Project Mahber

HashMapsData2Value

January 2, 2023

Contents

1 Introduction

At the time of writing holders of Algorand have no way of transacting in a *decentralized* privacy-preserving manner. While members of the Inc., as well as Silvio himself, have stated that they are investigating how privacy-preserving features could be added to the native protocol in a ”regulatory-compliant manner”, these features are likely not to come in the near-future.

Another project, Project Gjallarbro, seeks to introduce cross-chain atomic swaps between Algorand and Monero (and potentially other blockchains) using all the capabilities Algorand has at the present (i.e., the ed25519verify opcode). That solution would conveniently turn Monero into a L2 privacy layer (from the point-of-view of Algorand), but it would also come with other disadvantages such as latency and having to deal with another coin.

The aim of this paper, Project Mahber to highlight it as a different effort, sets out to show how holders on Algorand could deploy smart contracts that allow for some preservation of privacy in transactions, by only introducing small additions to the AVM. The cryptographic primitive that allow for this are called called ring signatures. They are also a core technology of Monero.

Note that privacy is a nebulous term that can mean a lot of different things. In this context we are referring to privacy of the end recipients, such that Alice can send *Algo/ASA* to Bob (or herself) without anyone being able to *directly* link the two.

Finally, there are many who want privacy but do not feel comfortable using their own honest activity to smokescreen potentially illicit activity, whether it be from a moral standpoint or a legal one[1]. Section 5 details how an oracle could potentially be included in the verification process, allowing the smart contract to adhere to sanctions by a trusted entity without compromising the privacy of unsanctioned account holders. This would be a completely optional feature.

Mahber (muh-ber) is Tigrinya **ማሕበር** and means ”association” or ”union” in the organizational sense.

2 Ring Signatures

Ring signatures are a type of digital signature that allow a member of a group to sign on behalf of that group, proving that they are a member of that group without leaking their exact identity.

This has a number of exciting use cases but in the context of cryptocurrencies

[1] Tornado Cash, a mixer service on Ethereum, was famously sanctioned by the United States government after having been used by North Korean hackers. One of the main developers was also apprehended by Dutch authorities. Note that Tornado Cash is built on zk-SNARK merkle proofs requiring a trusted setup and not ring signatures, though the point is the same.

one particularly useful thing is that it allows for privacy-preserving transactions. Imagine that the ”group” is ”the set of entities that have deposited money into a pool of funds”.

Alice could send her funds into a shared pool from one account and then extract it to another account without anyone being able to draw a straight connection between the two accounts. When extracting her funds she could prove that she is a member of the group of people who entered funds into the pool. Alternatively, Alice could have provided someone else, Bob, with the means to extract the money themselves to their own account.

The level of privacy afforded is dependent on the group size, also known as the anonymity set. The bigger the size, the more difficult it is to link accounts.

To prevent someone from repeatedly signing on behalf of the group and thus drain the shared funds, a so called 'key image' is introduced. By using it, we can make the ring signatures *linkable*, allowing a verifier to check if two ring signatures are linked. The key image itself becomes a proof of withdrawal for the verifier, allowing them to block subsequent withdrawal requests with the same key image.

2.1 Signing

Let $R = \{K_1, K_2, ..., K_n\}$ be a collection of n public keys, also known as a ring. Each public key has been generated by a corresponding private key k. H_p is a hash function that takes a point (public key) and outputs another point on the elliptic curve, ed25519 in this case. H_s is a normal scalar to scalar hash-function, e.g. Keccak256, which concatenates its individual inputs.

G is the generator such that $K = kG \pmod{l}$. The $+$ and $\cdot$ operators are either scalar or curve point operators depending on the case, with modular arithmetic done with mod l, the curve order. m is the message that is being signed.

1. Let k_π and K_π represent the prover.
2. Generate key image $\tilde{K} = k_\pi H_p(K_\pi)$.
3. Generate random integers a and r_i $\forall i \in \{1, 2, ..., n\}_{i \neq \pi}$.
4. Compute $c_{\pi+1} = H_s(m, aG, aH_p(K_\pi))$
5. $\forall i = \pi + 1, \pi + 2, ..., n, 1, 2, ..., \pi - 1$ (with $n + 1 \to 1$) compute:
   $c_{i+1} = H_s(m, [r_iG + c_iK_i], [r_iH_p(K_i) + c_i\tilde{K}])$
6. Define $r_\pi = a - c_\pi k_\pi$

The signature σ presented to verifier alongside m and the ring $R = \{K_1, K_2, ..., K_n\}$ can be defined as $\sigma(m) = \{c_1, r_1, ..., r_n, \tilde{K}\}$.

2.2 Verification

What about verification? It is done as follows.

1. Check that $l\tilde{K} = \mathbf{0}$ (the identity)
2. $\forall i = 1, 2, ..., n$ (with $n + 1 \rightarrow 1$) compute:
   $c'_{i+1} = H_s(m, [r_i G + c'_i K_i], [r_i H_p(K_i) + c'_i \tilde{K}])$.
3. If $c_1 = c'_1$ then all is good.

The reason why it works is because the only way someone could have produced a set of c' that seamlessly flowed together like this is if they had the private key for at least one of the public keys in the ring.

In particular, since $r_\pi = a - c_\pi k_\pi$ and $\tilde{K} = k_\pi H_p(K_\pi)$:
$\rightarrow c_{\pi+1} = H_s(m, [aG + c_\pi K_\pi - c_\pi K_\pi], [aH_p(K_\pi) - c_\pi k_\pi H_p(K_\pi) + c_\pi \tilde{K}])$
$\rightarrow c_{\pi+1} = H_s(m, aG, aH_p(K_\pi))$.

Intuitively, one can reason that had the key image not been defined from k_π and K_π, the signer would also not be able to set $r_\pi = a - c_\pi k_\pi$ and end up with the c_π needed for the ring signature.

Importantly, the value of $\tilde{K}$ is not dependent on the message being signed or the other public keys in the ring signature. In other words, it is the same every time k_π is used to sign, and can thus be used to link together multiple ring signatures signed by the same member of the group.

Regarding $l\tilde{K} = \mathbf{0}$, checking that the key image scalar multiplied with the curve order equals the identity element protects against a type of attack whereby the attacker is able to generate unlinked key images. For more information on this, refer to the following links:

- Crypto StackExchange - LibSodium x25519 and Ed25519 small order check?
- GetMonero Blog Post - Disclosure of Major Bug in CryptoNote Based Currencies
- Zero to Monero 2.0 - Page 31. Also the inspiration for the notation.

3 Suggested Opcodes

In the verification step, for each signature in the ring the verifier has to compute:

1. H_s once.
2. Scalar·Point multiplication four times. The $l\tilde{K} = \mathbf{0}$ check is done once, regardless of ring size.
3. Point+Point addition twice.

4. H_p once.

When making a deposit, the smart contract should check that K is a valid point, primarily as a guardrail to avoid users depositing funds they later can't extract but also to save withdrawers from having validate other submitted public keys they intend to include in their ring.

The Algorand core node software contains a fork of LibSodium within itself. We can make use of existing functions in it.

3.1 Point on the Curve

We need to check K at deposit time. The LibSodium fork has a function called crypto_core_ed25519_is_valid_point(). It performs several checks under the hood:

- ge25519_is_canonical
- ge25519_has_small_order
- ge25519_frombytes
- ge25519_is_on_curve
- ge25519_is_on_main_subgroup

Alternatively, especially in the short term, the already existing ed25519verify opcode could be utilized here by signing a message with k and passing it along. The smart contract could compare the signature with K and see that indeed, it is a valid signature and K is thus presumably also a valid point on the curve. The downside is that ed25519verify has a cost of 1900 units (three transactions worth) and it might be possible to get the cost down with a dedicated opcode.

3.2 Point Addition

The LibSodium fork has a function called crypto_core_ed25519_add.

3.3 Scalar · Point Multiplication

The LibSodium fork has two scalarmult functions. One is between a scalar and any point, called _crypto_scalarmult_ed25519. The second assumes the point is the base point _crypto_scalarmult_ed25519_base.

3.4 H_s

The AVM already offers several hash functions: Keccak256, SHA256 and SHA512_256. Monero uses Keccak256 in their ring signatures.

3.5 H_p: Curve Point → Curve Point

H_p is a special hash function that directly outputs a curve point. If it was possible to discover a scalar n such that $nG = Hp(x)$, the entire ring signature would be broken.

Monero has implemented the code for this hash function in (C/C++) hash_to_p3 and hash_to_ec. Original CryptoNote implementation here. A Python implementation, purely for educational and research purposes, can be found here.

For more information on how Monero has implemented H_p:

1. GetMonero PDF: Understanding ge_fromfe_frombytes vartime
2. GetMonero PDF: Monero is Not That Mysterious
3. Original CryptoNote 2.0 Paper. Makes a reference to Maciej Ula's paper Rational points on certain hyperelliptic curves over finite fields as the source of their H_p.
4. Delfr blog post: Cryptonote's Ring Signature Scheme.

The LibSodium fork has crypto_core_ed25519_from_uniform, a function explained in the docs as mapping a 32 byte vector to a point. It is an implementation of the Elligator 2 map. I am not sure if it is suitable for use here since according to the explanation there exists an inverse map from the point back to the 32 byte vector. Of course, we always hash public keys (public information), and the key image $\tilde{K} = k_\pi H_p(K_\pi)$ itself is a scalar multiplication with the private key and cannot be easily reversed (trap door function).

LibSodium implements the Map-to-Curve specified in the Hash to Curve RFC draft.

My hope is that it can be used straight up without any modifications. Advice and confirmation from someone more knowledgeable is welcome here.

4 Smart Contract

The funds are held in a smart contract that gate keeps deposits and withdrawals. The following are some considerations regarding that.

4.1 Making Deposits and Withdrawals Fungible

To achieve the privacy that we want, the smart contract must only be able to receive and send out funds in a set amount. For example, it could be set at 100 Algo. Algorand Standard Assets (ASA) representing fungible tokens are also appropriate, e.g. 10 of USDC.

By forcing everyone to only deposit and withdraw a fixed amount (e.g. 100 Algo) it is not possible to connect, using only transacted amounts, two accounts. I.e., deposits and withdrawals become fungible. More importantly, it also means the smart contract does not have to worry about the actual amount the withdrawer is allowed to take out as each $\sigma(m)$ with the valid, unspent key image, always only represents 100 Algo. In Monero this is handled using a type of non-interactive zero-knowledge range proofs called Bulletproofs.

Note that an account would not be limited to depositing only 100 Algo in total, they could repeatedly deposit in 100 Algo installments. For each installment they would need to generate a new key pair k and K. Of course, It might still be possible to connect two accounts if Account A deposited e.g. 19 installments of 100 Algo (1900 Algo in total) and account B just happened to also withdraw 19 installments of 100 Algo. The recommendation would be to split up and take, e.g., 11 installments into Account B and 8 installments into Account C.

4.2 Inspection, Storage and Validation of Public Keys and Key Images

Both the public keys and the key images need to be stored. To avoid any hiccups later the smart contract should verify that the public key K is actually a proper point on the curve.

With the introduction of Algorand Box, scaling storage has become not much simpler but also made this kind of smart contract possible. Since public keys and key images are both points on the curve, they can be represented with 32 bytes. Box names can be at most 64 bytes long and contain at most 32768 bytes (1024 keys).

When a public key K is provided in a deposit:

1. Check that K is a valid point.
2. Do a box_create with K set as the name and 32 bytes allocated. If K already exists box_create will fail, failing the deposit. Else the public key gets created and the deposit succeeds.

When a key image $\tilde{K}$ is provided in a withdrawal alongside the signature and the ring of public keys:

1. Check that $l\tilde{K} = \mathbf{0}$ (the identity)
2. Do a box_create with $\tilde{K}$ set as the name and 0 bytes allocated. If $\tilde{K}$ already exists box_create will fail, ending the submission. Else continue.
3. Follow the remaining steps of verification as mentioned in section 2.2.

If the ring provided contains public keys that do not exist as boxes already the withdrawal will fail.

4.3 Extracting Funds Safely and Privately

It is absolutely vital that the message m signed with the ring contains the final recipient account and that the smart contract uses it as a reference. If the message is generic it is be possible for rogue nodes to intercept the signature and simply withdraw the funds to themselves.

One issue that can arise is when the funds are meant to be withdrawn to an account that has not been activated on the blockchain yet. Not only does it not have any Algo to meet the minimum balance requirement, it does not have any Algo to pay for the Algorand fees (including verifying the ring signature) to withdraw in the first place.

A solution to this is to add logic such that while the bulk of the funds (e.g., 99 of the 100 Algo) are sent by the smart contract to an account specified in the message, the remaining Algo (e.g. 1 Algo) goes to the account that triggered the smart contract as a reward. With this logic, the signature and the entire contents of the transaction required for the withdrawal can be sent out to the world and anyone help the withdrawer out by triggering the withdrawal call on their behalf. This would mean that it does not matter to the actual withdrawer if rogue nodes grab and try to ”front run” the transaction, in fact they have a monetary incentive to do so and ensure they themselves get the reward.

4.4 Minimum Anonymity Set

Ring signatures only afford as much privacy to the signer as the number of public keys in the ring. Bigger ring, bigger obfuscation, but also more *costs*. The unfortunate reality with this signature scheme is that the verification cost grows linearly with the number of public keys.

It is still a good idea for the smart contract to force withdrawers to sign with a minimum ring size. The reason being, if Alice uses Bob's public key in the ring she creates and Bob later decides to be a cheapskate and create a very small ring signature, it a threat to the privacy of Alice's withdrawal.

The exact ring size number depends on just how low the total opcode cost can be brought down to for the verification of one element of the ring.

4.5 Ever-growing Storage

As entities use the smart contract to obfuscate links between accounts, the amount of Box storage occupied by public keys and key images will simply continue to grow. Since, by design and to our benefit, it is not possible to know which public keys are connected to which key image, and therefore which public key has already extracted its funds and what data can be safely purged.

Perhaps, as an agreed upon matter of common courtesy, the smart contracts can intentionally be created to simply max out after a certain amount of time, no longer accept any deposits and then self-destruct after all the funds have been withdrawn. This affects privacy negatively as every time a new smart contract is made there is always an initial period where the anonymity set is particularly small.

A different approach could be to use bloom filters. Bloom filters are space-efficient probabilistic data structures and could save a lot of space, at the cost of potential false-positives. While possible to implement, it opens the smart contract up to a number of vulnerabilities, such as a hacker being able to bruteforce a public key K that gives a false-positive satisfying the "has this K deposited 100 Algo?" bloom filter. Or a depositor being at risk of being locked out of withdrawing their funds because the "has this $\tilde{K}$ withdrawn 100 Algo?" bloom filter suddenly gave a false-positive.

Further research could be done here and advice is appreciated.

5 Adhering to Sanctions

Suppose that an account has been involved in scrupulous activity and the owner wants to obfuscate their funds through this smart contract. The owner could be anyone from a rug pull fraudster to a rogue state known for ransomware attacks. It could even be a legitimate business engaging in legitimate activity on-chain, such as selling adult-content or mind-altering substances, that some groups of people might not want to have their funds mixed with.

Could the contract be setup to address this, and thus potentially allow for more funds and public key deposits from those concerned by this? A potential solution is presented here.

Assume there is a trusted entity that can provides oracle data on-chain of sanctioned accounts. In other words, when pinged with an account address it will either return *true* (sanctioned) or *false* (not sanctioned).

We could adjust our smart contract's deposit and withdrawal methods such that when an account provides their public key (with the corresponding funds) the account address also gets stored. The smart contract could then check at both deposit and withdrawal times if an account is sanctioned.

At deposit time the smart contract only needs to check one account. At withdrawal time, the smart contract needs to check every account associated with each respective public key provided in the ring.

This hurts the "bad guys" without automatically putting at risk privacy of everyone else.

Typically, by the time a criminal gets sanctioned they will have already been able to move their illicit funds out of their sanctioned account. However, for the sake of privacy, they are unlikely to immediately withdraw their funds. It is thus more likely that criminal would be caught unable to withdraw.

In fact, there could be a time limit inside the smart contract set that ensures a certain amount of time has to have passed before a public key can be used in a ring. It could be trivially checked for by including a timestamp of when the public key was added in its box. This would give "law enforcement" a bigger time buffer. It goes without saying that immediately withdrawing funds after this time period by making use of a public key that has just become valid could potentially expose someone.

In the world of cryptocurrency we are naturally suspicious of centralized, trusted entities. A final piece of logic could be added that would allow for a sanctioned key to be used in a ring signature if the withdrawing account is itself the sanctioned account. Users could then be confident that even if their account was mistakenly placed in a sanctions list, they could take their funds back. This assumes that only one sanctioned account is involved, not the keys of multiple sanctioned accounts.

For law enforcement, provided they could get access to the sanctioned account owner they could also reclaim the illicit funds, rather than let them remain in perpetuity in the smart contract.

6 Bonus: Private e-Voting in DAO

There are other use-cases for ring signatures in Algorand beyond creating private transactions, use-cases that help justify the addition of new opcodes to the AVM.

Consider a DAO managing funds and various assets. The members of the DAO need to vote on how to spend the funds, and the vote needs to happen on the blockchain such that their will gets executed trustlessly by the smart contract. Ring signatures would allow the members to vote in an anonymous but secure manner. If the DAO is in charge of a lot of funds there might be a very compelling reason to want to have anonymous voting, beyond just avoiding the ”ruffled feathers” stemming from voting against someone’s proposal)

Assume that the DAO contract has the following logic:

- The DAO has a central managing smart contract, the DAO contract. Its logic executes the ”bylaws” of the DAO and holds the funds.
- To be a member of the DAO is to own a DAO-specific token. Different members hold different amounts of *stake*.
- The DAO token has the freeze address set to point to the DAO Contract. In fact, the token is by default frozen, and the only way to transact it is to ask the DAO contract to unfreeze it.
- Members of a DAO can ask the smart contract to initiate a vote to ”transact X amount of funds to address Y”, doing it publicly.
- If enough % of stake has signalled their support for initiating a vote the DAO contract proceeds forwar. If not enough agree to it, the DAO contract moves on. This has to happen within a specific time period.

- Within that time period the members of the DAO who voted to initiate a vote cannot ask the DAO contract to unfreeze their tokens. This is to ensure that they cannot initiate the vote and then pass their tokens around to other accounts, repeatedly voting. Meanwhile everyone else can ask for an unfreeze as usual, preventing a malicious attacker from permanently keeping everyone's tokens frozen. Since the total stake will remain static, it is enough for the smart contract to take total_stake minus frozen_initiator_stake divided by the total_stake to calculate the %.

The actual implementation of ”transact X amount of funds to address Y” could be handled by storing the tuple (Asset ID (0 if Algo), Amount, Recipient) in a ”Potential Outgoing Transactions” Box that is then moved to an ”Confirmed Outgoing Transactions” Box and executed following a successful vote.

Once the DAO contract has officially decided to put the matter to a vote (proposed publicly), the following procedure takes place:

1. The DAO contract enters ”voting mode” and *no* DAO tokens can be unfrozen.
2. Every token holder generates key pairs k/K and is allowed to pass in as many public keys K into the smart contract as is proportional to their stake. Depending on the market cap of the token and the Algorand fees associated with generation and validation of ring signatures it might be worth setting a minimum DAO token stake limit. E.g. the smart contract could demand each member have 10 DAO tokens per vote, where every vote corresponds to a key pair.
3. The members vote ”yes” or ”no” by using ring signatures of sufficient size to sign their vote message and passing along the key image. To avoid DAO members from having to vote from their own accounts, the DAO contract could send a small reward out to whomever does it on their behalf.
4. The DAO contract tallies the votes after a time period, implements the will of the members before cleaning up the store of public keys and key images.

In this way, a DAO could issue an anonymous (obfuscated) vote on-chain.

6.1 Repeatedly Initiating Costly Votes

It is possible for a malicious actor with sufficient stake to harm the DAO by repeatedly initiating costly votes and forcing the other DAO members to vote. The DAO creators need to have set appropriate stake requirements. Of course, normal proof-of-stake game theoretical factors are in play here, whereby the DAO token will simply lose its appeal (and thus its monetary value) to outsiders if enough is held by a malicious actor.

Additionally, the DAO token could have its clawback address set to the DAO contract. Besides allowing the DAO to charge a period membership fee, it would also allow the contract to have a ”slashing” functionality, where other members can vote to have a particularly egregious offending member unilaterally deprived of their tokens. Or, more mercifully, since the DAO contract is in charge of freezing/unfreezing tokens for members to transaction their tokens, it can also keep some kind of record of the rolling median DAO token price in Algo and offer the possibility for honest DAO members to buy out the offender. (Though this could of course also be gamed by the malicious actor...)

6.2 Bonus to the Bonus: Anonymous Tallying

There are situations where we don’t want just the voter to be secret but we also want the *message itself* to be secret.

Consider a situation where a DAO is in charge of voting for funding a Relay Node. A malicious Relay Node, either the one in question or malicious Relay Nodes in competition with it, might take offense and try to block votes that do not align with it. In this scenario it does not matter who actually made the vote in the first place, just that the vote was made.

If it is a simple matter of a ”yes/no” count then it is similar to that of the secure multi-party computation problem known as *Yao’s Millionaire’s problem.* Let’s consider this out of scope and leave it for the future.