys (the wallet itself), safe storage and facilitating backup and restore of the existing wallets. When a user creates a new wallet, random bytes would be generated and then turned into some kind of human friendly format that avoids users having to manipulate or write down representations of bytes. All of the surveyed wallets made use of mnemonic seed phrases which are a way to represent bytes (and a checksum) by mapping them to a dictionary of curated words in a familiar language that users can write down (See appendix 8 and 3 for more details). These phrases support the backup and restore of the bytes that the wallet is derived from. Complementary, applications must provide secure storage of these bytes in order to spend funds. There area known cases of wallets applications that are "View Only", often targeting use cases were users are delegated the task of receiving funds but must not have spend authority (like cashiers at a store). We treat those wallets as a subset of the ones under study on this paper. Wallets also need to provide certain basic operations which are receiving and sending funds. Receiving funds was found to be either by sharing addresses encoded in text or QR codes while spending funds was either found to be performed by a single or multiple steps form that needed the user's input of amount and recipients. To acknowledge these operations the wallet must stay up-to-date with the blockchain by being synchronized with the latest ongoing events as well as prior ones. Synchronization is the most important operation of a wallet application and it is the one that actually processes the information that conforms what the user visually perceives as the wallet application state. The state does not only refer to the status of the application which can be resumed by the sum of connectivity states and syncing (in progress or synced). The wallet state also refers to wallet's balance and the transaction history that composes it. Wallets must display the available and pending balance as well as the transactions that have been performed. Transactions are often split into received and sent. Both kinds of transactions can be confirmed or unconfirmed (also referred as 'pending'). Wallets have their own confirmation practices to consider that funds are certainly spent or received by counting how many blocks have passed from the one that included them on the blockchain's ledger. A further discussion can be found on the Wallet Synchronizer pattern section 10. These are the main responsibilities of wallets. Application will likely differ on how they implement them in terms of aesthetics and user experience. We have left these aspects out of the scope of this research.

How do Privacy Coins affect non-custodian mobile wallet requirements?

The section "Privacy and Cryptocurrencies" 1 discusses the most notable differences between public ledger transparent cryptocurrencies like Bitcoin or Ethereum [1] and anonymity enhancing ones like Zcash and Monero. The most notable being the fact that privacy can't be preserved if the wallets don't rely on themselves to process the blockchain and detect its own transactions. Every transaction must be trial-decrypted. This means that the wallet needs to attempt to decrypt the information in order to find out whether is intended for it or not. Failing to do so it's not considered an error, it means that probably the keys provided for decryption don't belong to the keys that encrypted it. This implies that wallet applications require a greater computing and storage capabilities just for the "passive" act of receiving transactions from a node that also plays the role of a "Server" for mobile clients (light clients). Mobile wallets are a specific kind of "light client". Desktop applications that don't run their own full-node and rely on other services to provide access to the blockchain are also considered light clients and therefore could be subject to the same patterns discussed in this paper.

Besides receiving transactions being more cumbersome, to generate a transaction to one or more recipients, there are extra requirements than those on their not-private counterparts. Zcash relies on Zero-Knowledge proofs (or ZK-Proofs) to preserve the information that otherwise would be public in protocols like Bitcoin or Ethereum. ZK-Proofs are a breakthrough in the privacy and cryptography space. They provide a way to prove information to

[1] Some cypherpunks refer to them as Surveillance Enhancing Cryptocurrencies (SEC) considering them a vehicle to mass surveillance of the crypto-economic ecosystem.

others without revealing its contents. For it, the state of the blockchain must be computed with the users' keys so that they can generate proofs that verifiers (observing the same blockchain) can agree on their soundness.

Another key factor that differentiates AECs from other cryptocurrencies is how they rely on the public ledger. Interacting with other peer nodes over public networks discloses metadata that otherwise would be private. Transparent ledger protocols like Bitcoin or Ethereum assume that every bit of information will be publicly persisted on the public ledger that is the blockchain. This is an assumption that influences how their wallets operate. Not having to factor in transaction (or graph) privacy means that a wallet cannot "leak" this information, since it is publicly available on the blockchain. When designing an interaction, developers can count on the fact that a lot of information will become public anyway, so there's no point on making efforts to make it private. For example, an Ethereum wallet can delegate processing the blockchain to a third party with little trade-offs. When an Ethereum wallet application needs to know of the transactions belonging to the user's keys, it has to query information that is already on the public domain, and it may do so by requesting that information to nodes that index the blockchain to respond to such queries. On the contrary, privacy coins would do this at much higher stakes. Mainly because it means telling a server to bring specific transactions o handing over keys that allow to decrypt information (but not spend) to "outsource" the trial-decryption effort, which allows that server to learn about the transaction graph of those keys. Something similar happens with custody of keys. Users can use custodian services to hold their coins for them. Although privacy would be lost since the custodian will learn every bit information related to those keys, Anonymity Enhancing Cryptocurrencies can't rely on custodians without losing their core purpose: Privacy.

Cryptocurrencies, wallets and user adoption.

Since their creation, cryptocurrencies have been presented as the opportunity for the average person to "be their own bank". Although it might have been far fetched in 2009, nowadays mobile banking adoption has been growing up to unexpected levels. The youngest generation of users operates with online banks naturally according to an article on Forbes [Shevlin 2021] showing that the penetration on Generation Z is 95%. As stated in *Crypto Pulse Report* [Bitstamp 2022] this generation is also the one who best uses cryptocurrencies. In the same way "Moneywork" around traditional finances poured into digital payments and online banking [Perry and Ferreira 2018], it could be expected that will also crypto-payments. However, this has its consequences. The fast pace of this industry plays against in-depth software engineering review and research on the field's best practices. Underestimating wallet engineering and poor design could lead on "Crypto Crime" as it was advised by wallet screening in Chainalysis report of 2022 [Chainalysis 2022]. Software errors at the end-user level in traditional centralized finance are highly recoverable when compared to the same kind of errors on a decentralized, sovereign custody of funds scheme. In a centralized financial system the trusted authorities can revert operations and reset passwords to protect their users. The opposite holds on a decentralized, self-custody financial system: loss-of-funds is an inherent risk at the tip of the users' fingers. This is where software engineering must come to aid to provide tools to identify and solve common problems and lower these risks.

In order to focus on presenting the patterns, several topics like wallet User Interface, User Experience, mobile development platform specific discussions and security implications were left out the scope of this paper. We advice readers implementing wallets to research their domain's threat models and security best practices. As is their nature to handle funds, wallet applications are more likely to be targeted by attackers than other applications and this fact should be factored in the threat model developers choose for their applications.

In the following section we will describe an overall architecture that will present the proposed patterns and their composition and interaction in a high level diagram. Afterwards, each one of the four proposed patterns will be described on their own dedicated section. These patterns condense complexity of the application domain in succinct interfaces and actors implementing them. We will present them following the "Gang of Four" template [Gamma et al. 1995] detailing intent, motivation, applicability, structure, participants, collaborations, sample code, known uses and related topics. We chose this format for it being widely adopted in non-pattern-niche environments and also being welcoming of contextual information that led to the identification and description of the pattern in

question as well as implementation details. A summary of the patterns presented and their intent can be found on table II.

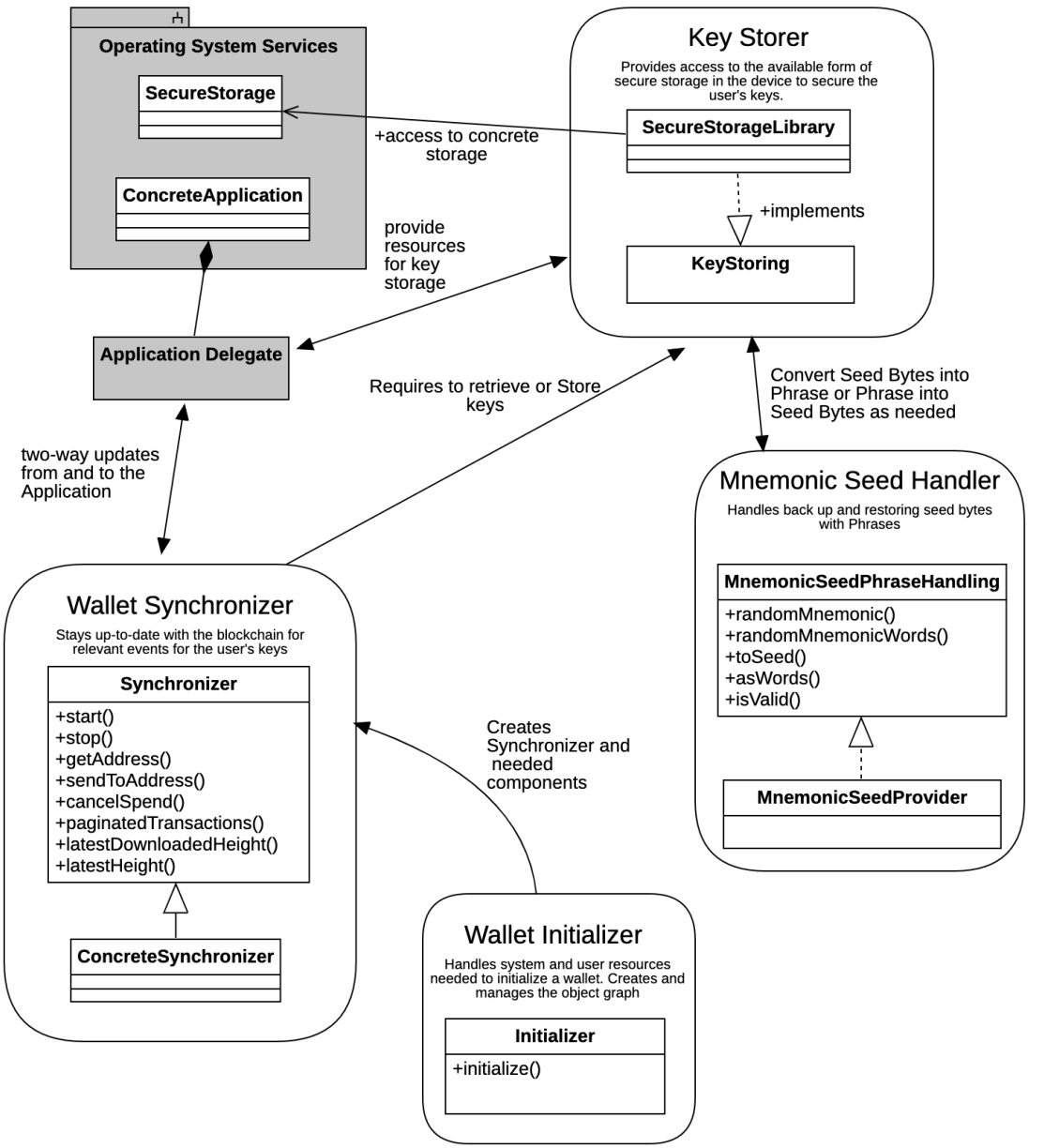

Fig. 3: Overview of how the proposed patterns play their role in the application's architecture

Pattern Name	Intent
Mnemonic Seed Handler	Encapsulates the logic behind handling seed bytes from and into mnemonic seed phrases from the wallet application logic.
KeyStorer	Provides an abstraction that describes the behavior required to store private keys securely on the user's device.
Wallet Initializer	Encapsulates and abstracts the complexity of starting up a wallet and all the needed resources
Wallet Synchronizer	Implements the functional and non-functional requirements to keep the wallet up-to-date with the blockchain

Table II. : Summary of the patterns presented in this paper

2. OVERALL ARCHITECTURE

The patterns we will present can be integrated altogether in an overall reference architecture for developing non-custodian AEC mobile wallets (or light clients). Figure 3 shows how these patterns will interact and collaborate at a high level. As a starting point, we situate the application delegate, which represents the entity that the operating system uses to delegate control to the application's code. Networking services are excluded since it is assumed that the blockchain is always accessed through them. This delegate is usually holding references to resources from the operating system such as connection pools, files on external storage or inside the application's sandbox. The OS's resources are places to the left of the figure. It is important to note that the application delegate is tied to the application process and once that process is terminated for whatever reason, the delegate will be too. The application uses *KeyStorer* to store or retrieve the user's keys. The absence of keys is considered as the application not being yet initialized. Once the user creates or imports a Seed Phrase with *MnemonicSeedPhraseHandler* it can be stored on the secure storage *KeyStorer* implements. If there are no errors, the *Initializer* is created to allocate the needed resources to create a *Wallet Synchronizer* and start syncing the wallet. The application delegate must also forward Operating System events to the *Wallet Synchronizer* and act accordingly depending on the events received. This architecture shown in figure 3 is abbreviated so that it focuses on the patterns. It leaves out many implementation details that change depending on the Operating System and the programming paradigms chosen by developers. Developers leaning to Reactive Functional Paradigms like RxJava, Co-Routines and Flow for the case of Android, or Rx-Swift and Combine for iOS will use publishers and subscribers for many of the attributes. We use Object-Oriented as a "common language" to communicate and describe the patterns and the suggested reference architecture, but it is not a requirement for its implementation.

3. MNEMONIC SEED HANDLER

Intent

This pattern abstracts the logic behind restore and generation of a sequence of randomly generated bytes (called "Entropy Bytes") from and into mnemonic seed phrases from the wallet application logic.

Motivation

A non-custodian wallet must be capable of creating, validating and restoring a sequence bytes that will originate the private and public cryptographic keys that conform a wallet. Private and Public keys that conform wallets are created from a sequence of randomly generated bytes. These bytes are picked from a search space broad enough to make collisions probabilistically not feasible. This is important since whomever controls a set of keys, controls the coins associated to them. Users must be guaranteed that others can't replicate keys by trial and error or other kind of attack vectors leveraged by an adversarial. Although the bytes are (or at least should be) generated by randomness, the keys derived from a set of bytes are deterministic. To regenerate its keys, users would have to hold on to the generated bytes. The breath of the randomness of the keys is one of the forces at play. While it mostly assures improbable odds for others to generate the same keys by intent or accident, it also makes these entropy bytes impossible to recover when lost partially or completely.

In addition to this, randomness represented in the shape of bytes is a foreign element to (human) users. Users might find them hard to remember, easy to tamper and to make mistakes while transcribing them, backing them up or restoring them into a wallet. Failing to input or back up keys correctly can mean that the user loses access to its keys and the funds associated to them. Translating these bytes into a format that is more familiar to human users has been a way wallet developers found useful to avoid loss of funds. There are many approaches to this problem. Mnemonic phrases is one of them. The mechanism consists on defining a dictionary of words (in English, Spanish, or any other written language) that can be used as a lookup table to represent a 2-way street to go from a word to a set of bytes.

Mnemonic Phrases used to represent entropy bytes that seed a set of keys, are called mnemonic seed phrases. They are only a convention on how to restore seed bytes used to derive the users' public and private keys. At the moment, BIP-39 mnemonic phrases [Palatinus et al. 2013] are a *de facto* standard to achieve this.

There are different conventions to generate and restore bytes with seed phrases. Their proliferation stopped once BIP-39 was defined but many still coexist. Hardware wallets manufacturers like Trezor® or Ledger® display warnings about only supporting restoring wallets with phrases generated with their own hardware beside them using the bitcoin standard. By reviewing different implementations of libraries implementing this BIP, we could gather a small set of requirements that these had in common. A library implementing BIP-39 must be capable of generating a random mnemonic phrase as String or Vector of words, but also to generate the seed bytes from a phrase and validate all of these accordingly. There are many implementations of BIP-39 in different languages. A compendium of them is maintained and curated by the authors of the BIP in the section section "Other Implementations" of the Bitcoin Improvement Proposal 39 [Palatinus et al. 2013].

Wallets might require to support many implementations of these libraries depending on different factors, those being requirements imposed by different cryptocurrency protocols that the wallet supports or limitations imposed by the concrete platform that the application is running on. This pattern provides a common set of requirements to meet while leaving implementation details to be handled depending on the particular scenario

Applicability

Although other mechanisms are in discussion and experimentation, currently, mnemonic seed phrases are the most popular one to generate and restore Seed Bytes on non-custodian cryptocurrency wallets. The proposed interface gathers these requirements hiding the complexity of generating such phrases and bytes and delegating it to the implementing component regardless it being developed in-house or by a third party.

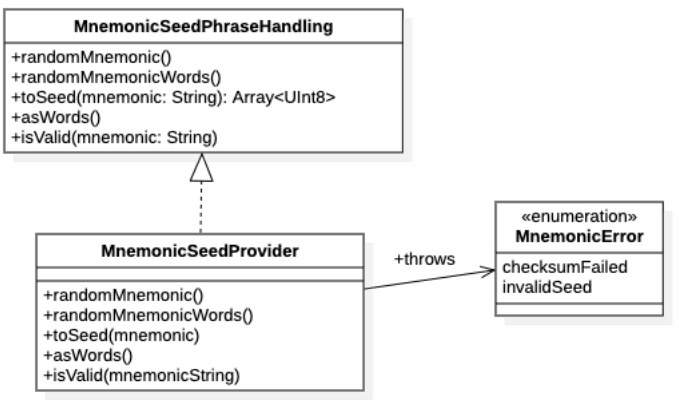

Fig. 4: Mnemonic Seed Handler: handling mnemonic phrases.

Structure

Figure 4 illustrates the pattern involving handling Mnemonic Phrases.

Participants

MnemonicPhraseHandling. The interface that captures the requirements to handle mnemonic seed phrases. It provides the ability to generate random bytes, represent them as a phrase (*as words*), conversely a bytes into a phrase and validating an existing phrase provided by user input.

MnemonicSeedProvider. The implementation of the interface. This could act as an *Adapter* when the library is provided by a third party.

MnemonicError. The possible errors derived from the requirements. The error *checksumFailed* refers to the checksum bytes contained on the phrase don't match the expected according to BIP-39. *InvalidSeed* signals that one or more of the provided words are not defined in the word dictionary.

Collaborations

This pattern can be used along with *KeyStorer* 4 when restoring a wallet from an existing mnemonic seed phrase backup. Figure 5 describes this collaboration assuming a mobile wallet application that only handles a single seed phrase. The use case corresponds to a user being in custody of its own keys in the form of a BIP-39 mnemonic seed phrase and a wallet birthday indicating the block height of the blockchain tip at the time the wallet was created. The wallet application will assume there are no transactions of interest on blocks prior to that given height. The user faces the application that has no keys stored and is shown a "Restore Wallet from Seed Phrase" button. When tapping the option the Wallet application provides the user the means to input the seed phrase. Some applications may provide a simple text field with an overall validation of the seed phrase and others might have more elaborate user experiences that take advantage of the word choices of the BIP-39 dictionary to have a safer, easier and more reliable input. Step four and five of figure 5 abbreviates the fact that there is a validation loop until a valid phrase is input which is then is converted to bytes. Those bytes may be atomically saved along the wallet birthday, but it could be the case that the user does not have the block height. In that case the wallet must start scanning from the initial block height. There's another collaboration for the case the user proceeds to create a new wallet instead that is covered on this section of the KeyStorer pattern.

Consequences

Application developers face the "Buy or Build" dilemma on every dependency they have to bring into their project. Implementing something like BIP-39 to create seed bytes and mnemonic phrases might be a fun programming challenge. Its specification is succinct and clear which makes it appealing to choose the **Build** path over the **Buy** path. Although this decision must be made with special care. While mnemonic phrases could be seen as "Yet Another Dependency", they are a critical piece on the wallet's architecture, a fundamental requirement and finally a security concern to be watched very closely. Any mistakes incurred in the implementation of this dependency will likely represent a «loss-of-funds» risk for users. When taking the *Build* route, developers must make sure others review their code, focusing specially on getting input from security and cryptography specialists. When taking the «Buy» path, developers must consider similar aspects as above. Being able to comply to the proposed interface of this pattern is a way of running a checklist over the basic requirements for a non-custodian wallet application but it is not sufficient. Developers should make sure that the dependency they will wrap around with the *Mnemonic Seed Handler* pattern is thoroughly audited, recommended by experts (and better if referenced by BIP-39 authors and maintainers) and open source. When possible should be built from source or included by using other mechanisms that ensure builds are both reproducible and auditable. Another factor to consider when choosing a library, is whether the concrete implementation of choice is already adapting another library. This will help avoiding an "Adapter Chain". Lastly, when different phrase conventions co-exist within the same application

(for example, a Monero and Zcash non-custodian wallet) this pattern could provide the false sense of uniformity and interchangeability of phrase implementations when it is not desirable. Developers must be then aware of the concrete implementations to properly differentiate one another at runtime.

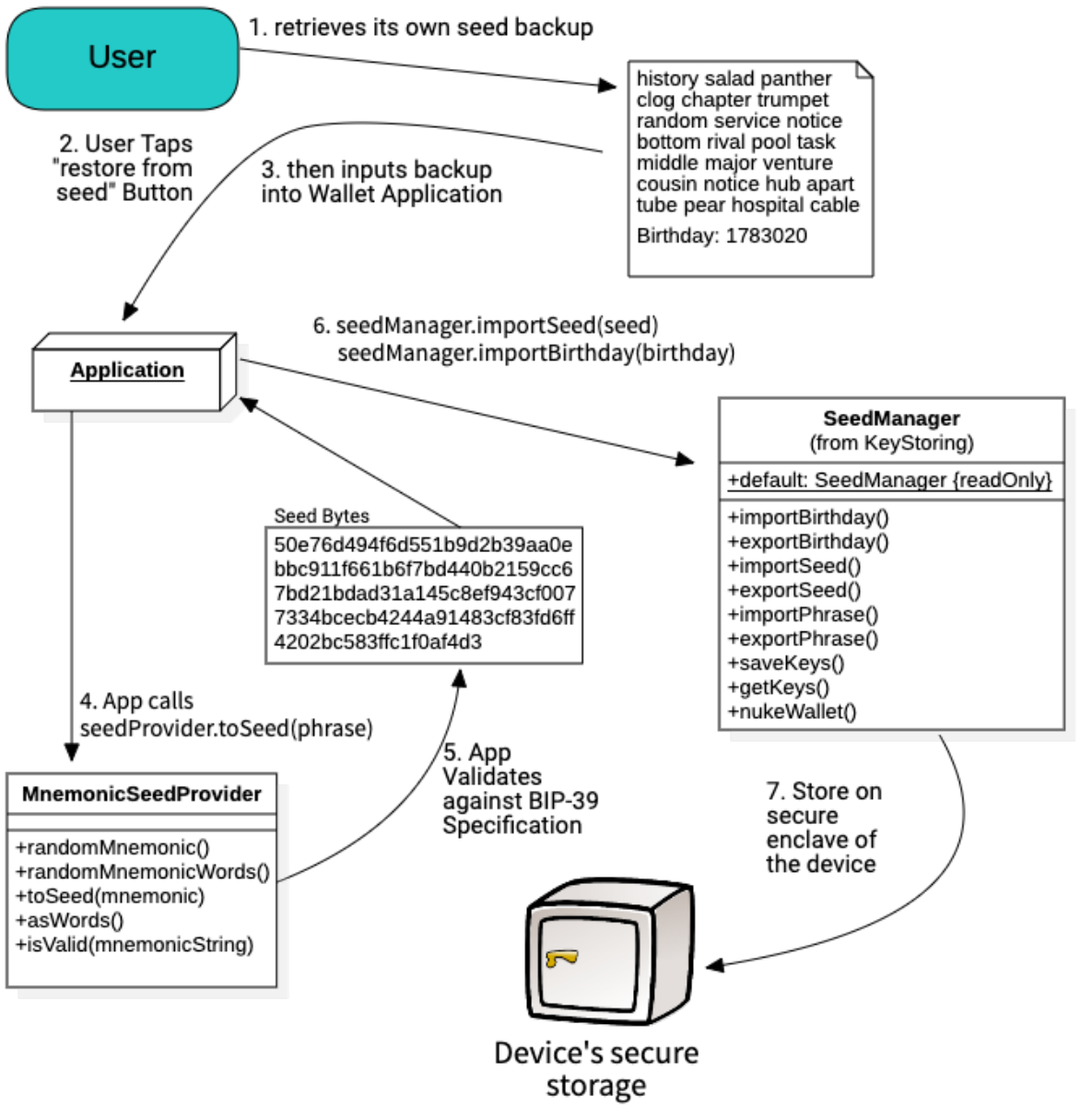

Fig. 5: Collaboration schema between Key Storer and Mnemonic Seed Handler when restoring a wallet from an existing Seed Phrase.

Implementation

The library *MnemonicSwift* [Electric Coin Company 2020b] implements BIP-39 using this interface.

Sample Code

Sample implementation can be found on Zcash's ECC Wallet source code [Electric Coin Company 2020a] which defines the interfaces and also implements them.

Known Uses

Mnemonic seed phrases can provide a false impression of standardization and uniformity, whereas there a guiding principle (the phrase) but then its interpretation can produce different outputs. Popular wallets as MetaMask use 12-word seed phrases, Ledger® or Trezor® wallets use 24-words and the "Mastering Monero" book defines a 25-word phrase. All of these produce different bytes but the users could not notice the difference between them. Multi-coin wallets like Unstoppable wallet for iOS and Android handle mnemonic seed phrases with this pattern to abstract this complexity away from users.

Although different ways of representing these phrases have been found. There are other kinds of restore mechanisms such as "Paper wallets" based on QR codes with the entropy bytes, or the novel "social recovery" which is based on users defining people they trust as "guardians" of a wallet. Guardians have a collateral custody of the keys in question. They own a partial representation of those keys that has enough information to assist the owner of the wallet to restore them but not to do so autonomously without the consent of the person that appointed the guardian. The Ethereum wallet "Argent" was one of the first wallets to use this approach and deploy it on production. Additionally, developers of this wallet implemented transaction thresholds that the user could set up so that if anomalies were detected wallet guardians could be notified. This intends to make it easier for users to find these guardians, because the wallet would let the know if something strange is happening instead of them needing to be experts on the matter to notice.

Related Patterns

This pattern works closely with the pattern Key Storer 4 and its implementation details can relate to:

—Adapter, when the wallet will rely on a specific implementation of the BIP-39 and there are differences to reconcile between the API provided by that component and the one that the application will use

—Proxy, when different implementations of the Mnemonic phrases need to coexist in the same application interchangeably depending on environment conditions or other factors.

4. KEYSTORER

Intent

Given the different secure storage capabilities and APIs present on a family of supported devices, provide an abstraction that describes the behavior required to store private keys securely on the user's device.

Motivation

The most evident objective of a non-custodian wallet is to store users' keys in an encrypted fashion by using a form of «secure enclave» or encrypted storage. Although most cases will rely on secure storage mechanisms provided by mobile operating systems (this is the case of the Overall Architecture presented in figure 3), developers might implement and provide their own. Whatever the case, the main goal is to provide access to store and retrieve the users' keys securely when required, allowing them the entitlement of the sovereign custody of their own cryptographic keys. Additionally, sovereign custody also entails the ability of destroying such keys. Depending of the application domain this type of interfaces can have different hierarchies. For the case of a *non-custodian* wallets, the sovereign custody of value is their core function. A cryptocurrency wallet can be seen as an interpretation

of a blockchain from the point of view of a set of public and private keys. There is no wallet without keys. This makes what it would be a non-functional requirement in other domains, a functional and central requirement in this domain. Secure storage components come in various forms and shapes whereas their end goal is specific. Different operating systems implement it their way, or in the case of Android, even different versions have distinct capabilities over the custody of sensitive data, those being bound either to software or hardware limitations. *KeyStorer* provides the needed behaviour that *non-custodian wallets* require and delegate implementation details to other components or libraries.

Applicability

As the motivation discussed the capability of secure storage varies greatly over time as well as their interfaces. *KeyStorer* isolates this implementation changes from affecting the core functionality of wallet applications.

Structure

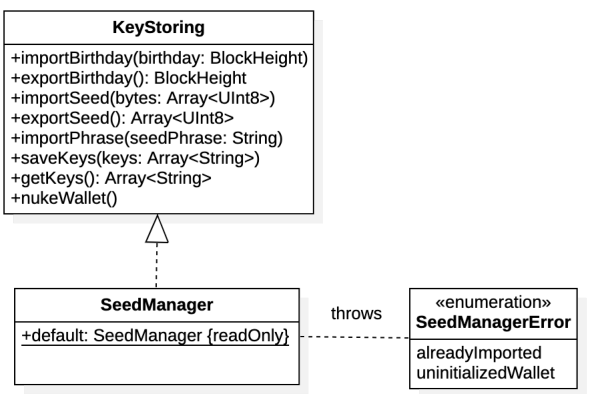

Fig. 6: KeyStorer: an interface describing the behaviour needed when securely storing keys and other sensitive data.

Participants

KeyStoring. The interface declaring the main methods needed for storing the wallet keys and birthday height.

SeedManager. Represents the concrete implementation of the interface. It communicates with (and adapts) the available (or selected) secure storage API to the requirements of a wallet defined by *KeyStorer*.

SeedManagerError. Transforms the errors of the underlying secure storage API into those that have a meaning on the wallet application domain.

Collaborations

This interface can be used with *Mnemonic Seed Handler* (described in section 3) when creating or restoring the wallet. Figure 7 describes in a simplified way the use case of creating a new wallet. This use case can relate to a wallet that only handles a single seed phrase or one that supports using many seed phrases at once. For the sake of simplicity we will refer to the single seed case. To begin using the app, the user will have to create a new wallet from a random mnemonic seed phrase by tapping a the “Create new wallet” option on the screen. This will trigger a series of events on the wallet application. The application will call *randomMnemonic* function on the *MnemonicSeedProvider* implementation which will generate new random seed bytes. Those seed bytes

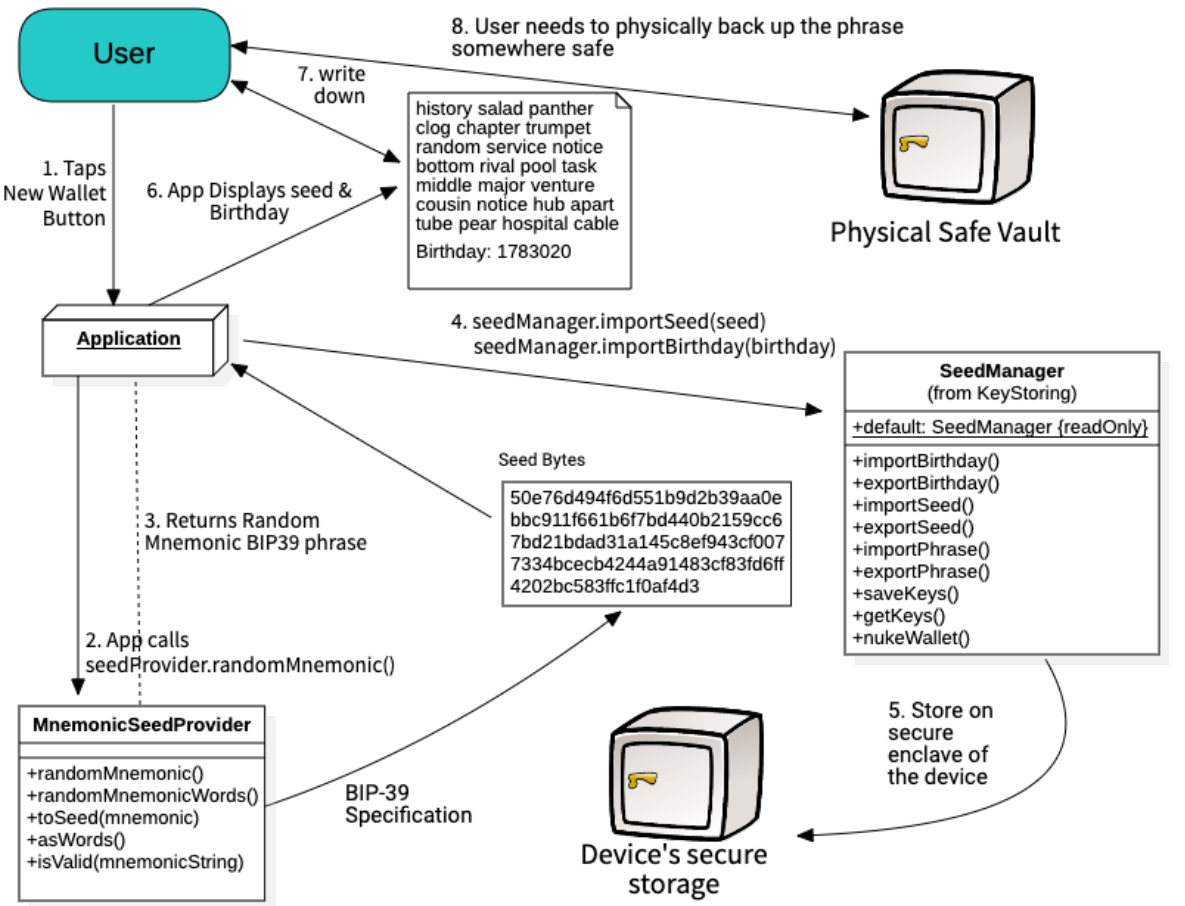

Fig. 7: Collaboration schema between Key Storer and Mnemonic Seed Handler in the creation of a new wallet.

must be stored securely on device along with the wallet's "Birthday" which indicates the current block height of the blockchain the wallet is targeting. Having the height of the block the wallet is being created at doesn't save any data on the chain itself but it will help the user when restoring the wallet on another application or devices more efficiently. It will be assumed that there are no transactions of interest before that given block height. Seed bytes and birthday must be stored on the encrypted storage of the device. The Wallet Application will call *importSeed()* and *importBirthday()* on the *KeyStorer* pattern implementation to do so. Once that operation is confirmed the application can consider that it's safe to show the seed bytes using the equivalent mnemonic seed phrase and indicated on the BIP-39 [Palatinus et al. 2013] specification. The order of these operations is an important implementation detail on the Wallet Application as they are enumerated on figure 7. If the user is shown the seed phrase before it has been successfully saved to the encrypted storage, being the case that storage operation fails afterwards, the user might have backed up a seed that is not stored on device. This would mean that the wallet might reset next time it is launched and will show another seed phrase instead, that the user might probably disregard as being backed up already and proceed to use the app without realizing the backed up phrase is not the one the application is using. Many applications feature a seed backup test that involves asking words of

the generated phrase on random indices. This kind of tests help both developers and users to be sure the keys have been backed up somewhere and that it is safe to proceed to sync the wallet and continue to its expected operation.

The number of instances accessing the secure storage could be limited used *Singleton*, or *FlyWeight*.

Consequences

Adapter interfaces have the risk of becoming an anti-pattern since the adapted interfaces can be an Adapter already. This causes an indirection layer that adds complexity and decreases traceability of code. Calling a function on one end of the adapter chain, can hop across different modules and even languages for the case that Adapted APIs involve Foreign Function Interfaces (FFI).

Implementation

An important factor of storing data on a secure storage is the lifetime and lifecycle of this information over time. What happens when the user deletes the application? Do the stored keys remain or are they deleted in cascade with the originating application? Is this information stored in system backups? How would an application re-install scenario be affected by this?

Another factor is the information to be stored. Key derivation can be computationally expensive. Systems with resource constraints must plan accordingly. Can this keys be derived on the fly or shall the storage preserve the derived results to avoid stressing the system and lagging the user interface when performing operations with the keys? The former only requires that seed bytes are stored whereas the latter has side-effects that must be handled. If derived keys are stored, they can't be undone. If there are protocol changes that affect Key derivation paths [Wuille et al. 2018][Wuille 2012], the wallets will probably lose the ability to migrate to this new derivation scheme, requiring the user to fully restore the wallet from the seed phrase that was used to create the wallet initially.

Wallet developers might choose to store other information on the device's secure storage depending on the features and User Experience they want to provide. That doesn't conflict the current specification of the pattern which only focuses on the minimum viable interface needed for the base requirements of a non-custodian privacy coin wallet.

Known Uses

The wallets analyzed as our case studies make use of interfaces (or libraries that provide them) to abstract secure key storage underpinnings from the applications' logic. Destruction of such keys was found in wallets like ECC Wallet [Company 2020] and Unstoppable [Horizontal Systems 2021d].

Related Patterns

—Adapter, when the target API that provides access to the secure storage needs to be transformed to comply with the public API.

—Proxy, create shim that exposes a desired interface that can be swapped depending on the underlying implementation or target platform

—Mnemonic Seed Handler (see section 3)

5. WALLET INITIALIZER

Intent

Provide a mechanism to encapsulate and abstract the complexity of starting up a wallet and all the needed resources for syncing it with the latest blockchain state. Limit the extension object graph and their associated entities.

Motivation

Non-custodian light clients of anonymity enhancing cryptocurrencies require a vast set of resources like databases, hosts and nodes URLs, Zero-Knowledge Proof parameters, libraries over FFIs, among others. All these resources (which could be non-functional requirements from a user-centric point of view) are used by many different entities and classes throughout the application. The *Wallet Initializer* helps to gather all this complexity in a single place while declaring them on its interface. It also brings a point for Dependency Injection of the aforementioned resources for Unit or Integration Testing and helps to avoid duplication of resources by acting as a *Visitor* parameter for other components like *Wallet Synchronizer*. For the case of "ZcashLightClientKit", the Zcash iOS SDK [Electric Coin Company 2019c] it also acts as creational actor by hosting many factory and convenience methods.

Applicability

Use this pattern when interacting with a blockchain requires many resources that are either dependent or independent of each other but yet cover a set of resources used in different parts of your application.

Structure

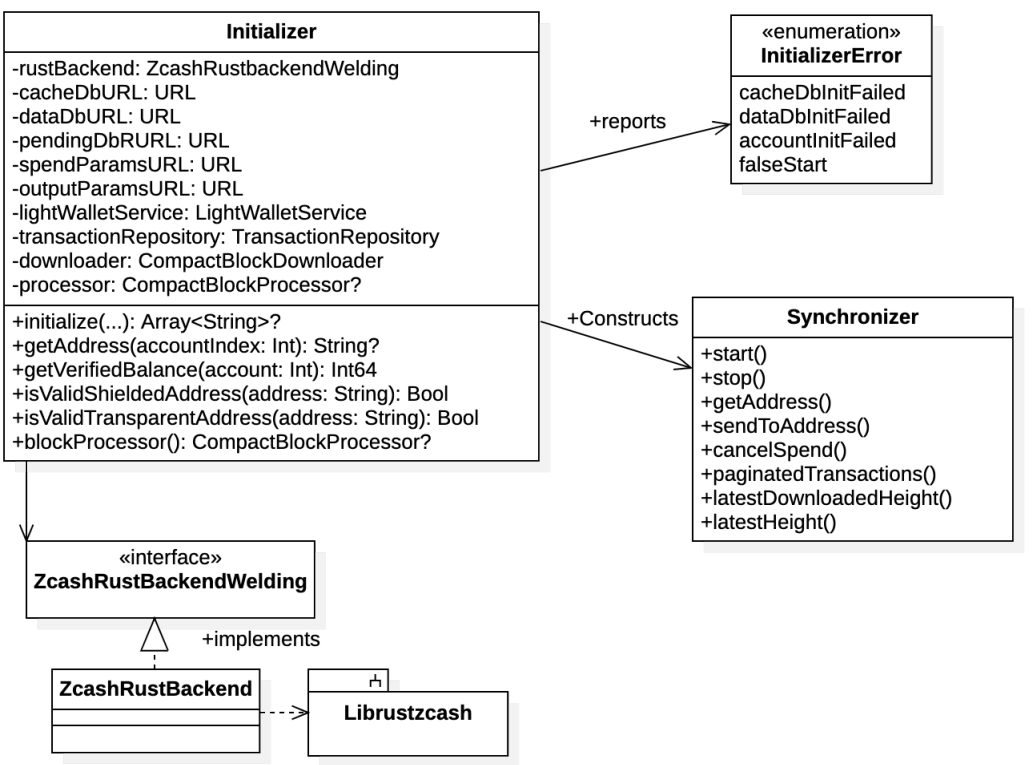

Fig. 8: Initializer on Zcash SDK: encapsulate the complexity of creating an object graph

Participants

A *Wallet Initializer* can have many implicit participants. It will deal with **System Resources** needed to sync the blockchain. It also needs to create and manage access to **Application Resources** like database files, zero-knowledge parameter files, or FFI modules that are needed to spin up a wallet. **User-provided Resources** are also managed by the *Initializer* if they are required to start the wallet up, perform database migrations or similar maintenance tasks. The *Initializer* also helps managing **Initialized Objects** which are instances that use the resources available for the *Initializer*. It can also be used as parameter or builder.

Initializer. Receives references to resources from the System, the Application environment and the user to start up a wallet. It can contain the logic needed to create other sub-components as well. Wallets backed up by SQL databases might require schema migrations, vacuum and other maintenance classes. If there are caches involved it would restore and populate them on startup.

InitializerError. The Initializer performs a lot of IO and other system critical operations that can fail and raise errors and exceptions. It should provide a abstraction over the possible errors that wallets can decode and translate into user-friendly messages.

Synchronizer. The Wallet Synchronizer can receive an *Initializer* instance as a constructor argument to obtain references to the resources it needs.

Protocol associated dependencies. The *Initializer* can be responsible or loading up dependencies like foreign function interfaces that encapsulate protocol requirements and handle that process' success or failure

Collaborations

An *Initializer* interacts with many actors. For the case of Zcash, it allows to initialize the databases used by a Rust component that is invoked through a Java or C FFI. It also interact with the *Wallet Synchronizer* by providing the needed resources for its construction. This Rust component is an FFI layer that leverages several features from *librustzcash* [Grigg and Others 2019] which is series of Rust crates that contain core logic of the Zcash protocol and implements many functionalities such as key derivation, cryptography, proof creation and verification or data storage APIs.

Consequences

The mere existence of the *Wallet Initializer* evidences the complexity of connecting a light client application to a decentralized protocol of privacy cryptocurrency. One of its potential side-effects is becoming a "God Object". Further reads about this topic can be found on chapter 3.3 of [Riel 1996]: "The God Class Problem (Behavioral Form)".

Implementation

Figure 9 shows an abbreviated diagram of the initialization sequence of the Zcash SDK. The many lanes depict the variety of objects that the *Initializer* interacts with and which are needed by light client applications to transact on the Zcash blockchain.

Wallet App. represents the wallet application that depends on ZcashLightClientKit.

Initializer. Instance of that class.

BlockStorage. Implements a Data Access Object interface in charge of persisting compact blocks on disk according to the ZIP-307 light client protocol [Grigg et al. 2018].

LWDService. Interface that handles the connection with *Lightwalletd*, light client server that implements the light client protocol ZIP-307.

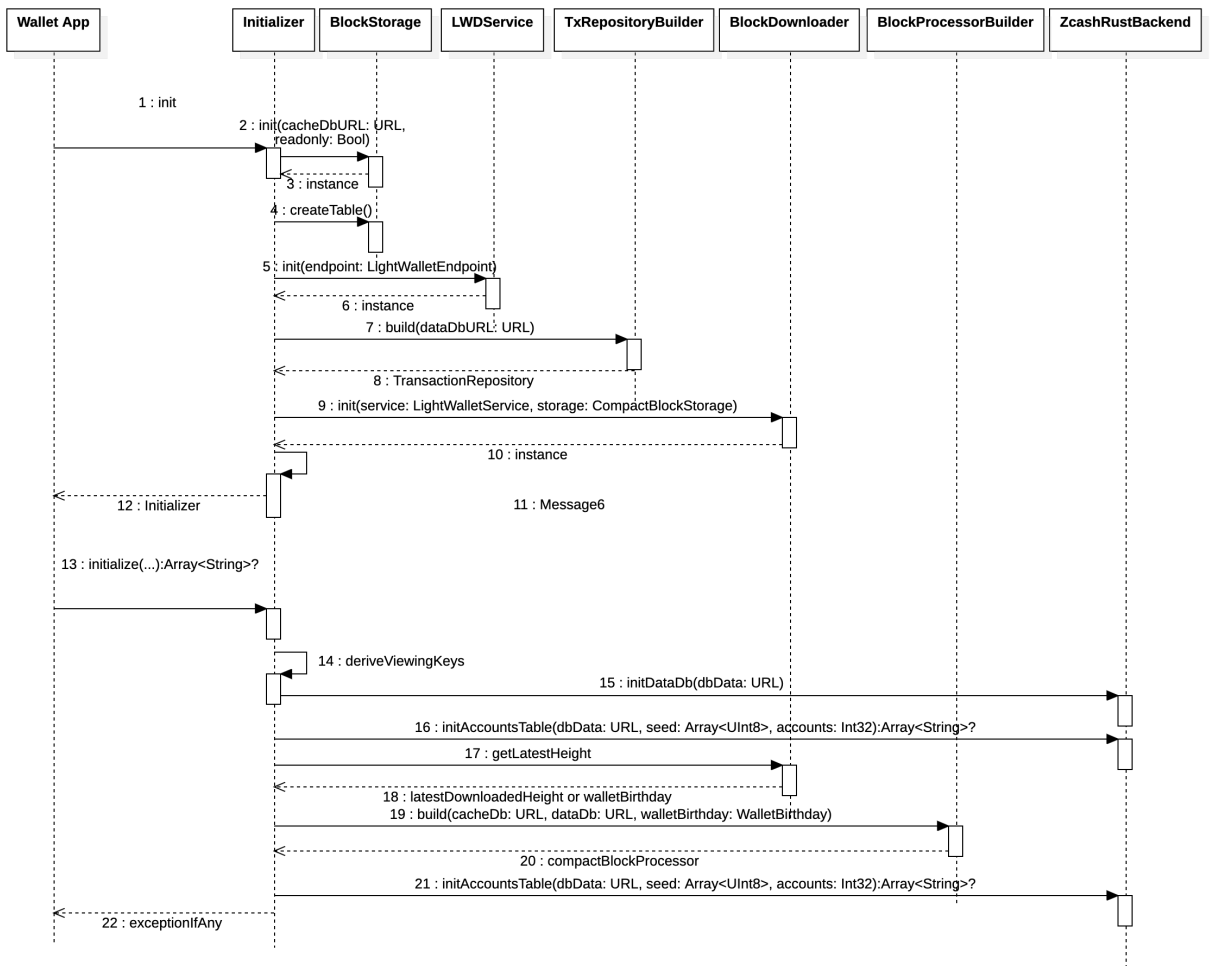

Fig. 9: Sequence diagram of the Initializer Object on a Zcash wallet.

TxRepositoryBuilder. Factory Class that builds concrete implementations of the *TransactionRepository* which acts as a transaction repository.

BlockDownloader. Class that downloads compact blocks from *LightWalletService* utilizing *BlockStorage* as persistence.

BlockProcessorBuilder. Builder class that instantiates the *CompactBlockProcessor* object.

ZcashRustBackend. Interfaces that encapsulates the C Rust FFI from *Librustzcash* providing core functionalities from the Zcash protocol.

The first lane (from left to right) on figure 9 shows that creating the *Initializer* instance has two steps: the construction of the object and the start up of its components under the *initialize()* function.

Code Sample

Sample code of *Initializer* can be found in the respective repositories of Zcash mobile SDKs for Android. This paper references version 1.3.0 [Electric Coin Company 2019b] and iOS 0.10.2 [Electric Coin Company 2019d].

Known Uses

The concept of an initializing class can be found in cryptocurrency frameworks and libraries such as Web3 SDK (See official docs of v1.3.4 [Ethereum 2017]) or the mentioned Zcash SDKs. The same role was found on Monero's "Wallet2 API" [The Monero Project 2021] used in most wallets of that privacy coin such as Monerujo [Monerujo Team 2020] or Cake Wallet [Cake Technologies 2020]. In this case the header file exposes an API condensing the required resources.

Related Patterns

The Initializer is a «Creational» actor. Which could be related to patterns of that kind such as

—Factory, facilitate creation a concrete instance of the Initializer by providing a Factory with predefined setups

—Builder pattern could be of use when there are many ways to customize initialization besides dependency injection through constructor parameters.

Additionally, there could be no use in creating more than one instance of an Initializer so it could be created as a *Singleton*.

6. WALLET SYNCHRONIZER

Intent

Transacting on a blockchain can be reduced to the fact of being up-to-date with the data produced on it from the point of view of a set of private and/or public keys. A *Wallet Synchronizer* implements the functional and non-functional requirements to meet those ends.

Motivation

In the appendix section 8 we introduce the use case of a merchant that accepts payments in cryptocurrencies. There is nothing that says "Bob's wallet" on the blockchain, neither a "Wallet" representation. Cryptocurrencies are made of a chain of blocks, that contain transactions composed of inputs and outputs representing the transfer of value from cryptographic key to another. It is the software which takes a set of keys and begins reading the blocks and figuring out the balances. A wallet is nothing else than a representation of the blockchain from the point of view of a set of keys. The wallet application takes Bob's keys, initializes the necessary resources, catches up with all relevant events on the blockchain and once it is in sync with the latest block, it can display an accurate balance and his past or ongoing transactions for him. It is also valid that for user experience reasons, wallets could display less accurate "optimistic balances" to the user until a more certain one is available after synchronization is complete.

Operating on a blockchain requires knowledge of the protocol, that is how the consensus and network of peers work. To transact (receiving or submitting transactions) it is necessary to be synced with the blockchain. That logic is described in the documentation of the different blockchains. There is a great amount common grounds depending on the type consensus used, that being Proof-of-Work, Proof-of-Stake, Proof-of-Space, etc. but still every protocol has its own nuances. The *Wallet Synchronizer* condenses all the knowledge required to sync the blockchain, receiving and producing transactions. A wallet can be synced locally or by delegating decryption to other actors. Monero is one case of this last use case where the wallet delegates syncing to full nodes by handing over *Viewing Keys* that are used to trial decrypt and return the results to the application. Another case is the Oblivious Message Retrieval[Liu and Tromer 2021] which proposes a way for wallets to retrieve information from

the blockchain anonymously without revealing any viewing keys or significant metadata to the server they connect to. OMR makes use of special detection keys that would allow the server to retrieve the users' transactions of interest without learning specifically which ones they are. Either way, it is the *Wallet Synchronizer*'s goal to abstract this logic from the wallet application.

Applicability

The role of a *Wallet Synchronizer* exists in every blockchain in different shapes. In some cases delegated to an intermediate server, like fiat-crypto centralized exchanges where Key custody is in charge of the exchange company. The opposite approach is the decentralized non-custodian light client wallets where users make sovereign custody of their keys and sync the blockchain locally through an intermediate server that eases the storage burden like Bitcoins Simple Payment Verification [Nakamoto 2009] or Light Client Protocol for Payment Detection [Grigg et al. 2018] on Zcash. The intermediate point is viewing key delegation to a trusted server like Monero does [SerHack 2018]. In any case it has been found that the resulting interface is similar at a light client level by analyzing the interfaces of different Bitcoin, Zcash and Monero non-custodian wallets. The wallets analyzed were Zec Wallet full-node [Kulkarni 2020b], Lite [Kulkarni 2020a] and mobile [Zcash Foundation 2019]; ECC Wallet iOS [Company 2020] and Android [Electric Coin Company 2019a]; Unstoppable android [Horizontal Systems 2021c] and iOS [Horizontal Systems 2021d], its EthereumKit [Horizontal Systems 2021b] and BitcoinKit [Horizontal Systems 2021a] SDKs; Monerujo Wallet [Monerujo Team 2020], Cake Wallet [Cake Technologies 2020] (also its legacy native implementation) [Cake Technologies 2018], and the Wallet2_Api header [The Monero Project 2021] which has been extensively found in on GitHub more than five thousand times.

Structure

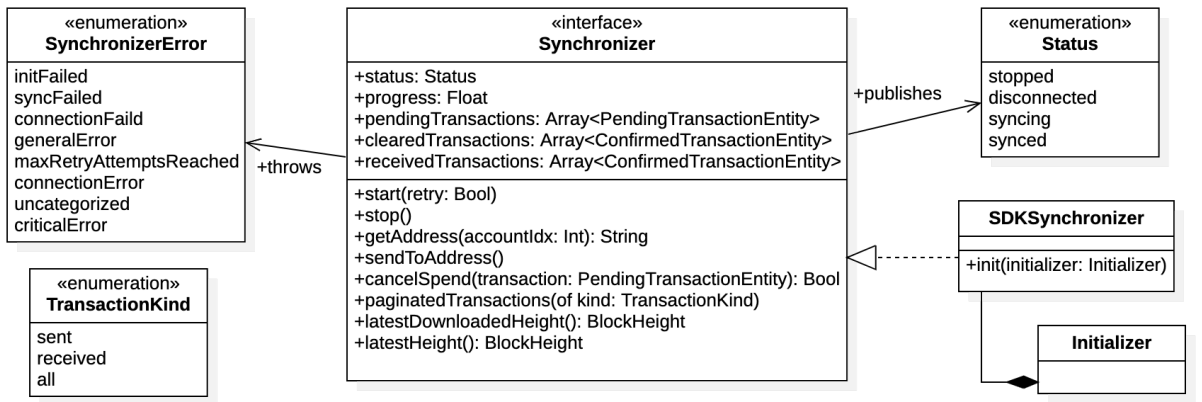

Fig. 10: Wallet Synchronizer: concentrates the requirements behind blockchain synchronization from set of user-provided keys.

Participants

Synchronizer Interface. A public interface that exposes clear intents by its functions and attributes like *start*, *stop*, etc. It can contain convenience or accessory functions that could be performed with other libraries like *getAddress()*. Emits or publishes its state through the *«Status»* Enum and synchronization progress through *«progress»*. Transaction found while syncing are exposed by *PendingTransactions*, *ClearedTransactions* y *ReceivedTransactions*. Pending transactions are those submitted by the user that have not been yet included on a block or confirmed. Cleared transactions are those that have been mined and confirmed. Received transactions

are those that have outputs that belong to the user's wallet. Wallets might have different confirmation times. While some wallets have 1 or 2 block confirmations others may have ten or more. Some protocols may advice to treat different confirmation times for funds depending on their source. If a user is moving funds from another account the same wallet is tracking, this transaction could be considered confirmed almost instantly. This affects how wallets count their balance, since unconfirmed funds can't (or shouldn't) be spent.

Concrete instance, SDKSynchronizer. Implementation of the interface. It additionally has the responsibility of knowing and responding to the application life cycle. Different operating systems (or their versions) can have substantial differences on how application life cycle is handled. The Wallet Synchronizer must be aware of the application being backgrounded, foregrounded, launched or shut down.

Status. Current status of the Wallet Synchronizer. This has great influence over the user interface.

SynchronizerError. Possible failure states on a succinct set of cases to be caught by client applications and exposed to users in a comprehensive way.

TransactionKind. Kind of transaction (pending, confirmed, etc.).

Collaborations

The *Wallet Synchronizer* can collaborate with many components of the system. Its role is mainly to concentrate the requirements to operate in a blockchain in a concise and clear way and expose that interface to the client applications. For its creation, a *Wallet Initializer* can be of aid since *Wallet Synchronizer* instances require a vast set of resources from the user and the operating system. In the Zcash mobile SDKs, the *Wallet Synchronizer* encapsulates other more granular and lower level components that are in charge of processing blocks or deriving keys and addresses.

Consequences

The counterpart of the benefits of a succinct, assertive and (for sure) opinionated interface is that many design decisions have already been taken by third parties. This discussion resembles to the *Buy or Build* dilemma which has been discussed extensively in Software Engineering literature and shouldn't be taken lightly. A good compendium of this topic is available in several chapters of MCconnell's "Code Complete" [McConnell 2004] for consultation.

Implementation

The implementation details of a *Wallet Synchronizer* is tightly coupled to the blockchain it will sync and the platform it will be executed on. It must be able to handle platform specifics (such as lifecycle) as well as protocol ones. The programming environment also influences the implementation significantly. Object oriented or imperative programming environments handle events quite differently than it Reactive Functional Programming ones. This has significant impact on the source code that can be appreciated on projects like ECC-Wallet or Unstoppable Wallet for iOS. There the Zcash iOS SDK implements the *Wallet Synchronizer* interface with *UIKit* OOP in an imperative fashion while the wallets make extensive use of Reactive functional programming frameworks such as *Combine* or *RxSwift*[2] respectively. In both cases there's in important amount of "glue code" to adapt those two different paradigms.

Sample Code

A public interface that matches the proposed interface can be found on version 0.9.3 of *ZcashLightClientKit* [Electric Coin Company 2019e]. Uses of SDKSynchronizer can be found on the *ZcashAdapter* class in Unstoppable Wallet

[2] RxSwift [ReactiveX 2015] is a RFP framework that pre-dated Apple's Combine Framework. URL: `https://github.com/ReactiveX/RxSwift`

for iOS [Horizontal Systems 2021e], where key derivation delegation to a component called *DerivationTool* can be observed. Developers construct the *Wallet Synchronizer* concrete class with the aid of an *Initializer* object and the subscribe the constructed object to operating system events and then then hook their application to the *Wallet Synchronizer* events.

Known Uses

Similar uses of *Wallet Synchronizer* can be found on other SDKs such as *BitcoinKit* by Horizontal Systems [Horizontal Systems 2021e], where the class *BitcoinCore* condenses the behavior expressed in the *Wallet Synchronizer* interface. We transcribed the Swift version of that interface on listing 6.

Listing 1: *BitcoinCore* on *BitcoinKit* SDK. Similarities with *SDKSynchronizer*.

```
extension BitcoinCore {
    public func start()
    internal func stop()
}

extension BitcoinCore {
    public var lastBlockInfo: BlockInfo? { get }
    public var balance: BalanceInfo { get }
    public var syncState: BitcoinCore.KitState { get }
    public func transactions(fromUid: String? = nil,
                             limit: Int? = nil
                         ) -> Single<[TransactionInfo]>
    public func transaction(hash: String) -> TransactionInfo?
    public func send(to address: String,
            value: Int,
            feeRate: Int,
            sortType: TransactionDataSortType,
            pluginData: [UInt8 : IPluginData] = [:]
    ) throws -> FullTransaction

    ....
```

Related Patterns

—Façade. The *Wallet Synchronizer* is an interface that can be considered the visible interface to a more complex back-end whose implementation details are meant to be isolated to the API client.

—Builder, it can be useful when there are several ways of creating a Wallet Synchronizer that add extra logic that can't be dealt with only with constructors.

—Factory Method, provide a convenience one-line way to instantiate a wallet synchronizer with a default setup that would cover a great percentage of the API clients.

7. CONCLUSIONS AND FUTURE WORK

In this work we presented the problem of implementing non-custodian mobile wallets for anonymity enhancing cryptocurrencies. We introduced readers to the concept of wallets, their different kinds, how they differ in terms of access to the blockchain and custody of the keys. We described the responsibilities of AEC wallets and how privacy affects them. Section Overall Architecture presented four patterns that combined together gather the

basic requirements for building a non-custodian mobile wallet for (but not necessarily) privacy coins. The pattern *Mnemonic Seed Handler* shows the challenges of encoding seed bytes in a sustainable way and the forces at play that originate the pattern. Later *Key Storer* describes the nuances of providing safe storage to the seed bytes while maintaining availability when needed. *Wallet Initializer* discusses and how to address the complexity of instantiating these wallets and how it can be encapsulated and controlled by showing practical implementation details found in existing projects. Lastly we present *Wallet Synchronizer*, being the core element of a non-custodian privacy preserving wallet, its requirements, consequences and contradicting forces. The presented patterns are a product of a thorough analysis of Non-custodian Privacy Coin Open Source Mobile Wallets' Architecture which covered the Zcash and Monero protocols, their documentation and the flagship wallets presented on their respective websites. We reviewed their source code, documentation and surveyed the present use cases on released application builds and user stories on the repositories and compared each other. that captures common scenarios and generalizes how they could be approached by developers that need to build this kind of wallets.

To continue improving and validating the proposal we plan to conduct a series of "development experience" experiments to test the ergonomics of the patterns. Given a group of developers, they will be introduced to the matter of non-custodian privacy coin wallets to set a common ground among them, then we will present and discuss the proposed patterns. Then a "Refactoring To Patterns" exercise will be conducted, where the subjects will be given an initial working wallet source code and their work will be applying the patters to it. This will allow gathering feedback from the developers and continue improving the patterns. Another branch of future work is to study the feasibility and impact of adopting a novel technique called Oblivious Message Retrieval (OMR) [Liu and Tromer 2021] to synchronization. According to its authors, a wallet with OMR capabilities can delegate detection of relevant items on a blockchain to an OMR capable server that even when serving these items will not be able to learn anything about them or the interested party. This plays a key role not only for wallet privacy at a metadata level, but also in wallet performance. As blocks in a chain are more full of transactions, synchronization becomes more resource consuming and optimizations are needed in order to keep wallets from being a resource drain for mobile devices.

Acknowledgements

Authors would like to thank reviewers and PLoP'22 organizers for carrying out a great Patterns conference, Mary Tedeschi and Rebecca Wirfs-Brock for shepherding the paper with their valuable and dedicated feedback. We also want to mention participants of the writer's workshop Eduardo Fernandez, Valentino Vranic and André Cordeiro who provided dedicated feedback that helped improved this publication. This paper was possible thanks to the remarkable Open Source work of developer teams of Electric Coin Company, Zcash Foundation, Horizontal Systems and ZecWallet, contributors and communities of CakeWallet, Monerujo. Special thanks to Jack Grigg, Kevin Gorham and Steven Smith from ECC. This paper was carried out on LIFIA and CONICET collaboration and thanks to the open, tuition-free and public education and scientific system of Argentina.

8. APPENDIX: INTRODUCTION TO MOBILE CRYPTOCURRENCY WALLETS AND PAYMENTS

Cryptocurrencies emerge to fulfill the promise of decentralized "electronic cash". A digital equivalent to physical currency as Mean-of-Exchange (MoE) between two of more parties [Nakamoto 2009]. They are complex distributed systems deployed all across the globe operating under consensus protocols that covers a great amount of use cases. For this brief introduction we will focus on a basic known use case where Alice and Bob (the most quoted theoretical human beings) want to exchange goods or services for a certain amount of value using non-custodian mobile crypto wallets.

Bob is known to be an excellent baker who has a small traditional shop in a local flea market and also a tech enthusiast. He likes to innovate and explore new ways of reaching all kinds of customers. He was interested in cryptocurrencies the moment he heard they could be used to exchange goods and services worldwide in a person to person fashion, a global flea market. He researched the different kinds of wallets (see section 1) and decided

that since his bakery was a family legacy he would like to continue to be completely in control of its finances as it always have been since his grandparents started the business. His choice was to create a wallet that allowed him both to be in custody of his funds (and keys) and also provided a mobile experience that could be used in the flea market in the suburbs. And he's determined to go all the way even building from source if it's necessary!

Bob ends up downloading an open source non-custodian wallet because being a baker is a busy job already. When he starts it up the wallet will give him a random mnemonic phrase of 12 or 24 words (more words is safer) like it is shown on figure 11. You can think of the "Seed Phrase" as if it was Bob's secret recipe for his traditional Alfajores. If it went public, everyone could make them and they would certainly be less special. Seed phrases are even more secret! Anyone knowing Bob's seed phrase can see and spend the funds. In a decentralized currency system there's nothing or no one that can undo that [3]. So it's very important that Bob stores it in a very safe place.

The wallet will treat the mnemonic seed phrase as the bytes they represent. Some Wallets also offer QR code Paper wallets as an alternative for backing up the seed words, still phrases are the most common way non-custodian wallets provide the seed bytes to users. Those bytes can be used to generate any cryptocurrency keys. Figure 12 shows how seed bytes are the identity from where keys are derived from. Bob could have the same bytes represented as a QR code as well or even write down the hex-bytes representation. The important fact is that the bytes can be restored when they are needed. For example, when Bob accidentally drops his phone into his industrial size mixer and needs to get a new phone (and surely a make new batch of dough). The same bytes can be used to derive Bitcoin, Ethereum, Zcash, Monero or whatever coin Bob wants to use [4].

Once his wallet is set up, the application will synchronize itself with the blockchain by connecting to either a node or an intermediary server that allows the wallet to access figure out the account balance (see Pattern *Wallet Synchronizer* 6). Bob is ready to head down to the flea market to sell his renowned traditional bakery produces and accept crypto as payment.

Finally, crypto-customer approaches. Alice wants to try "Alfajores". She heard of them while playing around with the Celo testing blockchain named after them [5] to honor delightful piece of Argentinean bakery consisting of two cookies paired together with a filling of "Dulce de Leche" (a superior version of caramel cream) or fruit marmalade often with a chocolate or sugar coating. Alice found out that Bob's Bakery specialty are indeed Alfajores, and he just posted on his social media that now accepts crypto! "What a coincidence!"- Alice thought while she headed down to the flea market to taste this renowned delicacies.

At the store, Alice greets Bob and asks for a chocolate and "Dulce de Leche" alfajor. Bobs hands over the alfajores and asks Alice how she wishes to pay. She mentions having some cryptocurrency that Bob accepts as payment. Bob says that it would be a total of 0.045 units of that crypto and extends his wallet showing the QR code with the address. Alice takes out her own wallet application, scans the address, and performs a payment. She inputs the 0.045 value for the transaction plus a Fee (and the wallet might suggest a fee for the transaction as well). Figure 13 shows how the wallet composes the payment. First as cryptocurrencies are electronic cash systems, Alice's wallet has a certain available balance that is composed of unspent notes. The wallet will add up notes until the *amount + fee* value is equated or surpassed, then create and sign a transaction for Bob's address that will be relayed to the peer to peer network. The transaction will say how much of that value is for Bob, how much change will Alice get back and the remainder will be considered the "Fee". That transaction will have to be included in the blockchain. Bob and Alice will have to wait a prudent time to consider that transaction confirmed (see subsection 1 for a brief discussion on confirmations) and settled on the blockchain's ledger. This transaction is no different as the old double-entry ledger that Bob's grandparents used many years ago at the bakery. Table III shows how this exchange would look like on a double-entry ledger.

[3] Please don't send any funds to the addresses of this seed phrase!

[4] If you want to experiment with phrases, bytes and addresses, we recommend you checking out `https://iancoleman.io/bip39/` a site that has many derivation tools and can also be used offline from source.

[5] See `https://docs.celo.org/network#alfajores-the-developer-testnet`

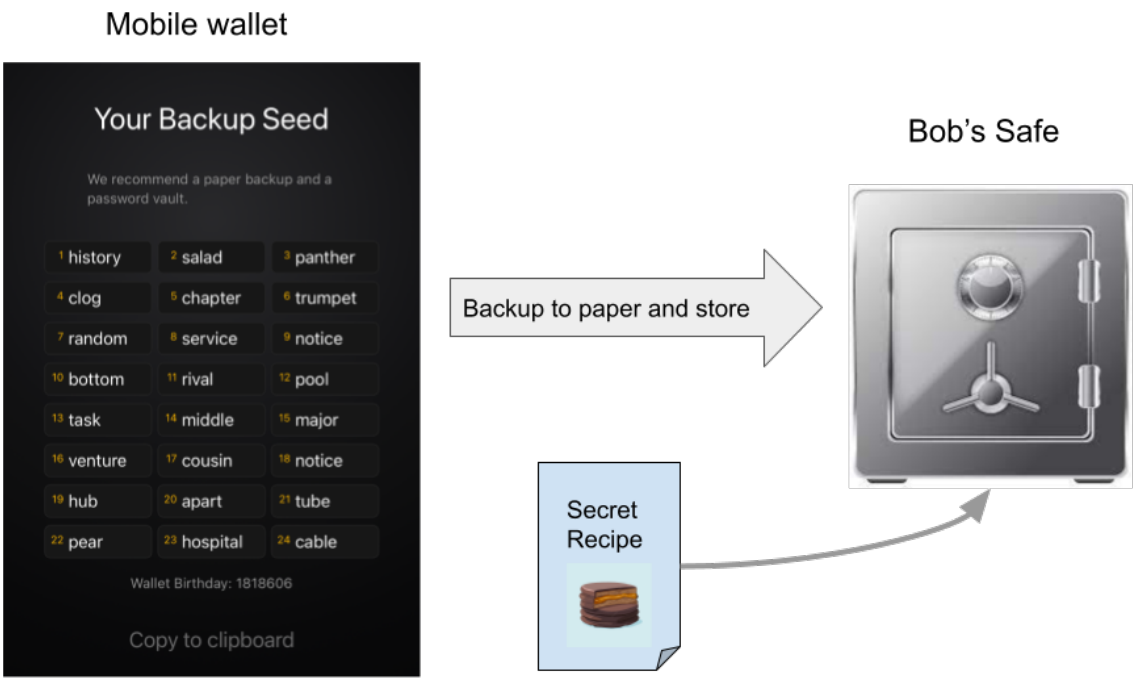

Fig. 11: Bob's seed phrase backup

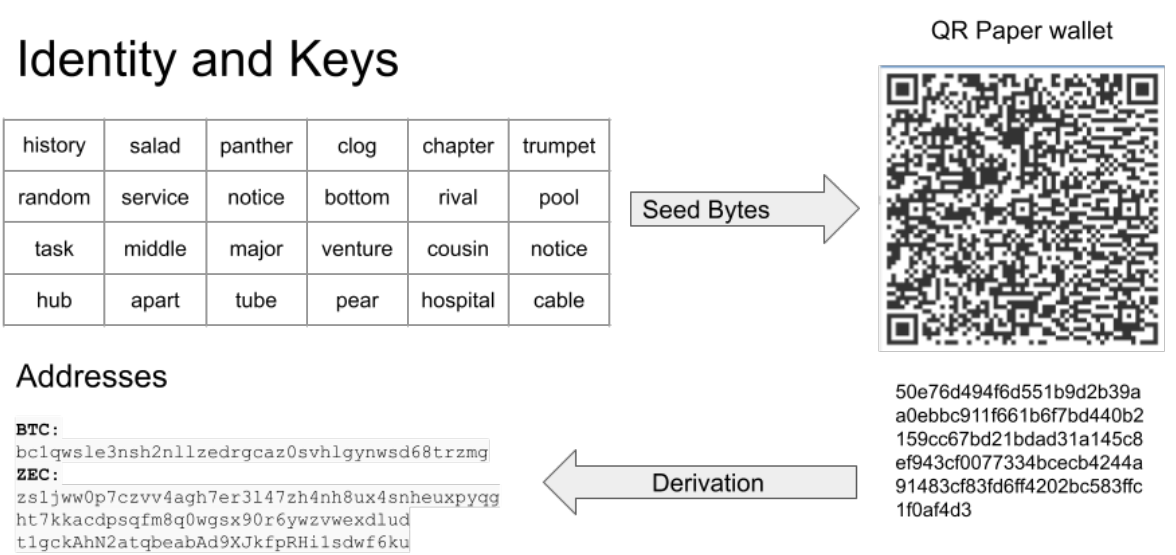

Fig. 12: Seed Bytes and derivation

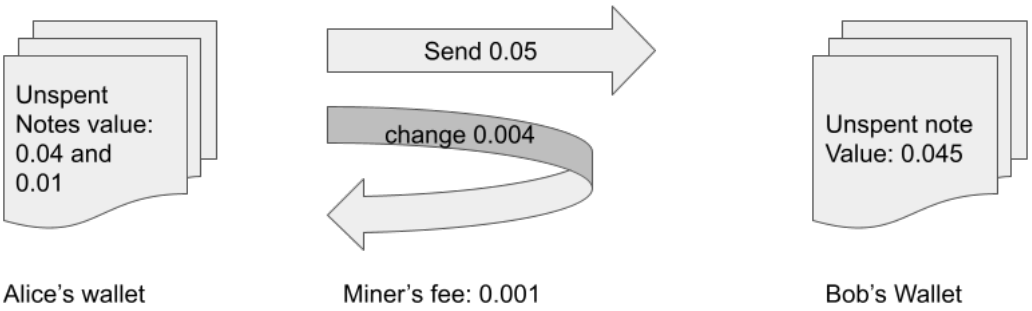

Fig. 13: Flow of a transaction between 2 parties

Inputs	Value	Outputs	Value	Recipient
input 1	0.01	output 1	0.045	Bob
input 2	0.04	output 2	0.004	Alice (change)
total	0.05	total	0.049	
Miner/Validator Fee	0.001			

Table III. : Alice's purchase in terms of Input-Output accounting in the blockchain's ledger.

Privacy implications of cryptocurrencies as mean-of-exchange

If Alice was using traditional cash, the transaction would be very similar to the one described before. Alice greets Bob and asks for a chocolate and "Dulce de Leche" alfajor. She hands some currency notes to Bob and he gives her the alfajor and some change. This is a common transaction in the real world and usually people don't have further thoughts about such a scenario.

Taking a few steps back, we can focus on the privacy aspect of this transaction between Alice and Bob. Alice knows the public information Bob has published for costumers to find his place at the flea market. At the place Alice can see the goods that Bob has for sale in display. Both can physically see each other. Bob can learn that Alice has a currency bill of the denomination she handed to him. Alice can learn that Bob had at least the change he handed back. Bob does not necessarily need to know Alice's identity unless she discloses it herself. No other information is shared between the parties than the one needed for the exchange. Alice can't know how many alfajores Bob sold, whether his name is really Bob or the recipe of the alfajores. Neither Bob knows how much money Alice has or her true identity. Eavesdroppers won't learn much more either. Will this hold for the case of cryptocurrencies? At first, from a usage perspective a study on "Moneywork" [Perry and Ferreira 2018] supports the hypothesis that once electronic payment methods are recognized as "valid", people adopt them following the same patterns as traditional payment methods. So, we can expect crypto payments growing and being more usual as adoption increases.

If Alice were to purchase the alfajor using Bitcoin, she would have had to agree an amount in BTC for the exchange, then request a Bitcoin address to Bob. Then he would monitor it with his wallet until that transaction was mined into the blockchain and confirmed. At this point Alice and Bob have learned each others' addresses, potentially their full balances depending on how well they manage to use their wallets to avoid transaction *linkability*.

Something they don't know is that a new baker, Chad, is in town and is decided to debunk Bob's legacy at all costs. Chad hired Mallory (a surveillance specialist) to find out everything about Bob's bakery business. Mallory has made her homework and read many papers on how to track wallet activity and learn about the actors involved Biryukov A. [2019; Lero A.R.S. [2019; Kus Khalilov M.C. [2018; Aiolli et al. [2019; Chen et al. [2020]. The flea market is away from downtown and cellphone reception isn't good and their offer free Wi-Fi for merchants and customers. So, Mallory passively observes the Wi-Fi network particularly interested in Bob's activity. Learning metadata like IP addresses and timestamps that could help to deanonymize transactions involving Bob's wallet by contrasting that information with the one present on the blockchain. By doing so, Mallory would have learned a lot about Bob's marketing and pricing policies that will help Chad's evil plans.

As we know, Bob is a tech enthusiast and uses privacy coins. He created a wallet with his mnemonic seed phrase that he stored in a safe place to be in control of his private keys and uses a non-custodian wallet to sync the blockchain. He is aware that eavesdroppers might try to track his activity so he uses servers that have "canary warrants" and usually uses a VPN on public Wi-Fi but for some reason it's not working for him today. When Alice sends him the transaction they have agreed upon, Mallory can learn about that network activity, but then the information that is present in the blockchain of AECs doesn't reveal anything meaningful she can report back to Chad. Neither Alice or Bob can learn a lot from their respective addresses. Even if Alice was found to be working for Mallory, privacy is preserved and Bob's bakery is safe from Chad's market dumping attempt. Privacy coin mobile wallets can be seen as warrants of fairness in the crypto-economy.

When Bob the baker had to decide which kind of wallet he would use, he made a series of choices as if he was walking down a path with many yields. First, centralized or decentralized finance; then custodian or non-custodian and finally public or private currency. Each time he chose the latter option, he prioritized control over delegation. These choices have an important impact on the software supporting them. Delegation is an instrument to handle complexity, by moving it to another place at the expense of relinquishing control over the process and trusting the results to a third party. Bob's choice embraces control and therefore entails some complexity which developers and users need to manage. A further discussion on these topics can be found on the Introduction subsections of this paper.

REFERENCES

Fabio Aiolli, Mauro Conti, Ankit Gangwal, and Mirko Polato. 2019. Mind Your Wallet's Privacy: Identifying Bitcoin Wallet Apps and User's Actions through Network Traffic Analysis. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19)*. Association for Computing Machinery, New York, NY, USA, 1484–1491. DOI:http://dx.doi.org/10.1145/3297280.3297430

Andreas M. Antonopoulos. 2017. *Mastering Bitcoin: Programming the Open Blockchain* (2nd ed.). O'Reilly Media, Inc., 1005 Gravenstein Hwy N, Sebastopol, CA.

Tikhomirov S. Biryukov A. 2019. Security and privacy of mobile wallet users in Bitcoin, Dash, Monero, and Zcash. (2019).

Bitstamp. 2022. Bitstamp Crypto Pulse Q1 2022. website. (29 04 2022). https://blog.bitstamp.net/post/crypto-pulse-crypto-to-overtake-traditional-investments.

Cake Technologies. 2018. Cake Wallet iOS. github. (27 03 2018). https://github.com/fotolockr/CakeWallet/tree/master/CakeWallet/Domain/Monero.

Cake Technologies. 2020. Cake Wallet. github. (20 11 2020). https://github.com/cake-tech/cake_wallet.

Chainalysis. 2022. THE 2022 CRYPTO CRIME REPORT. website. (6 01 2022). https://go.chainalysis.com/rs/503-FAP-074/images/Crypto-Crime-Report-2022.pdf.

Ting Chen, Zihao Li, Yuxiao Zhu, Jiachi Chen, Xiapu Luo, John Chi-Shing Lui, Xiaodong Lin, and Xiaosong Zhang. 2020. Understanding Ethereum via Graph Analysis. *ACM Trans. Internet Technol.* 20, 2, Article 18 (apr 2020), 32 pages. DOI:http://dx.doi.org/10.1145/3381036

Electric Coin Company. 2020. ECC Wallet for iOS. github. (20 11 2020). https://github.com/zcash/zcash-ios-wallet.

Electric Coin Company. 2019a. Zcash SDK for Android. github. (7 7 2019). https://github.com/zcash/zcash-android-wallet-sdk.

Electric Coin Company. 2019b. Zcash SDK for Android. github. (7 7 2019). https://github.com/zcash/zcash-android-wallet-sdk/blob/v1.3.0-beta09/src/main/java/cash/z/ecc/android/sdk/Initializer.kt.

Electric Coin Company. 2019c. ZcashLightClientKit, Zcash SDK for iOS. github. (12 12 2019). https://github.com/zcash/ZcashLightClientKit.

Electric Coin Company. 2019d. ZcashLightClientKit, Zcash SDK for iOS. github. (12 12 2019). https://github.com/zcash/ZcashLightClientKit/blob/0.10.2/ZcashLightClientKit/Initializer.swift.

Electric Coin Company. 2019e. ZcashLightClientKit, Zcash SDK for iOS. github. (12 12 2019). https://github.com/zcash/ZcashLightClientKit/blob/0.9.2/ZcashLightClientKit/Synchronizer.swift.

Electric Coin Company. 2020a. Mnemonic Seed Handling. Github repository. (28 02 2020). https://github.com/zcash/zcash-ios-wallet/blob/0.5.0-130/wallet/wallet/Utils/MnemonicSeedPhraseHandling.swift.

Electric Coin Company. 2020b. MnemonicSwift: An implementation of BIP39 in Swift. Github. (2020). https://github.com/zcash-hackworks/MnemonicSwift.

Electric Coin Company. 2022. Zcash. Website. (2022). https://z.cash.

Electric Coin Company and Kevin Gorham. 2020. ECC Wallet for Android. github. (20 11 2020). https://github.com/zcash/zcash-android-wallet.

Ethereum. 2017. Web3 Instance Documentation. Website. (04 05 2017). https://web3js.readthedocs.io/en/v1.3.4/web3.html.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA.

Jack Grigg and Others. 2019. Librustzcash - Zcash Rust Crates. Github repository. (19 11 2019). https://github.com/zcash/librustzcash.

Jack Grigg, George Tankersley, and Matthew Green. 2018. Light Client Protocol for Payment Detection. Zcash ZIP Repository. (17 09 2018). https://zips.z.cash/zip-0307.

Laszlo Hanyecz. 2010. Bitcoin Pizza. Bitcoin Blockchain. (22 05 2010). https://www.blockchain.com/btc/tx/a1075db55d416d3ca199f55b6084e2115b9345e16c5cf302fc80e9d5fbf5d48d.

Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2020. Zcash Protocol Specification. Version 2020.1.14 Overwinter + Sapling + Blossom + Heartwood + Canopy. Zcash ZIP Repository. (2020). https://zips.z.cash/protocol/protocol.pdf.

Horizontal Systems. 2021a. BitcoinKit. github. (16 05 2021). https://github.com/horizontalsystems/bitcoin-kit-ios, https://github.com/horizontalsystems/bitcoin-kit-android.

Horizontal Systems. 2021b. EthereumKit. github. (16 05 2021). https://github.com/horizontalsystems/ethereum-kit-ios, https://github.com/horizontalsystems/ethereum-kit-android.

Horizontal Systems. 2021c. Unstoppable Android Wallet. github. (16 05 2021). https://github.com/horizontalsystems/unstoppable-wallet-android.

Horizontal Systems. 2021d. Unstoppable iOS Wallet. github. (16 05 2021). https://github.com/horizontalsystems/unstoppable-wallet-ios.

Horizontal Systems. 2021e. Unstoppable Wallet site. website. (22 04 2021). https://github.com/horizontalsystems/unstoppable-wallet-ios/blob/0.21/UnstoppableWallet/UnstoppableWallet/Core/Adapters/ZcashAdapter.swift.

Aditya Kulkarni. 2020a. Zec Wallet Lite. github. (20 11 2020). https://github.com/adityapk00/zecwallet-lite.

Aditya Kulkarni. 2020b. Zec Wallet Mobile. github. (20 11 2020). https://github.com/zecwalletco/zecwallet-mobile.

Levi A. Kus Khalilov M.C. 2018. A survey on anonymity and privacy in bitcoin-like digital cash systems. (2018).

Gear A.L. Lero A.R.S., Lero J.B. 2019. Privacy and security analysis of cryptocurrency mobile applications. (2019).

Zeyu Liu and Eran Tromer. 2021. Oblivious Message Retrieval. Cryptology ePrint Archive, Paper 2021/1256. (2021). https://eprint.iacr.org/2021/1256.

Steve McConnell. 2004. *Code Complete, Second Edition*. Microsoft Press, USA.

Ian Miers, Christina Garman, Matthew Green, and Aviel D. Rubin. 2013. Zerocoin: Anonymous Distributed E-Cash from Bitcoin. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. IEEE Computer Society, USA, 397—411. DOI:http://dx.doi.org/10.1109/SP.2013.34

Monero Labs. 2022. Get Monero. Website. (2022). https://www.getmonero.org/.

Monerujo Team. 2020. Monerujo Wallet. github. (20 11 2020). https://github.com/m2049r/xmrwallet.

Satoshi Nakamoto. 2009. Bitcoin: A peer-to-peer electronic cash system". http://bitcoin.org/bitcoin.pdf. (2009).

Marek Palatinus, Pavol Rusnak, Sean Bowe, and Aaron Voisine. 2013. Mnemonic code for generating deterministic keys. Bitcoin BIP. (10 09 2013). https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki.

Mark Perry and Jennifer Ferreira. 2018. Moneywork: Practices of Use and Social Interaction around Digital and Analog Money. *ACM Trans. Comput.-Hum. Interact.* 24, 6, Article 41 (jan 2018), 32 pages. DOI:http://dx.doi.org/10.1145/3162082

ReactiveX. 2015. RxSwift. Github. (9 04 2015). https://github.com/ReactiveX/RxSwift.

Arthur J. Riel. 1996. *Object-Oriented Design Heuristics*. Addison-Wesley, Reading, MA.

Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. 2014. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*. IEEE Computer Society, USA, 459—-474. DOI:http://dx.doi.org/10.1109/SP.2014.36

SerHack. 2018. Mastering Monero: The future of private transactions. (2018).

Ron Shevlin. 2021. Mobile Banking Adoption In The United States Has Skyrocketed (But So Have Fraud Concerns). website. (29 07 2021). https://www.forbes.com/sites/ronshevlin/2021/07/29/mobile-banking-adoption-has-skyrocketed-but-so-have-fraud-concerns-what-can-banks-do/.

The Monero Project. 2021. Monero “Wallet2 Api”. github. (21 03 2021). https://github.com/monero-project/monero/blob/b6a029f222abada36c7bc6c65899a4ac969d7dee/src/wallet/api/wallet2_api.h.

Nicolas Van Saberhagen. 2013. CryptoNote v 2.0. https://cryptonote.org/whitepaper.pdf. (2013).

Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. https://github.com/ethereum/yellowpaper, *Ethereum project yellow paper* 151 (2014), 1–32.

Pieter Wuille. 2012. Hierarchical Deterministic Wallets. Github repository. (11 02 2012). https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki.

Pieter Wuille, Marek Palatinus, and Pavol Rusnak. 2018. Shielded Hierarchical Deterministic Wallets. ZIPS repository. (22 05 2018). https://zips.z.cash/zip-0032.

Zcash Foundation. 2019. Zec Wallet Full-node. github. (19 02 2019). https://github.com/ZcashFoundation/zecwallet.