Universal Private Key Management for Cryptocurrencies

Omer Shlomovits, KZen Networks

June 20, 2019

Abstract

The custody of cryptocurrencies has been an issue since bitcoin inception and is becoming more problematic as new blockchains are introduced into the space. Today there is no solution to manage private keys in a way that can achieve both high level of security and availability. In this work we will give the cryptographic foundations and concepts of a new-generation non-custodial decentralized key management as a service. The design is structured in layers. We first describe the cryptographic building blocks, protocol and architecture and then the security and user-experience services that can be built on top.

Contents

1 Introduction

1.1 Multi-Party ECDSA

The efficiency of Multi Party Computation (MPC) has been improved over the last decades of research to become a useful tool for secure computation in numerous fields. An interesting and hard MPC problem that has been addressed in various research papers over the past years is Elliptic Curve Digital Signature Algorithm (ECDSA) Key Generation and Signing [1, 2, 3, 4, 7]. ECDSA signature scheme is the de-facto scheme used for cryptocurrencies, i.e Bitcoin and its forks (cash, gold, Zcash, litecoin etc..) Ethereum and its ERC20 derivatives, Ripple, Neo and more. Blockchains that do not use ECDSA, use signatures scheme like Schnorr, $ed25519$, ring signatures (Monero, Cardano etc..) and more that are better designed to be computed among multiple parties.
Since Bitcoin gain popularity in the last couple of years there is more focus from the scientific community to address threshold-ECDSA or how to create a set of two protocols: (1) ECDSA private key generation scheme such that the private key is shared among n participants, (2) and a compatible signing scheme such that only $t < n$ participants are needed to create a valid signature. Most recent works [3, 7] are aimed to provide efficient protocols for the specific case of $n = t = 2$. Although using sophisticated cryptography protocols, the primitives and assumptions are actually simple. This latest breakthroughs results in very fast computations that requires modest computing and memory resources and minimal number of communication rounds to achieve $(2, 2)$-threshold ECDSA efficiently between personal mobile computing devices.

1.2 Crypto Asset Custody

Digital assets custody, wether its cryptocurrency, identity, data or any other tokenizable assets, can be defined loosely as the ownership of private keys. The owner of a private key controls who knows it, has the ability to recover it and can sign on publicly verifiable statements about assets tied to the private key, for example - "move funds to address A". Wallets are divided to custodian and non-custodian wallets. Custodian wallets, i.e. Coinbase [39] offer centralized management for private keys, meaning the service provider is the owner of the private keys. A user of this service put trust in the service to manage the private keys on his behalf. The service provider is a single point of failure and is prone to hacking but is also able to provide a vast range of utilities for the wallet user. On the other end, a non-custodian wallets are putting the private key management at the hand of the user. Examples of such wallets are hardware wallets such as Ledger [50] and most of existing mobile wallets, i.e. BRD[57], jaxx [58]. This type of wallets do not require trust in a third party but they are giving the user full responsibility on his own security, which creates another problem of dealing with human errors on top of existing security concerns. This led to numerous cases of lost or stolen keys, i.e. [59, 60]. As a direct consequence non-custodial wallets are suffering from bad user experience. Figure 1 presents our analysis

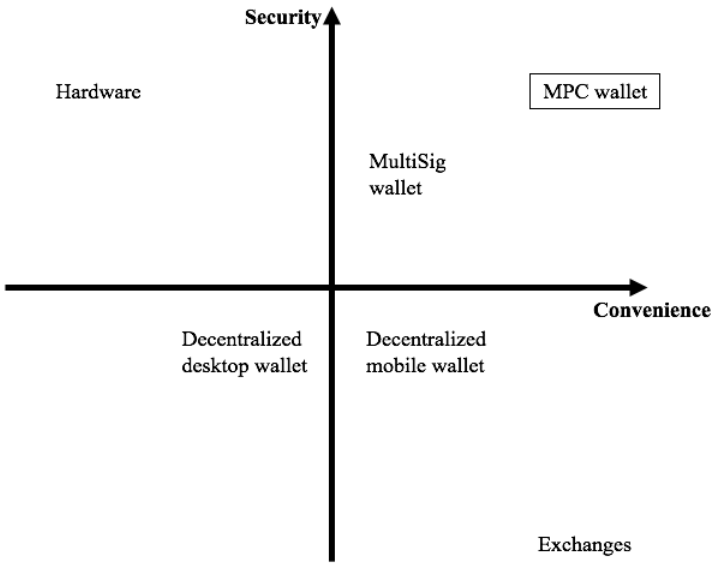

Figure 1: Landscape of custody solution

of current custody solutions landscape measured with respect to security and convenience. The complete analysis can be found in [5]. As can be seen from the figure we currently lack an optimal solution with high level of security and Exchange like convenience. In this paper we propose and provide details on a new solution that answer those requirements.

1.3 Our Framework

Architecture Our model is built in layers: at the bottom we are utilizing MPC as a core technology. We are wrapping MPC with novel cryptographic protocols that will be used as our building blocks in the higher layers. The second layer is an architecture suitable for two party computation to achieve decentralized non-custodial basic wallet functionality. Above it we have a security layer that consists of industry best practices with adaptation to cryptocurrencies landscape. The top layer is the service layer, where we rely on our stack to provide user experience matching a centralized service.
We chose to implement two party ECDSA scheme, where one share of the private key is held by the user (Client) and the other share is held by the service provider (Server). The two shares are generated separately as a result of distributed Key Generation protocol based on Lindell's protocol [3]. Both shares are required to sign a transaction in a blockchain. The signing protocol is also distributed [3] such that the client share never leaves the device that created it. Same goes for the server private key share.
In our architecture the server runs in the cloud and the clients are mobile devices running

native code. Also on the cloud is our database and a host of full nodes, one or more for each blockchain we support. This design allows us to provide services like a centralized service but without trusting a third party for key management.
The Key Generation protocol is boosted to provide the required speed from a mobile application. The Signing protocol is tailored to our client-server-blockchain architecture with added stage for message authentication to make sure the client sign the correct transaction. On the blockchain the transactions looks like regular transactions, i.e. no Multisig scripts or smart contracts.

Building Blocks In the process of building a secure non-custodial service we have developed some utility protocols:

1. **Efficient $(2,3)$-threshold ECDSA** We designed a new protocol, using efficient 3 party computation framework from [10]. We use this variant for our recovery scheme - how to gracefully handle a not-functioning server.
2. **Secure Share Transfer** This new primitive allows to secretly transfer a private key share between users in a publicly verifiable way. We use it for making off-chain transactions with on-chain impact which has many derivatives such as Atomic Cross Chain Swaps.
3. **Private Key Refreshing** We implemented proactive secret sharing for our scheme. We periodically replace the private key shares of the client and the server without effecting the public key and recovery. This makes our architecture a true No Single Point Of Failure since an attacker must get two private key shares at the same time. An attacker with one share after refreshing and the other party share from before the refreshing will have nothing.
4. **Public Key Refreshing** We created our own BIP32 [16] hierarchical deterministic wallet that works for two parties. With this method we can create non-interactively any number of change addresses. We believe our scheme is superior to the scheme offered in [19] subsection 4.4.

Wallet As a Service using our building blocks together with blockchain technologies we are now able to provide users with a wide service layer. Universality or multi-chain support with the highest level of security. Move funds to secure Cold Storage with a click, done with the server deleting his share, leaving a copy of it in cold backup. Minimal fees with fast confirmations and built in mixing using batch transactions. Or no fees at all between clients. clients can also swap assets from different chains using the server as a mediator, improving [20]. Finally, we took care of the corner cases where the server cannot provide service (DoS, Hacked, the team is dead, etc..) and came up with full recovery options. The user can disengage the server at any time without lost of funds.

1.4 Related Work

Multi Signatures (Multisigs) are the traditional way that some blockchain-based cryptocurrencies add threshold security to improve private key management. Multisig requires t private keys out of n to each sign on a transaction for it to be valid. A comparison between Multisig and threshold signatures can be found in [1]. The comparison is for Bitcoin only and therefore we believe that it is lacking some fundamental aspects.

Scalability Multisigs are blockchain specific: Every chain provides a unique way to do Multisig - script (for example Bitcoin), smart contract (For example Ethereum), built-in (for example Ripple) or not at all (for example Neo). We consider MultiSig to be part of the second layer of a blockchain and therefore blockchain specific. Adding a new coin requires building a new code specific implementation. Therefore the addition of coins cannot be scaled. On the other hand threshold signatures are blockchain agnostic: MPC involves changing the digital signature protocol of a blockchain which is in the lowest level of a blockchain. There are only few different possible constructions and therefore once you mastered all of them adding a new chain can be done by just replacing an existing math function with an equivalent already prepared MPC function.Therefore threshold signatures can be scaled to multiple chains.

Address creation In Multisig there is a potential vulnerability during private key generation since it can be done in single point (app, browser). For example BitGo [27] describes address creation as their major weakness because two out of three keys are generated in the client browser. In MPC threshold signatures Key generation is distributed and a private key share never leaves the device.

Privacy In Multisig the access-control policy is encoded in the transaction and eventually publicly revealed. In threshold signatures the transactions looks regular and do not convey the access policy. Mixing can be done to increase privacy.

Key-Evolving In Multisig In order to refresh private keys we need a transaction and we also change the public address. In threshold signatures private key shares can be constantly rotated without going through the blockchain and without changing the public address.

Number of rounds In multisig there is a single round. Transactions are signed offline by multiple parties which creates a delay by design until enough transactions are collected. In threshold signatures we need multiple rounds. Transactions are signed online by multiple parties. The user experience is the same experience as sending regular transaction.

Flexibility In Multisig a signature can be accomplished with any (t, n)-threshold combination. In practice n is a low number (3 or 5) because otherwise the transaction can become very large. In Threshold signatures any (t, n)-threshold combination is possible, with the same transaction size.

Cost Multisig transaction size is linear with the number of signers and in most cryptocurrencies the fees are high than standard. In threshold signature the transaction size is the same as regular transactions and therefore the fees remains the same.

2 Preliminaries

2.1 Definitions

Signature Scheme A signature scheme $\mathcal{S}$ is a triple of efficient randomized algorithms **(Key-Gen, Sig, Ver)**. **Key-Gen** is the key generator algorithm: on input the security parameter 1^λ , it outputs a pair (y, x), such that y is the public key and x is the secret key of the signature scheme. **Sig** is the signing algorithm: on input a message m and the secret key x, it outputs sig, a signature of the message m. Since **Sig** can be a randomized algorithm there might be several valid signatures sig of a message m under the key x; with **Sig**(m, x) we will denote the set of such signatures. **Ver** is the verification algorithm. On input a message m, the public key y, and a string sig, it checks whether sig is a proper signature of m, i.e. if $sig \in$ **Sig**(m, x). The notion of security for signature schemes was formally defined in [41] in various flavours. The following definition captures the strongest of these notions: existential unforgeability against adaptively chosen message attack.

Definition 1. We say that a signature scheme $\mathcal{S}$ =**(Key-Gen,Sig,Ver)** is unforgeable if no adversary who is given the public key y generated by **Key-Gen**, and the signatures of k messages $m_1, ..., m_k$ adaptively chosen, can produce the signature on a new message m (i.e.,$m \in m_1, ..., m_k$) with non-negligible (in λ) probability.

Threshold Secret Sharing Given a secret value x we say that the values $(x_1, ..., x_n)$ constitute a (t, n)-threshold secret sharing of x if t (or less) of these values reveal no information about x, and if there is an efficient algorithm that outputs x having $t + 1$ of the values x_i as inputs.

Threshold Signature Schemes Let $\mathcal{S}$=**(Key-Gen, Sig, Ver)** be a signature scheme. A (t, n)-threshold signature scheme $\mathcal{TS}$ for $\mathcal{S}$ is a pair of protocols **(Thresh-Key-Gen, Thresh-Sig)** for the set of players $P_1, ..., P_n$. **Thresh-Key-Gen** is a distributed key generation protocol used by the players to jointly generate a pair (y, x) of public/private keys on input a security parameter 1^λ . At the end of the protocol, the private output of player

P_i is a value x_i such that the values $(x_1, ..., x_n)$ form a (t, n)-threshold secret sharing of x. The public output of the protocol contains the public key y. Public/private key pairs (y, x) are produced by **Thresh-Key-Gen** with the same probability distribution as if they were generated by the **Key-Gen** protocol of the regular signature scheme $\mathcal{S}$. In some cases it is acceptable to have a centralized key generation protocol, in which a trusted dealer runs **Key-Gen** to obtain (x, y) and the shares x among the n players. **Thresh-Sig** is the distributed signature protocol. The private input of P_i is the value x_i . The public inputs consist of a message m and the public key y. The output of the protocol is a value $sig \in$**Sig**(m, x). The verification algorithm for a threshold signature scheme is, therefore, the same as in the regular centralized signature scheme $\mathcal{S}$

Definition 2. We say that a (t, n)-threshold signature scheme $\mathcal{TS}$ = (**ThreshKey-Gen, Thresh-Sig)** is unforgeable, if no malicious adversary who corrupts at most t players can produce, with non-negligible (in λ) probability, the signature on any new (i.e., previously unsigned) message m, given the view of the protocol **Thresh-Key-Gen** and of the protocol **Thresh-Sig** on input messages $m_1, ..., m_k$ which the adversary adaptively chose. This is analogous to the notion of existential unforgeability under chosen message attack as defined by Goldwasser, Micali, and Rivest [41]. Notice that now the adversary does not just see the signatures of k messages adaptively chosen, but also the internal state of the corrupted players and the public communication of the protocols. Following [41] one can also define weaker notions of unforgeability.

A Note on Namings Throughout the paper we use several names to describe the participants of the system. In most cases we will use the following convention: Users represent the human interface of the system. Client and server will be used to describe two different types of software in our architecture. Party A and Party B will be referenced when describing cryptographic protocol. Service provider is the entity that control the server when services are required.

3 Building Blocks

3.1 Two Party Key Generation and Signing for ECDSA

The problem of two party ECDSA has gained popularity over the last years since the extensive use of ECDSA for cryptocurrencies [1]. Solutions include a set of two protocols: Key Generation for share creation and Signing for using the shares for threshold signatures. The first to tackle the problem were MacKenzie and Reiter [2] back in 2004. Their solution which is extremely inefficient was the only solution for more than a decade until Gennaro et al. [1] tried to generalize the $(2, 2)$-threshold scheme of [2] to (t, n)-threshold with Bitcoin wallet as the main use case. The work of [1] introduces a broadcast channel to the communication model, needs more rounds of communication for signing (6 vs

[1] We describe the building blocks in respect to ECDSA for brevity and since it is considered at the moment the most challenging. Equivalent building blocks for other types of digital signatures with multiplicative or additive secret sharing are also part of our system.

4), requires the same length of Paillier key [6] ($> q^8$ where q is the order of the elliptic curve) and uses Distributed Paillier encryption implemented with a trusted dealer. Even with those limitations the benchmarks for optimized setup were impressive - in the order of seconds for signing operation. In our setting the full Key Generation protocol would need to run only several times, therefore it is not required to be very fast but it is required to be efficient in communication and computing power costs. the Signing protocol would be used repititively and thus should be very fast. We conclude that [1] is not equipped for our needs.
A newer paper, joint work of Boneh, Gennaro and Goldfeder [4] revisited the original Gennaro paper and with the aid of homomorphic encryption claimed to achieve improvement. The number of rounds indeed was reduced (4) but the major caveats remained - length of Paillier key, the distributed Paillier setup and the trusted dealer. From the published benchmarks we can deduce that even with optimizations the time per signature did not improve. Both [4] and [1] are protocols optimized for the general case of (t, n)-threshold and when we try to reduce them to the specific case of $(2, 2)$-threshold we get unnecessary overhead.
Lindell's work [3] published first in CRYPTO17', came out between [4] and [1]. This work also demonstrates a use of Paillier homomorphic encryption but optimized for the specific case of $(2, 2)$-threshold. The result is highly efficient implementation that takes only tens of milliseconds to produce a signatures. The trade-off is expensive key generation which still in comparison to the other works is extremely efficient, taking not much more than 2 seconds. Lindell's protocols do not assume a trusted party or distributed Paillier key generation and the length of the Paillier key becomes much shorter ($> q^5$). The communication model do not assume broadcast channel.
The newest work on ECDSA threshold signature by Doerner et. al [7] rely only on elliptic curve primitives and assumption, just like regular ECDSA and without using Paillier crypto system and its related assumptions. The result is the fastest Key Generation and Signing protocols up-to-date being mainly bounded by network latency. Moreover the Signing protocol requires only one round which can make the $(2, 2)$-threshold signing non interactive. Compared to Lindell's protocol [3] there is order of magnitude improvement in both Key Generation and Signing, but Lindell's protocol is two orders of magnitude smaller in communication cost per signing (769B vs 86kB).
Surprisingly, for certain applications, Lindell's protocol seems to be more efficient than [7]. For example as a building block for payment channels [64] without using scripts or smart contracts[67].

3.1.1 Plain $(2, 2)$-Multi-Signature ECDSA

In this section we describe how we implemented Key Generation and Signing based on [3].
We first describe ECDSA signature scheme:
Assume that Alice has a private key x. the coresponding public key is $x \cdot G$ and she wants to sign a message m. Alice can compute a signature on a message m as follows:

1. Choose ephemeral key k.

2. Compute $R = k \cdot G$.
3. Assume R is the point (r_x, r_y). Then, set r to be the x coordinate of the curve point R.
4. Compute the signature: $s = k^{-1} \cdot (H(m) + r \cdot x)$, where H denotes hash function.
5. Output (r, s)

In our setting the two parties that jointly compute the signature are a server and a client. Naturally the server is has access to more resources and will take the role in the protocols of the party that requires more computing power. In [3] terminology the server will play the role of party P_1 wheres the client will play the role of party P_2.

$(2,2)$ ECDSA Key Generation subprotocol

1: Client initiates protocol
2: Server Chooses a random x_1 and computes $Q_1 = x_1 \cdot G$
3: Server commits to (a) Q_1 and (c) non-interactive zero knowledge proof of knowledge (ni-zkpok) of Q_1 discrete log. Server sends the commitments to the client
4: Client Chooses a random x_2 and computes $Q_2 = x_2 \cdot G$ and sends to the server Q_2 together with a ni-zkpok of Q_2 discrete log.
5: Server verifies the client proof (abort if result is not true) and decommits his commitments and sends to client.
6: The server generates Paillier key-pair [6] and computes $c_{key} = Enc_{pk}(x_1)$ where pk is the Paillier public key. c_{key} and pk are sent to the client.
7: Client verifies the server proof and initiates 3-step zero-knowledge protocol for the server to proof correctness of Paillier key generation. Proof is based on [68] section 3.3 and [69] section 4.2.
8: Client initiates a 2-round zero-knowledge protocol for the server to prove that the value encrypted in c_{key} is the discrete log of Q_1. The protocol is given in [3] section 6.
9: Client initiates a 2-round zero-knowledge protocol for the server to prove that $x_1 \in \mathbb{Z}_q$ where q is the order of the elliptic curve. The protocol is given in [3] Appendix A.

All commitments in the protocols are done using SHA256. Our Paillier cryptosystem and zero-knowledge key correctness implementation are optimized and can be found in [70]. The protocol code can be found in [72]

In 4.1 we show (a) how to generate a session ID for every instantiation of the Signing subprotocol, and (b) how to get to the point that the server and client have the same message to sign view.

3.2 $(2,3)$-threshold ECDSA Protocol

Lindell's Key Generation protocol works for $(2,3)$-threshold if we implement the functionality $\mathcal{F}_{rand}$ from [10] for generating shares of random values non-interactively. This will allow three parties to create random shares in the following way:

$(2,2)$ ECDSA Signing subprotocol

1: Server and client repeat steps 1 - 4 of the Key Generation protocol but for ephemeral key-pairs: k_1, R_1 and k_2, R_2 for the server and client respectively.
2: Server computes $R = k_1 \cdot R_2$. Client computes $R = k_2 \cdot R_1$. From the same R both can extract the x-coordinate $r = r_x$.
3: Client computes $c_1 = Enc_{pk}(k_2^{-1} \cdot m' + \rho q)$ and $c_2 = {c_{key}}^{x_2 \cdot r \cdot k_2^{-1}}$. Here ρ is some random number. The client then computes and sends $c_3 = c_1 \oplus c_2$ where $\oplus$ denotes the additive homomorphic operation of Paillier cryptosystem.
4: Server decrypts c_3 to get s'. The server computes $s = s' \cdot k_1^{-1}$ and output (r, s) as the ECDSA signature.

Each party i chooses a random r_i in the elliptic curve field and sends it to party $i + 1$. The full private key will be $r_1 r_2 r_3$. The roles of prover and verifier between each two parties that want to run Key Generation protocol will be determined according to the following rules:

rule1 Each party will be a prover for the party that received its r_i and will be the verifier for the party that sent to her the r_{i-1} key.

rule2 The private key of the Prover will always be $r_i r_{i-1}$ and the verifier private key will always be r_{i+1}.

The Key Generation protocol stays the same at its core. We will replace only the first stages of the protocol with a 3 party protocol to create the random shares for malicious adversaries. The new Key Generation protocol is now:

Protocol 1: $(2,3)$-threshold ecdsa protocol

1: Each party i calculates $\alpha_i = r_{i-1} \cdot G - r_i \cdot G$ and sends it to the other two parties
2: Each party i checks that $\sum \alpha_i = 0$ if not it aborts
3: For any two parties: the prover P_{i-1} (who will be the prover is decided according to **rule1**) defines $x_1 = r_{i-1} \cdot r_{i-2}$ and sends $Q_1 = r_{i-1} r_{i-2} \cdot G$ to the Verifier
4: The verifier V_i check that $Q_1 = r_{i-1} \cdot (\alpha_{i-1} + r_{i-1} \cdot G)$. if not it aborts.
5: The verifier V_i defines $x_2 = r_i$ and sends $Q_2 = r_i \cdot G$ to the Prover.
6: The prover P_{i-1} check that $Q_2 = \alpha_{i-2} + r_{i-1} \cdot G$. If not it aborts
7: prover P and verifier V continue with the original Key Generation protocol. starting at stage 1

Note, numbering in this protocol is circular, for example for P_1 and V_2 we get $x_1 = r_1 r_3$ and $\alpha_1 = r_3 \cdot G - r_1 \cdot G$.
correctness. If all three parties are honest, every two parties will have together:
$x_1 \cdot x_2 = r_1 \cdot r_2 \cdot r_3$.
security. We use elliptic curve as a PRF. There are two issues we need to address here. The first is to make sure that one party can extract the full private key with negligible

probability. The second is that Q_1, Q_2 are public keys with private keys that correspond to the random shares generated before. The first issue is reducible to elliptic curve security. The second problem is answered with the checks we do inside the protocol. The second issue is not relevant to the two party case because we don't care about the structure of x_1, x_2 all we care is that the party that sent the public key Q_j knows the private key for the specific elliptic curve. The rest is taken care of by the protocol. In the 3 party case we need to make sure that the public key is from a specific private key. In this case we use the replicated secret sharing to verify the correctness of the public keys.

3.3 Share Transfer

In our suggested server-client architecture, two shares of the private key are generated separately. This construction makes the server power restricted since it will never control or own the full private key. Another implication is that the client that controls the second private key share is the owner of the matched public address/ account / unspent transaction. This fundamental change from the classical private key ownership gives rise to a new way to off-chain-but-on-chain (OCOC) transfer of funds/assets. Described here is a method to transfer the client private key share to another client in a secret and verifiable way. It is considered off-chain since no new data is appended to the chain (i.e. transaction) and it is effectively on-chain change since the control of the private key has changed hands. We require that the server will have only validation power and no change to the minimal level of trust we need in normal operation. As will be explained later, this primitive building block is the basis for many services such as shared wallet, frictionless peer to peer transactions, inheritance and more.

Formalizing the initial setting: We have three parties, C_1, S, C_2. All communication goes through S. party C_1 wants to transfer to C_2 a secret value x_2. Party S wants to validate that the encrypted values is indeed C_1 private share x_2 encrypted under C_2 public key and that C_2 will be able to extract x_2. C_1 is the prover+encryptor, S is the validator, C_2 is the receiver.
This setting reflects the state after a client and the server run Key Generation protocol and now the client wants to do OCOC transfer to another registered client. The full protocol is given in protocol 2.

Share Transfer Ideal Functionality $\mathcal{F}_{st}$:
Functionality $\mathcal{F}_{st}$ works with parties P_1, P_2 as follows:

- Upon receiving (transfer, sid,x) from party P_i record (sid,x,i). If sid already exists then ignore the message.
- Upon receiving (verify, sid, Q): if record with sid exist and $Q = xG$ for G generator of the EC add to the record (verified). Otherwise ignore the message.
- Upon receiving (share, sid) from party P_{3-i}: if there exist a record (sid,x,i, verified) then send x to party P_{3-i}. Otherwise ignore the message.

Protocol 2: Share transfer protocol

Joint input: (1) elliptic curve parameters (2), C_2's public key: Y ,(3) C_1's public key $Q_2 = x_2 \cdot G$

Verification:

1: C_1 computes n tuples of 5 group elements. $\forall\ 0 < i \leq n$:

$$\{D_1^i, D_2^i, D_3^i, D_4^i, D_5^i\} = \\ = \{t_i[x_2]_i G,\ t_i r_i Y,\ t_i r_i G,\ [x_2]_i G + r_i Y,\ r_i G\}$$

$t_i,\ r_i$ are random field elements, $[x_2]_i$ is the i'th segment of x_2 shifted by $s_i = 256\frac{i-1}{n}$ bits to the right (there are $bl = 256/n$ bits in each segment).

2: C_1 sends the n tuples to S

3: Zero-Knowledge proof of correct encryption:

  1: C_1 chooses random $z \in Z_q$, computes $a = z \cdot Y$ and sends a to S

  2: S computes $D_4 = \sum_i 2^{s_i} D_4^i$ and then $T = D_4 - Q_2 (= r \cdot Y)$ where $r = \sum_i 2^{s_i} r_i$

  3: S sends C_1 a challenge c

  4: C_1 computes $f = z + r \cdot c$ and sends f to S

  5: S checks that $f \cdot Y = a + c \cdot T$. Otherwise abort.

6: Proof of extractability:

  1: C_1 generates n range proofs for, proving that all D_4^i are Pedersen Commitments to values $< 2^{bl}$. Range proofs can be done efficiently using Bulletproofs [13]

  2: C_1 generates $2n$ Proofs of Membership of the same scalar using Chaum and Pedersen protocol [14] adapted for elliptic curves (protocol is given below): for $0 < i \leq n$: EC-DLEQ(Y, D_2^i, G, D_3^i), EC-DLEQ$(D_4^i, D_1^i + D_2^i, D_5^i, D_3^i)$

  3: C_1 and S runs a sigma protocol where C_1 proves knowledge of k such that $D_1^i = kG$

Sharing:

1: S transfers all n pairs D_4^i, D_5^i to C_2

2: C_2 extracts $[x_2]_i$ from every pair using brute force on $D_4^i - yD_5^i$ where y is Y matched private key, known only to C_2

3: C_2 combines the $[x_2]_i$'s to get the secret $x_2 = \sum_i 2^{s_i}[x_2]_i$

4: C_2 verifies that $Q_2 = x_2 \cdot G$

5: C_2 and S initiates Key Rotation (protocol 3) to get new shares x_1' and x_2'.

6: S deletes his old share x_1

CP93 DH tuple proof of membership

For group elements G_1, H_1, G_2, H_2 prove knowledge of x such that $H_1 = xG_1$ and $H_2 = xG_2$. We denote this protocol by EC-DLEQ(G_1, H_1, G_2, H_2)

1: The prover sends $a_1 = wG_1$ and $a_2 = wG_2$ to the verifier with $w \in_R \mathbb{Z}_q$

2: The verifier sends a random challenge $c \in_R \mathbb{Z}_q$

3: The prover responds with $r = w - xc (mod\ q)$

4: The verifier checks that $a_1 = rG_1 + cH_1$ and $a_2 = rG_2 + cH_2$

Analysis $[x_2]_i$'s are splits of x_2 such that the rest of x_2 is zeroed, for example $[x_2]_1 = 00..00[x_2]_1$ and $[x_2]_n = [x_2]_n 00..00$.
As can be seen from the protocol, the Verification part between the sending client and the server is more complex so we start with explaining the Sharing part. Sharing is run between the server and the receiving client. The server can decide to deny the client from receiving or send it wrong group elements. the receiving client gets n group element pairs. For each pair i the client strips the blinding using its private key y and brute force to get $[x_2]_i$ (guessing a value x and compare xG to $D_4^i - yD_5^i$). Finally, combining the $[x_2]_i$ segments, the receiving client tests that the secret share corresponds to the public key of the sending client. A cheater server could be detected in such a way. There is an open question on how to make sure that the sending client public key is indeed public (and also that the receiving client public key is public) but here we assume that this is the case. Ideally it would be good to use segments of size around two bytes such that it would be very fast to brute force and n would not be large.
By deleting x_1 the server finalizing the transfer because even if C_1 will hack the server the corresponding share is lost forever therefore he cannot use this private key anymore.
Moving to the $Verification$ protocol: **zero-knowledge:** can be shown by noticing that D_4^i's are perfect hiding Pedersen commitments and that $\{D_1^i + D_2^i,\ D_3^i\} = t_i\{D_4^i,\ D_5^i\}$ such that the only possibility for the server to extract $[x_2]_i$ is brute forcing D_1^i which has negligible success probability for random t_i assuming EC-dlog is hard.
Correctness and Soundness: Lets parameterize the i'th tuple such that from the point of view of the server we have 5 group elements: (A, B, C, D, E). Using (6.2) and (6.3) we get: $(1)B = xY, (2)C = xG, (3)C = nE, (4)A + B = nD, (5)A = kG$ where x, n, k are field elements unknown to the server. Plugging (1) and (5) in (4) we get $(6)nD = kG + xY \Rightarrow D = n^{-1}kG + n^{-1}xY$ which is a Pedersen commitment as wanted. Comparing (2) and (3) we get $(7)nE = xG \Rightarrow E = n^{-1}xG$. Plugging $Y = yG$ for y the private key of the receiving client (not known to the server), equation (6) becomes $(8)D = n^{-1}kG + n^{-1}xyG$. Finally plugging (7) in (8) the server can verify that $D - yE = n^{-1}kG$ meaning the blinding was cancelled. Combine with range proof makes $n^{-1}k$ a small field element that can be brute-force (true for all i's). The zk proof of correct encryption proves that sum $D - yE$ over all i's equals x_2G concluding the proof that a client with private key y will be able to extract the secret share x_2. A sending client that will try to cheat will fail with high probability in proving one of the EC-DLEQ or in the proof of correct encryption (reduced to hardness of EC-dlog). The three proof types in Verification can be made non-interactive using Fiat-Shamir heuristic. It can be publicly verified by any party that learns the public keys Q_2 and Y in a trusted way. It is best if C_1 will learn Y using a direct channel between C_1 and C_2 i.e. email or QRcode. This is to avoid the situation where the server publish a Y' to C_1 such that only the server knows the matched private key.

3.4 Private Key Rotation

We want to enable key rotation , a method that will keep the private key and public key the same but will change the private key shares [11]. The immediate benefit is that an attacker

have to get the two key shares before a KeyRotate operation. If the attacker has one share before a rotation and the other after a rotation he will not be able to recover the private key. Likewise if the attacker has few rotated shared keys of one side it does not help him at all if he can not get one of the shares of the other side from one of these specific rotations. Assuming the server and the client already run the two party protocol for key generation at least once the key rotation protocol is :

Protocol 3: Key rotating protocol

- Run a two party coin toss between the server and client and get a random string s
  1: Server computes: $x'_1 = s \cdot x_1 \mod (q)$
  2: Client computes: $x'_2 = s^{-1} \cdot x_2 \mod (q)$
  3: Both parties compute: $Q'_1 = s \cdot Q_1$, $Q'_2 = s^{-1} \cdot Q_2$ and $c'_{key} = c^s_{key}$
  4: Server and Client run (2, 2) ECDSA Key Generation subprotocol and verify the signature using Q as verification key. If verification fails - abort.
- The coin toss protocol is Blum coin toss :
  1: Client generate random numbers u_2, and v_2, both $\mod (q)$.
  2: Client sends $com = H(u_2, v_2)$ to server.
  3: Server generate a random number $u_1 \mod (q)$ and sends to client.
  4: Client calculates $crs = u_2 \oplus u_1$.
  5: Client sends the server u'_2, v'_2.
  6: Server checks If $H(u'_2, v'_2) = com$. If not - abort.
  7: Server calculates $crs^{-1} = (u'_2 \oplus u'_1)^{-1} \mod (q)$.

The coin toss protocol is secure in the random oracle model and allows for the client to abort after seeing u_1.
A full security proof in the hybrid model can be found in [25]. $(s)^{-1}$ is the inverse of s modulo q.
We take advantage of the homomorphic properties of Paillier encryption to update the server secret share.
We use $xor : \oplus$ operation between the client and server shares to avoid a situation where one of the parties use zero for his coin toss input which will zero the private keys. Using $\oplus$ means that we will get pseudorandom new keys in the presences of one malicious party. Moreover, If one of the parties had control over the crs it could potentially have led to compromising the other party secret share. For example the malicious party can use engineered strings to find the original secret share of the other party (using a zero string with one on bit and then guessing this bit). The coin toss defends the secret shares from the other party. This protocol can theoretically require only one pre-processing randomly generated string and produce then the parties can rotate non interactively indefinitely. The problem is that it will miss the point of rotation because an attacker that attacks one party and get the random string can generate from the random string all future rotations such that next when we attacks the second party he can get the full key. Thus we require interaction for every rotation. We assume secure communication channel. Otherwise a man in the middle can learn the coin toss string. This does not help much to an attacker but it means that as

long as the attacker knows the coin tosses if he have an old version of one of the shares, he can roll this share himself. We can mitigate it by using a more sophisticated coin toss protocol that does not assume secure communication. The protocol assumes the parties will dump the old shares but in fact even after rotation the old shares will still work. A better solution will therefore be to invalidate the old shares as part of the key rotation protocol. This protocol is suited for multiplicative sharing of the private key such that $key = s_1 \cdot s_2$, for additive shares there is a similar protocol.

3.5 Two Party HD Wallet

In BIP32 [16] there is a method for child key derivation from a parent key. This method is extended using indices and tree structure to create infinite number of public-private key pairs from one master key. Using an agreed hierarchical system, as specified in another BIP [18] enables the creation of hierarchical deterministic (HD) set of elliptic curve key pairs. The HD property can be used for variety of use cases such as generating change addresses or assigning one master key for many cryptocurrencies for example. In our setting we use a two party key generation algorithm - the server holds a share of the secret key and the client holds the second share. Together and without revealing each other share of the secret key the server and client can sign together ECDSA . The two party key generation scheme (TPKG) assumes both parties know the base point G of the secp256k1 elliptic curve and all the rest of this elliptic curve parameters [15].

After master key generation, the client C knows $\{x_2; Q = x_2 \cdot Q_1\}$, the server S knows $\{x_1; Q = x_1 \cdot Q_2; d\}$ and their joint input is $\{Q_1; Q_2; C_{key}; N\}$. Where N is the public key of Paillier encryption system, d is the private key and C_{key} equals an encryption of x_1 with public key N.
Two party HD wallet protocol target is to allow for the deterministic generation of change addresses for the same user account. Therefore, the purpose of the scheme is to change the public key in an efficient way. The security lies on the Key Generation protocol. All the derivation is based on public parameters beside the derivation of the first chain code which is intended to make sure that the HD public keys are private. This is in contrast to the "Key Rotation" scheme where the target is to refresh the private key shares without affecting the public key. The implication is that in this scheme we do not need to replace the Paillier public key N and in "Key Rotation" we need to do replacement as part of the refreshing. In the HD protocol scheme the derivation should rely on public parameters.
We will work only for normal, not hardened, childs ($index < 2^{31}$) because the secret key is not known to any of the parties. The scheme below is based on the scheme from [16] and uses some of its terminology.

We can repeat this scheme for different $index$ values to get different key pairs for the same hierarchy level. For each $index$ we can calculate the next hierarchy level by repeating the scheme with inputs k_S, k_C for the server and client respectively and common inputs $\{cc_{new}, C_{key_{new}}, K_s, K_c, Q_{child}\}$.

Protocol 4: Two party HD wallet

- Create common chain code: we invoke Diffie-Helman (DH) key exchange between client and server to create a common secret chain code cc
- Both parties calculate:
  1: $I = hmac - sha512(key = cc, data = Q||index||0)$
  2: $I = I_L||I_R$ where $|I_L| = |I_R| = 32bytes$
  3: $K_c = I_R I_L \cdot Q_2$
  4: $K_s = I_R^{-1} \cdot Q_1$
  5: $Q_{child} = I_L \cdot Q$
  6: $C_{key_{new}} = C_{key}^{Enc(I_R^{-1})}$
  7: $cc_{new} = hmac - sha512(key = cc, data = Q||index||1)[0..255]$
- C private key share is $k_c = I_R I_L \cdot x_2$
- S private key share is $k_s = I_R^{-1} \cdot x_1$
- The two parties create a joint signature for message string of zeros and verify it alone.

Analysis Another implementation can be found in an earlier version of [1] (can be found in [19] section 4.4). There are a few problem with the referenced implementation:

1. It is not specified how chain code is generated and shared.
2. There is only one chain code, meaning it is not rotating and once you know the chain code you immediately break privacy (attacker knows all public keys in the chain).
3. It suffices to know one private key of non-leader to know all of his private keys (they stay the same across the tree).
4. The protocol does not state how leader is elected.
5. Not Compatible with BIP32.

In our protocol:

1. Chain code is generated and shared using DH key exchange. This way guarantees that for one honest party we will get pseudorandom chain code, known only to the two parties.
2. The chain code is refreshed for every new key in the chain, the new chain code derivation guarantees it will be pseudorandom
3. In our scheme you need to know the extended public key (chain code included) plus child private key to get all the secret keys descended from that public key of that party.

4. Verifier stays verifier and prover stays prover.

Both schemes are non interactive.
In our scheme we do not need to run proofs from Key Generation protocol again. Specifically no need to make range proof or proof of Paillier encryption. Both parties use the existing C_{key} together with shared I_R to calculate the new C_{key}. If one of the parties decides to cheat, the signature will not be verified. The scheme described here is suited for specific elliptic curve and ECDSA. For different elliptic curves and digital signature algorithms a similar variant of the protocol will be used.

4 Wallet As a Service

4.1 Securely Signing Transactions

Implementing [3] Signing Subprotocol Sign(sid, m) requires the parties to agree on two elements: (1) the session identification sid and (2) the message to sign. In our server and client architecture only the server has local access to blockchain full nodes. Assuming we cannot rely on API requests the server is the client only channel to communicate with the blockchain for sending and receiving funds. This can potentially lead to an attack of malicious server, which is now able to manipulate the blockchain view of the client or to change the message to be signed when client wishes to generate a transaction Reusing sid can lead to further attacks.
Process of establishing a transaction: The digital signature is on the hash of the message and from the pre-image resistance property of the hash function it is impossible for the client to know what is the original message that he signs on. Even if we assume that there is an easy solution for this, the server will still have the power to manipulate the message by changing amounts for example. In some cryptocurrencies that are based on transfer of ownership and not on account the transaction could also be manipulated by changing the input transaction IDs which are the hash digests of previous transactions that the current transaction wish to redeem. Here are the steps we advice to take prior to jointly signing a transaction. We will describe the steps in Bitcoin framework and terminology but the concept can be generalized trivially to other cryptocurrencies.

1. The server should query the Bitcoin full node periodically for any changes in the user balance. The server should maintain a list with all the user's confirmed unspent transactions (UTXOs) and send regular updated to the client software while it is online. The update includes:
   (a) total balance
   (b) $txid$ (transaction ID) of all UTXOs
   (c) all available details in the blockchain on every UTXO
2. The client verifies:

(a) Every $txid$ received in (b) is the hash digest of the raw transaction of that UTXO as received in (c).

(b) The updated balance is equal to the sum of the UTXOs amounts.

3. When client initiates a new transaction he chooses the output amount and to what address to transfer the amount and send this parameters to the server.
4. The server compose the transaction tx in optimal way, according to predefined policy about fees and other variables such as number of UTXOs to redeem.
5. The server verifies that tx inputs never before appeared in the blockchain, otherwise - abort.
6. The server sends the client the transaction details and the hash m on which it wants to sign.
7. The client verifies the transaction variables is according to the policy and that the digest of the raw transaction is indeed the message hash m he received from the server.
8. The server initiate protocol to **generate sid** (see below).
9. The server and client run Sign(sid, m).

Analysis In case the server uses wrong $txid$ the transaction is invalid and the full node will block it upon sending. In case the server tries to change the output value the client hash will be different from the hash he received from the server. In case the server tries to change the output address the client hash will be different from the hash he received from the server. The server cannot add or remove outputs. In case of high fee transaction, adding another output could have tunnelled some of the fees to a new output address. Adding an output by the server will result in mismatched transaction hash digest. The outputs must be part of the signature hash (SIGHASH_ALL in bitcoin terminology). Otherwise the server could send the client the correct values but create a malleable transaction with different outputs. The server cannot add more $txids$ because he is limited by the balance known to the client. The server cannot hide part of the $txids$ to get the client to pay more fees because of more inputs. This is because the client keeps in memory the last known balance. Only using MPC signature protocol the balance could have become lower and since the user was not connected, the balance has no way to become lower than what is saved on the device. It is possible that incoming transactions created more UTXOs while the user was offline. Theoretically it is possible that the server will hide them and not update the client. It is likely that the account will be expecting an income transaction and if not seeing it he will not hurry to commit a new transaction before understanding where are the funds (sending to himself to self check the server for example). In the case the client do not expect a new payment he will not count on it when making a transaction. This is a possible weakness because using the new UTXO the client could have potentially achieved a better fee. It is highly unlikely since the server has no incentive to do this.

	Bitcoin	**Ethereum**	**Monero**
network	bitcoind [34]	etherscan-api [31]	monerod[35]
wallet	bitcoin-cli [34]	custom	monero-wallet-cli [35]
modified code	bitcoinjs-lib [32]	ethereumjs-util[33]	monero project [35]

Table 1: Implementation details

Protocol **generate sid** is based on two party implementation of the $\mathcal{F}_{init}$ protocol from [55]. Basically: The server and the client exchange random numbers and the sid will be equal to the concatenation of the two random numbers $r_s||r_c$. Each party makes sure that her random number appears in sid. It gets a bit more complicated for three or more parties but the concept stays the same.

4.2 Universality

One of the major advantages of using threshold cryptography is the scalability of few signature scheme to work with multiple chains. This is in contrast to Multi-Signatures.
Scanning the top market cap currencies [26] we found only a handful of used elliptic curves: $secp256k1$, P-256 and $ed25519$ [30] to name a few, only a handful of signature schemes, most dominant are ECDSA, EdDSA, Schnorr, and Ring Signatures. To prove our statement we completed two proof of concepts (POCs):

1. show that we can use the same Key Generation and Signing protocols for Bitcoin and for Ethereum. The only wallet service today that can support Multisig in both Ethereum and bitcoin is BitGo [27] where Ethereum support is still considered work in progress and has started at 2016. We have successfully created a $(2, 2)$-threshold Ethereum signature in less than a day once we had ECDSA keyGen and Sign protocols.
2. support two party computation for Monero digital signatures. We chose Monero because it allowed us to test the ability to support $ed25519$ elliptic curve and ring signatures with confidential transactions.

Table 1 shows the different tools and code we used and modified in the blockchains we handled. Our POCs were composed of two stages to test address generation and transaction validity. In the first stage we used the appropriate MPC key generation protocol, either for ECDSA or for $ed25519$ together with the cryptocurrency specification on how to create private-public key pair. Once a public address is generated we sent coins to this address from a regular address we generated in the wallet. The second stage was meant to test the MPC sign-transaction protocol and our ability to integrate our code in existing libraries. The all process of creating and signing a transaction remains the same in high level except no private key is introduced at no point in time and we replaced the original call to the sign function with a wrapper that eventually call MPC sign. We made sure that we can

send spendable funds to new regular addresses and to new MPC addresses. We validated the transactions by sending the funds from the receiving addresses and using a blockchain explorer to track them later. Finally we checked that properties of the cryptocurrencies still holds, i.e. confidential transactions in Monero, Segwit in Bitcoin, etc. For completeness we mention that the only difference to the ECDSA sign protocol between Bitcoin and Ethereum is the addition of Recovery ID [36] to the digital signature.

4.2.1 (2,2)-threshold Monero Confidential Transactions

We believe that two party Monero ring signatures is of separate interest since it was not done before. The original Monero protocol was based on CryptoNote [28], which uses ring signatures and one-time keys to hide the destination and origin of transactions. It was later changed to a modified type of ring signature called A Multi-layered Linkable Spontaneous Anonymous Group signature (MLSAG) which allows for hidden amounts on top of hidden origins and destinations [29] thus achieving a confidential transaction according to definition noted in [37]. We will first describe Key Generation (KeyGen) in Monero and then the signature scheme.

(2,2)-threshold Key Generation KeyGen in Monero requires to (a) understand how addresses are composed and (2) work with $ed25519$ elliptic curve. As it appears to be Monero key is actually two keys - a spend key and a view key. Spend key is required for transactions and view key is good for tracking, audit etc. see end of section 4.3 in [28]. Since we care for signing and we want all the rest to be made as easily as possible to manage we decided to use MPC only on the spend key. The view key is not splitted but instead known to both parties, server and client. This can only affect privacy but since we already trust the server for privacy this is not an issue. The two private shares of the spend key are pseudorandom numbers in the finite field for $ed25519$. The spend private key will be the equal to the addition of the shares $b = b_1 + b_2$ and the public spend key will be $Q = b \cdot B$ where B is the elliptic curve base point and $(\cdot)$ means we add B to itself b times.

(2,2)-threshold Ring Signature The original Cryptonote ring signature [28] is still maintained in Monero wallet code and used when a user wants to sign an arbitrary message, not a transaction. The current method for signing, MLSAG, creates a full confidential transaction, including the amount [29]. Protocol 5 describes how to make two party computation of the original ring signature scheme and then we provide the adaptation for MLSAG. All the parameters symbols (L, R, c, r) are the same as in the Cryptonote signature protocol.

Security Analysis: The ephemeral secret key is $i = b_1 + b_2 + H_s(aR)$. As can be seen from the protocol it is never at one place. The server share b_1 never leaves the server and the client share b_2 never leaves the client. The image public key I, and L_s, R_s are all created using local scalar to point calculation and addition. Reversing a point back to the

Protocol 5: (2,2)-threshold Monero Ring Signature protocol

1. Client generates a random share b_2
2. Server generates a random share b_1
3. Client and server generate together (Blum coin toss) a view key a
4. Per transaction we generate the image public key (ephemeral key) like this:
   (a) Client generates $I_2 = b_2 \cdot B$
   (b) Server generates $I_1 = b_1 \cdot B + H_s(aR) \cdot B$
5. Per transaction we generate two shares of a random point and use them to securely compute the signature :
   (a) Client generates a random number q_2
   (b) Server generates a random number q_1
   (c) Client computes $R_{s2} = q_2 \cdot \mathcal{H}_p(P_s)$. Client sends to the server a commitment to R_{s2} and zero knowledge proof of knowledge ($zkpok$) of its discrete logarithm.
   (d) Server computes $R_{s1} = q_1 \cdot \mathcal{H}_p(P_s)$. Server sends to the client a $zkpok$ of R_{s1} discrete logarithm.
   (e) Client sends the Server decommitment of R_{s2}
   (f) Client and server repeat steps (c) – (e) for $L_{s2} = q_2 \cdot B$ and $L_{s1} = q_1 \cdot B$ respectively.
   (g) Client and server repeat steps (c) – (e) for the ephemeral key shares I_2 and I_1 respectively.
   (h) Client and server computes $R_s = R_{s2} + R_{s1}$
   (i) Client and server computes $L_s = L_{s2} + L_{s1}$
   (j) Client computes $r_{s2} = q_2 - c_s \cdot b_2$ and send r_{s2} to the server.
   (k) Server checks that $r_{s2} \cdot B$ is equal to $L_{s2} - c_s \cdot I_2$. If not - server aborts.
   (l) Server computes $r_{s1} = q_1 - c_s \cdot (b_1 + H_s(aR))$ and send r_{s1} to the client.
   (m) Client checks that $r_{s2} \cdot B$ is equal to $L_{s1} - c_s \cdot I_1$. if not - client aborts.
   (n) Client and server computes $r_s = r_{s1} + r_{s2}$.
   (o) Server computes L_i, R_i, c_i, r_i for all $i \neq s$ according to the Cryptonote ring signature protocol and output signature σ

scalar is hard assuming elliptic curve discrete logarithm is hard problem. The scalars r_{s1} and r_{s2} are using q_1 and q_2 as mask values. If we look on r_{s2} (same analysis goes for r_{s1}) we can build a reduction to Decisional Diffie Helman (formal prove is in the extended version). An adversary that can tell **whp** if $r_{s2} \cdot B$ is composed from $q_2 \cdot B$ and $b_2 \cdot B$ or from a random scalar can be used to break DDH assumption. Another way to look at it is that the only possible way for the server to know b_2 from r_{s2} is by knowing q_2 which means he can calculate it from $q_2 \cdot B$ which is hard according to elliptic curve discrete log assumption.

Adapting to MLSAG The original protocol is described on section 2.2 in [29]. The concept stays the same. The ephemeral key calculation stays the same. Instead of q_1, q_2, we have α_1 and α_2. Instead of r_{s1} and r_{s2} we have: $s_{j_1} = \alpha_1 - c \cdot x_1$ and $s_{j_2} = \alpha_2 - c \cdot x_2$ Where $x_1 = b_1 + H_s(aR)$ and $x_2 = b_2$ are the ephemeral private key shares as before. The security stays the same for the same reasons and same assumptions.

4.3 Master Key

In this section we give a formal description of a wallet key management lifecycle. We translate the concept of key management in our setting to the following requirements:
(1) After authentication the server side system is required for each user to generate all historically used addresses of this user for all different possible coins.
(2) Both client and server need to load all the corresponding private key shares, updated to the latest rotation.
(3) The client must have an efficient way to encode and recreate all his secret key material in case of a needed recovery.
In order to answer all this demands it is essential to introduce a notion of Master Key (mk). The mk will be a singular point in the key-pairs tree that will enable to build the tree without having the client deal with memory requirements or keeping state. As par BIP32[16] we need initial raw entropy to generate mk. We will now describe how to create this entropy, how to create mk from this entropy and how to create the key-pair tree from mk.

4.3.1 Passphrase → Entropy

The common practice in todays wallets is to use BIP39[17] where 12-24 words mnemonic + optional passphrase are cryptographically converted to seed entropy. We will use aezeed, which is a newer method, used currently by Lightning Labs [54]. This method provides a ciphertext mnemonic, which means it can be public and the user only needs to remember or keep secret a short passphrase. This passphrase can later be used as one option of user key recovery. Moreover, since the mnemonic is a ciphertext enciphered with the passphrase it is possible to replace passphrase and to ”rotate” the mnemonic by decrypting and encrypting the mnemonic with the new passphrase. Aezeed also contains two extra fields that might come up handy in storing useful information like timestamp and version.

4.3.2 Entropy $\rightarrow mk$

We start with the same practice from BIP32[16]: calculate $I = hmac(Entropy)$ with key equals to "master seed" and $sha512$ hash function. We will get 512 random bits. We then use $hmac$ again for key material expanding according to the construction of *hmac key derivation function* (HKDF) [56]. We will expand to get new seed for every used elliptic curve and every digital signature algorithm according to a predefined order. For example the first seed will be for ECDSA using $secp256k1$ curve, the second seed for ECDSA using $P256$ curve, the third seed for $ed25519$ and so on. Since not many types of elliptic curve based DSA are being used for blockchains and in general the diversity is small it will make the list manageable with small memory footprint.

4.3.3 $mk \rightarrow$ Key-pair Forest

Upon user request to add a coin that utilizes a seed's designated elliptic curve and DSA, the seed will be used as the randomness for running $(2, 2)$-threshold Key Generation with the server. i.e. for Bitcoin the client and server will run $(2, 2)$-threshold ECDSA. The resulting client secret share will be used as root for HD tree, i.e. for Bitcoin we use HD-wallet (protocol 4). Other HD protocols will be similar with only a change of the elliptic curve or in worst case scenario a change from multiplicative to additive parameters. We get extra random bits (usually $256bit$) with every new root seed. This bits can be used for the chain code part of the client. This root shares will be saved, using client encryption, on the server side, and will be used as another option for user key share recovery as detailed in 4.6.4. It is possible to support more than one coin within the same tree because the coin type can be defined as part of the hierarchy fields as in BIP44[18]. Dividing the trees based on algorithm is therefore preferable on division based on coins.

Commutativity of HD-tree and Key Rotation Since both HD-tree (3.5) and Key Rotation (3.4) protocols are multiplicative we get the important property of commutativity modulo the field order between them:

$$\mathrm{Rot}(\mathrm{HD}(mk)) = \mathrm{HD}(\mathrm{Rot}(mk)) \tag{1}$$

The application for a wallet service is that the client needs only keep in static memory the mk's. The master key is the only rotating element in the system and it facilitates rotation for the entire HD-tree client shares without changing the HD-tree public addresses. New addresses can be generated and they will stay consistent after master key rotation. In fact, the server as part of the message integrity protocol (4.1) is in charge of holding the user tree indices of active addresses, the client can reconstruct the tree either from the server or locally in case it saves this indices to static memory.
It is important to mention that HD and Rot in this paper are for the case of multiplicative shares and a second version that fits to additive shares will be used for the linear digital signature schemes trees.

4.4 Atomic Cross Chain Swaps

Atomic cross-chain swaps is the notion of trading cryptocurrency from one blockchain with its value equivalent in cryptocurrency from a different blockchain. Atomic means that the mutual transfer will occur fully or not at all. In this section we will suggest in-wallet atomic swaps protocol that relies on our architecture of communication between two clients through trustless server.

The problem: describing it in simple words the core security issue is trust. Let us try to describe the existing landscape today:

(a) In centralized-custodial exchanges, for example Binance [42], the exchange is a trusted third party. Funds are deposited to the exchange platform, trading is done within the platform and finally the funds can be withdrawn to a private wallet. From the moment of deposit and until withdraw the funds are owned by the exchange which at any moment can be hacked, stop working or become malicious and steal all the funds currently on the platform. There are many examples to security problems with this model, the most famous is Mt. Gox [49].

(b) There are centralized but non-custodial trading systems, for example ShapeShift [43]. This platforms guarantee security before and after the trade but in the middle after funds has been sent to the system and before they returned we put trust in the trading platform, which again can be hacked,suffer denial of service (DOS) or become malicious and steal the funds.

(c) Decentrelized peer to peer exchanges, such as $0x$ [44], are trustless platforms that settles trades on the blockchain. They can suffer from issues in security such as frontrunning [45] and performance since they cannot offer real-time cross chain trades. The on-chain trading makes scalability another issue.

Two new protocols that worth mentioning are Tesseract project [21] and Commonwealth Crypto [46]. Tesseract is using Intel SGX enclave as trusted execution engine to provide a better trusted centralized custodial exchange. The problem with this method is that the secure enclave is a single point of failure that also can be compromised [47]. Commonwealth Crypto protocol offers centralized non custodial approach based on building blocks that are used in the Lightning network protocol [48, 62], namely escrow service with off-chain trading and on-chain settlement. The problems with this protocol are the same as the problems with the Lightning network, for example: 1) limited capacity, 2) an exchange needs to have many coins tied up to escrow channels, 3) fits only blockchains with specific features such as expiry time transactions and requires a new implementation for every blockchain added.

We offer a new protocol for in-wallet exchange of assets between two chains. The key ingredients to make this protocol work are our building blocks: Universality to support multiple coins on our platform. Multiparty computation to reduce the trust in the system such that the user is in full control of his funds at all times. Share Transfer protocol to securely trade assets off chain with immediate on-chain effect. Protocol 6 initalization stage requires the two clients, c_1 and c_2 to send to themselves the exact amount of funds to be traded. This new addresses will switch hands simultaneously or not at all.

Protocol 6: Atomic Swaps

1: Run $(2, 2)$-threshold Key Generation in chain A between client c_1 and Server s. The public key will be c_1's public key Q_1 (client has the full private key) and the resulting address is add_A
2: Run $(2, 2)$-threshold Key Generation in chain B between client c_2 and s. The public key will be c_2's public key Q_2 (client has the full private key) and the resulting address is add_B
3: c_1 sends c_A coins from c_1 account to add_A
4: c_2 sends c_B coins from c_2 account to add_B
5: All parties check the validity of the transactions from steps 3-4 on their respective chains. s will not continue to next step until x blockchain confirmations.
6: c_1 and s run Share Transfer.Verification (protocol 2) to share c_1 share of add_A
7: c_2 and s run Share Transfer.Verification to share c_2 share of add_B
8: If s do not posses the two verified encrypted shares in an agreed upon time frame - abort.
9: s and c_1 run Share Transfer.Sharing phase to share c_2 private share of add_B with c_1
10: s and c_2 run Share Transfer.Sharing to share c_1 private share of add_A with c_2
11: s and c_1 create a transaction in chain B to send the funds in address add_B to a c_1 address in his HD-tree.
12: s and c_2 create a transaction in chain A to send the funds in address add_A to a c_2 address in his HD-tree.

Steps $\{1, 3, 6, 9, 11\}$ can run in parallel to $\{2, 4, 7, 10, 12\}$

Protocol Analysis
privacy: We use Share Transfer with delay. Therefore the server s has his private shares and at no time during the protocol execution he gets access to one of the clients private shares. The swap will look like two regular transactions on two difference chains. In case both amounts are not confidential it will be possible to match between the two amounts.
correctness: Given that all public keys are known, c_1 and c_2 will own add_B in chain B and add_A in chain A respectively.
fairness: The protocol is fair (Atomic) - if s aborts the previous state remains, if s does not abort both clients get their new keys.
security: s enforce no double spend. if c_1 or c_2 deviate from the protocol, the s verification will fail and s will abort. Malicious s can authorize non-verified secret shares and transfer them, but s cannot control the transferred shares (unless colluding with one of the clients) so this attack will only damage s reputation. Honest s is incentivized to complete good trades and in case of s being hacked the broken swap is reversible using cold storage keys once s become honest again. If s colludes with one of the clients protocol 6 is not secure.
There are possible attack vectors by the service provider that need to be addressed when implementing the scheme:

1. The server has the power to do denial of service or unfair swap: if it chooses to it can transfer the shares to neither party or to only one of the parties. This is the main reason why we need to end the protocol with on-chain transactions. In case of dispute claiming that s did an unfair swap the offended party can provide a proof for s misbehavior: Let us take for example c_1. In this case c_1 proves (in zero knowledge or by simply exposing) that they know a private key x_1 such that $Q_1 = x_1 G$ and Q_1 is the public key that generated the address add_A where add_A is the address that c_1 sent amount according to an order in the order book. This claim can be made by the taker or the maker since both are able to provide receipts from the order book. The maker because it will probably create the transaction before there is a taker and the taker will signal the order book that the maker offer is being taken.
   If s was honest and this is a false claim s can easily provide similar proof for a transaction of the matched amount in the other chain. To validate that the receiver of this transaction is indeed the client (c_1 in our example), s will expose the HD chain-code cc (this can be done without exposing secret keys see 3.5) and show that the outgoing transaction address and ingoing transaction address are from the same tree. This can be done by exposing the path connecting the sent payment to the received payment. If c_1 chooses to not finish the protocol and transfer the funds from add_B to his native HD tree then s can provide add_B with the match amount and the public proof that the owner of x_1 can claim this funds. Bottom line - the dispute can be settled legally using hard evidence.

2. s can play a role of fake maker or taker and collect funds to itself. The best defense is the same dispute solving mechanism we just described. It is legitimate for the

server to posses the role of maker/taker as long as the server completes the swap. To further mitigate abuse of the platform the order book or books will be created and maintained by different authorities and not the server.

Note on privacy risk between Clients: Our wallet model is based on hierarchical deterministic (HD) structure with one master key. The suggested atomic swap protocol adds a key from another HD structure. Finalizing the swap is done by transferring the funds to the local HD tree. Exposure of a private key from a different HD tree will not reveal the entire structure because the chain code will remain unknown.

Note on Order Books and public key infrastructure: The first step of the protocol for atomic swaps is to find a match between a taker and a maker. This can be done using one or more order books. Generating and maintaining the order book can be done by anyone theoretically. The Matching step is an important preliminary step from cryptographic perspective as well: We are assuming that Share Transfer is secure given that correct public keys are known to all parties and specifically that c_1 knows c_2 public key and vice versa. If s is in charge of distributing the public keys it is easy for it to cheat and give c_1 a public key that s generated such that s will get the full private key. Here we outline the matching step and required availability of each party:

1. (step 1 of the Atomic Swap protocol:) A maker creates a transaction with the amount she wants to trade to a new address add_m in her local chain. The transaction parameters are: maker secret share x_m, maker public key Q_m.
2. The order book records the specification of the trade: what assets the maker wants to trade, amount, time frame.
3. (step 2 of the Atomic Swap protocol:) A taker is found. The taker creates a transaction to a new address add_t . Parameters are: x_t, Q_t.
4. The taker asks s to provide details on the maker offer. s will provide the public key Q_m
5. The taker will make sure that Q_m corresponds to add_m and will use Q_m for the encryption key in the Share Transfer protocol. The taker can run Share Transfer.Verification (step 3) independently of the maker.
6. The maker will be notified about the taker and will ask the server to provide details about the taker. s will provide the public key Q_t.
7. The maker will verify correctness in the same way the taker did and afterwards can run Share Transfer.Verification (step 4) independently of the taker
8. If both Share Transfer.Verification's ended before the time frame and they are correct, s will complete the two Sharing's.

Notice that s can still cheat by provide a private key that it generated or by collaborating with the order book providers. We thus advice that the order book will be a smart contract or distributed over many providers. In this scenario s can create a private key alone and publish an offer in the order book and it will be a legitimate play. In case s will try to abort before the swap is complete it will be possible to publicly detect as we showed before.
To avoid the case where s cheats using his own private key, running Share Transfer.Verification and then abort but steals the funds from add_A/add_B not immediately we add a rule that abort means that c_1/c_2 will transact the funds from add_A/add_B back to the original address (that's not been part of the Share Transfer).

4.5 0-fee internal payment channels

Going one level up in scale, with enough active users that want to transact with each other the server and clients are forming a star like topology network. In the general case the server as the service provider will need to publish every transaction to the blockchain paying the mining fees and waiting non-deterministic amount of time for confirmation. In this section we explore how we can harness the star topology network to provide users with better performance in terms of fees and confirmation speed. The analysis will be focused on UTXO based cryptocurrencies and Bitcoin specifically. In address based cryptocurrencies the formalism is different but possible using similar arguments.
The first option that comes to mind is to utilize the Share Transfer primitive, protocol 5, to transfer private key ownership from one client to another. Two important facts are 1) the owner of the UTXO is the one that can spend it and 2) the server co-signing is required for spending no matter what. Combining this two facts we get that the receiving client has now full ownership of the UTXO. The benefits here are (a) immediate settlement - no need to wait for on-chain confirmation, (b) zero fee - no need to generate new UTXO, and (c) privacy - the new owner cannot be traced through the public block chain.
This method comes with two big problems:

1. **Address reuse:** After client share transfer from client A to client B both parties are equal with relation to ownership of the public address that can be derived from the full private key. It is true that the server can enforce policy that will keep the receiving party (client B) from spending funds from this address that are not capped by the original transaction amount but if the server is down and recovery is activated (see 4.6.1) both users will own the same private key and will be able to use it to sign transaction that spend the shared UTXO (if you are the sender) or to spend more UTXOs that can be spent by this private key (if you are the receiver). Continuously changing, single-use receiving addresses can help but not mitigate this issue.

2. **UTXO Atomicity and Round numbers:** The transfer method can work only for transfer of full UTXOs. As opposed to Atomic Swap described in 4.4 here

we cannot allow a user to first make a self transfer with the amount they want to send because it will miss the point of lowering the fees. Looking on the Bitcoin blockchain explorer (random example: [61]) we see that almost without exception all transactions are with many digits after the decimal point which means the users still think in fiat and not in native crypto. This will make it nearly impossible to find a perfect match between existing UTXOs and the amount that user wish to send. There are several ideas on how to increase the matching such as rounding amounts in a way that will still benefit the sender (immediate settlement + lower than market fee) and creating a cashier role on the network that will hold a variety of UTXOs of different amounts and will be always online. In case no exact match is found the sender can change a UTXO with few smaller ones including one with matched amount. Occasional users in the network are also incentivized to offer such exchange since it will consolidate their balance which will later result in low transactions fee. Unfortunately all this proposals are not good enough and carry high overhead.

We propose the following two part solution for the two problems:

1. **Private Payment Channel:** Inspired by lightning network specification [62] we can use a variant of payment channels to create a bond between UTXO sender and receiver such that there is only one possible valid state to how the UTXO output is divided. As long as the transactions are within the internal network the sender and receiver are free to spend the UTXO according to the appropriate updated division. The payment channel invalidates every previous transaction and all participants will hold the current unpublished signed transaction. For chain settlement - any one of the parties can close the channel at any time by publishing the updated transaction to the blockchain. Sending outdated signed transaction not through the server will be detected by the server and the sending client will be penalized.
2. **Delegated Share Transfer:** We make a small change to the Share Transfer protocol 5 to create a-symmetry between original secret share owner and the secret share receiver. We now have three levels of ownership to the private key instead of the previous two: (a) full ownership (sender), (b) delegated ownership (receiver) dependent on server policy with no possibility for key recovery, and (c) validation only (server).

In the following subsections we provide more details on the private payment channel and delegated share transfer:

4.5.1 Private Payment Channel

The channel creation in our case will start by sender and server co-signing without publishing a transaction that spends the sender UTXO. This signed transaction will be

sent by the server to the receiver together with the sender encrypted secret share using 5. This completes the channel creation. In Bitcoin since Segwit [63] the hash of the signed transaction will be identical to the future on-chain *txid*. We call it $txid_1$

Observation1: Sharing a private secret share between different clients means that each one of them can independently re-co-sign a transaction to allocate funds according to updated division.

Assuming the receiver (party B) is a new account there are three destination types for party B to pay amount <= $txid_1.amount$: (a) party C controlled by a user in the internal network, (b) back to party A , and (c) party D outside of the internal network - need to go through the blockchain.

Based on Observation1 above it is easy to see that for the original sender party A, the three possible action will work in reciprocal way thus we cover all possibilities of the system and it is enough to analyze the cases with respect to party B.

Cases (a) and (b) are similar. In both - party B will generate new updated transaction. In the first case party B will divide its output to two outputs such that one output goes to party C and will run Delegate Share Transfer protocol to share the delegated ownership rights with party C. In the second case party B will update the original transaction with the new amounts and co-sign the new transaction. To avoid the situation where party B get the old transaction which benefits party B and publish it on the blockchain by itself the transactions in the private payment channel will have a spacial defined structure that will enable the server to penalize party B if an old transaction goes public. Please see [62] BOLT3 for the details on how to build such script for Bitcoin and [64] in the section on channel updates for overview. Basically if party B will publish the old transaction the server will catch it and will have 3 days period to punish party B by making a transaction that will sweep all of party B funds in this channel. This is done by following a simple rule: each party that updates the transaction will put time delay on the output that goes back to itself and a condition on that output that given a secret known to the remote party which is the server in our case plus a secret known to the local party, which is party B, then this output can be spent immediately. The local party secret value of the old transaction will be sent to the server as part of signing the new updated transaction. Eventually, the server share $txid_2$ and the signed transaction to all parties with any ownership over the secret share.

In case (c), Party B will co-sign with the server a new transaction that includes the current channel state in addition to other outputs with addresses outside of the internal network. This signed transaction will be published on the blockchain, effectively closing the channel. This transaction will incur a fee since this is a transaction to the blockchain. One can claim that this will be a bigger than normal transaction and therefore will produce more fees. Few arguments against this claim are: (A) This is an implementation of multi-hop channel meaning that there can be almost infinite number of internal transactions for one on-chain transaction, improving even on lightning network [64] since no fees are required for opening a channel and no fees are due to routing. (B) The transaction outputs can be batched saving up to eighty precent on fees [65]. (C) The service provider can help with fees from certain amount. (D) In Bitcoin, inputs are more dominant for setting the

transaction length and therefore the price. Normally in UTXO based blockchains there are many inputs [66]. In our case the number of inputs will be similar to regular transaction.
Finally we note that in case the original transaction of party A to party B includes more than one input it does not make a difference as long as this inputs are for the same address. If inputs are from more than one address the Delegated Share Transfer will have to be for each input separately. If party B is not a new account it does not make a change since the analysis is separate per UTXO.

4.5.2 Delegated Share Transfer

Here we provide the construction that enables one party to delegate partial ownership over private secret share that corresponds to a specific UTXO input. The difference between partial and full ownership is that in case the server is down and the system enters a recovery mode where all the server shares go public the partial owner will not be able to access the full private key while the full owner will have access to the private key. This force the partial owner to behave according to the service provider policy enforcing access only to the designated channel with restriction on the original amount that was dedicated to the partial owner.
During the computation we require that the server will not be exposed at any moment to the full private key.
The way to do this is by letting the server to do a biased key rotation such that only the server knows the random rotation. See algorithm 1

Algorithm 1: Delegated Share Transfer

1: Server s chooses a random number $z \in \mathbb{Z}_q$
2: In the sharing phase of Share Transfer protocol 5 S send n pairs $\{[z]_i D_4^i, [z]_i D_5^i\}$
3: The receiving client c extracts zx_2 and public key $Q_2' = zx_2G$.
4: s proves knowledge of z such that $Q_2' = zQ_2$
5: c secret share will be $x_2' = zx_2$ and s secret share will be $x_1' = z^{-1}x_1$ (for the case of multiplicative shares) for KeyGen protocol.

We require the use of different random value z every run of the protocol.
The algorithm works well also for delegating a delegated share. This is necessary for example in the case where party B wish to transfer her delegated share to party C to copy its delegated ownership on the private key. In this situation party B will transfer without other option x_2' to the server encrypted under party C public key. The server will pill off z and multiply by z_2 before transferring to party C. The way it is done is depicted in Algorithm 2.

Algorithm 2: Delegate of Delegated Share Transfer

1: Server s chooses a random number $z_2 \in \mathbb{Z}_q$
2: In the verification phase of Share Transfer protocol 5 S compute $T = D_4 - zQ_2$
3: In the sharing phase of Share Transfer protocol 5 S send n pairs $\{[z_2]_i[z^{-1}]_i D_4^i, [z_2]_i[z^{-1}]_i D_5^i\}$
4: The receiving client c extracts $z_2 x_2$ and public key $Q_2'' = z_2 x_2 G$.
5: s proves knowledge of z_2 such that $Q_2'' = z_2 Q_2$
6: c secret share will be $x_2' = z_2 x_2$ and s secret share will be $x_1' = z_2^{-1} x_1$ (for the case of multiplicative shares) for KeyGen protocol.

4.6 Private Key Extraction

4.6.1 Server Exceptions

There is an inherent flaw with the architecture of $(2, 2)$-threshold signatures. Namely, it requires the availability and honesty of both parties. In a normal operation mode there are incentives for each of the parties which puts us in Nash equilibrium. The client is the owner of the money so he has absolutely no reason to act against the rules because it can only lose. The server is the service provider and will gain more users if he can gain trust and for that he needs to be honest. There are three exceptions that can cause a different server behaviour: (1) the server is hacked and becomes malicious (2) the server is "dead", meaning a long denial of service (3) the company employees that know the passwords to the server are all gone, for example died in a plane crash. In all three exceptions we assume the worst case scenario that the server is now fully controlled by irrational player. We offer two general solutions to server exceptions. Our suggestion is to look at the solutions as two layers of protection. Implementing both will assure that user funds will always have high availability and recovery.
The schemes below are for recovery of one secret. This is because We assume the use of HD-wallet therefore all private keys could be derived from a single master key which is the key we wish to recover. In practice the schemes can be generalized to recover multiple keys.

4.6.2 Recovery Party

The concept of this solution is to relax the $(2, 2)$-threshold to some extent to allow recovery. We have two tools in the toolbox that can help: (a) $(2, 3)$-threshold key generation (b) key rotation.
The first possible scheme will use $(2, 3)$-threshold in the following way:
1) Run $(2, 3)$-threshold key-generation between the server, the client main device and the client recovery device.
2) In case of server exception the user can recover his key using his recovery device.
We use $(2, 3)$-threshold instead of sending the server private key share to the recovery

device because in this way we get that all 3 shares are locally generated at 3 different places and no share goes ”over the air” between devices at creation time, making it less prone to attacks. This setting can also use to sign a transaction locally between the two client devices and send it to the server to publish to the blockchain. The local-signing ability can be important because eventually, assuming server exception, the link between the client and the full nodes is down and the best way to do export is by a transaction that moves all the funds to a different account/wallet. Reconstructing the private key alone will demand an extra effort from the user to export it.
The second possible scheme will use key rotation in the following way:
1) After running Key Generation protocol the server and client will immediately run Key Rotation to get a second pair of secret shares that correspond to the same master key.
2) The server will send his secret recovery share to a second device owned by the client.
3) In case of server exception the user will use his two devices to combine the recovery shares for the full private key.
Notice that with this scheme the second user device can be memory-only device wheres in the first scheme the second device should have some computing power. Another differentiation between the schemes is that in the second scheme we use a communication channel to transfer a recovery secret share. The extra risk can be mitigated because now the sender (the server) can encrypt the recovery share, for example with a user public key. In case that an attacker gets access to both user devices he will need a user password to get both shares of the private key.

There is yet another possibility for recovery, parallel to both scheme, such that instead of using a second device we use a Key Recovery Service (KRS). The KRS should be separated to the maximal extent from the original server, i.e. on a different cloud with different admins and passwords, even geographically separated. The advantages here are stricter access control and that the two server can also be another option for recovering user key.

4.6.3 Social Recovery of Server Secret Share

There are some cases in which the service provider (the ’server’) cannot be trusted anymore. For this cases we want to create a way for a collective of users to vote to break the dependency in the server. By doing so the users wallets will become stand-alone with private key stored only on the device. Users will now be able to choose a different service provider or to use the same. This voting mechanism should be available at all times and not dependent on the server. We require that the activation of the procedure would need a large quorum of users cooperating, assuming there are no more than threshold malicious users and more than threshold honest users. Another requirement for the protocol to work is that a user will get a vote only after running Key Generation. This makes sense because only once there is a private key then the user puts a stake in the system and risk that received funds will get locked because of server exceptions. Our protocol is based on Schoenmakers Publicly Varifiable Secret Sharing (PVSS) protocol [71] and Share Transfer protocol 5.

In high level: after Key Generation the user will get from the server :
(a) The server share x_2 encrypted with encryption key y. The same y will be used for all users.
(b) a secret share $p(i)$ where the secret is the encryption key y used in (a).
The user will save this data locally and if there is a point in time where the user will want to vote against the server she will publish her secret share. Once above threshold shares are collected the user will get y and will decrypt the server share locally, which will end in a fully owned, attached to device private key.
We require the following:

1. In secret share distribution:
   (a) Proof of correct encryption: a proof from the Server that the ciphertext sent to the user is indeed an encryption of the Server private key share.
   (b) Proof of correct secret share: a proof from the Server that the secret share the user received is a share of the encryption key.
2. Upon secret share reconstruction:
   (a) Every user will have the ability to prove that her secret share is correct and can be used to recover the encryption key.

The first demand can be accomplished by a variant of Share Transfer 5. The last demand can be accomplished using PVSS. Demand 1(b) will require a combination of properties from PVSS and Share Transfer.
We notice that usually Secret Sharing protocols are described formally such that all participants are known in advance for shares distribution. This is not a security requirement and it can be shown that as long as the Dealer (Server in our case) decided on a scheme (a threshold t) and published public parameters the number of shares n and the generation of new shares can be done dynamically as long as $n > t$. Participation can be done one user at a time communicating with the Dealer starting with user initialization. Our protocol is adjusted accordingly: The server generates a secret share for a user once this user starts using the system.

Analysis During normal procedure the user can request to terminate the account at any moment. The server will send him x_1 or simply co-sign transactions to export the account funds to different addresses solely owned by the user. If for any reason the server fails to comply, the user should check the official service provider channels, i.e. website/facebook/twitter to get the master decryption key $S_i^j|_{j=1}^n$, skip the PVSS reconstruction step and go directly to brute-force x_1 (last two step of Reconstruction). With it all user can decrypt locally the ciphertext of x_1 and get ownership of the full root private key. This will answer the scenario of server going through a long denial of service attack or takedown. If an attacker wants to block this method, he needs to attack the server and the social media account of the service provider meaning no single point of failure (SPOF).
The extreme case where the service provider itself becomes the attacker or if for any other reason, the service provider is not trusted anymore to provide services each user will

Protocol 7: Social Recovery

Initalization (Preprocessing):

1: The Server picks n random polynomials $p(x)^j|_{j=1}^n$ of degree $t-1$ with coefficients α_i^j, for $0 \le i < t$ for the j'th polynomial.
2: The encryption key is set to $y = \{\alpha_0^j\}_{j=0}^{n-1}$
3: The Server uses hash of the generator G of the elliptic curve group and map the hash result to another generator group element H (can be done for example by taking sha256 output to be the x coordinate)
4: For every polynomial the Server publishes commitments to the coefficients: $C_i^j = \alpha_i^j \cdot H$ for $0 \le i < t$. Let $X^j(k) = \sum_{i=0}^{t-1} k^i \cdot C_i^j$

Distribution :

1: Server and Client runs Key Generation 1. Client gets x_2 and Server gets x_1, both know Q_1, Q_2, Q where $Q_n = x_n \cdot G$ and $Q = x_n \cdot Q_{3-n}$
2: The Server publishes the Client encrypted shares $Y_i^j = p(i)^j \cdot Q_2$ for $0 \le j < n$
3: Proof that the secret shares are consistent: as part of PVSS the Server and Client will run EC-CP93 PoK protocol (see 5 for details) to prove for all j that $p(i)^j$ satisfy $X^j(i) = p(i)^j H,\ Y_i^j = p(i)^j Q_2$ - EC-DLOQ$(H, X^j(i), Q_2, Y_i^j)$
4: The Server and the Client run the following protocol (a variant of $\mathrm{ShareTransfer.Verification}$ 5 but in opposite roles):
    1: The Server computes n tuples of 5 group elements. $\forall\ 0 < i \le n$:

$$\{D_1^j, D_2^j, D_3^j, D_4^j, D_5^j\} = \\ = \{t_j[x_1]_j G,\ t_j s_j Q_2,\ t_j C_0^j,\ [x_1]_j G + s_j Q_2,\ t_j H\}$$

    t_j is a random field elements, $[x_1]_j$ is the j'th segment of x_1 shifted by $sr(j) = 256\frac{j-1}{n}$ bits to the right (there are $bl = 256/n$ bits in each segment). $C_0^j = \alpha_0^j H = s_0^j H$ is the shared publicly commitment to the secret s_0^j. The actual shared secret group element that the i'th PVSS scheme will reconstruct given threshold secret shares is $s_j G$.
    2: the Server sends the n tuples to the Client
    3: Zero-Knowledge proof of correct encryption:
        1: The Client computes $D_4 = \sum_j 2^{sr(j)} D_4^j$ and then $T = D_4 - Q_1 (= sQ_2)$ where $s = \sum_j 2^{sr(j)} s_j$
        2: The Server generate a proof using EC-CP93 PoK that $T = \sum_j 2^{sr(j)} s_j Q_2$ and $C_0 = \sum_j 2^{sr(j)} C_0^j = \sum_j 2^{sr(j)} s_0^j H$ using the same scalar $\sum_j 2^{sr(j)} s_j$: EC-DLEQ(Q_2, T, H, C_0)
    3: Proof of extractability:
        1: The Server generates n range proofs for, proving that all D_4^j are Pedersen Commitments to values $< 2^{bl}$. Range proofs can be done efficiently using Bulletproofs [13]
        2: The Server generates the following $3n$ EC-CP93 PoK : for $0 < j \le n$: EC-DLEQ(C_0^j, D_3^i, H, D_5^t), EC-DLEQ$(H, D_5^t, D_4^j, D_1^j + D_2^j)$, EC-DLEQ$(Q_2, D_2^j, H, D_3^j)$
        3: The Server and Client runs a sigma protocol where the Server proves knowledge of n such that $D_1^j = vG$ for $0 < j \le n$

Protocol 7: Social Recovery

Reconstruction:

1: Using its private key x_2 the Client finds the share $S_i^j = p(i)^j G$ from Y_i^k by computing $S_i^k = {Y_i^j}^{1/x_2}$.
2: The Client publishes the S_i^j's plus a proof that the value S_i^j is a correct decryption of Y_i^j (non interactive version of ZKPoK of dlog, see [71]).
3: Suppose at least t different clients with valid shares (valid proof of same exponent for all Y_i^j and X_i^j, for $0 < i \leq n$ valid proof of correct decryption). The secrets $\alpha_0^j G$ is obtained by Lagrange interpolation (see [71]).
4: The Client computes $[x_1]_j = D_4^j - x_2 \alpha_0^j G$ and brute force to get $[x_1]_j$
5: The Client combines all $[x_1]_j$ to get x_1 and calculates the full private key $x_1 \cdot x_2$

publish his secret shares $S_i^j|_{j=1}^n$ which by the properties of PVSS can be publicly verified to belong to a specific secrets. Each user will run Reconstruction to get his own x_1. A Client will be verified only once the set of secret shares given to the user was proved to match the public parameters (commitments). This is straight forward from PVSS we just need to make sure that the public parameters are indeed public and immutable. Using a blockchain smart contract or a blockchain transaction attesting to hash of IPFS [**?**] using Sign-to-Contract [52, 53] link are a good options for this.

This leaves us with the problem of making sure that the secrets reconstruct by the PVSS are enough to extract the full private key. We have a public commitment $C_0^j = s_j H$ in the j'th PVSS scheme and the corresponding reconstructed secret is $s_j G$. We want to show in zero knowledge that the ciphertext a Client gets after Key Generation for $[x_1]_j$ can be decrypted using $s_j G$.

zero-knowledge: we are using a tuple of 5 group elements for every s_j. Four group elements are contain t_j which is a random field element, unique for the tuple. D_4^j is $[x_1]_j$ masked with $x_2(s_j G)$, where $s_j G$ can be obtained in two ways: (1) extract s_j from the public commitment C_0^j (2) extract the secret protected by the PVSS. Therefore all 5 group elements are random looking group elements and zero knowledge can be reduced to hardness of DDH in elliptic curves. Since t_j, s_j are different for every tuple stands independently and the claim is true for all tuples.

Correctness and Soundness: Lets parameterize the j'th tuple such that from the point of view of the server we have 5 group elements: (A, B, C, D, E). F_1^i will be denoted F. K_1^i will be denoted K. Using the EC-DLEQ relations we get the following: $(1) C = xC_0^j, (2) E = xH, (3) xD = A + B, (4) B = uQ_2, (5) C = uH, (6) A = vG$ where x, u, v are field elements unknown to the client. Plugging (4) and (6) in (3) we get $(7) xD = vG + uQ_2 \Rightarrow D = x^{-1}vG + x^{-1}uQ_2$. Comparing (5) and (1) we get $xC_0^j = uH$ and since $C_0^j = s_j H$ we get $(8) u = xs_j$. Plugging (8) in (7) we get $(9) D = x^{-1}vG + s_j Q_2$. Since the secret from the j'th PVSS is $S_j = s_j Q_2$. Subtracting S_j from (9) we get $D = x^{-1}vG$. Combine with range proof makes $x^{-1}v$ a small field element that can be brute-force (true for all j's). The zk proof of correct encryption proves

that sum D over all j's equals $Q_1 + sQ_2$ where sQ_2 is the sum of all secrets from the n PVSS instances. The server doesn't know the relation between Q_2 and G if the server adds Since we showed that for every solved PVSS the client can cancel precisely the s_iQ_2 element and be left with a brute-forceable segment of the server private key. Combining all the extracted $[x_1]_j$ segments times G the same way we sum over the D's followed by subtraction of the same form of sum over the PVSS secrets result in Q_1 concluding the proof that a client with secrets s_iG for $0 < i \leq n$ will be able to extract the secret share x_1. A sending server that will try to cheat will fail with high probability in proving one of the EC-DLEQ or in the proof of correct encryption (reduced to hardness of EC-dlog).
Choosing parameters: We remark that this scheme requires a tailored parameters to work best. At every point in the lifetime of the service we would like to allow a threshold of users proportional to the total users (say 50%). An approximation for this can be achieved by creating multiple Social Recovery Scheme with different thresholds and give each user secret shares according to the current number of registered users at the time the user registers. For example if we would like that 50 out of the first 100 users will be able to vote and that 500 our of the first 1000 - the server will generate 2 schemes, one with threshold of 50 and one with threshold 500. The first 100 users will get secret shares of the schemes while the rest will get secret shares only of the larger scheme. IPFS [**?**] can be used as a distributed way to save the data. Each user will get IPFS file and will keep the hash link.

4.6.4 Client Share Recovery

Up until now we dealt with server share recovery in case the server becomes corrupted, doesn't matter if on purpose or by accident. We will now briefly explain how to recover the private key for the case where the client secret share is gone. This can happen for example in case the user lost his device or that the device was stolen. For such cases the server can hold an encrypted version of the client master key mk, where the decryption key can be derived from a user known password. Upon authentication the server will provide the encrypted share to the client. The client will then decrypt the secret share using the user password and complete the recovery.
The more interesting case is when the user don't have a password or that he forgot it. In this case recovery can be done using the following guidelines:
The user will divide his private share to two random parts using SSSS. One part can be stored in one of the user's other cloud providers. The second part will be sent to the server. In recovery the user will load the first "secret share of secret share" (ss-ss) from his cloud provider and upon authentication he will receive from the server the second ss-ss. This division of shares guarantees that at no point one party holds the full key. We assume that the server is honest, see 5.1 for more details.

4.7 Shared Wallet

A shared wallet consists of group of members who can approve transactions according to some logic. Some examples are: (a) A family account in which any of the parents can

sign, (b) partners account in which 2 out of 3 can sign,(3) firm expense account: one of employees, one of the accountants and the CFO.
The group creation is done by a single user - the group admin. Before running Key Generation the group admin will define group members and signing logic. After Key Generation the group admin use Share Transfer (protocol 2) to share his private key share with the group. The signing can be initiated by one of the group members. The server will run Signing protocol with this member but will not publish immediately the transaction to the blockchain. Instead the server will enforce the signing logic that was set up at Key Generation time. Once the Signing logic is satisfied the server will publish the transaction.
Key Rotation is done according to protocol 3 with one of the members that are online and this member will Share Transfer the rotated private key share to the rest of the group.
In a more advance form of this feature a group governance mechanism can be plugged in. For example above threshold of votes the group can remove a user or change the signing logic. This option will be greatly assisted by instant massaging capability. Generally, there can be many scenarios and many complicated signing logic. The user experience of a shared wallet can be very much like groups in popular IM apps (e.g. Whatsapp). The description of such is not specific to cryptocurrency and can probably leverage existing code like [51].
Shared Wallet will need a stateful server and recovery should be done using $(2, 3)$-threshold digital signature where the third party should be trusted by most of the group members.

4.8 Cold Storage

The ability to keep the private keys in an isolated environment that prevents online access is how we define cold storage. Cold storage can be implemented in many forms. A user can use hardware or paper wallets and in centrelized wallets cold storage can be offered as-a-service. Cold storage service usually requires physical security measures, such as vault and bodyguards, and strict access policy, see for example XAPO [38] and Coinbase [39]. In our architecture it is extremely easy to implement cold storage. The private key is never actually created and even when doing actions that require the private key like signing transactions, at no time the private key is assembled at one place. Therefore by just moving one of the necessary shares to be offline the entire private key is considered offline. We can move user funds to cold storage as a service by just deleting the server private share from the cloud and not retrieve it from backup. In this case an attacker will now have to get the online user share and the offline (backup) share of the server instead of getting two online shares simultaneously. Going back online requires access to the offline backup, something that can be done only manually. We can define a delay for going back to hot wallet, for example one day, one week etc.

5 Discussion

5.1 The Server Model

During this work we made several assumptions on the server behaviour without a formal definition of adversary model. Indeed there are some times when we assumed a measure of trust between client and server or even regarded the server as a trusted party, for example in Atomic Swaps. In other times we assumed the server can be malicious, for example in key generation protocol. We postpone the formal definition of the server model for a future work but we do want to explain the logic beyond our assumptions. We generally treat the server as malicious for all key management concerns, meaning that at any point the server can get hacked and be in the control of the attacker. The server owners can also become malicious, even without economical incentive. We even consider the possibility that the server operators will die as mentioned in the section on recovery. We differentiate between the key management layer including all the building blocks composing this layer such as key refreshing, and between the wallet-as-a-service layer where the server is becoming a service provider. In the service layer we still assume that the server is malicious adversary except for cases where the system has already one malicious participant. We do not protect against more than one malicious party at one time in the system. For example, in Atomic Swaps if one of the clients acts maliciously and do not follow the protocol then the server is considered to be honest. A counter example is when constructing the right message, we assume the client is honest and will follow protocol because he has strong incentive to generate the transaction properly. In that case we assume the server is malicious.

5.2 Authentication

The service needs to support multiple authentication methods so when one is not available (e.g. device based authentication and device is lost), the server can use alternative authentication methods. The user key share is stored in encrypted versions on the server, where each encrypted version corresponds to an authentication method. The versions are stored on the server, as the server is better equipped for protecting their availability through backup. Each of the encrypted versions is encrypted with a corresponding client-side generated public key, which its private key is generated on the client side and never sent to the server. On authentication, the server sends the encrypted version corresponding to the server desired authentication scenario (normally, the main authentication method, unless specified otherwise) to the client and the client decrypts it locally using its private key. Whenever the client key share is updated (rotated, changed or added), the client needs to encrypt it in all the versions using their respective public key and send it to the Server. To store and communicate secrets, we suggest the use of SSL/TLS protocol and Public Key Infrastructure (PKI): Server Certificates, Client Certificates, Certificate Signing Request (CSR). At on-boarding the client generates CSR and sends to the server. The server creates a certificate from the CSR and sends to client. The client use a certificate to authenticate.

5.3 More Services

We mentioned in this paper some essential services that we believe are must have as a minimal setting. There are a lot more services that we do not mentioned but they can be absolutely supported for the relevant use case. In short, every service possible in Multisig wallet is possible for threshold signature wallet with the ability to scale to almost all cryptocurrencies. To give example of such services we can do; $(2, 2)$-threshold saving account, multiple factor authentication wallet, $(2, 3)$-threshold escrow service and more Multisig enabled services that can be found in [40]. Mixing and Atomic Swaps are unique to MPC based signatures because of scalability, flexibility and privacy issues in Multisigs.
Generally speaking, threshold signatures are native to the blockchain layer which means that every future application that will require private key and digital signature will be supported. Good examples are Identity and data management.

References

[1] R. Gennaro, S. Goldfeder, A. Narayanan. Threshold-optimal DSA/ECDSA signatures and an application to Bitcoin wallet security. ACNS 2016

[2] P. MacKenzie and M. Reiter. Two-party Generation of DSA Signatures. Int. J. Inf. Secur. 2004.

[3] Y. Lindell. Fast Secure Two-Party ECDSA Signing. IACR Cryptology ePrint Crypto 2017.

[4] D. Boneh, R. Gennaro, S. Goldfeder. Using Level-1 Homomorphic Encryption To Improve Threshold DSA Signatures For Bitcoin Wallet Security. Latincrypt 2017.

[5] O. Ohayon, The sad state of crypto custody. https://techcrunch.com/2018/02/01/the-sad-state-of-crypto-custody/.

[6] P. Paillier, D. Pointcheval (1999). Efficient Public-Key Cryptosystems Provably Secure Against Active Adversaries. ASIACRYPT. Springer. pages. 165-179, 1999

[7] J. Doerner, Y. Kondo, E. Lee, A. Shelat. Secure Multi-party Threshold ECDSA from ECDSA Assumptions, to be appear in Oakland SP2018

[8] G. Maxwell, A. Poelstra, Y. Seurin, P. Wuille. Simple Schnorr Multi-Signatures with Applications to Bitcoin. Cryptology ePrint Archive: Report 2018/068 2018. https://eprint.iacr.org/2018/068

[9] F.Boudot. Efficient Proofs that a Committed Number Lies in an Interval. Lecture Notes in Computer Science, Vol. 1807, pages 431-444, EUROCRYPT 2000

[10] Y. Lindell, A. Nof. A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority. pages 259-276. CCS 2017

[11] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public-key and signature schemes. In Proceedings of the Fourth Annual Conference on Computer Communications Security, pages 100?110. ACM, 1997

[12] T. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. Lecture Notes in Computer Science, Vol. 576, pages 129-140, CRYPTO 1991

[13] B. Bunz, B. Bootle, D. Boneh, A. Poelstra, P. Wuille, G. Maxwell, Bulletproofs: Short proofs for Confidential Transactions and more.In Proceedings of the IEEE Symposium on Security and Privacy, (2018).

[14] D. Chaum, T. P. Pedersen. Transferred cash grows in size. In Advances in Cryptology, EUROCRYPT , volume 658 of Lecture Notes in Computer Science, pages 390 - 407, (1993).

[15] https://en.bitcoin.it/wiki/Secp256k1

[16] https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

[17] https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki

[18] https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki

[19] S. Goldfeder, R. Gennaro, H. Kalodner, J. Bon- neau, J.A.Kroll, E.W. Felten, A. Narayanan. Securing Bitcoin wallets via a new DSA/ECDSA threshold signature scheme. Manuscript, 2015. http://stevengoldfeder.com/papers/threshold_sigs.pdf

[20] S. Goldfeder, J. Bonneau, R. Gennaro, A. Narayanan. Escrow protocols for cryptocurrencies: How to buy physical goods using Bitcoin. In Financial Cryptography and Data Security. 2017

[21] I. Bentov, Y. Ji, F. Zhang, Y. Li, X. Zhao, L. Breidenbach, P. Daian, A. Juels. Tesseract: Real-time cryptocurrency exchange using trusted hardware. https://eprint.iacr.org/2017/1153, 2017

[22] https://ipfs.io/

[23] https://www.blocknotary.com/timestamp

[24] https://opentimestamps.org/

[25] Y. Lindell. "How to simulate it " a tutorial on the simulation proof technique. Tutorials on the Foundations of Cryptography. Springer, Cham, 2017.

[26] https://coinmarketcap.com/

[27] https://www.bitgo.com/

[28] CryptoNote v2.0. https://cryptonote.org/whitepaper.pdf

[29] S. Noether. Ring Confidential Transactions. https://eprint.iacr.org/2015/1098.pdf

[30] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, B.Y. Yang. High-speed high-security signatures. Journal of Cryptographic Engineering 2. pages 77-89, 2012

[31] https://etherscan.io/apis

[32] https://github.com/bitcoinjs/bitcoinjs-lib

[33] https://github.com/ethereumjs/ethereumjs-util

[34] https://bitcoin.org/en/developer-examples

[35] https://github.com/monero-project/monero

[36] R.L.Brown. SEC 1: elliptic curve cryptography. Standards for Efficient Cryptography Group (SECG). http://www.secg.org/sec1-v2.pdf. 2009

[37] G. Maxwell. Confidential Transactions. https://people.xiph.org/ greg/confidential_values.txt, 2015

[38] https://xapo.com

[39] https://coinbase.com

[40] https://en.bitcoin.it/wiki/Multisignature

[41] S. Goldwasser, S. Micali, and R.L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. SIAM J. Computing, pages 281-308, 1988

[42] https://www.binance.com

[43] https://shapeshift.io/

[44] https://0xproject.com/

[45] I. Bentov, L. Breidenbach, P. Daian, A. Juels, Y. Li, X. Zhao. The cost ofdecentralization in 0x and EtherDelta

[46] https://www.commonwealthcrypto.com/

[47] https://github.com/lsds/spectre-attack-sgx

[48] J. Poon, T. Dryja .The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf

[49] https://en.wikipedia.org/wiki/Mt._Gox

[50] https://www.ledgerwallet.com/

[51] https://github.com/signalapp/Signal-iOS

[52] https://blog.eternitywall.com/2018/04/13/sign-to-contract/

[53] I. Gerhardt, T. Hanke Homomorphic payment addresses and the pay-to-contract protocol. arXiv preprint arXiv:1212.3257 (2012).

[54] https://github.com/lightningnetwork/lnd/tree/master/aezeed

[55] B. Barak, Y. Lindell, T. Rabin. Protocol initialization for the framework of universal composability. Cryptology ePrint Archive, Report 2004/006, 2004.

[56] H. Krawczyk, Cryptographic Extraction and Key Derivation: The HKDF Scheme, Proc. 30th Ann. Conf. Advances in Cryptology (CRYPTO 10), (2010).

[57] https://brd.com/

[58] https://jaxx.io/

[59] https://www.coindesk.com/hacker-returns-225-btc-taken-blockchain-wallets

[60] https://bitcoin.stackexchange.com/questions/64446/wallet-gone-and-lost-recovery-phrase-how-to-get-back-my-bitcoins

[61] https://blockchain.info/block/00000000000000000002680100499d956f2c45894b82ff109fed11ca6ce9c0ced

[62] https://github.com/lightningnetwork/lightning-rfc

[63] https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki

[64] https://dev.lightning.community/overview/

[65] https://bitcointechtalk.com/saving-up-to-80-on-bitcoin-transaction-fees-by-batching-payments-4147ab7009fb

[66] C. Perez-Sola, S. Delgado-Segura, G. Navarro-Arribas, J. Herrera-Joancomarti. Another coin bites the dust: An analysis of dust in UTXO based cryptocurrencies https://eprint.iacr.org/2018/513.pdf, (2018).

[67] G. Malavolta, P. Moreno-Sanchez, C. Schneidewind, A. Kate, M. Maffei. Multi-Hop Locks for Secure, Privacy-Preserving and Interoperable Payment-Channel Networks. https://eprint.iacr.org/2018/472 (2018)

[68] C. Hazay, G. L. Mikkelsen, T. Rabin, T. Toft. Efficient rsa key generation and threshold paillier in the two-party setting. In CT-RSA, pages 313 - 331, (2012).

[69] I. Damgard and M. Jurik. A generalisation, a simplification and some applications of paillier?s probabilistic public-key system. In Public Key Cryptography, pages 119 - 136, (2001).

[70] https://github.com/snipsco/rust-paillier

[71] B. Schoenmakers. A Simple Publicly Verifiable Secret Sharing Scheme and its Application to Electronic Voting. In CRYPTO, pages 148-164, (1999)

[72] https://github.com/KZen-networks/multi-party-ecdsa