Zero to Monero: Multisig Chapter

June 19, 2018 Draft 0.1.0

Kurt M. Alonso
kurt@oktav.se

koe
ukoe@protonmail.com

(for questions and comments, please CC both authors)

Contents

CHAPTER 1

Multisignatures in Monero

Cryptocurrency transactions are not recoverable. If someone steals your private keys or scams you, the money lost could be gone forever. Dividing signing power between people can weaken the potential danger of a miscreant.

For example, say you deposit money into a joint account with a security company that monitors for suspicious activity related to your account. Transactions can only be signed if both you and the company cooperate. If someone steals your keys, you can notify the company there is a problem, and the company will stop signing transactions for your account. This is usually called an 'escrow' service.[1]

Cryptocurrencies use a 'multisignature' technique to achieve so-called 'M-of-N multisig'. In M-of-N, N people cooperate to make a joint key, and only M people (for $M \leq N$) are needed to sign with that key. We begin this chapter by introducing the basics of N-of-N multisig, progress into N-of-N Monero multisig, generalize for M-of-N multisig, and then explain how to nest multisig keys inside other multisig keys.

In the Monero source code only N-of-N and (N-1)-of-N are currently available. Possible future developments include robust key aggregation, M-of-N multisig, and nesting multisig keys. Credit for robust key aggregation belongs to pseudonymous Surae Noether, who is working on a Monero Research Lab paper which also aims to prove Monero's multisignature scheme is secure.[2] Our only

[1] Multisignatures have a diversity of applications, from corporate accounts to newspaper subscriptions.

[2] This MRL paper might be published after Zero to Monero: First Edition. It is set to be Monero Research Bulletin MRL-0007, and will be found here (among other places) https://lab.getmonero.org/.

contributions are formally describing M-of-N multisig, and a novel approach to nesting multisig keys.

1.1 Communicating with co-signers

Building joint keys and joint transactions requires communicating secret information between people who could be located all around the globe. To keep that information secure from unwanted observers, co-signers need to encrypt the messages they send each other.

A very simple way to encrypt messages using elliptic curve cryptography is via Diffie-Hellman exchange (ECDH). We already mentioned this in Section **??**, where Monero output commitment mask and amount are communicated to receivers via the shared secret rK^v. It looked like this:

$$mask_t = y_t + \mathcal{H}_n(\mathcal{H}_n(rK^v_B, t))$$
$$amount_t = b_t + \mathcal{H}_n(\mathcal{H}_n(\mathcal{H}_n(rK^v_B, t)))$$

src/ringct/ rctOps.cpp ecdhEncode()

We could easily extend this to any message. First encode the message as a series of bits, then break it into chunks equal in size to the output of $\mathcal{H}_n$. Generate a random number $r \in \mathbb{Z}_l$ and perform a Diffie-Hellman exchange on all the message chunks using the recipient's public key K. Send those encrypted chunks to the intended recipient along with the public key rG, who can then decrypt the message with the shared secret krG. Message senders should also sign encrypted messages (usually the encrypted message's hash) so they can't be easily tampered with.

Since encryption is not essential to the operation of a cryptocurrency like Monero, we do not feel it necessary to go into more detail. Curious readers can look at this excellent conceptual overview [1], or see a technical description of the popular AES encryption scheme here [2]. Also, Dr. Bernstein developed an encryption scheme known as ChaCha [3, 7], which the primary Monero implementation uses to encrypt certain sensitive information related to users' wallets (such as key images for owned outputs).

src/wallet/ ringdb.cpp

1.2 Key aggregation for addresses

1.2.1 Naive approach

Let's say there are N people who want to create a shared multisignature address, which we denote $(K^{v,sh}, K^{s,sh})$. Funds can be sent to that address just like any normal address, but, as we will see later, to spend those funds all N people have to work together to sign transactions.

Since all N participants should be able to view funds received by the shared address, we can let everyone know the shared view key $k^{v,sh}$ (recall Sections **??**, **??**). To give all participants equal power, the view key can be a sum of view key components that all participants send each other

src/multisig/multisig.cpp generate_multisig_view_secret_key()

securely. For participant $e \in \{1, ..., N\}$, his view key component is $k^v_e \in_R \mathbb{Z}_l$, and all participants can compute the shared private view key

$$k^{v,sh} = \sum_{e=1}^{N} k^v_e$$

In a similar fashion, the shared spend key $K^{s,sh} = k^{s,sh}G$ could be a sum of private spend key components. However, if someone knows all the private spend key components then they know the total private spend key. Add in the private view key and he can sign transactions on his own. It wouldn't be multisignature, just a plain old signature.

Instead, we get the same effect if the shared spend key is a sum of public spend keys. Say the participants have public spend keys K^s_e which they send to each other securely. Now let them each compute

src/multisig/multisig.cpp generate_multisig_N_N()

$$K^{s,sh} = \sum_{e} K^s_e$$

Clearly this is the same as

$$K^{s,sh} = (\sum_{e} k^s_e)G$$

1.2.2 Drawbacks to naive approach

Using a sum of public spend keys is intuitive and seemingly straightforward, but leads to a few issues.

Key aggregation test

An outside adversary who knows all the public spend keys K^s_e can trivially test a given public address (K^v, K^s) for key aggregation by computing $K^{s,sh} = \sum_e K^s_e$ and checking $K^s \stackrel{?}{=} K^{s,sh}$. This ties in with a broader requirement that aggregated keys be indistinguishable from normal keys, so observers can't gain any insight into users' activities based on the kind of address they publish.[3]

We can get around this by creating brand new spend keys for each multisignature address, or by masking old keys. The former case is easy, but may be inconvenient.

The second case proceeds like this: given participant e's old key pair (K^v_e, K^s_e) with private keys (k^v_e, k^s_e) and random masks μ^v_e, μ^s_e,[4] let his new private key components for the shared address be

$$k^v_e = \mathcal{H}_n(k^v_e, \mu^v_e)$$
$$k^s_e = \mathcal{H}_n(k^s_e, \mu^s_e)$$

[3] If at least one honest participant uses share components selected randomly from a uniform distribution, then keys aggregated by a simple sum are indistinguishable [6] from normal keys.

[4] The random masks could easily be derived from some password. For example, $\mu^s = \mathcal{H}_n(password)$ and $\mu^v = \mathcal{H}_n(\mu^s)$.

If participants don't want observers to gather the new keys and test for key aggregation, they would have to communicate their new key components to each other securely.

If key aggregation tests are not a concern, they could publish their public key components (K_e^v, K_e^s) as normal addresses. Any third party could then compute the shared address from those individual addresses and send funds to it, without interacting with any of the joint recipients [5].

Key cancelation

If the shared spend key is a sum of public keys, a dishonest participant who learns his collaborators' spend key components ahead of time can cancel them.

For example, say Alice and Bob want to make a shared address. Alice, in good faith, tells Bob her key components (k_A^v, K_A^s). Bob privately makes his key components (k_B^v, K_B^s) but doesn't tell Alice right away. Instead, he computes $K_B'^s = K_B^s - K_A^s$ and tells Alice $(k_B^v, K_B'^s)$. The shared address is:

$$\begin{aligned} K^{v,sh} &= (k_A^v + k_B^v)G \\ &= k^{v,sh}G \\ K^{s,sh} &= K_A^s + K_B'^s \\ &= K_A^s + (K_B^s - K_A^s) \\ &= K_B^s \end{aligned}$$

This leaves a shared address $(k^{v,sh}G, K_B^s)$ where Alice knows the private shared view key, and Bob knows both the private view key *and* private spend key! Bob can sign transactions on his own, fooling Alice, who might believe funds sent to the address can only be spent with her permission.

We could solve this issue by requiring each participant, before aggregating keys, make a signature proving they know the private key to their spend key component.[5] This is inconvenient and vulnerable to implementation mistakes. Fortunately a solid alternative is available.

1.2.3 Robust key aggregation

We can easily resist key cancellation with a small change to spend key aggregation (leaving view key aggregation the same). Let the set of N signers' spend key components be $\mathbb{S} = \{K_1^s, ..., K_N^s\}$, ordered according to some convention (such as smallest to largest numerically, i.e. lexicographically).[6] The robust aggregated spend key is

$$K^{s,sh} = \sum_e \mathcal{H}_n(K_e^s, \mathbb{S}) K_e^s$$

Now if Bob tries to cancel Alice' spend key, he gets stuck with a very difficult problem.

[5] Monero's first iteration of multisignature, made available in April 2018, used this naive key aggregation, and required users sign their spend key components.

[6] $\mathbb{S}$ needs to be ordered consistently so participants can be sure they are all hashing the same thing.

$$\begin{aligned}
K^{s,sh} &= \mathcal{H}_n(K^s_A, \mathbb{S})K^s_A + \mathcal{H}_n(K'^s_B, \mathbb{S})K'^s_B \\
&= \mathcal{H}_n(K^s_A, \mathbb{S})K^s_A + \mathcal{H}_n(K'^s_B, \mathbb{S})K^s_B - \mathcal{H}_n(K'^s_B, \mathbb{S})K^s_A \\
&= [\mathcal{H}_n(K^s_A, \mathbb{S}) - \mathcal{H}_n(K'^s_B, \mathbb{S})]K^s_A + \mathcal{H}_n(K'^s_B, \mathbb{S})K^s_B
\end{aligned}$$

We leave Bob's frustration to the reader's imagination.

Just like with the naive approach, any third party who knows $\mathbb{S}$ and the corresponding public view keys can compute the shared address.

Since participants don't need to prove they know their private spend keys, or really interact at all prior to signing transactions, our robust key aggregation meets the so-called *plain public-key model*, where "the only requirement is that each potential signer has a public key"[5].[7]

Robust key aggregation has not yet been implemented in Monero, but since participants can store and use private key $k_e = H(K'_e, \mathbb{S})k'_e$ (for naive key aggregation, $k_e = k'_e$), updating Monero to use robust key aggregation will only change the key merge process.

1.3 Thresholded Schnorr-like signatures

It takes a certain amount of signers for a multisignature to work, so we say there is a 'threshold' of signers below which the signature can't be produced. A multisignature with N participants that requires all N people to build a signature, usually referred to as *N-of-N multisig*, would have a threshold of N. Later we will generalize this to M-of-N (M $\leq$ N) multisig where N participants create the shared address but only M people are needed to make signatures.

Let's take a step back from Monero and explore general N-of-N thresholded signatures. Say there are N people who each have a public key in the set $\mathbb{S} = \{K'_1, ..., K'_N\}$, where each person $e \in \{1, ..., N\}$ knows the private key k'_e. Their shared public key, which they will use to sign messages, is (using robust key aggregation)

$$K^{sh} = \sum_e \mathcal{H}_n(K'_e, \mathbb{S})K'_e$$

1.3.1 Simple threshold Schnorr-like signatures

All signature schemes in this document lead from Maurer's general zero-knowledge proof of knowledge [4], so we can demonstrate the essential form of thresholded signatures using a simple Schnorr-like signature (recall Section **??**).

Signature

Suppose the people of set $\mathbb{S}$ want to jointly sign a message $\mathfrak{m}$ using their shared public key K^{sh}. They could collaborate on a basic Schnorr-like signature like this

[7] As we will see later, key aggregation only meets the plain public-key model for N-of-N and 1-of-N multisig.

1. Each participant $e \in \{1, ..., N\}$ does the following:
    (a) picks random component $\alpha_e \in_R \mathbb{Z}_l$,
    (b) computes $\alpha_e G$,
    (c) and sends $\alpha_e G$ to the other participants securely.
2. Each participant computes
$$\alpha^{sh} G = \sum_e \alpha_e G$$
3. Each participant $e \in \{1, ..., N\}$ does the following:
    (a) computes the challenge $c = \mathcal{H}_n(\mathfrak{m}, [\alpha^{sh} G])$,
    (b) defines their response component $r_e = \alpha_e - c * k_e \pmod{l}$,
    (c) and sends r_e to the other participants securely.
4. Each participant computes
$$r^{sh} = \sum_e r_e$$
5. Any participant can publish the signature $\sigma(\mathfrak{m}) = (c, r^{sh})$.

Verification

Given K^{sh}, $\mathfrak{m}$, and $\sigma(\mathfrak{m}) = (c, r^{sh})$:

1. Compute the challenge $c' = \mathcal{H}_n(\mathfrak{m}, [r^{sh} G + c * K^{sh}])$.
2. If $c' = c$ then the signature is legitimate except with negligible probability.

We included the superscript *sh* for clarity, but in reality the verifier has no way to tell K^{sh} is aggregated unless a participant tells him, or unless he knows the key components K'_e.

Why it works

Response r^{sh} is the core of this signature. Participant e knows two secrets in r_e (α_e and k_e), so his private key k_e is information-theoretically secure from other participants (assuming he never reuses α_e). Moreover, verifiers use the shared public key K^{sh}, so all key components are needed to build signatures.

$$\begin{aligned}
r^{sh}G &= (\sum_e r_e)G \\
&= (\sum_e (\alpha_e - c * k_e))G \\
&= (\sum_e \alpha_e)G - c * (\sum_e k_e)G \\
&= \alpha^{sh}G - c * K^{sh} \\
\alpha^{sh}G &= r^{sh}G + c * K^{sh} \\
\mathcal{H}_n(\mathfrak{m}, [\alpha^{sh}G]) &= \mathcal{H}_n(\mathfrak{m}, [r^{sh}G + c * K^{sh}]) \\
c &= c'
\end{aligned}$$

1.3.2 Back's Linkable Spontaneous Threshold Anonymous Group (bLSTAG) signatures

Naturally, bLSTAG follows after simple Schnorr-like signatures. The concept remains the same, so we can jump right in. Recall Section **??**.

Signature

Let $\mathfrak{m}$ be the message to sign, $\mathcal{R} = \{K_1, K_2, ..., K_n\}$ a ring of n distinct public keys and K^{sh}_{π} the N-of-N shared public key to be signed with, where π is a secret index in $\mathcal{R}$.

1. Each participant $e \in \{1, ..., N\}$ does the following:
   (a) computes key image component $\tilde{K}_e = k_{\pi,e}\mathcal{H}_p(K^{sh}_{\pi})$,
   (b) and sends $\tilde{K}_e$ to the other participants securely.
2. Each participant computes the shared key image
$$\tilde{K}^{sh} = \sum_e \tilde{K}_e$$
3. Each participant $e \in \{1, ..., N\}$ does the following:
   (a) generates seed component $\alpha_e \in_R \mathbb{Z}_l$ and computes $\alpha_e G$ and $\alpha_e \mathcal{H}_p(K^{sh}_{\pi})$,
   (b) generates random numbers $r_{i,e} \in_R \mathbb{Z}_l$ for $i \in \{1, 2, ..., n\}$ but excluding $i = \pi$,
   (c) and sends $\alpha_e G$, $\alpha_e \mathcal{H}_p(K^{sh}_{\pi})$, and all $r_{i,e}$ to the other participants securely.
4. Each participant computes
$$\alpha^{sh}G = \sum_e \alpha_e G$$
$$\alpha^{sh}\mathcal{H}_p(K^{sh}_{\pi}) = \sum_e \alpha_e \mathcal{H}_p(K^{sh}_{\pi})$$

and all

$$r_i^{sh} = \sum_e r_{i,e}$$

5. Each participant $e \in \{1, ..., N\}$ can now build the signature:

   (a) Compute
   $$c_{\pi+1} = \mathcal{H}_n(\mathfrak{m}, [\alpha^{sh} G], [\alpha^{sh} \mathcal{H}_p(K_\pi^{sh})])$$

   (b) For $i = \pi+1, \pi+2, ..., n, 1, 2, ..., \pi-1$ calculate, replacing $n+1 \rightarrow 1$,
   $$c_{i+1} = \mathcal{H}_n(\mathfrak{m}, [r_i^{sh} G + c_i K_i], [r_i^{sh} \mathcal{H}_p(K_i) + c_i \tilde{K}^{sh}])$$

6. To close the signature, each participant $e \in \{1, ..., N\}$ does the following:

   (a) defines $r_{\pi,e} = \alpha_e - c_\pi * k_{\pi,e} \pmod{l}$,

   (b) and sends $r_{\pi,e}$ to the other participants.

7. Each participant can now compute
   $$r_\pi^{sh} = \sum_e r_{\pi,e}$$

The signature will be $\sigma(\mathfrak{m}) = (c_1, r_1^{sh}, ..., r_n^{sh})$, with key image $\tilde{K}^{sh}$.

Verification

Verification does not change, so we will not repeat ourselves. See Section **??**.

Why it works

Opening and closing the signature loop requires all key component owners to participate. Furthermore, the key image depends on each participants' private key components.

1.3.3 Multilayer Linkable Spontaneous Threshold Anonymous Group (MLSTAG) signatures

Expanding bLSTAG to MLSTAG is fairly trivial. Instead of one layer, there are m layers each containing a shared public key at the same secret index π. It does not matter if one group of people owns all the shared keys or if separate groups own them. In the end, each step of signature generation requires components from everyone. Recall Section **??**.

1.4 MLSTAG Ring Confidential signatures for Monero

Monero thresholded ring confidential transactions add some complexity because MLSTAG signing keys are one-time addresses and commitments to zero (for input amounts).

Recalling Section **??**: one-time addresses assigning ownership of a transaction's t^{th} output to whoever has public address (K^v_t, K^s_t) go like this

$$\begin{aligned} K^o_t &= \mathcal{H}_n(rK^v_t, t)G + K^s_t = (\mathcal{H}_n(rK^v_t, t) + k^s_t)G \\ k^o_t &= \mathcal{H}_n(rK^v_t, t) + k^s_t \end{aligned}$$

We can update our notation for shared address $(K^{v,sh}_t, K^{s,sh}_t)$:

$$\begin{aligned} K^{o,sh}_t &= \mathcal{H}_n(rK^{v,sh}_t, t)G + K^{s,sh}_t \\ k^{o,sh}_t &= \mathcal{H}_n(rK^{v,sh}_t, t) + k^{s,sh}_t \end{aligned}$$

Just as before, anyone with $k^{v,sh}_t$ and $K^{s,sh}_t$ can discover $K^{o,sh}_t$ is their address' output, and can decode the Diffie-Hellman terms for output amount and corresponding commitment mask (Section **??**).

This also means multisig subaddresses are possible (Section **??**). Multisig transactions using funds received to a subaddress require some fairly straightforward modifications to the following algorithms, which we mention in footnotes.

1.4.1 `RCTTypeFull` N-of-N multisig

Let's say the owners of a shared address $(K^{v,sh}, K^{s,sh})$ have received $j \in \{1, ..., m\}$ outputs with one-time addresses $K^{o,sh}_j$ and amounts $a_1, ..., a_m$, and now want to spend them with $t \in \{1, ..., p\}$ new outputs with amounts $b_1, ..., b_p$. Recall Section **??**.

Most parts of a multisig transaction can be completed by whoever initiated it. Only the MLSTAG signatures require collaboration. An initiator needs to do these things:

1. Generate a transaction private key $r \in_R \mathbb{Z}_l$ and communicate it to his fellow signers securely (Section **??**).
2. Select a random mask for output t, $y_t \in_R \mathbb{Z}_l$, and send the ECDH terms $mask_t$ and $amount_t$ to his fellow signers securely (Section **??**).
3. Produce range proofs for each output t, and send the signatures to his fellow signers securely. Recall Sections **??**, **??**, **??**.

Now the group of signers is ready to build input signatures.

Preparing transaction inputs

For each input j the participants select v sets of size m of additional unrelated addresses and their commitments from the blockchain, corresponding to apparently unspent outputs. They mix the addresses in a ring with their own m unspent outputs' addresses, adding fake commitments to zero, as follows (Section **??**):

$$\begin{aligned}\mathcal{R} = \{&\{K^o_{1,1}, ..., K^o_{1,m}, (\sum_j C_{1,j} - \sum_t C^b_t)\},\\ &...\\ &\{K^{o,sh}_{\pi,1}, ..., K^{o,sh}_{\pi,m}, (\sum_j C^a_{\pi,j} - \sum_t C^b_t)\},\\ &...\\ &\{K^o_{v+1,1}, ..., K^o_{v+1,m}, (\sum_j C_{v+1,j} - \sum_t C^b_t)\}\}\end{aligned}$$

Here the private keys for $\{K^{o,sh}_{\pi,1}, ..., K^{o,sh}_{\pi,m}, (\sum_j C^a_{\pi,j} - \sum_t C^b_t)\}$ are $k^{o,sh}_{\pi,1}, ..., k^{o,sh}_{\pi,m}, z$, where, using u_j as the output index in the transaction where $K^{o,sh}_{\pi,j}$ was sent to the multisig address, the private key

$$k^{o,sh}_{\pi,j} = \mathcal{H}_n(rK^{v,sh}, u_j) + \sum_e k^s_e$$

is a composite, which no single participant knows.[8]

Commitment to zero z is, assuming input amounts equal output amounts plus transaction fee, simply input commitment masks (obtained by Diffie-Hellman from received inputs, using the shared view key) minus output commitment masks (non-initiator participants can compute these from $mask_t$), which all participants can compute:

$$z = \sum_j x_j - \sum_t y_t$$

MLSTAG RingCT

First they construct the shared key images for all inputs $j \in \{1, ..., m\}$ with one-time addresses $K^{o,sh}_{\pi,j}$.

src/wallet/wallet2.cpp sign_multisig_tx()

1. For each input j each participant e does the following:

   (a) computes $\tilde{K}^o_{j,e} = k^s_e \mathcal{H}_p(K^{o,sh}_{\pi,j})$,

   (b) and sends $\tilde{K}^o_{j,e}$ to the other participants securely.

[8] If $K^{o,sh}_{\pi,j}$ is built from an i-indexed multisig subaddress, then (from Section **??**) its private key is a composite:

$$k^{o,sh}_{\pi,j} = \mathcal{H}_n(k^{v,sh} r_{u_j} K^{s,sh,i}, u_j) + \sum_e k^s_e + \mathcal{H}_n(k^{v,sh}, i)$$

2. Each participant can now compute[9]
$$\tilde{K}_j^{o,sh} = \mathcal{H}_n(k^{v,sh}rG, u_j)\mathcal{H}_p(K_{\pi,j}^{o,sh}) + \sum_e \tilde{K}_{j,e}^o$$

Then they build the MLSTAG signature.

1. Each participant e does the following for $j \in \{1, ..., m+1\}$:
   (a) generates seed components $\alpha_{j,e} \in_R \mathbb{Z}_l$ and computes $\alpha_{j,e}G$ and $\alpha_{j,e}\mathcal{H}_p(K_{\pi,j}^{o,sh})$,
   (b) generates, for $i \in \{1, ..., v+1\}$ except $i = \pi$, random components $r_{i,j,e}$,
   (c) and sends all $\alpha_{j,e}G$, $\alpha_{j,e}\mathcal{H}_p(K_{\pi,j}^{o,sh})$, and $r_{i,j,e}$ to the other participants securely.
2. Each participant can compute all
$$\alpha_j^{sh}G = \sum_e \alpha_{j,e}G$$
$$\alpha_j^{sh}\mathcal{H}_p(K_{\pi,j}^{o,sh}) = \sum_e \alpha_{j,e}\mathcal{H}_p(K_{\pi,j}^{o,sh})$$
$$r_{i,j}^{sh} = \sum_e r_{i,j,e}$$
3. Each participant can build the signature loop (see Section **??**).
4. To finish closing the signature, each participant e does the following for $j \in \{1, ..., m\}$:
   (a) defines $r_{\pi,j,e} = \alpha_{j,e} - c_\pi k_e^s \pmod l$, and $r_{\pi,m+1,e} = \alpha_{m+1,e} - c_\pi z \pmod l$,
   (b) and sends $r_{\pi,j,e}$ to the other participants securely.
5. Everyone can compute[10]
$$r_{\pi,j}^{sh} = \sum_e r_{\pi,j,e} - c_\pi * \mathcal{H}_n(k^{v,sh}rG, u_j)$$
$$r_{\pi,m+1}^{sh} = \sum_e r_{\pi,m+1,e}$$

src/wallet/wallet2.cpp signMultisig()

The signature is $\sigma(\mathfrak{m}) = (c_1, r_{1,1}^{sh}, ..., r_{1,m+1}^{sh}, ..., r_{v+1,m+1}^{sh})$ with $(\tilde{K}_1^{o,sh}, ..., \tilde{K}_m^{o,sh})$.

Since in Monero the message $\mathfrak{m}$ and the challenge c_π depend on all other parts of the transaction, any participant who tries to cheat by sending the wrong value, at any point in the whole process, to his fellow signers will cause the signature to fail. $r_{\pi,j}^{sh}$ is only useful for the $\mathfrak{m}$ and c_π it is defined for.

[9] If the one-time address corresponds to an i-indexed multisig subaddress, add
$$\tilde{K}_j^{o,sh} = ... + \mathcal{H}_n(k^{v,sh}, i)\mathcal{H}_p(K_{\pi,j}^{o,sh})$$

[10] If the one-time address $K_{\pi,j}^{o,sh}$ corresponds to an i-indexed multisig subaddress, include
$$r_{\pi,j}^{sh} = ... - c_\pi * \mathcal{H}_n(k^{v,sh}, i)$$

1.4.2 `RCTTypeSimple` N-of-N multisig

`RCTTypeSimple` uses pseudo-output commitments for inputs, and signs inputs separately. To accommodate these differences from `RCTTypeFull`, we need one more sequence of events prior to building input MLSTAGs. The initiator must construct pseudo-output commitments (Section **??**).

He selects, for each input $j \in \{1, ..., m-1\}$, mask components $x'_j \in_R \mathbb{Z}_l$, then computes the m^{th} mask as

$$x'_m = \sum_t y_t - \sum_{j=1}^{m-1} x'_j$$

He communicates all x'_j to the other participants securely.

With pseudo-output commitments $C'^a_j = x'_j G + a_j H$, all participants will know the commitments to zero $z_j = x_j - x'_j$. MLSTAG signatures use just one input at a time with distinct secret indices π_j, but otherwise proceed the same as in Section 1.4.1.

1.4.3 Simplified communication

It takes a lot of steps to build a multisignature Monero transaction. We can reduce interactions between signers by consolidating communication into two parts with four total rounds.

1. *Key aggregation for a multisig public address.* Anyone with a set of public addresses can merge an N-of-N address, but no participant will know the shared view key unless they learn all the components, so the group starts by sending k^v_e and K'^s_e to each other securely. Any participant can merge and publish $(K^{v,sh}, K^{s,sh})$, allowing the group to receive funds to the shared address. M-of-N aggregation requires more steps, which we describe in the next section.

2. *Transactions*:

   (a) Some participant or sub-coalition (the initiator) decides to write a transaction. They choose m inputs with one-time addresses $K^{o,sh}_j$ and amount commitments C^a_j, v sets of m additional one-time addresses and commitments to be used as mixins, pick p output recipients with public addresses (K^v_t, K^s_t) and amounts b_t to send them, decide a transaction fee f,[11] pick a transaction private key r,[12] if using `RCTTypeSimple` generate pseudo-output commitment masks x'_j with $j \neq m$, construct the ECDH terms $mask_t$ and $amount_t$ for each output, and produce range proofs for each output. They also prepare their contribution for the next communication round.

[11] The transaction fee does not have to be explicit, because anyone can infer it from $\sum_j a_j - \sum_t b_t = f$.

[12] Or transaction private keys r_t if sending to at least one subaddress.

The initiator sends all this information to the other participants securely.[13] The other participants can signal agreement by sending their part of the next communication round, or negotiate for changes.

(b) Each participant chooses their components of the MLSTAG signature(s), and sends all of it to other participants securely.

MLSTAG Signature(s): key image $\tilde{K}^o_{j,e}$, signature randomness $\alpha_{j,e}G$, $\alpha_{j,e}\mathcal{H}_p(K^{o,sh}_{\pi,j})$, and $r_{i,j,e}$ with $i \neq \pi_j$.

(c) Each participant closes their part of the MLSTAG signature(s), sending all $r_{\pi_j,j,e}$ to the other participants securely.

Assuming the process went well, all participants can finish writing the transaction and broadcast it on their own. Transactions authored by a multisig coalition are indistinguishable from those authored by individuals.

1.5 Recalculating key images

If someone loses their records and wants to calculate their address' balance (received minus spent funds), they need to check the blockchain. View keys are only useful for reading received funds; users need to calculate key images for all received outputs to see if they have been spent, by comparing with key images stored in the blockchain. Since members to a shared address can't compute key images on their own, they need to enlist the other participants' help.

Calculating key images from a simple sum of components might fail if dishonest participants report false keys. Given a received output with one-time address $K^{o,sh}$, the group can produce a simple 'linkable' Schnorr-like signature (basically single-key bLSTAG) to prove the key image $\tilde{K}^{o,sh}$ is legitimate without revealing their private spend key components or needing to trust each other.

Signature

1. Each participant e does the following:
   (a) computes $\tilde{K}^o_e = k^s_e \mathcal{H}_p(K^{o,sh})$,
   (b) generates seed component $\alpha_e \in_R \mathbb{Z}_l$ and computes $\alpha_e G$ and $\alpha_e \mathcal{H}_p(K^{o,sh})$,
   (c) and sends $\tilde{K}^o_e$, $\alpha_e G$, and $\alpha_e \mathcal{H}_p(K^{o,sh})$ to the other participants securely.
2. Each participant can compute:[14]
$$\tilde{K}^{o,sh} = \mathcal{H}_n(k^{v,sh} rG, u)\mathcal{H}_p(K^{o,sh}) + \sum_e \tilde{K}^o_e$$

[13] He doesn't need to send the output amounts b_t directly, as they can be computed from $mask_t$.

[14] If the one-time address corresponds to an i-indexed multisig subaddress, add
$$\tilde{K}^{o,sh} = ... + \mathcal{H}_n(k^{v,sh}, i)\mathcal{H}_p(K^{o,sh})$$

$$\alpha^{sh}G = \sum_e \alpha_e G$$
$$\alpha^{sh}\mathcal{H}_p(K^{o,sh}) = \sum_e \alpha_e \mathcal{H}_p(K^{o,sh})$$

3. Each participant computes the challenge
$$c = \mathcal{H}_n([\alpha^{sh}G], [\alpha^{sh}\mathcal{H}_p(K^{o,sh})])$$

4. Each participant does the following:
   (a) defines $r_e = \alpha_e - c * k^s_e \pmod{l}$,
   (b) and sends r_e to the other participants securely.

5. Each participant can compute[15]
$$r^{sh} = \sum_e r_e - c * \mathcal{H}_n(k^{v,sh}rG, u)$$

The signature is (c, r^{sh}) with $\tilde{K}^{o,sh}$.

Verification

1. Check $l\tilde{K}^{o,sh} \overset{?}{=} 0$.
2. Compute $c' = \mathcal{H}_n([r^{sh}G + c * K^{o,sh}], [r^{sh}\mathcal{H}_p(K^{o,sh}) + c * \tilde{K}^{o,sh}])$
3. If $c' = c$ then the key image $\tilde{K}^{o,sh}$ corresponds to one-time address $K^{o,sh}$ (except with negligible probability).

1.6 Smaller thresholds

At the beginning of this chapter we discussed escrow services, which used 2-of-2 multisig to split signing power between a user and a security company. That setup isn't ideal, because if the security company is compromised, or refuses to cooperate, your funds may get stuck.

We can get around that potentiality with a 2-of-3 multisig address, which has three participants but only needs two of them to sign transactions. Escrow services can offer 2-of-3 multisig where they provide one key and users provide two keys. Users can store one key in a secure location (like a safety deposit box), and use the other for day-to-day purchases. If the escrow service fails, a user can use the (secure key + day key) to withdraw funds.

Multisignatures with sub-N thresholds have a wide range of uses.

[15] If the one-time address $K^{o,sh}$ corresponds to an i-indexed multisig subaddress, include
$$r^{sh} = ... - c * \mathcal{H}_n(k^{v,sh}, i)$$

1.6.1 1-of-N key aggregation

Suppose a group of people want to make a multisig key K^{sh} they can all sign with. The solution is trivial: let everyone know the private key k^{sh}. There are three ways to do this.[16]

1. One participant or sub-coalition selects a key and sends it to everyone else securely.
2. All participants compute private key components and send them securely, using the simple sum as merged key.
3. Participants extend M-of-N multisig to 1-of-N. This might be useful if an adversary has access to the group's communications.

In this case, for Monero, everyone would know the private keys $(k^{v,sh,\text{1xN}}, k^{s,sh,\text{1xN}})$. Before this section all shared keys were N-of-N, but now we use superscript 1xN to denote 1-of-N merged keys.

1.6.2 (N-1)-of-N key aggregation

In an (N-1)-of-N shared key, such as 2-of-3 or 4-of-5, any set of (N-1) participants can sign. We achieve this with Diffie-Hellman shared secrets. Lets say there are participants $e \in \{1, ..., N\}$, with public keys K_e which they are all aware of.

Each participant e computes, for $w \in \{1, ..., N\}$ but $w \neq e$,

$$k^{\text{(N-1)xN}}_{e,w} = \mathcal{H}_n(k_e K_w)$$

Then he computes $K^{\text{(N-1)xN}}_{e,w} = k^{\text{(N-1)xN}}_{e,w} G$ and sends it to all other participants securely.

Each participant will have (N-1) private key components corresponding to each of the other participants, making N(N-1) total keys between everyone. All keys are shared by two Diffie-Hellman partners, so there are really N(N-1)/2 unique keys. Those unique keys compose the set $\mathbb{S}$.

All N(N-1)/2 key components can be merged together using the robust key aggregation approach from Section 1.2.3. Furthermore, all N(N-1)/2 private key components can be assembled with just (N-1) participants since each participant shares one Diffie-Hellman secret with the N$^{\text{th}}$ guy.

For example, say there are three people with public keys $\{K'_1, K'_2, K'_3\}$, to which they each know a private key, who want to make a 2-of-3 multisig key. After Diffie-Hellman and sending each other the public keys they know the following:

1. Person 1: $k'^{\text{2x3}}_{1,2}$, $k'^{\text{2x3}}_{1,3}$, $K'^{\text{2x3}}_{2,3}$
2. Person 2: $k'^{\text{2x3}}_{2,1}$, $k'^{\text{2x3}}_{2,3}$, $K'^{\text{2x3}}_{1,3}$
3. Person 3: $k'^{\text{2x3}}_{3,1}$, $k'^{\text{2x3}}_{3,2}$, $K'^{\text{2x3}}_{1,2}$

[16] Note that key cancellation is largely meaningless here because everyone knows the full private key.

Where $k'^{2x3}_{1,2} = k'^{2x3}_{2,1}$, and so on. The set $\mathbb{S} = \{K'^{2x3}_{1,2}, K'^{2x3}_{1,3}, K'^{2x3}_{2,3}\}$.

The merged key is (robust method):

$$\begin{aligned} K^{sh,2x3} = &\mathcal{H}_n(K'^{2x3}_{1,2}, \mathbb{S})K'^{2x3}_{1,2} + \\ &\mathcal{H}_n(K'^{2x3}_{1,3}, \mathbb{S})K'^{2x3}_{1,3} + \\ &\mathcal{H}_n(K'^{2x3}_{2,3}, \mathbb{S})K'^{2x3}_{2,3} \end{aligned}$$

Now let's say persons 1 and 2 want to sign a message $\mathfrak{m}$. We will use a basic Schnorr-like signature to demonstrate.

1. Each participant $e \in \{1, 2\}$ does the following:
   (a) picks random component $\alpha_e \in_R \mathbb{Z}_l$,
   (b) computes $\alpha_e G$,
   (c) and sends $\alpha_e G$ to the other participants securely.
2. Each participant computes
$$\alpha^{sh}G = \sum_e \alpha_e G$$
$$c = \mathcal{H}_n(\mathfrak{m}, [\alpha^{sh}G])$$
3. Participant 1 does the following:
   (a) computes $r_1 = \alpha_1 - c * [k^{2x3}_{1,3} + k^{2x3}_{1,2}]$,
   (b) and sends r_1 to participant 2 securely.
4. Participant 2 does the following:
   (a) computes $r_2 = \alpha_2 - c * k^{2x3}_{2,3}$,
   (b) and sends r_2 to participant 1 securely.
5. Each participant computes
$$r^{sh} = \sum_e r_e$$
6. Either participant can publish the signature $\sigma(\mathfrak{m}) = (c, r^{sh})$.

The only change with sub-N threshold signatures is how to 'close the loop' by defining $r_{\pi,e}$ (in the case of ring signatures, with secret index π). Each participant must include their shared secret corresponding to the 'missing person', but since all the other shared secrets are doubled up there is a trick. Given the set $\mathbb{S}_o$ of all participants' original keys, only the *first person* - ordered by index in $\mathbb{S}_o$ - with the copy of a shared secret uses it to calculate his $r_{\pi,e}$.

In the previous example, participant 1 computes

$$r_1 = \alpha_1 - c * [k^{2x3}_{1,3} + k^{2x3}_{1,2}]$$

while participant 2 only computes

$$r_2 = \alpha_2 - c * k^{2\text{x}3}_{2,3}$$

The same principle applies to computing the shared key image in sub-N threshold Monero multisig transactions.

1.6.3 M-of-N key aggregation

We can understand M-of-N by adjusting our perspective on (N-1)-of-N. In (N-1)-of-N every shared secret between two public keys, such as K_1 and K_2, contains two private keys, $k_1 k_2 G$. It's a secret because only person 1 can compute $k_1 K_2$, and only person 2 can compute $k_2 K_1$.

What if there is a third person with K_3, there exists shared secrets $k_1 k_2 G$, $k_1 k_3 G$, and $k_2 k_3 G$, and the participants send these public keys to each other (making them no longer secret)? They each contributed a private key to two of the public keys. Now say they make a new shared secret with that third public key.

Person 1 computes shared secret $k_1 * (k_2 k_3 G)$, person 2 computes $k_2 * (k_1 k_3 G)$, and person 3 computes $k_3 * (k_1 k_2 G)$. Now they all know $k_1 k_2 k_3 G$, making a three-way shared secret (so long as no one publishes it).

The group could use $k'^{sh} = \mathcal{H}_n(k_1 k_2 k_3 G)$ as a shared private key, and publish $K^{sh,1\text{x}3} = \mathcal{H}_n(K'^{sh}, \mathbb{S}^{1\text{x}3}) K'^{sh}$ as a 1-of-3 multisig address.

In a 3-of-3 multisig every 1 person has a secret, in a 2-of-3 multisig, every group of 2 people has a shared secret, and in 1-of-3 every group of 3 people has a shared secret.

Now we can generalize to M-of-N: every possible group of (N-M+1) people have a shared secret. If (N-M) people are missing, all their shared secrets are owned by at least 1 person who remains. There are M people remaining, and they can collaborate to sign with the group's aggregated key.[17]

M-of-N algorithm

Given participants $e \in \{1, ..., N\}$ with initial private keys $k_1, .., k_N$ who wish to produce an M-of-N merged key (M $\leq$ N; M and N $\geq$ 2 since 1-of-N is trivial), we can use an interactive algorithm.

We will use $\mathbb{S}_s$ to denote all the *unique* public keys at stage $s \in \{0, ..., (N-M)\}$. The set $\mathbb{S}_{N-M}$ is ordered according to a sorting convention (such as smallest to largest numerically, i.e. lexicographically).[18]

We will use $\mathbb{S}^K_{s,e}$ to denote the set of public keys each participant created at stage s of the algorithm. In the beginning $\mathbb{S}^K_{0,e} = \{K_e\}$.

The set $\mathbb{S}^k_e$ will contain each participant's private keys at the end, when they hash their shared secrets and move into key merging. Participants use these keys to collaborate on signatures.

[17] Monero M-of-N multisig is currently under development, though may be available soon after this report is published.

[18] Notation: we use $\mathbb{S}[n]$ to denote the n$^{\text{th}}$ element of the set.

1. Each participant e sends their original public key set $\mathbb{S}^K_{0,e} = \{K_e\}$ to the other participants securely.
2. Each participant builds $\mathbb{S}_0$ by collecting all $\mathbb{S}^K_{0,e}$ and removing duplicates.
3. For merge stage $s \in \{1, ..., (N-M)\}$ (skip if M = N)
   (a) Each participant e does the following:
      i. For each element of $\mathbb{S}_{s-1} \notin \mathbb{S}^K_{s-1,e}$, compute new shared secret
      $$k_e * \mathbb{S}_{s-1}[g_{s-1}]$$
      ii. Put all new shared secrets in $\mathbb{S}^K_{s,e}$.
      iii. If $s = (N-M)$, compute the shared private key for each element x in $\mathbb{S}^K_{N-M,e}$
      $$\mathbb{S}^k_e[x] = \mathcal{H}_n(\mathbb{S}^K_{N-M,e}[x])$$
      and overwrite the public key by setting $\mathbb{S}^K_{N-M,e}[x] = \mathbb{S}^k_e[x] * G$.
      iv. Send the other participants $\mathbb{S}^K_{s,e}$.
   (b) Each participant builds $\mathbb{S}_s$ by collecting all $\mathbb{S}^K_{s,e}$ and removing duplicates.[19]
4. Each participant sorts $\mathbb{S}_{N-M}$ according to the convention.
5. The merged key is computed like this
$$K^{sh,\text{MxN}} = \sum_{g=1}^{\text{size of } \mathbb{S}_{N-M}} \mathcal{H}_n(\mathbb{S}_{(N-M)}[g], \mathbb{S}_{(N-M)}) * \mathbb{S}_{(N-M)}[g]$$
6. Each participant e overwrites each element x in $\mathbb{S}^k_e$ with
$$\mathbb{S}^k_e[x] = \mathcal{H}_n(\mathbb{S}^k_e[x]G, \mathbb{S}_{(N-M)}) * \mathbb{S}^k_e[x]$$

Note: if users want to have unequal signing power in a multisig, like 2 shares in a 3-of-4, they should use multiple key components instead of reusing the same one.

1.7 Key families

Up to this point we have considered key aggregation between a simple group of signers. For example, Alice, Bob, and Carol each contributing key components to a 2-of-3 multisig address.

Now imagine Alice wants to sign all transactions from that address, but doesn't want Bob and Carol to sign without her. In other words, (Alice + Bob) or (Alice + Carol) are acceptable, but not (Bob + Carol).

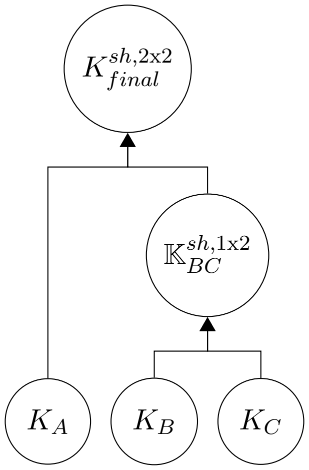
We can achieve that scenario with two layers of key aggregation: first a 1-of-2 multisig key $\mathbb{K}^{sh,1\text{x}2}_{BC}$ between Bob and Carol, then a 2-of-2 multisig key $K^{sh,2\text{x}2}_{final}$ between Alice and $\mathbb{K}^{sh,1\text{x}2}_{BC}$. Basically, a (1+(1-of-2))-of-2 multisig address. We will explain what $\mathbb{K}$ means soon.

This implies access structures to signing rights can be fairly open-ended.

[19] Participants should keep track of who has which keys at the last stage ($s = N-M$), to facilitate collaborative signing, where only the first person in $\mathbb{S}_0$ with a certain private key uses it to sign. See Section 1.6.2.

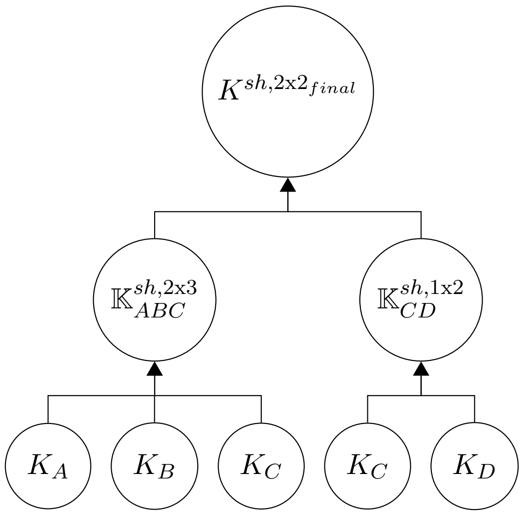
1.7.1 Family trees

We can diagram the (1+(1-of-2))-of-2 multisig address like this:


$K^{sh,2\text{x}2}_{final}$
$\mathbb{K}^{sh,1\text{x}2}_{BC}$
K_A
K_B
K_C

The keys K_A, K_B, K_C are considered *root ancestors*, while $\mathbb{K}^{sh,1\text{x}2}_{BC}$ is the *child* of *parents* K_B and K_C. Parents can have more than one child, though for conceptual clarity we consider each copy of a parent as distinct. This means there can be multiple root ancestors that are the same key.

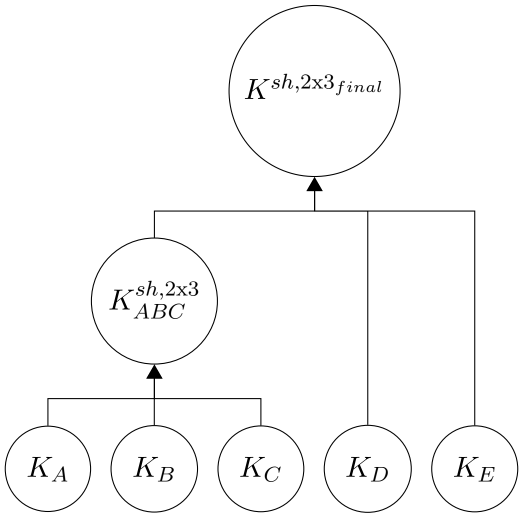
For example, in this 2-of-3 and 1-of-2 joined in a 2-of-2, Carol's key K_C is used twice and displayed twice:


$K^{sh,2\text{x}2_{final}}$
$\mathbb{K}^{sh,2\text{x}3}_{ABC}$
$\mathbb{K}^{sh,1\text{x}2}_{CD}$
K_A
K_B
K_C
K_C
K_D

Separate sets $\mathbb{S}$ are defined for each multisig sub-coalition. There are three sets in the previous example: $\mathbb{S}_{ABC} = \{K_{AB}, K_{BC}, K_{AC}\}$, $\mathbb{S}_{CD} = \{K_{CD}\}$, and $\mathbb{S}_{final} = \{\mathbb{K}^{sh,2\text{x}3}_{ABC}, \mathbb{K}^{sh,1\text{x}2}_{CD}\}$.

1.7.2 Nesting multisig keys

Suppose we have the following key family



If we merge the keys in $\mathbb{S}_{ABC}$ corresponding to the first 2-of-3, we run into an issue at the next level. Let's take just one shared secret, between K^{sh}_{ABC} and K_D, to demonstrate:

$$k_{ABC,D} = \mathcal{H}_n(k^{sh}_{ABC} K_D)$$

Now, two people from ABC could easily contribute key components so the sub-coalition can compute

$$k^{sh}_{ABC} K_D = \sum k_{ABC} K_D$$

The problem is everyone from ABC can compute $k_{ABC,D} = \mathcal{H}_n(k^{sh}_{ABC} K_D)$! If everyone from a lower-tier multisig knows all its private keys for a higher-tier multisig, then the lower-tier multisig might as well be 1-of-N.

We get around this by not completely merging keys until the final child key. Instead, we just do the first part, $\mathcal{H}_n(K, \mathbb{S})K$, for all keys output by low-tier multisigs, and put those in a key set

$$\mathbb{K}^{sh,out} = \{[\mathcal{H}_n(K_1, \mathbb{S})K_1], [\mathcal{H}_n(K_2, \mathbb{S})K_2], ...\}$$

To use $\mathbb{K}$ in a new multisig, we pass it around just like a normal key, with one change. Operations involving $\mathbb{K}$ use each of its member keys, instead of the whole merged key. For example, the public 'key' of a shared secret between $\mathbb{K}_x$ and K_A would produce a new key set that looks like

$$\mathbb{K}^{sh}_{x,A} = \{[\mathcal{H}_n(k_A \mathbb{K}_x[1])G], [\mathcal{H}_n(k_A \mathbb{K}_x[2])G], ...\}$$

This way all members of $\mathbb{K}_x$ only know shared secrets corresponding to their private keys from their lower-tier multisig. An operation between a keyset of size two ${}^2\mathbb{K}_A$ and keyset of size three ${}^3\mathbb{K}_B$ would produce a keyset of size six ${}^6\mathbb{K}_{AB}$. We can generalize all keys in a key family as keysets, where single keys are denoted ${}^1\mathbb{K}$. Elements of a keyset are ordered according to some convention (i.e. smallest to largest numerically), and sets containing keysets are ordered by the first element in each keyset, according to some convention.

We let the key sets propagate through the family structure until the final child appears, at which point every single key in all key sets is summed together.

More formally, we can say there is an operation `premerge` which takes in a set $\mathbb{S}$ and outputs a keyset $\mathbb{K}$ of equal size, containing each keyset $\mathbb{K}_i$ from $\mathbb{S}$ transformed into $\mathcal{H}_n(\mathbb{K}_i, \mathbb{S})\mathbb{K}_i$. There is another operation `merge` which takes in a set $\mathbb{S}$, `premerges` it into $\mathbb{K}$, and then outputs a keyset ${}^1\mathbb{K}$ that is the sum of all keysets of size one in $\mathbb{K}$. `Premerge` is used on the output sets of nested multisigs, and `merge` is used on the final child multisig's output set.[20]

While this may seem very complicated, a well-designed algorithm can easily keep track of everything.

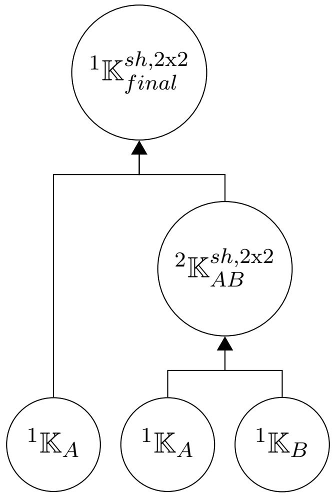
Note that the operation $\mathcal{H}_n(\mathbb{K}, \mathbb{S})\mathbb{K}$ (an abbreviated notation - the key set's members need to be separated out) needs to be done to the outputs of *all* nested multisigs, even when an N'-of-N' multisig is nested into an N-of-N, because the set $\mathbb{S}$ will change.

1.7.3 Implications for Monero

Each sub-coalition contributing to the final key needs to contribute components to Monero transactions, and so every sub-sub-coalition needs to contribute to its child sub-coalition.

This means every root ancestor, even when there are multiple copies of the same key in the family structure, must contribute one root component to their child, and each child one component to its child and so on. We use simple sums at each level.

For example, let's take this family


${}^1\mathbb{K}^{sh,2\text{x}2}_{final}$
${}^2\mathbb{K}^{sh,2\text{x}2}_{AB}$
${}^1\mathbb{K}_A$
${}^1\mathbb{K}_A$
${}^1\mathbb{K}_B$

Say they need to compute some shared value x for a signature. Root ancestors contribute: $x_{A,1}$, $x_{A,2}$, x_B. The total is $x^{sh} = x_{A,1} + x_{A,2} + x_B$.

There are currently no implementations of nested multisig in Monero.

[20] `Merge` can also be used on the output set of a nested multisig, if the sub-coalition composing it wants to use their shared address for other purposes.

Bibliography

[1] Cryptography tutorial. https://www.tutorialspoint.com/cryptography/index.htm [Online; accessed 05/19/2018].

[2] Federal information processing standards publication (FIPS 197). Advanced Encryption Standard (AES), 2001.

[3] Daniel J. Bernstein. Chacha, a variant of salsa20, 2008.

[4] Ueli Maurer. Unifying zero-knowledge proofs of knowledge. In Bart Preneel, editor, *Progress in Cryptology – AFRICACRYPT 2009*, pages 272–286, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[5] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. Simple schnorr multi-signatures with applications to bitcoin. 2018.

[6] Paola Scozzafava. Uniform distribution and sum modulo m of independent random variables. *Statistics & Probability Letters*, 18(4):313 – 314, 1993.

[7] A. Langley Google Inc. Y. Nir, Check Point. Chacha20 and poly1305 for ietf protocols. Internet Research Task Force (IRTF), May 2015. https://tools.ietf.org/html/rfc7539 [Online; accessed 05/11/2018].