Bitcoin Wallet Powered by Two-Party ECDSA - Extended Abstract

Team KZen

https://github.com/KZen-networks

Abstract. We propose a Bitcoin wallet that utilizes two party ECDSA (2P-ECDSA [3]) Key Generation and Signing. Our architecture relies on a simple client-server communication model. The proposal includes: two-party hierarchical deterministic key derivation (2P-HD) a-la BIP32 [20], secret share rotation and verifiable backup. We discuss the opportunities and challenges of using a multi-party wallet.

1 Background

For end-users, cryptocurrencies and blockchain-based assets are hard to store and manage. One of the reasons is the tradeoff between security and usability. Storing private keys safely requires dedicated hardware or extreme security measures which make using the coins on a daily basis difficult. Threshold cryptography provides ways to distribute the private keys and to produce signatures in a distributed manner. This can potentially benefit security but at the same time introduces new challenges such as availability, ownership and recovery. Bitcoin employs ECDSA as the signing scheme. There is an active line of research for practical and efficient multi-party ECDSA schemes [1–6].

2 A General ThreshSig Wallet

Informally, a threshold wallet is described by the following algorithms:

- **Distributed Key Generation (DKG):** A multi-party algorithm to generate a public key (a blockchain address) and secret shares of the private key.
- **Distributed Signing:** a multi-party algorithm that takes the DKG secret shares and additional public data to generate a valid signature on a given message for the DKG public key (a blockchain transaction).
- **Deterministic Child Address Derivation:** a multi-party algorithm to deterministically derive new public key and secret shares from public key and secret shares.
- **Rotation:** a multi-party protocol to update the distributed secret shares.
- **Secret Shares Recovery:** a protocol to recover a secret share.

We stress that in a threshold wallet, at no point in time a single party holds the complete private key. All multi-party protocols must be secure against malicious adversaries.
A wallet also requires read and write access to a full node. The messages (transactions) being signed are known to all parties running a threshold wallet. It is recommended that as many parties as possible will have independent access to the blockchain network. A party without a trusted connection to the blockchain will have to run an additional protocol to authenticate the message to be signed.

2.1 Comparison to Multisig wallet

Multisignature (Multisig) is the current way to add threshold security to private key management. We highlight some the properties and advantages of using threshold signatures (ThreshSig) compared to Multisig:

Enhanced security. In both MultiSig and ThreshSig an adversary must break into multiple machines, But, thanks to key rotation, an attacker must simultaneously attack the different parties to get access to the complete private key. In ThreshSig wallet this is done without changing the public key and without going through the blockchain, but only in a few rounds of communication between the participating parties. On the other hand, in Multisig wallet the secret keys are non-proactive. Updating the keys can be done only by transmitting a blockchain transaction, transferring the assets to the new public keys. This requires paying the necessary transaction fees and waiting the necessary confirmation time.

Reduced fees. A ThreshSig transaction is indistinguishable from a regular transaction, while Multisig transactions have a more unique structure that can attest to the Multisig policy for other observers of the network. For example, in Bitcoin terms, a ThreshSig transaction is indistinguishable from a transaction sent to or from a single Bitcoin address, whereas a Multisig transaction would need to redeem a special type of output, namely a *pay-to-script-hash* output specifying the exact public keys of all participants.
Therefore, The cost of a ThreshSig transaction is the same as of a regular transaction, since they have the same size. A Multisig transaction on the other hand, has a varied size and therefore its cost is proportional to the number of participating parties.

Flexibility. ThreshSig can be easily adopted for different types of blockchains as it depends only on cryptography. On the other hand, Multisig relies on the application layer of a blockchain, i.e. Script in Bitcoin, smart contracts in Ethereum etc., and therefore more effort is required to add support for new chains.

3 Design Considerations for Choosing 2P-ECDSA

We chose to work in the two-party setting to optimize on simplicity and security. To explain this points we first define two roles in our system: Owners and

Providers. In the blockchain use case we assume single owner to a private key. The Owner in our system is the same owner and we define it as the signer on the transaction that moved assets to the system. All other parties taking part in computations are Providers. By using a threshold signing scheme and assuming availability of Providers we get high security standard as there is no single point of failure and high throughput of signatures which answers the usability issue. However, to ensure that the Owner keeps control over its assets, we introduce the following two requirements:

1. No transaction can be made without the Owner participation.
2. At any point in time the Owner can recover the full private key.

In a wallet we have a single Owner. The question we want to ask is how many Providers do we need. We assume a rational Provider, driven to maximize its economic gain. The first requirement eliminates the possibility of Providers colluding without the Owner. As it appears to be there are very specific access structures that can support this claim. The simplest of those is the two party setting where the Owner is one party and the Provider is the second party. This case fits well a classical client-server communication model with the Provider running a server. Providers are encourage to support multiple Owners which by charging very small fees for the extra security layer can make good economic model (but needs to be further defined). On the other hand, more complex access structure will involve running a full blown p2p network with no substantial added value to security or incentivation over the two party case. Concretely, having hybrid access structure that includes Owner plus a t out of n Providers can be seen as a an extension on top of the two party setting where now the sub structure of $\{t, n\}$-threshold between Providers do not have to maintain the first requirement.

4 Implementation Details for 2P-Thresh-Wallet

Let $\mathbb{G}$ be an elliptic curve group of prime order q and a base point (generator) G. All commitments in the protocol are hash based. We use SHA256. {2,2} Key Generation and {2,2} Signing is based on Lindell's CRYPTO17 paper [3].

4.1 {2,2} Key Generation

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

{2,2} Key Gen algorithm

1. Init: The server chooses a random x_1 and computes $Q_1 = x_1 \cdot G$. The client chooses a random x_2 and computes $Q_2 = x_2 \cdot G$.

2. The server and the client run (a variant of) the ECDH key exchange (see [3] for details). $Q = x_1 x_2 G$ is the resulting joint public key.
3. The server generates Paillier [15] key-pair and computes $c_{key} = Enc_{pk}(x_1)$ where pk is the Paillier public key. c_{key} and pk are sent to the client.
4. The server sends to the client a non-interactive zero-knowledge proof that the Pailler key is well-formed (we use the proof given in [8] with the parameters from [6]).
5. The 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.
6. The 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.

4.2 {2,2} Signing

We first describe ECDSA signature scheme:
Assume that Alice has a private key x. The corresponding 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 an ephemeral key $k \in [1, q]$.
2. Compute $R = k \cdot G$.
3. 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)

Here is the equivalent two party signing algorithm that results in the same signature.

{2,2} Signing algorithm

1. Server and client repeat step 1 of the Key Generation protocol but for ephemeral key-pairs. The protocol outputs 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 $c_1 = Enc_{pk}(k_2^{-1} \cdot H(m) + \rho q)$ 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 if (r, s) validates as a signature on H(m) outputs (r, s) as the ECDSA signature.

4.3 2P-HD

The purpose of the 2P-HD protocol is to allow derivation of child keys in the two party setting. The parties will run {2,2} Key Generation once to generate one master key each. The master key can derive all other keys and it is the one to be recovered if needed. This protocol achieves the security guarantees in the spirit of BIP32 [20] but is incompatible to the standard. Instead it was designed to support natively the two party case with much more efficiency. We do not try to be compatible to BIP32 [20]. Compliance in this case would have helped when a user wishes to export her wallet to another wallet system or to import from another wallet to the threshold wallet system. Importing from BIP32 supported wallet to a threshold wallet cannot be done without a trusted dealer to secret share the private key thus creating single point of failure. Exporting from threshold wallet to a BIP32/BIP39 wallet is also not possible because there is no way to get a BIP39 mnemonic for the threshold wallet. Therefore we see only small benefits to have a multiparty version of BIP32 instead of having a highly efficient new protocol.

2P-HD algorithm

1. Init: The server and client inputs to the protocol is the outputs from {2,2} Key Generation. Specifically, public keys Q_1, Q_2, Q. After a key derivation the protocol outputs a new set of public keys Q'_1, Q'_2 and private keys x'_1, x'_2 respectively
2. The server and client run ECDH to get a shared secret which we call chain code cc. The chain code is used for all key derivations.
3. The protocol takes as input a vector of indices and repeats the following:
   (a) For a given i, both parties compute :
      i. $f = \text{HMAC512}(key = cc, data = Q||i)$
      ii. $f = f_l||f_r$ where $|f_l| = |f_r| = 256$
      iii. $Q'_2 = f_l \cdot Q_2$
      iv. $Q' = f'_l \cdot Q$
      v. $cc' = f_r \cdot cc$
      vi. Update $\{Q_1, Q_2, Q, cc\} = \{Q_1, Q'_2, Q', cc'\}$
   (b) The client updates his secret share to $x'_2 = f_l \cdot x_2$. The server secret share remains the same $x'_1 = x_1$.

4.4 2P-Rotation

The goal of this protocol is to generate two new secret shares x'_1, x'_2 from the old ones x_1, x_2 without changing the joint public key Q. The old secret shares can then be discarded; This mitigates most transient attacks on the private key since no combination of new and old secret shares can used to get the full private key. The secret shares are multiplicative, i.e. $Q = x_1 x_2 G$. Therefore, updating $x'_1 = r x_1$ and $x_2 = r^{-1} x_2$ for random r results in re-randomization of the secret shares with the same public key.

2P-Rotation

1. Run a string coin toss protocol (we use optimal rounds version of coin toss from [9]), the result is a random field element r.
2. Client updates $x_2' = r^{-1}x_2$ and $Q_1' = rQ_1$.
3. Server generates a new Paillier key pair and encrypts $x_1' = rx_1$.
4. The Server and the client run the same zero knowledge proofs as in $\{2, 2\}$ KeyGen, namely - proof of correct Paillier public key and L_{PDL} proof ([3] section 6.1) including range proof, using Q_1' instead of Q_1

It is important to notice that 2P-HD and Rotation are commutative and can be applied in different order with the same outcome. This means that only the master key needs to be rotated.

4.5 Recovery

We start with the case where the client wants the server to backup the server's secret share x_1 such that in a set of specified cases it will be recovered by the client - resulting in the client hold of the full private key x_1x_2. We use a protocol based on DLog Verifiable Encryption [10]. The backup of a secret share is done with the help of an Escrow. The escrow has a known public key Q_e (might be attached to a blockchain address). The server will send the client an encryption c of x_1 under the escrow public key. The server will also send the client a zero knowledge proof proving that (a) the encryption is for a DLog corresponds to Q_1 and (b) the encryption can be decrypted using the Escrow's private key k_e. The Escrow can periodically send a life signal that it still owns the private key k_e corresponds to Q_e by sending a signature with k_e on a message from today's paper. When the client to recover the share, if for example the server did not send a life signal to the Escrow (or some smart contract) for long time, the Escrow publishes its private key and the client can decrypt c and get x_1.
Similarly, the Server can keep the encryption c and use it as a backup for x_1, such that if something happens to it, the Server can recover by asking the Escrow for its private key.

Verifiable Recovery

1. *Encryption:* The server divides x_1 to m small enough segments $[x_1]_i, i \in \{1, ..., m\}$ such that is is feasible to compute the discrete logarithm. Each segment is encrypted using the Escrow public key Q_k (such that $Q_k = k_eG$) using ElGamal "encryption in the exponent": $(D_i, E_i) = ([x_1]_iG + r_iQ_e, r_iG)$. The server sends the ciphertext to the client.
2. *Proofs:* The server sends zero-knowledge proofs:
   (a) $x_1 = \sum_{i=1}^{m} [x_1]_i$
   (b) proof of knowledge that $(D_i, E_i) = ([x_1]_iG + r_iQ_e, r_iG)$
   (c) proof that the segments bit size is small (we use Bulletproofs [11])

3. The Escrow sends periodically a signature of a time-stamped message using k_e. The client Verifies the signature using Q_e.
4. *Decryption:* After k_e becomes public, the user will decrypt segment $[x_1]_i$ from $D_i - k_e E_i$ by using best known algorithm for finding DLog over a small space of options.

We comment that a reciprocal treatment can be done for the client backup.

5 Code Architecture

In general, we try to maximize the amount of code we re-use. Our stack is built completely in Rust. At the bottom we use secp256k1 elliptic curve (EC) library [12]. On top we have a utility library that implements simple EC cryptographic primitives [13]. At the same level we also wrote a library for zero knowledge proofs over Paillier cryptosystem [14] that uses another library for basic Paillier operations [15]. On top we have our 2P-ECDSA library [16] that consumes Paillier and ECC that we implemented below. higher than that we have a key management system (KMS) library [17] that packs 2P-ECDSA communication messages and adds the concept of master key with rotation, 2P-HD and recovery. Finally we have a wallet app with network, DB and full node access [18]. We construct Bitcoin transactions using Rust Bitcoin library [19]

6 Challenges

Achieving good performances without compromising on security represents the main challenge. For a good user experience we need fast signing and key derivation times since these operations are reoccurring. Key Generation and backup are done once but still need to be reasonably fast. To achieve this we optimized on multiple fronts: We minimized communication rounds by choosing zero-knowledge proofs that can run non-interactively (unless the communication tradeoff was worse). We aggregated different steps of the protocols into the same message whenever it was possible without harming security. We used software optimizations like parallelism of different Paillier encryptions. We used mathematical best practices like using Chinese Reminder Theorem for Paillier encryptions. Finally, we used native Rust all the way to not suffer when bridging between languages.

References

1. P.D. MacKenzie and M.K. Reiter. Two-party generation of DSA signatures. International Journal of Information Security,pp 218239, (2004). An extended abstract appeared at CRYPTO 2001
2. D. Boneh, R. Gennaro, S. Goldfeder. Using Level-1 Homomorphic Encryption To Improve Threshold DSA Signatures For Bitcoin Wallet Security. In Latincrypt 2017.

3. Y. Lindell. Fast Secure Two-Party ECDSA Signing. In CRYPTO 2017, Springer (LNCS 10402), pages 613-644, 2017.
4. J. Doerner, K, Yashvanth, L. Eysa, A, Shelat. Secure Two-party Threshold ECDSA from ECDSA Assumptions. In 2018 IEEE Symposium on Security and Privacy (SP), pp. 980-997. IEEE, 2018.
5. R. Gennaro, S. Goldfeder. Fast Multiparty Threshold ECDSA with Fast Trustless Setup ACM Conference on Computer and Communications Security (CCS), 2018.
6. Y. Lindell and A. Nof. Fast Secure Multiparty ECDSA with Practical Distributed Key Generation and Applications to Cryptocurrency Custody. ACM Conference on Computer and Communications Security (CCS), 2018.
7. P. Paillier, D. Pointcheval (1999). Efficient Public-Key Cryptosystems Provably Secure Against Active Adversaries. ASIACRYPT. Springer. pages. 165-179, 1999
8. S. Goldberg, L. Reyzin, O. Sagga and F. Baldimtsi. Certifying RSA Public Keys with anEfficient NIZK. Cryptology ePrint Archive: Report 2018/057, 2018
9. Y. Lindell, How to Simulate iT - A Tutorial on the Simulation Proof Technique, https://eprint.iacr.org/2016/046.pdf
10. J. Camenisch, and V. Shoup. Practical verifiable encryption and decryption of discrete logarithms. Advances in Cryptology CRYPTO 2003. pages 126-144, Springer, Berlin, Heidelberg, 2003.
11. B. Bunz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, Bulletproofs: Short proofs for confidential transactions and more. In Security and Privacy(SP) IEEE Symposium, 2018.
12. https://github.com/rust-bitcoin/rust-secp256k1
13. https://github.com/KZen-networks/curv
14. https://github.com/KZen-networks/zk-paillier
15. https://github.com/mortendahl/rust-paillier
16. https://github.com/KZen-networks/multi-party-ecdsa
17. https://github.com/KZen-networks/kms-secp256k1
18. https://github.com/KZen-networks/gotham-city
19. https://github.com/rust-bitcoin/rust-bitcoin
20. https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki