Bitcoin Wallet Powered by Two Party ECDSA - Extended Abstract

Team KZen

https://github.com/KZen-networks

Abstract. We demonstrate a Bitcoin wallet that utilizes two party ECDSA (2P-ECDSA [3]). Our architecture relies of a simple client-server communication model. We show support for 2 party deterministic child derivation (2P-HD), secret share rotation and verifiable recovery. 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 availability. 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 key and digital signing. This can potentially benefit security but at the same time reveal new challenges such as availability, ownership and recovery. Bitcoin is utilizing 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 state that in a threshold wallet at no point in time one party will hold the full private key. All multi-party protocols must be secure against malicious adversaries.

A wallet requires also read and write access to a full node. We do not define it here but emphasize that there is no privacy between 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 blockchain will run additional protocol to authenticate the message to be signed.

2.1 Comparison to Multisig wallet

We can learn on the properties and advantages of using threshold signatures (ThreshSig) by comparing to MultiSig. Multisig is the current way to add threshold security to private key management. Key rotation means that an attacker must attack simultaneously different parties to get access to the full private key. In ThreshSig wallet this is done without changing the public key and without going through the blockchain. In MultiSig wallet the secret keys are non-proactive. Updating the keys can be done by making a blockchain transaction. Referring to bitcoin blockchain, a ThreshSig transaction (tx) is indistinguishable from a regular tx while Multisig tx has a more unique structure that can attest to the MultiSig policy for other observers of the network. The cost of a ThreshSig tx is the same as regulat tx since they have the same size. A MultiSig tx on the other hand, has a size and therefore cost, proportional to the number of participating parties. Finally, ThreshSig can be easily adapted for different types of blockchains as it depends only on cryptography. MultiSig is relying on the application layer of a blockchain , i.e. Script in Bitcoin, smart contract in Ethereum etc.. and therefore requires more effort to add support for new chains.

3 Design Considerations for Choosing 2P-ECDSA

We define two roles: Owners and Providers. An Owner is the end user who owns the funds in the account and holds one secret share of the private key. The Provider is another share holder of the private key but has no funds tied to this private key. Its role is to provide the additional security in the system, enabling the owner to generate keys and transact in distributed fashion. For maintaining maximal power at the hands of the Owner we have two minimal claims:

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

Working with 2 parties allows us to use classical server-client communication model. The Client will be the Owner and the Server will be the Provider. Working in a more general $\{t, n\}$ threshold model will have an efficiency penalty, will require p2p communication network with broadcast channel and will violate the first claim. For reasons of simplicity and security we chose to work in the 2 party setting.

4 Implementation Details for 2P-Thresh-Wallet

Let G be an elliptic curve group of prime order q with base point (generator) G.

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. 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} Key Gen algorithm

1. Init: Server Chooses a random x_1 and computes $Q_1 = x_1 \cdot G$, Client Chooses a random x_2 and computes $Q_2 = x_2 \cdot G$
2. Server and Client run ECDH key exchange (in fact a bit more involved version of EC-DH [3]). $Q = x_1 x_2 G$ is the resulting joint public key.
3. The Server generates Paillier 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 and Client run a proof of correct Paillier public key. We use a non interactive version of the proof given in [7] with parameters given in [6]. The Server is the prover and the client is the verifier.
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. Client initiates a 2-round zero-knowledge protocol for the server to prove that $x_1 \in Z_q$ where q is the order of the elliptic curve. The protocol is given in [3] Appendix A.

All commitments in the protocol are hash based. We use SHA256.

4.2 {2,2} Signing

We first describe ECDSA signature scheme:
Assume that Alice has a private key x. t 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 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)

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: 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.

4.3 2P-HD

The purpose of 2P-HD protocol is to allow each party to hold one master key. 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 but is incompatible to the standard. Instead it was designed to support natively the two party case with much efficiency. Forcing BIP32 will make an inefficient protocol and since there is no way to import or export between BIP39/BIP32 compatible wallet to a 2 party wallet there is only small benefit for supporting the standard.

2P-HD algorithm

1. The Server and Client run ECDH to get a shared secret which we call chain code cc.
   (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. $cc' = f_r \cdot cc$
   (b) The Client updates his secret share to $x'_2 = f_l \cdot x_2$

The protocol takes as input a vector of indices and repeats the algorithm each input Q' and next index i.

4.4 2P-Rotation

The secret shares are multiplicative, i.e. $Q = x_1 x_2 G$. Therefore, updating $x'_1 = rx_1$ 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 [8]), 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 Client run the same zero knowledge proofs as in $\{2, 2\}$ KeyGen, namely - proof of correct Paillier public key and PDL proof 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 use a protocol based on DLog Verifiable Encryption [9]. The backup of a secret share is done with the help of an Escrow. Lets assume the Client wants the Server to backup its x_1. 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 (b) the encryption can be decrypted using the Escrow 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 needs a recovery, 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 $[x_1]_i, i \in \{1, ..., m\}$. 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 zk proofs: (1) $x_1 = \sum_{i=1}^{m} [x_1]_i$ (2) proof of knowledge that $(D_i, E_i) = ([x_1]_iG + r_iQ_e, r_iG)$ 3. proof that the segments bit size is small (we use Bulletproofs [10]).
3. The Escrow sends periodically 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 [11]. On top we have a utility library that implements simple EC cryptographic primitives [12]. At the same level we also wrote a library for zk proofs over Paillier cryptosystem [13] that uses another library for basic Paillier operations [14]. On top we have our 2p-ecdsa library [15] that consumes Paillier and ECC that we implemented below. higher than that we have a key management system (KMS) library [16] 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 [17]. We construct bitcoin transactions using rust-bitcoin library [18]

6 Challenges

Performance without compromising on security represents the main challenge. For a good user experience we need fast signing and key derivation times since this 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 zk 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. S. Goldberg, L. Reyzin, O. Sagga and F. Baldimtsi. Certifying RSA Public Keys with anEfficient NIZK. Cryptology ePrint Archive: Report 2018/057, 2018
8. Y. Lindell, How to Simulate iT - A Tutorial on the Simulation Proof Technique, https://eprint.iacr.org/2016/046.pdf
9. 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.
10. 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.
11. https://github.com/rust-bitcoin/rust-secp256k1
12. https://github.com/KZen-networks/curv
13. https://github.com/KZen-networks/zk-paillier
14. https://github.com/mortendahl/rust-paillier
15. https://github.com/KZen-networks/multi-party-ecdsa
16. https://github.com/KZen-networks/kms-secp256k1
17. https://github.com/KZen-networks/gotham-city
18. https://github.com/rust-bitcoin/rust-bitcoin