Unlocking Mass-Market Self-Custody

Secure and Private Smartphone Bitcoin Wallets

Jesse Posner
jposner@block.xyz

Jurvis Tan
jurvis@block.xyz

Jordan Mecom
jm@block.xyz

Wilmer Paulino
wilmer@block.xyz

Clay Garrett
cgarrett@block.xyz

Jonathan Pollack
jpollack@block.xyz

Block, Inc.

October 2024

1 Abstract

This paper proposes a smartphone-based Bitcoin wallet design that aims to make self-custody safe and accessible to the mass market, recognizing that requiring specialized hardware to safely own bitcoin limits self-custody adoption. We address many of the security, availability, and privacy challenges inherent with mobile platforms by leveraging a variety of cryptographic techniques such as FROST-based multi-party computation (MPC), secure key backups, Oblivious Pseudorandom Functions (OPRF) for PINs, and zero-knowledge proofs, alongside protective procedures like server signing policies and time-delayed security mechanisms. We detail the key management architecture, backup and recovery mechanisms, and security features designed to protect against unauthorized access and actions. Building upon the widespread ownership of smartphones, our approach aims to demonstrate a practical pathway toward widespread self-custodial adoption.

2 Introduction

The pursuit of widespread self-custodial adoption in Bitcoin faces a fundamental challenge: balancing security with usability. Today, the benchmark for secure Bitcoin storage is often considered to be a multi-signature setup utilizing specialized hardware wallets. While this approach offers formidable protection against a range of attack vectors, it can present significant barriers to entry for the average customer. Expecting the mass market to invest in and manage specialized hardware is impractical and hinders the broader adoption of self-custody solutions.

Mobile phones, ubiquitous and integral to daily life, represent a compelling platform for Bitcoin wallets. The "holy grail" is to transform these devices into secure wallets that approach the security properties of specialized hardware setups. However, this is no small feat. Phones are general-purpose computation devices, perpetually connected to the internet, and constantly carried on one's person—conditions that inherently expose them to increased security risks such as malware, remote exploits, and physical theft.

To address these challenges, we start by identifying key properties that Bitcoin wallets should embody.

- **Usability:** Familiar user experience on familiar devices without creating operational security burdens.
- **Security:** Safeguard against both remote and physical threats.
- **Availability:** Provide wallet recovery mechanisms after loss of devices, cloud accounts, or both.
- **Privacy:** Protect identity and transaction details even in collaborative custody configurations.

In this paper, we introduce a smartphone-based wallet design that addresses these challenges. By leveraging a combination of cryptographic techniques and security features, we demonstrate how to significantly improve upon existing software wallet designs while approaching some of the desirable security properties of hardware wallets. Recognizing that all wallet designs involve a series of trade-off decisions, we believe the combination presented herein offers a practical path toward bringing self-custody to the mass market.

2.1 Open Development

At Bitkey, our mission is to bring simple and secure self-custody solutions to a global audience. We believe that transparency and community collaboration are essential to achieving this goal. Committed to open development, we share our ideas before building them and release the code once completed. This paper serves as an invitation for community scrutiny and feedback on our current thinking and proposed designs as we work towards making self-custody accessible to the mass market.

2.2 Outline

In the following sections, we delve into the technical details of the proposed architecture, demonstrating how it addresses the inherent challenges of phone-based self-custodial Bitcoin wallets and contributes to advancing the field toward practical, secure, private, and user-friendly solutions.

- **Section 3. Key Management:** We introduce the FROST 2-of-2 MPC key arrangement, detailing how keys are generated, updated, and used for signing.
- **Section 4. Key Backups:** We describe the mechanisms employed to provide customers with their app key and a backup of the server key, ensuring their self-sovereignty.
- **Section 5. Privileged Actions:** We explain how certain actions are protected to mitigate the loss of funds if parts of the wallet are compromised.
- **Section 6. Recovery:** We outline the processes for recovering the wallet if key material is lost and how to mitigate attacks that attempt to exploit recovery mechanisms to gain wallet control.
- **Section 7. Privacy:** We discuss several methods for maintaining customer and wallet privacy even in collaborative custody arrangements.
- **Section 8. Summary:** Closing thoughts on the powerful combination of all techniques covered in the paper.

3 Key Management

Effective key management is the cornerstone of a secure and user-friendly Bitcoin wallet. Traditional single-signature (1-of-1) solutions present significant risks due to their reliance on a single point of failure. Customers are often required to export mnemonic seed phrases and store them securely—a process that is both cumbersome and fraught with potential for loss or theft. In many cases, customers are not even aware of the sensitivity and responsibility they are undertaking.

To mitigate these risks, we explore a two-of-two (2-of-2) multi-party computation (MPC) approach. In this setup, both the customer's mobile device and the server each hold a distinct key share, and both are required to authorize transactions. Notably, Coinbase has made significant contributions in this area with their 2-of-2 MPC wallet architecture [1], where normal spending requires collaboration between the customer's device and Coinbase's servers, while the customer retains a self-custodial backup of the server's key share.

We propose enhancements to the 2-of-2 MPC architecture by optimizing it for Bitcoin and addressing practical considerations unique to mobile wallets. We recognize that involving more keys in a k-of-n setup, without introducing additional devices, adds complexity without benefits. Therefore, we focus on improving the 2-of-2 arrangement to maximize security and usability.

3.1 Key Arrangement

Our wallet architecture utilizes a two-of-two (2-of-2) configuration with keys generated by the app and the server.

3.1.1 Normal Spending

For normal spending, transactions require signatures from both the customer's mobile app and the server. Since the server's key share is essential for authorizing transactions, the server can implement additional signing policies to enhance fund safety. This is covered in Section 6.2.

3.1.2 Self-Sovereign Spending

While involving the server in the transaction authorization process enhances security, self-custody requires that customers have the ability to move their funds independently of the server. To enable this, the customer receives an encrypted copy of the server's key share that is decryptable only by the phone's secure enclave. This enclave-based encryption ensures that the server's key share can only be accessed on the customer's physical device and by the authorized app, protected by biometric authentication and an application enforced time delay. This is covered in Section 4.1.

3.2 Key Security

The security of any wallet fundamentally depends on protecting cryptographic keys from unauthorized access or misuse. Our wallet design features a key share held by the customer's mobile device, and a key share held by the server. We'll refer to these as the "mobile key share" and "server key share."

In our proposed design, the customer's mobile key is stored locally—using the Keychain on iOS and `EncryptedSharedPreferences` backed by the local Keystore on Android. This mobile key share is decrypted and held in memory for transaction signing. The security of mobile key shares relies on the underlying mobile platform's application sandboxing and the presence of a trusted execution environment, which together help prevent key exfiltration, indirectly through various hardware security and OS mechanisms.

On the server side, customer signing keys are secured through a layered encryption approach involving Data Encryption Keys (DEKs) and a Customer Master Key (CMK). The signing keys are encrypted using the DEKs. These DEKs are themselves encrypted with a CMK managed by AWS Key Management Service (KMS) and stored at rest in DynamoDB. The code which manages these keys and signs customer transactions is called the Wallet Security Module (WSM), and runs in an AWS Nitro Enclave.

The CMK has a strict policy that defines how it can be used. Specifically, it can only decrypt DEKs when a valid cryptographic attestation is presented, proving that the code executing within an AWS Nitro Enclave matches a specific hash value known as PCR0. This policy condition ensures that only an enclave running the approved code can decrypt the DEKs; unauthorized or altered code cannot use the CMK to decrypt them. The policy effectively binds the CMK's decryption capability to a particular deployment, preventing misuse even if someone attempts to deploy a malicious enclave.

3.2.1 Data Encryption Keys (DEKs)

The server's signing keys will be encrypted using Data Encryption Keys (DEKs), which will rotate approximately every 2 million uses of a DEK. This system will generate and cache DEKs, utilizing AWS KMS and AWS DynamoDB to store and update usage counts in batches to minimize database interactions. By employing local caching and a leasing mechanism, we aim to efficiently retrieve DEKs while mitigating nonce-collision risks and reducing the potential impact if a DEK is compromised.

3.2.2 Customer Managed Key (CMK)

DEKs themselves are encrypted with the AWS Customer Managed Key (CMK). WSM's CMK has a policy on it that binds its use to a specific WSM deployment via the AWS Nitro Trusted Platform Module (TPM)'s PCR0 value:

```
"Action": "kms:Decrypt",
"Resource": "*",
"Condition": {
    "StringEqualsIgnoreCase": {
        "kms:RecipientAttestation:PCR0": "${var.enclave_attestation_pcr0}"
    }
}
```

The `PCR0` is a hash of the code deployed within the enclave, stored and locked within the TPM. More on this later; but this means that the CMK cannot be used outside of an allowlisted WSM deployment.

3.2.3 Safeguarding KMS Key Policies

We use a dedicated AWS account to manage the CMK, ensuring strict isolation and control. The AWS admin account is dual control across two groups. The account requires MFA, and the U2F tokens are stored in a secure location that requires a third, unrelated, group to access. This ensures that changes to the CMK policy are extremely difficult to make.

All infrastructure is provisioned using AWS CloudFormation, and key updates or policy changes are governed through a CI pipeline. The WSM deployment requires multi-party approval for any changes, with engineers in a signing quorum providing cryptographic signatures for authorization. A separate pipeline manages the trusted engineer certificates, which are stored in an S3 bucket, and signed changes must meet the required signature threshold.

3.2.4 Signing WSM

WSM's binary (the `.eif`) file can be signed, and the hash of the signing certificate goes in `PCR8`:

$$\texttt{PCR8} = \texttt{sha384}(\texttt{0x00*48} \,||\, \texttt{sha384}(\texttt{SignatureSection[0].cert}))$$

This allows one to verify that WSM executed code signed by the appropriate private key. This protects against an attacker maliciously deploying their own instance of an enclave with a matching PCR0, because they cannot sign the binary correctly.

3.2.5 Enclave Attestation

In our design, the mobile app, as well as individuals such as security researchers, could request WSM's attestation document and inspect it. Signed `.eif` files enable us to use the enclave to establish a secure channel with strong attestation guarantees. The mobile application can receive an attestation document containing the enclave signature and then:

1. Verify the attestation document.
2. Extract and validate the `PCR8` contents from the attestation document.
3. Use the enclave's public key, included in the attestation document, to establish a secure channel.

This enables third parties to inspect and verify the enclave software while also ensuring that the code is actually executing. Additionally, this process ensures that the client is indeed communicating with the intended enclave. Consequently, our customers can be more confident that the enclave is not acting maliciously, as a signed mobile app will only communicate with a signed backend enclave.

3.3 FROST

Traditional multi-signature Bitcoin wallets typically involve each participant in a collaborative custody setup holding a unique key. To authorize spending, the wallet relies on a script that mandates the presentation of a quorum of valid signatures from the associated public keys. While effective, this approach has notable limitations.

Firstly, when an output is spent, the entire script—including all public keys—is revealed on-chain. This increases transaction costs due to additional data and reduces privacy by exposing the wallet's multi-signature arrangement. Secondly, changing the set of signers similarly necessitates a "sweep" transaction — moving all funds to a new wallet, a process that can be cumbersome, expensive, and privacy revealing. Thirdly, in certain key arrangements, a lost key requires a sweep to a new wallet using the remaining keys.

To mitigate these issues, we propose leveraging Multi-Party Computation (MPC) to manage multi-signatures off-chain. Specifically, we suggest utilizing the Flexible Round-Optimized Schnorr Threshold Signatures (FROST) protocol. Introduced by Komlo and Goldberg in 2020 [2], FROST provides an efficient and secure framework for threshold cryptography in collaborative settings. FROST enables participants to collaboratively generate valid Schnorr signatures verified against a single public key, without ever assembling the corresponding private key, and without revealing their individual private key shares.

As a result, transactions become more cost-effective by encoding fewer keys and signatures. Additionally, they are more private, revealing no details about the multi-signature arrangement on-chain. Furthermore, FROST is compatible with protocols that facilitate seamless secret refreshes and configuration changes, enabling the replacement, removal, and addition of keys entirely off-chain. This avoids moving funds, incurring fees, linking UTXOs, or losing access to previous addresses whenever changes to the key set are required. An especially valuable application of this capability is the ability to update a wallet by seamlessly adding or removing hardware devices.

FROST consists of two main components: a Distributed Key Generation (DKG) protocol and a signing protocol that enables signature aggregation among participants.

3.3.1 Key Generation

In the original Komlo and Goldberg FROST paper, the authors introduced an enhanced Pedersen-style DKG protocol, often referred to as PedPop, which incorporated proofs of possession to safeguard against rogue key attacks. The protocol requires two communication rounds and assumes the availability of a secure broadcast channel.

Building on the original protocol, the Olaf paper [3] presented SimplPedPop, a refined version of PedPop that reduced the required communication rounds and added a final broadcast phase. This broadcast ensures unanimous agreement on the final public key among all participants, streamlining the DKG process and helping to guard against active adversaries [4]. However, like its predecessor, SimplPedPop still requires implementers to provide their own secure broadcast mechanism.

In our design, we propose a variant called NoisePedPop. This protocol integrates the round optimization from SimplPedPop and employs the Echo Broadcast Mechanism as suggested by EncPedPop to guard against active adversaries. Additionally, NoisePedPop utilizes the **NOISE_IK** protocol to provide integrated secure communication channels, effectively addressing the need for a secure broadcast mechanism within the protocol.

	PedPop	**SimplPedPop**	**NoisePedPop**
Communication Rounds	Two	One (with broadcast)	One (with broadcast)
Secure Broadcast Channel	Assumed required but not specified	Required; implementers must provide their own	Required; using NOISE_IK
Secure Channels	Assumed but not explicitly integrated	Assumed but not explicitly integrated	Integrated with NOISE_IK
Echo Broadcast Mechanism	No	No	Yes

Table 1: Comparison of PedPop, SimplPedPop, and NoisePedPop

The key generation protocol can be found in Appendix A.1.

3.3.2 Key Tweaking and Signing

To prevent address reuse in Bitcoin, wallets typically use BIP 32 [5] to derive child key pairs for different outputs. In the context of Taproot transactions, derived public keys require an additional "tweak" to obtain a Taproot output key, which is then encoded to generate a receive address.

The signing protocol to generate a valid Schnorr signature is achieved through a secure aggregation process where each participant contributes to the final signature in such a way that the aggregate signature remains indistinguishable from a standard Schnorr signature. Importantly, the verification of signatures produced using FROST is identical to the conventional Schnorr signature verification procedure, making it compatible with **BIP 340** [6].

The methodology for tweaking FROST-based keys is detailed in Appendix A.2, and Appendix A.3 outlines how signature aggregation operates in this context.

3.3.3 Key Refreshes

Key refreshes generate a new set of cryptographic keys that are entirely incompatible with any previous key material. This mechanism plays a critical role in mitigating attacks. An adversary attempting to achieve a signing quorum must gather multiple artifacts from various participants. In instances where only partial key material is compromised, key refreshes ensure that the exposure is temporary, and once all participants have refreshed and discarded their old keys, previously compromised keys become obsolete.

Notably, the refresh process is performed entirely off-chain and can occur frequently, potentially every time the application comes online. Key refreshes are accomplished using the **Refresh Protocol** in Proactive Secret Sharing (PSS), as described by Herzberg [7]. The specific protocol is detailed in Appendix A.4.

3.3.4 Key Enrollment

Adding a new key to the key setup can be accomplished using the **Repair Protocol** in Proactive Secret Sharing (PSS), as outlined by Laing [8]. This is useful for setups where the customer may want to expand beyond using just the App and Server, and add a third offline device to the collaborative custody setup.

In this instance, we propose using the Refresh protocol in PSS to eliminate the on-chain visibility of signer updates. This means key configuration changes can occur without creating a transaction, allowing customers to avoid fees and maintain privacy throughout the process. The specific protocol is detailed in Appendix A.5.

3.3.5 Key Repair

In a k-of-n multi-signature setup where $k < n$, customers who may lose one of the signing devices within the quorum need to re-establish a key share. This process does not introduce any novel protocols beyond those already discussed. Specifically, to repair a key share, the quorum of signers first conducts the **Refresh Protocol** to invalidate the lost share. Then, they execute the **Repair Protocol** to enroll a new key share.

This approach allows for seamless key share recovery without affecting the wallet's on-chain activity, preserving privacy and avoiding transaction fees.

3.3.6 Secure Channels

Establishing secure communication channels is essential for FROST. A secure channel refers to a peer-to-peer communication protocol that provides both authenticity and confidentiality guarantees. Since FROST relies on secure channels for key generation, signing, and other critical operations, it is imperative that each participant has a long-term identity to verify they are communicating with the correct party.

Our proposed secure channel is based on **NOISE_IK** [9], but with `Secp256r1` as the Diffie-Hellman function; the full protocol name is `Noise_IK_p256_ChaChaPoly_SHA256`. This protocol employs `Secp256r1` due to Apple's iOS Secure Enclave exclusive support of this curve, which allows us to root the key exchange in the enclave. In our design, the Bitkey hardware will also support this same protocol to allow upgrading from a 2-of-2 to a 2-of-3 wallet. **NOISE_IK** runs suitably on embedded systems, making it an ideal choice for our purposes.

The Noise framework offers various two-way authentication patterns. Given our requirement for long-term identity verification, patterns involving `N` (no static key) or `X` (delayed key transmission) are unsuitable. This

leaves `K` (known static key) and `I` (immediate key transmission) as options. For the server, `K` can be fulfilled by embedding its long-term identity in the mobile app and firmware. The mobile app and hardware, lacking an out-of-band communication method, will use `I`, transmitting their static key during the handshake. The server will trust-on-first-use the key on the initial connection and verify it on subsequent connections.

Rather than directly embed the server public key into the firmware and mobile app, the public key will be placed in a certificate with a standard three-tier PKI, designed specifically for this purpose. While not strictly necessary, this simplifies server key rotation in case of compromise, enabling affected clients to transition into a secure state more easily.

4 Key Backups

In the previous sections, we defined the key arrangements of our wallet, detailing how they are generated, stored, and utilized within our 2-of-2 spending paths. We also discussed how these keys are secured, and the use of FROST to move key management operations off-chain, enhancing both privacy and security. Having established the operational keys of the wallet, we now turn our attention to the crucial aspect of key backup, which ensures both self-sovereignty and recovery from loss. This section is anchored on two main considerations:

- **Loss of the Server:** This scenario encompasses situations where the server becomes inaccessible, becomes uncooperative, attempts to censor transactions, or when the customer simply wishes to exercise full self-sovereignty without relying on the server's participation. To facilitate unilateral control over their funds, we provide customers with a Self-Sovereign Backup (SSB). The SSP contains the necessary key material to assemble the Self-Sovereign Backup, allowing customers to independently access and transfer their funds to any destination of their choosing.
- **Loss of the App:** In cases where the customer loses their mobile device, the app key share stored on the phone is no longer accessible. To address this, we securely back up the app key share in the customer's cloud account. Importantly, this key share is encrypted in a manner that does not rely on the phone, nor is the secret revealed to the server.

4.1 Self-Sovereign Backup

To ensure the wallet remains self-sovereign, we propose providing customers with a Self-Sovereign Backup (SSB). This enables customers to independently recover their funds without relying on the server's participation. The SSP works by encrypting the server's private signing key share to a public key corresponding to a private key stored within the mobile device's Trusted Execution Environment (TEE). The security of this approach relies on the TEE for two primary reasons: (1) to provide isolation from the application processor, which may have malware; and (2) to enforce customer authentication, such as PIN or biometrics, to gate access to the key.

The combination of these two means that malware cannot secretly access the key. Even if the device's operating system is compromised (assuming the TEE itself remains secure), malware cannot secretly access the key because any access attempts must prompt for customer authentication within the secure environment. Note that malware present on the mobile phone at the time of decryption still poses a security risk. This can be mitigated by encrypting to an external hardware token, instead of a mobile phone TEE.

4.1.1 Defining a Trusted Execution Environment

A Trusted Execution Environment (TEE) provides isolation of sensitive code and data from the rest of the system without relying on external hardware. It ensures that, even if the main operating system or applications are compromised, security-critical resources remain protected. Below, we outline the criteria for what we consider a TEE:

On iOS devices, the Secure Enclave has been available since the iPhone 5s. For Android devices, the Keystore API is used, but the underlying mechanism can vary. Our design supports two specific security levels:

Category	**Isolation Mechanism**	**Is a TEE?**
ARM TrustZone	Splits the System-on-Chip (SoC) into two worlds: a "secure world" and a "normal world." Sensitive operations and data are executed/stored in the secure world, isolated from the normal world.	Yes
Android StrongBox	StrongBox utilizes a separate hardware-backed security coprocessor (usually integrated within the SoC) to protect cryptographic keys and operations. It runs isolated from the main OS and apps.	Yes
iOS Secure Enclave	The Secure Enclave is a dedicated hardware module integrated into Apple devices. It has its own microkernel and secure memory, isolated from the main processor and iOS.	Yes
Android Keystore at `SECURITY_LEVEL_SOFTWARE`	OS-provided, such as app sandboxing.	No

Table 2: Defining Trusted Execution Environments

- `SECURITY_LEVEL_TRUSTED_ENVIRONMENT` indicates the mobile device is using a TEE, which is a separate execution environment but runs *on the same chip as the main application processor*. This enforcement happens via TrustZone for Cortex-A.
- `SECURITY_LEVEL_STRONGBOX` indicates, essentially, the presence of a secure enclave, similar to what iPhones provide. The exact feature set is described in the Android CDD.

Finally, the protocol for generating keys, encrypting, and decrypting to the TEE is described in Appendix A.6.

4.1.2 Additional Security Enhancements

Beyond the Trusted Execution Environment, there are a variety of additional techniques that can be used to further harden the setup:

- **Timer enforcement:** Application logic may implement a local timer using a hardware timer, such as those that back `clock_gettime(2)` with `CLOCK_MONOTONIC_RAW`, that must expire before decryption occurs. While malware could circumvent this, it adds a layer of defense against scenarios where a customer might be coerced into decrypting the server key.
- **Publicly-verifiable backups:** To ensure the server actually encrypted and provided its private key, as opposed to something else, we employ publicly verifiable encryption techniques [1], which can be deployed using the Elliptic Curve Integrated Encryption Scheme (ECIES). This allows customers to verify, through zero-knowledge proofs, that the correct private key share was encrypted without revealing the key share itself, ensuring that their wallet is truly backed up and self-sovereign.

- **Key refresh:** The backup must be refreshed if the key shares change. Since the encryption scheme uses double Diffie-Hellman, the mobile application only needs to provide the non-enclave-backed key when it needs to issue a new ciphertext. See Appendix A.6 for the full protocol.
- **Cloud-only storage:** To mitigate risks associated with device loss or upgrade scenarios—where lingering backups could pose a security risk—the backup is stored exclusively in the cloud, not locally on the phone. On iOS, this can be achieved using methods like `evictUbiquitousItemAtURL`; similar approaches are available on Android platforms, via Google Drive APIs.

4.2 App Key Share

It is essential to provide a secure backup mechanism for the mobile application's key share. Loss of the mobile device is a frequent event experienced by many and we need to ensure that the app key share is not lost with the phone. This can be addressed with a secure backup of the app key share, stored in the customer's cloud account.

4.2.1 Oblivious Pseudorandom Function

This backup is encrypted with a key derived from the customer's PIN. To facilitate this, the mobile application and the server would engage in an Oblivious Pseudorandom Function (OPRF) protocol. This allows the customer to derive a strong encryption key from their short PIN without revealing the PIN or the derived key to the server during the process. The backup is updated whenever key shares are rotated (see Key Repair).

The protocol is based on the 2-Hash Diffie Hellman OPRF (2HashDH) [10]. In this scheme, the client blinds their PIN by generating a random nonce and multiplying it by a point derived from a hash of the customer's PIN using a hashing to curve protocol such as `secp256k1_XMD:SHA-256_SSWU_RO`. This point, a blinded PIN, is sent to WSM, which returns a blinded value that incorporates the entropy from a unique 256-bit key assigned to each account. The mobile application then unblinds the returned value to derive the customer's secret. The secret and a domain separation tag are passed to an HKDF to derive the encryption key for the app key share (the "PIN Encryption Key"). A description of 2HashDH can be found in Appendix A.7.

If an attacker gained access to both the encrypted app key share backup as well as the secret key for the customer stored in WSM, they would be able to use a brute-force attack to determine the customer's PIN. It is fundamental to our design that WSM's OPRF key is always segregated from any data that is encrypted with the OPRF to prevent brute-force attacks on the PIN.

4.2.2 Rate Limiting

We propose that the server's OPRF endpoint is rate-limited to prevent external brute-force attacks. A policy that rate limits at 10 queries per hour per account and 300 queries per month would require 1.4 years [11] to brute-force a 4-digit PIN. These parameters can be adjusted as needed to find an optimal balance between API accessibility, brute-force resistance, and PIN length. A 6-digit PIN is preferable to a 4-digit PIN to help protect customers with very common and/or frequently reused 4-digit PINs, while still being a manageable length.

In addition, a per-IP address rate limit is useful to help prevent attackers from maliciously consuming the endpoint to prevent genuine attempts at recovery by the customer. To mitigate against this attack, we require a signature from an evaluation request key that is stored in the customer's cloud, so that an attacker must compromise the customer's cloud to attempt an OPRF evaluation with a guessed PIN.

4.3 Chain Code

In addition to this signing key share, we also store an encrypted chain code using the same method described in Section 4.1. The chain code is required to recover funds, however, it also reveals the entire transaction history of the wallet. By including the chain code in the encrypted backup, we ensure only the customer has access to the transaction history, which is not revealed to the server during the recovery process.

5 Recovery

In the preceding sections, we detailed the architecture of our wallet, covering key management, backup mechanisms, and security features designed to protect against unauthorized actions. This section examines situations where the customer loses access to parts of their wallet—such as the mobile application, the cloud backup, or both. We address these scenarios by explaining how customers can recover wallet access, the mitigations used to prevent the recovery tools being useful for attackers.

5.1 Loss of Mobile Application

To handle the scenario of customers losing access to their mobile application, we utilize a recovery mechanism that leverages an OPRF (see Section 4.1) with a PIN to derive **PIN Authentication Key (PAK)**.

The PAK, in conjunction with a **Delay and Notify (D&N)** process, acts as a mechanism to prevent an attacker who has cloud access from decrypting the key or restoring the wallet to an attacker's phone. The customer's PIN is also employed to derive the decryption key needed to decrypt the app key share backup stored in the cloud.

5.1.1 PIN Authentication Key (PAK)

We utilize an OPRF to make the PIN difficult to brute-force both for the server and external attackers. After deriving the PIN Encryption Key (PEK) using the protocol described in (See Section 4.1), the mobile application generates a PAK and encrypts it with the PEK, and stores the encrypted PAK in the cloud.

5.2 Loss of Cloud Backup

In the event that the mobile application detects the loss of cloud backup data, it will notify the customer and recreate the cloud backup using the data stored locally on the device. As outlined in Section 4.1, the app key share is encrypted using an encryption key derived from the customer's PIN using the server's OPRF, to secure the backup during this process.

5.3 Loss of Both App and Cloud Backup

To safeguard customers who lose access to both their mobile device and cloud backup data simultaneously, we propose an optional **Social Recovery** feature. This mechanism is particularly useful in scenarios such as:

- **Switching between platforms:** Switching between Android and iOS results in a change to both a mobile device and cloud provider simultaneously. If a customer neglects to retain the cloud backup data, then making this platform switch risks a loss of funds held in the wallet.
- **Security limitations on Android:** Unlike iOS, Android doesn't provide developers a way to restrict access to cloud data to the signed application that initializes it. In addition, this data can be deleted without any "step-up" authentication. This makes the wallet vulnerable to loss of funds when an attacker gains physical access, even temporarily, to a customer's unlocked Android mobile device.

5.3.1 Trusted Contacts

The Social Recovery system relies on one or more **Trusted Contacts** who assist the customer in decrypting a backed-up key share. The design is based on the **Portable Blind Cloud Storage (PBCS)** protocol [12]. PBCS requires two servers: a key server and a data server. WSM plays the role of the key server and each trusted contact plays the role of a data server. WSM issues an enclave-backed attestation document (see Enclave Attestation) that gives the mobile application some assurance that OPRF requests will enforce a rate limit for the customer's key to defend against brute-force attacks.

When registering a trusted contact, the customer establishes an authenticated secure channel using the SPAKE2 protocol [13]. The customer initiates this process by sending an invitation, with a short code, to the trusted contact over SMS or another out-of-band communication medium. This SPAKE2 channel is used

to transfer data for the interactions between the customer and the trusted contact in the PBCS Register Procedure and Give Procedure.

5.3.2 Portable Blind Cloud Storage

The customer then performs the PBCS Register Procedure and Give Procedure. We combine the procedures to minimize communication rounds. The customer begins by deriving a social recovery key from their PIN using WSM's OPRF. The key is registered with the trusted contact using the SPAKE2 channel, along with a seed and an encryption nonce.

Then the seed and the customer's PIN are used to derive an authentication token. The customer also derives an encryption key from their nonce and PIN and uses it to encrypt their key share, and then stores the authentication token, ciphertext, and an integrity tag in WSM. This procedure is detailed in Appendix A.8.

When the customer needs to recover funds, they establish a new SPAKE2 channel with the trusted contact. This SPAKE2 channel is used to transfer data for the PBCS Take Procedure interactions. The customer derives their social recovery key from their PIN using WSM's OPRF and sends it to the trusted contact. If the key verifies, the trusted contact returns the seed and nonce. The customer then uses the seed and nonce to derive their authentication token and encryption key. The customer authenticates with WSM using the token, which returns the ciphertext and integrity tag. Lastly, the customer verifies the integrity of the ciphertext with the tag and decrypts the ciphertext. This procedure is detailed in Appendix A.9.

Once the customer has decrypted their key share, the server will still refuse to co-sign transactions, refresh shares, or create a new Self-Sovereign Backup until a D&N process has completed. During this process, the customer can use their PIN Authentication Key to cancel a malicious recovery, and in the event of a contested recovery, control of communication channels can be used to resolve stalemates.

6 Privileged Actions

To enhance the security of our wallet, we introduce mechanisms to protect certain **privileged actions**. These mechanisms leverage time delays as a defensive strategy, making it significantly more difficult for attackers to execute unauthorized operations—especially if they have only momentary or partial access to the customer's device or credentials. By incorporating these time-based defenses, we increase the likelihood that only the true owner can perform sensitive operations.

6.1 Delay & Notify

The Delay and Notify mechanism is designed to protect sensitive actions by introducing a mandatory delay before execution. When a protected action is initiated the server imposes a predetermined delay period, typically several days, before carrying out the action.

During this delay, the server sends notifications to the customer's pre-configured communication channels, such as push notifications, emails, and text messages. These notifications are sent repeatedly to maximize the chances of reaching the customer. If the customer receives a notification and did not initiate the action, they have the opportunity to veto the pending operation directly from the communication channel or through the app. Vetoing cancels the action, preventing it from being executed. If the delay period elapses without a veto, the server proceeds to execute the action.

This mechanism mitigates risks by making it challenging for attackers to perform sensitive actions without the customer's knowledge. Attackers who gain temporary access to the customer's device—perhaps it was left unlocked, or they discovered the PIN—cannot immediately execute protected actions. The delay period provides a critical window during which the legitimate customer can detect and respond to unauthorized attempts.

6.1.1 PIN

Provided the customer has access to their cloud data and their PIN, they can utilize the server's OPRF to derive their PEK, and then use the PEK to decrypt the PAK stored in the cloud, and sign with the PAK

to authorize the privileged action (e.g., D&N veto). The public key for the PAK is registered with the server during the initial wallet onboarding.

6.1.2 Compromised Communication Channels

We acknowledge that attackers may attempt to compromise the customer's communication channels to suppress notifications or prevent vetoes. However, the attacker must maintain exclusive control over all communication channels throughout the length of the delay. If the victim receives a single notification on any channel they will have the opportunity to block the attacker's action. Over time, we believe regaining control over communication channels typically favors the legitimate customer, who can leverage identity verification processes (e.g., with a telecom provider for a phone number) to restore access.

6.1.3 Breaking Stalemates

By design, in a situation when both the attacker and the victim have access to the communication channel, the wallet is essentially in a stalemate where both sides can veto the actions of the other. If either party is able to gain exclusive access to the communications channel for the duration of the delay, then the stalemate is broken.

To further empower the legitimate customer, and to deter the stalemate from being used as an extortion mechanism, when actions are repeatedly initiated and vetoed the wallet enters a contested state. Once this state has been triggered, we require the customer's PIN for both initiating and vetoing protected actions.

If the attacker does not possess the PIN, then the legitimate customer can use it to break the stalemate and regain control over the wallet, even without exclusive access to the communication channels.

6.2 Vaults

Since the server is required to co-sign during normal spending, customers can set signing policies enforced by the server to further enhance security. One such policy is the use of **Vaults**, which allow customers to protect a predefined portion of their balance by setting a minimum balance threshold — the server will adhere to this threshold by refusing to co-sign any transaction that would cause the balance to drop below the threshold.

Customers can freely spend funds above the vault threshold without delay, facilitating day-to-day transactions. To lower the threshold and make vaulted funds available, the customer must undergo the Delay and Notify process described above. This mechanism mitigates the risk of unauthorized depletion of the customer's core funds. If an attacker gains access to the wallet, they can only spend funds above the vault threshold immediately. Attempting to access the vaulted funds triggers the Delay and Notify process, giving the customer time to intervene.

6.2.1 Interaction with Self-Sovereign Backup

While the vault mechanism protects funds through server-enforced policies, we must also consider that the customer would have control of the self-sovereign backup, which allows them to spend unilaterally without the server's involvement. However, this backup is encrypted using the phone's secure enclave and can only be decrypted by the signed app that created it. The app enforces a mandatory delay—say three days—before decrypting the backup key.

This means an attacker cannot use the Self-Sovereign Backup to circumvent the vault's time delays and access the vaulted funds immediately. The delay in the app mirrors the server's delay, ensuring consistent security measures across different recovery paths. For more details on the Self-Sovereign Backup, see Section 4.0.1.

By integrating these time-based defenses at both the server and app levels, we create a layered security model that mitigates various attack vectors. The combination of the Delay and Notify mechanism and Vaults increases the effort and time required for an attacker to compromise the wallet fully, favoring the legitimate customer and enhancing overall security without significantly impacting usability.

7 Privacy

Maintaining customer privacy is a critical challenge in collaborative custody arrangements, where the server actively participates in wallet security. The key issue lies in leveraging the server's capabilities without compromising sensitive information about the customer's wallet and transactions. In our proposed design, we demonstrate how privacy can be preserved even when the server is involved in critical operations. We address this by focusing on descriptor privacy, signing privacy, and vault privacy.

7.1 Descriptor Privacy

The wallet descriptor reveals the entire transaction history of the wallet because it contains all the requisite information to derive its child keys. To protect customer privacy, the server must not store the plaintext wallet descriptor. However, preventing the server from learning this information while still being able to sign is a challenge.

The solution is two-fold. First, we task the mobile application with the sole responsibility of generating and interacting with the chain code. Secondly, we use predicate blind signing [14] to allow the server to sign without learning the child key for which it is signing, and without being able to recognize the final signature, while also being able to enforce signing policies (see Section 7.2).

When performing BIP 32 [5] key tweaking with FROST (see Section 3.3.2), the tweak is added during the signature aggregation step. When the signatures are aggregated, they take the form of:

$$s' = r + x \cdot c$$

where r is the nonce, x is the group private key, and c is the challenge hash. To apply the BIP 32 tweak t to the signature, the aggregator adds $t \cdot c$ to s' to produce the signature:

$$s = s' + (t \cdot c) = r + x \cdot c + (t \cdot c) = r + c(x + t)$$

In our case, the mobile application is the signature aggregator, which applies the tweak. During signing, the server only learns blinded values. Therefore, the server does not need to know the key tweak, or the chain code, to produce its partial signature.

7.2 Signing Privacy

Predicate blind signing [14] combines blind Schnorr signatures with zero-knowledge proofs that make assertions about transaction attributes. This allows the server to enforce signing policies without learning any identifying information about the transaction. However, this protocol must be modified to be compatible with FROST.

To perform the protocol, the mobile application receives a nonce commitment from the server and uses it to compute a blinded nonce commitment and a blinded challenge hash. The server then uses the blinded challenge hash to create a blinded signature and returns it to the mobile application, which unblinds it. In the process, the server does not learn the resulting signature, message, challenge hash, or nonce commitment and is unable to derive those values.

We have a preliminary understanding of the modifications that need to be made to the predicate blind signing protocol to make it compatible with FROST, BIP 340, BIP 341, and BIP 32 [2] [6] [15] [5]. However, we are still actively working on proving that the modified scheme is secure.

7.3 Vault Privacy

Velocity controls are enforced by the server, which presents a privacy challenge because the server needs to be able to condition its signature on the verification of whether a customer has sufficient un-vaulted funds. To solve this, we use a series of zero-knowledge proofs that allow the mobile application to prove to the server that it has sufficient funds without revealing any other information, such as transaction details and wallet balances.

A proof of sufficient funds can be modeled as a solvency proof. We repurpose and adapt a proof-of-solvency protocol designed for exchanges called Provisions [16], which gives us a method of efficiently proving sufficient

un-vaulted funds without requiring a large and complex circuit using a general-purpose proof system such as zk-SNARKs.

7.3.1 Provisions Proof of Solvency

We use the Pay-to-Taproot (P2TR) UTXO set as the anonymity set in the protocol. P2TR addresses are compatible with Provisions because they both correspond to a single public key, rather than a hashed script, and embed that key directly, as opposed to a hashed key. This set can be pruned as needed to meet performance requirements on mobile devices by ejecting unowned outputs using a randomized sampling.

The set only includes UTXOs that were created subsequent to the creation of the wallet. Earlier UTXOs don't provide anonymity because the server knows the wallet did not send transactions before it was created. As a result, the anonymity set for each wallet will grow over time. The initial set isn't built until the first vault deposit, and we delay the ability to enable the vault until a sufficient time period has elapsed to populate the set.

We use the Provisions proof-of-assets protocol to generate a commitment to the sum of wallet assets. This entails generating a Pedersen commitment for each UTXO in the anonymity set and a corresponding proof-of-knowledge that asserts that (1) the mobile application knows the private key for the UTXO and the commitment opens to the balance of the UTXO, or (2) that the commitment opens to zero. The sum of these commitments is a commitment to the sum of wallet assets. These commitments allow the server to enforce vault policies without learning the amount of coins that are stored in the vault.

When the mobile application requests a deposit of coins into the vault, it creates a Pedersen commitment of the amount to be deposited and sends it to the server. If this is the first deposit commitment received from the customer, the server stores it as the initial commitment, otherwise, the server adds this commitment to the existing deposit commitment to generate a new deposit commitment sum. The mobile application also includes a commitment to the sum of its assets so the server can verify that the deposit commitment sum is greater than or equal to the sum of wallet assets. As described in Provisions, range proofs are included with the Pedersen commitments that assert the committed values do not exceed 2^{51} bits to prevent overflows.

When the mobile application requests a withdrawal of coins from the vault, it creates a Pedersen commitment of the amount to be withdrawn and sends it to the server. The server then subtracts the Pedersen commitment from the deposit commitment sum to generate the new cumulative commitment sum. The deposit commitment sum is equivalent to the commitment to the sum of the exchange's liabilities in the Provisions proof-of-liabilities protocol.

When a customer requests that the server co-sign a transaction, the server enforces its vault policy with the following proof components: (1) a commitment to a sum of wallet assets, (2) a commitment to a sum of vault deposits, (3) a commitment to the sum of all non-change outputs in the transaction, and (4) a commitment to the surplus, if any, when the sum of the deposits and the transaction amount is subtracted from the sum of assets. The server checks whether the assets minus the deposits minus the transaction amount minus the surplus is a commitment to the value of 0.

Using the predicate blind signing protocol (see Section 7.2), the mobile application generates a zero-knowledge proof that the transaction Pedersen commitment opens to the sum of the non-change outputs by including that assertion as a predicate. In addition, it generates a Pedersen commitment to the surplus amount, if any, when subtracting the vault deposits and transaction amount from the sum of wallet assets.

8 Summary

The landscape of Bitcoin self-custody has often faced a balance between security and usability. Our wallet design brings together cryptographic techniques like FROST-based MPC, secure key backups, OPRF-based PINs, and zero-knowledge proofs into a cohesive solution tailored for the mass market. Our goal is to maximize security, privacy, and availability, without compromising on the user experience provided by a smartphone-based solution.

A Appendix

A.1 FROST Key Generation Protocol

The three phases of DKG that both the App and Server will engage in order to generate an aggregate public key and VSS (Verifiable Secret Sharing) commitments are shown below.

1. **Generate**
   (a) App generates a proof of possession (R_1, σ_1), a list of coefficient commitments to its polynomial $\overrightarrow{C}_1$, and an intermediate share, $f_1(2)$.
   (b) Server does the same, generating a proof of possession (R_2, σ_2), a list of coefficient commitments to its polynomial $\overrightarrow{C}_2$, and an intermediate share, $f_2(1)$.
   (c) Note, $\overrightarrow{C}_i = \langle \phi_{i0}, \phi_{i1} \rangle$, and each ϕ_{il} is called a coefficient commitment.
   (d) Communication Round: Then, both the App and Server send $\langle (\tilde{R}_i, \tilde{\sigma_i}), \overrightarrow{C}_i, f_i(\ell) \rangle$ to each other.
2. **Aggregate**
   (a) First, the App and Server will validate the coefficient commitments and the proof of possession to ensure that the provided intermediate share is valid.
   (b) Then, they would assemble their own Shamir share by taking the sum of the intermediate shares they have: $s_i = \sum_{l=1}^{2} f_i(l)$. Additionally, they would take the first element of the coefficient commitment given to them, and add it to their own to obtain the aggregate public key: $Y = \prod_{l=1}^{2} \phi_{l0}$.
   (c) Lastly, they would each compute the aggregate VSS commitments: $\overrightarrow{C} = \langle \prod_{\ell=1}^{2} \phi_{\ell 0}, \prod_{\ell=1}^{2} \phi_{\ell 1} \rangle$
3. **Equality Check**
   (a) Broadcast Round: In this step, the App and Server would send $\overrightarrow{C}$ and Y to each other, and ensure that the one they computed is the same as the one they receive from each other.
   (b) The DKG is only considered a success and completed once both App and Server performs this equality check.

Both the communication and broadcast rounds would be done through a secure channel based on NOISE_IK described in Section 3.3.6.

A.2 FROST Key Tweaking

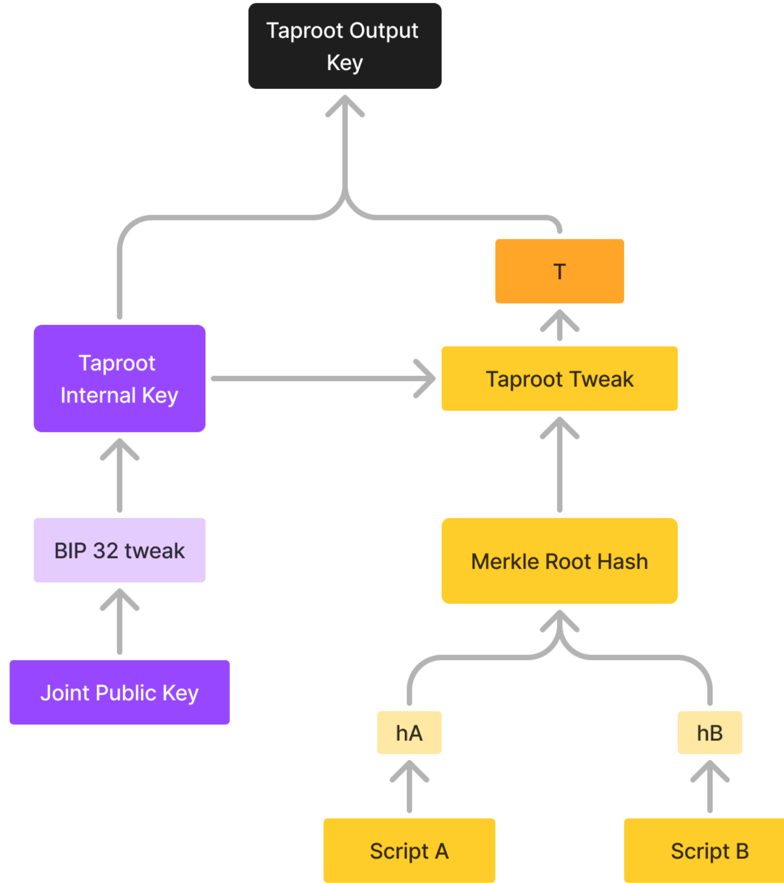


Figure 1: FROST key tweaking

1. First we take the BIP 32 tweak, $t_{b_{32}}$, and apply it to the joint public key as follows:

$$\hat{Y} = Y \cdot g^{t_{b_{32}}}$$

2. Then, we check if y-coordinate of the Taproot internal key $\hat{Y}$ is even. If it isn't, we negate it and call this value $\hat{t}$.

$$\hat{t} = \begin{cases} t_{\text{b32}} & \text{if } \hat{Y}_y|2 \\ -t_{\text{b32}} & \text{else} \end{cases}$$

3. Additionally, we hold on to the parity bit for use in signing later.

$$\pi = \begin{cases} 0 & \text{if } \hat{Y}_y|2 \\ 1 & \text{else} \end{cases}$$

4. Then, we add the $\hat{t}$ we computed above with the Taproot tweak, t_{xonly} to obtain $\tilde{t}$, the final tweak.

$$\tilde{t} = \hat{t} + t_{\text{xonly}} \mod q$$

5. Lastly, we negate to make sure Y' has an even y-coordinate before tweaking it with the Taproot key to obtain the Taproot output key, $\tilde{Y}$. This would be encoded to give us a Bitcoin address.

$$Y' = \begin{cases} \hat{Y} & \text{if } \hat{Y}_y | 2 \\ -\hat{Y} & \text{else} \end{cases}$$

$$\tilde{Y} = Y' \cdot g^{t_{\text{xonly}}}$$

6. Lastly, we negate the value of $\tilde{t}$ based on the parity of $\tilde{Y}_y$

$$t = \begin{cases} \tilde{t} & \text{if } \tilde{Y}_y | 2 \\ -\tilde{t} & \text{else} \end{cases}$$

In order to produce the signature, we need to compute $\tilde{Y}$ and π, and we'll spell out what we do with them in the next section

A.3 FROST Signature Aggregation

The App and Server would need to independently compute the Taproot output key $\tilde{Y}$ and parity bit π as inputs to the signing protocol. The signing protocol works as follows.

1. **Nonce Generation**
   (a) App and Server generates nonces $(d_a, e_a) \xleftarrow{\$} \mathbb{Z}_q \times \mathbb{Z}_q$ and $(d_s, e_s) \xleftarrow{\$} \mathbb{Z}_q \times \mathbb{Z}_q$, respectively.
   (b) Then, the compute their nonce commitments $(D_i, E_i) = (g^{d_i}, g^{e_i})$ and send it to each other.

2. **Signing**
   (a) When both the App and Server receive nonce commitments from each other, they would first check $(D_\ell, E_\ell) \in \mathbb{G}$. Where $\mathbb{G}$ is the group of prime order q, and ℓ is the index of their counterparty.
   (b) Then, App and Server would both aggregate their own nonce commitments with their counterparty's: $D \leftarrow \prod_{\ell=1}^{2} D_\ell$ and $E \leftarrow \prod_{\ell=1}^{2} E_\ell$. We use the values here to define $\rho = (D, E)$.
   (c) Both App and Server would then compute a binding value b such that $b = H_{non}(\tilde{Y}||1||2||\rho||m)$ and m is the message.
   (d) This value is then used to derive the group commitment $R = D \cdot E^b$. Additionally, they would use R to compute a challenge $c = H_{sig}(\tilde{Y}||R||m)$.
   (e) Now, both the App and Server have all they need to construct their partial signature: $\sigma_i = d_i + (e_i \cdot b) + \tilde{s}_i \cdot \lambda_i \cdot c$.
      i. Here, $\tilde{s}_i = -s_i$ if $\tilde{Y}_y | 2 \neq p$, else $\tilde{s}_i = s_i$, where s_i refers to the respective participant's secret share.
      ii. Let λ_i be a Lagrange coefficient such that $\lambda_i = \prod_{j \in \{1,2\} \setminus \{i\}} \frac{j}{j-i}$
   (f) App and Server would now each send their partial signatures, σ_i, to each other.

3. **Aggregation**
   (a) When both the App and Server receive partial signatures from each other, they would need to validate the responses. They do it as follows: $g^{\sigma_\ell} \stackrel{?}{=} D_\ell \cdot (E_\ell)^b \cdot Y_\ell^{c \cdot \lambda_\ell}$
      i. Here, $Y_\ell = \prod_{k=0}^{1} \phi_k^{\ell^k}$
   (b) Finally, when both App and Server are convinced they have valid partial signatures they can aggregate them: $\sigma = \sum_{\ell=1}^{2} \sigma_\ell + ct$
   (c) Finally, they can publish the signature, (σ, R)

A.4 FROST Key Refresh Protocol

Like Distributed Key Generation, key refreshes take a similar, 2-round, 3-step procedure: (1) Generate, (2) Aggregate, (3) Equality Check.

1. **Generate**
   (a) Both App and Server generate a random value $a'_{i1} \leftarrow \mathbb{Z}_q$. They would use this as the coefficient to their refresh polynomial, f'_i such that $f'_i(x) = a'_{i1}x$.
   (b) Then, they would each evaluate three values.
      i. An intermediate share of their refresh polynomial evaluated at their index: $f'_i(i) = a'_{i1} \cdot i$. They will keep this for themselves.
      ii. An intermediate share of their refresh polynomial evaluated at their counterparty's index: $f'_i(\ell) = a'_{i1} \cdot \ell$.
      iii. A commitment to the coefficient of their refresh polynomial $\phi'_{i1} = g^{a'_{i1}}$.
   (c) Communication Round: Then, both the App and Server would share the Refresh Package, $\langle \phi'_{i1}, f'_i(\ell) \rangle$, with each other.
2. **Aggregate**
   (a) When both the App and Server receive refresh packages from each other, they would first need to validate it. They do this by ensuring that the coefficient commitment does indeed correspond to the intermediate share: $g^{f'_\ell(i)} \stackrel{?}{=} \phi'^{i}_{\ell 1}$.
   (b) If and only if the verification succeeds, both the App and Server would proceed to compute their refreshed share, $s'_i = s_i + \sum_{\ell=1}^{2} f'_\ell(i)$.
   (c) Additionally, they would need to update their VSS commitments: $\overrightarrow{C}' = \langle \phi_0, \phi_1 \cdot \prod_{\ell=1}^{2} \phi'_{\ell 1} \rangle$
   (d) Broadcast Round: Both App and Server would then store **but not replace** their refreshed share s'_i, and send the updated VSS commitment, $\overrightarrow{C}'$, with each other.
3. **Equality Check**
   (a) Here, the App and Server would independently verify that the VSS commitment they obtained with their counterparty matches what they've computed.

$$\overrightarrow{C} \leftarrow P_\ell$$
$$\overrightarrow{C}' \stackrel{?}{=} \overrightarrow{C}$$

   (b) If and only if this equality check passes do we securely store and update the cloud backup with the new refreshed share s'_i and $\overrightarrow{C}'$.

A.5 FROST Key Enrollment Protocol

In order to add a new key to the set up, the App and Server would serve as the threshold of participants needed to come together to collaboratively assemble a set of information for the new signer to derive its share.

First, let the Lagrange coefficient $\lambda_i = \prod_{j \in \{1,2\} \setminus \{i\}} \frac{j}{j-i}$.

1. **Generate**
   (a) The App and Server would both take their share, and multiply it by the lagrange coefficient λ_i evaluated at the new participant's index, r: $s_i \lambda_i(r)$.
   (b) Then, they would both use split this up to additive shares δ_{ij} such that $s_i \lambda_i(r) = \delta_{i1} + \delta_{i2}$.

(c) They would then generate repair share commitments $\overrightarrow{R}_i = \langle \psi_{i1}, \psi_{i2} \rangle$, where $\psi_{ij} = g^{\delta_{ij}}$.

(d) Finally, the App and Server would keep δ_{i1} for themselves, and share $(\delta_{i2}, \overrightarrow{R}_i)$ with each other.

2. **Aggregate Repair Share**

Recall that the VSS commitments is $\overrightarrow{C} = \langle \prod_{\ell=1}^{2} \phi_{\ell 0}, \prod_{\ell=1}^{2} \phi_{\ell 1} \rangle$, and ϕ_{ij} refers to the coefficient commitment of each participant's i's polynomial such that if $f_i(x) = a_{i1} + a_{i2}x + \cdots + a_{i(t-1)}x^{t-1}$, such that $\phi_{ij} = g^{a_{ij}}$.

(a) The App and Server would first both verify the additive share it receives from each other: $g^{\delta_{i2}} \stackrel{?}{=} \psi_{i2}$.

(b) Then, they would each verify the repair share by taking the summation of the repair share commitments $(\overrightarrow{R})$ from their counterparty and check that they match the result of the interpolation evaluated with the public verification share of their counterparty, Y_ℓ.

$$g^{\delta_{\ell 1} + \delta_{\ell 2}} \stackrel{?}{=} \lambda_\ell(r) \cdot Y_\ell$$

(c) Once the verification is complete, both the App and Server would assemble an aggregate repair share, ξ_i, such that $\xi_i = \delta_{ii} + \delta_{i\ell}$.

(d) Finally, both the App and Server would share their aggregate repair share (ξ_i), repair share commitments $(\overrightarrow{R})$, and VSS commitments $(\overrightarrow{C})$ with the new signer.

3. **Enrollment**

From here, we use the index a to refer to the App, and s to refer to the Server.

(a) Now, the new signer receives a repair package from the App and Server: $(\xi_a, \overrightarrow{R}_a, \overrightarrow{C}_a) \leftarrow P_a$, $(\xi_s, \overrightarrow{R}_s, \overrightarrow{C}_s) \leftarrow P_s$.

(b) First, they would verify that the VSS commitments are indeed the same: $\overrightarrow{C}_a \stackrel{?}{=} \overrightarrow{C}_s$.

(c) Next, the new signer would compute the public verification shares for the App and Server.

$$Y_a = \prod_{k=0}^{1} \phi_k^{a^k}$$
$$Y_s = \prod_{k=0}^{1} \phi_k^{s^k}$$

(d) Then, they would verify the repair share commitments against the public verification shares they computed.

$$\psi_{a1} \cdot \psi_{a2} \stackrel{?}{=} \lambda_a \cdot Y_a$$
$$\psi_{s1} \cdot \psi_{s2} \stackrel{?}{=} \lambda_s \cdot Y_s$$

(e) Then, the new signer would also verify the aggregate repair share against the repair share commitments.

$$g^{\xi_a} \stackrel{?}{=} \psi_{a1} \cdot \psi_{s2}$$
$$g^{\xi_s} \stackrel{?}{=} \psi_{s1} \cdot \psi_{a2}$$

(f) Once they're both verified, the new signer would sum both aggregate repair share to construct the repair share: $s_r = \xi_a + \xi_s$. Else, terminate.

(g) The new signer then stores both the repair share s_r and VSS commitments $\overrightarrow{C}$.

A.6 TEE Cryptographic Protocols

Generation: The mobile app will generate two wrapping keypairs in the TEE. One will be generated in the TEE and gated by biometric or PIN authentication, and the other will be stored in the TEE without an authentication requirement. This is necessary so that refreshing the FROST shares does not require customer interaction. We'll call these keypairs the local wrapping keypairs with auth, and no auth — LKA and LKN. The server will receive the LKs over a secure (authenticated, encrypted) channel.

$$
\begin{aligned}
(lka_{\text{priv}}, lka_{\text{pub}}) &= \text{GenerateKeypair}_{secp256r1} \\
(lkn_{\text{priv}}, lkn_{\text{pub}}) &= \text{GenerateKeypair}_{secp256r1} \\
\text{return } (lka, lkn)
\end{aligned}
$$

Encryption: The server, to encrypt to the mobile device's TEE, will do the following:

$$
\begin{aligned}
(eph_{\text{priv}}, eph_{\text{pub}}) &= \text{GenerateKeypair}_{secp256r1} \\
s_1 &= \text{ECDH}(lka_{\text{pub}}, eph_{\text{priv}}) \\
s_2 &= \text{ECDH}(lkn_{\text{pub}}, eph_{\text{priv}}) \\
iv &= \text{urandom}(12) \\
k &= \text{HKDF}_{\text{SHA256}}(s_1 || s_2) \\
ct &= \text{AES-GCM}(k, iv, \text{message}) \\
\text{return } (eph_{\text{pub}}, ct)
\end{aligned}
$$

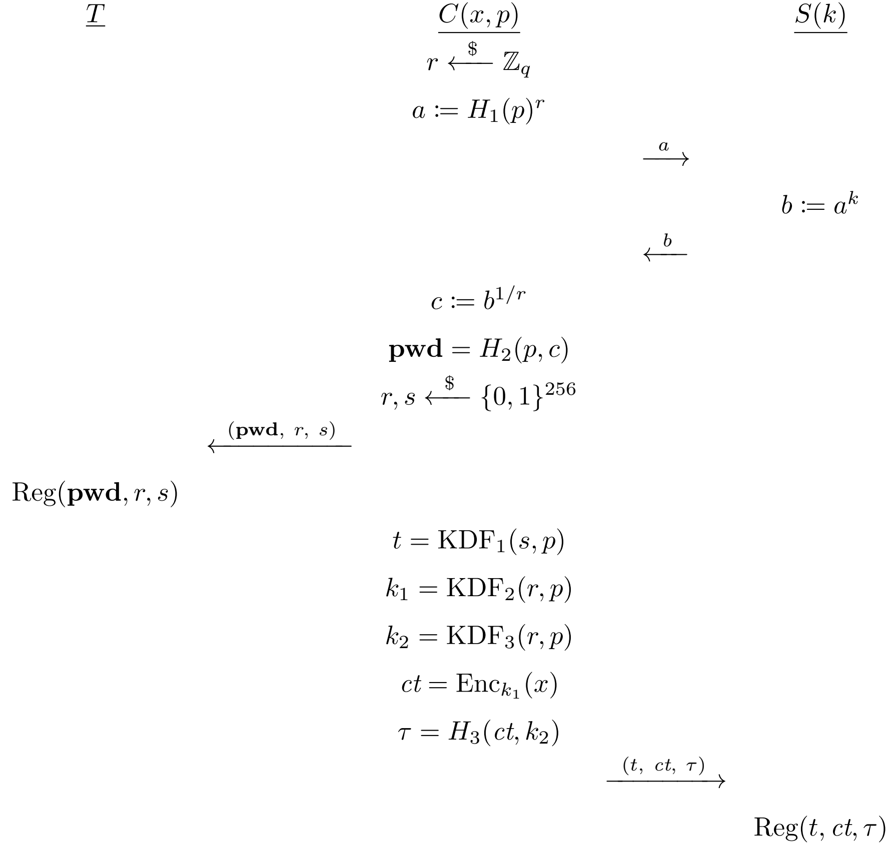
Decryption: And the mobile phone will decrypt as follows:

$$
\begin{aligned}
s_1 &= \text{ECDH}(lka_{\text{priv}}, eph_{\text{pub}}) \\
s_2 &= \text{ECDH}(lkn_{\text{priv}}, eph_{\text{pub}}) \\
(k, iv) &= \text{HKDF}_{\text{SHA256}}(s_1 || s_2) \\
pt &= \text{AES-GCM}^{-1}(k, iv, ct) \\
\text{return } pt
\end{aligned}
$$

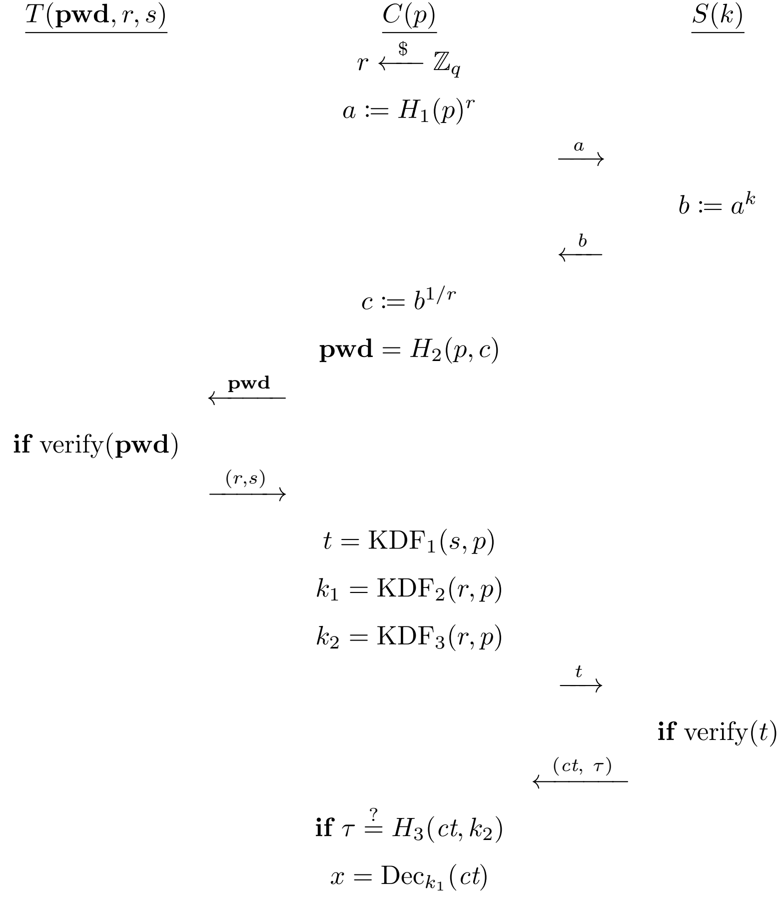
A.7 2HashDH

$$
\begin{array}{ccc}
\underline{C(x)} & & \underline{S(k)} \\
r \xleftarrow{\$} \mathbb{Z}_q & & \\
a := H_1(x)^r & & \\
 & \xrightarrow{a} & \\
 & & b := a^k \\
 & \xleftarrow{b} & \\
c := b^{1/r} & & \\
\textbf{return } H_2(x, c) & &
\end{array}
$$

A.8 Portable Blind Cloud Storage Register and Give Procedure



A.9 Portable Blind Cloud Storage Take Procedure



References

[1] Y. Lindell Coinbase, “Cryptography and mpc in coinbase wallet as a service (waas),” 2023. [Online]. Available: https://coinbase.bynder.com/m/687ea39fd77aa80e/original/CB-MPC-Whitepaper.pdf

[2] C. Komlo and I. Goldberg, “Frost: Flexible round-optimized schnorr threshold signatures,” 2020. [Online]. Available: https://eprint.iacr.org/2020/852.pdf

[3] H. Chu, P. Gerhart, T. Ruffing, and D. Schröder, “Practical schnorr threshold signatures without the algebraic group model,” Cryptology ePrint Archive, 2023. [Online]. Available: https://eprint.iacr.org/2023/899

[4] ZcashFoundation, “Lack of agreement despite broadcast · issue 577 · zcashfoundation/frost,” GitHub, 2024. [Online]. Available: https://github.com/ZcashFoundation/frost/issues/577

[5] P. Wuille, “Hierarchical deterministic wallets,” GitHub, 02 2012. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[6] P. Wuille, J. Nick, and T. Ruffing, “Schnorr signatures for secp256k1,” GitHub, 01 2020. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki

[7] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, “Proactive secret sharing or: How to cope with perpetual leakage,” *LNCS*, vol. 963, pp. 339–352, 01 1995.

[8] T. M. Laing and D. R. Stinson, “A survey and refinement of repairable threshold schemes,” Cryptology ePrint Archive, 2017. [Online]. Available: https://eprint.iacr.org/2017/1155

[9] T. Perrin, “The noise protocol framework,” 07 2018. [Online]. Available: http://www.noiseprotocol.org/noise.pdf

[10] S. Jarecki, A. Kiayias, and H. Krawczyk, “Round-optimal password-protected secret sharing and t-pake in the password-only model,” 08 2014. [Online]. Available: https://eprint.iacr.org/2014/650.pdf

[11] A. Everspaugh, R. Chaterjee, S. Scott, A. Juels, T. Ristenpart, C. Tech, R. Chatterjee, and R. Holloway, “The pythia prf service the pythia prf service,” 2015. [Online]. Available: https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-everspaugh.pdf

[12] L. Chen, Y.-N. Li, Q. Tang, and M. Yung, “End-to-same-end encryption: Modularly augmenting an app with an efficient, portable, and blind cloud storage end-to-same-end encryption: Modularly augmenting an app with an efficient, portable, and blind cloud storage,” 08 2022. [Online]. Available: https://www.usenix.org/system/files/sec22-chen-long.pdf

[13] W. Ladd and Akamai, “Rfc 9382 spake2, a password-authenticated key exchange,” 09 2023. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9382.pdf

[14] G. Fuchsbauer and M. Wolf, “Concurrently secure blind schnorr signatures,” Cryptology ePrint Archive, 2022. [Online]. Available: https://eprint.iacr.org/2022/1676

[15] P. Wuille, J. Nick, and A. Towns, “Taproot: Segwit version 1 spending rules,” GitHub, 01 2020. [Online]. Available: https://github.com/bitcoin/bips/blob/master/bip-0341.mediawiki

[16] G. Dagher, B. Bünz, J. Bonneau, J. Clark, and D. Boneh, “Provisions: Privacy-preserving proofs of solvency for bitcoin exchanges full version *,” 2015. [Online]. Available: https://jbonneau.com/doc/DBBCB15-CCS-provisions.pdf