Contents

Ring Signature based mixing on Ethereum

This is a report written by me, @noot, for my University of Toronto course ESC499. It's written assuming the person reading knows nothing about blockchain and/or Ethereum and/or cryptography. If you have some background knowledge, you can skip the intro :)

1. Introduction and Background

1.1 Ethereum

A **blockchain** is a decentralized network of Byzantine fault tolerant nodes which transition from one state to another through network consensus. Byzantine fault tolerance is an idea first introduced regarding distributed computer systems, particularly multi-core processors. If one part of the system is faulty and gives incorrect signals to the rest of the system, we want to be assured that the system overall maintains the correct state. In a **proof-of-work** blockchain, such as Ethereum, as long as over 50% of the nodes are honest, we can be assured that the correct state is maintained.

Ethereum is a blockchain which includes, alongside the transaction-based state, a quasi-Turing-complete stack machine also known as the Ethereum Virtual Machine (**EVM**), the specifications of which can be found in the Ethereum yellow paper [1]. This allows for the execution of **smart contracts** on the network. Smart contracts are pieces of software that live on the network, written in a Turing-complete language (commonly Solidity) and executed on the EVM. Smart contracts are deterministic and are able to make changes to state if certain conditions are met. The software used to connect to and participate in the network is called a **client**. The two most popular Ethereum clients are go-ethereum, or **geth**, which is written is Golang, and **parity-ethereum**, written in Rust. A client is able to sync the network, verify the transactions on the network, as well as send transactions to the network. The client also contains the EVM and is able to execute and verify smart contracts.

On Ethereum, addresses may be either user accounts *or* smart contracts. Transactions can be directed towards either a user account, usually involving transfer of ether, or a smart contract, involving execution of one of its functions. A user account has a corresponding **private key** which can be used to access the funds in the account. A contract *does not* have a corresponding private key; no one *owns* the contract once it is deployed. Functionality for ownership must be programmed into the contract from the start; there is no inherent owner of the contract.

All transactions on the network involve the native currency or fuel called **ether**. Every transaction on the network uses a certain amount of **gas** which is used to prevent smart contracts from using all the network capacity. For example, an infinite loop cannot occur as it will eventually run out of gas.

Each Ethereum block consists of a **block header** which contains information about the current state of the network. This current state includes both account balances (`txHash` and `receiptHash`) and values stored in smart contracts (`root` or `stateRoot`). The following is the structure of the block header in **geth**: [2]

```
type Header struct {
  ParentHash  common.Hash
  UncleHash   common.Hash
  Coinbase    common.Address
  Root        common.Hash
  TxHash      common.Hash
  ReceiptHash common.Hash
  Bloom       Bloom
  Difficulty  *big.Int
  Number      *big.Int
  GasLimit    uint64
  GasUsed     uint64
  Time        *big.Int
  Extra       []byte
  MixDigest   common.Hash
  Nonce       BlockNonce
}
```

In this report, the `gasLimit` and `gasUsed` parameters are relevant. Each transaction included in the block has a certain gas cost associated with it. The `gasLimit` of a block is the maximum amount of gas that can be consumed by the total of transactions in the block. The `gasUsed` is the amount of gas used actually by all of the transactions. `gasUsed` can range from 0 to `gasLimit.`

Transactions in Ethereum consist of a `to` address, a `from` address, a `value` in ether, and a `data` field, as well as information about gas. Additionally, the transaction needs to be cryptographically signed by the account it is coming from. The following is a `transaction` struct:

```
type transaction struct {
    AccountNonce uint64
    GasPrice     *big.Int
    GasLimit     uint64
    To      *common.Address
    Value        *big.Int
    Data         []byte
    V *big.Int
    R *big.Int
    S *big.Int
}
```

Relevant to this report is `data`, `to`, `value`, `gasPrice`, `gasLimit`, and v, r, s (the cryptographic signature parameters.) Notice there is no `from` field; the sender of the transaction is inferred from the signature.

In a transaction, `value` and `gasPrice` are measured in *wei*, not in ether. Wei is the smallest denomination of the currency; 10^18 wei == 1 ether.

`data` is a byte array of data used in contract calls. When the `to` address is the address of a contract, `data` contains encoded information about which function to call and with what parameters. If the `to` address is a user account, `data` is irrelevant.

`value` is the amount of wei that is being transferred from the sender to the recipient in this transaction. The `gasLimit` specifies the maximum amount of gas that may be used by this transaction and `gasPrice` specified the price of gas, in wei per unit of gas. Multiplying `gasLimit` by `gasPrice` gives the actual maximum cost, in wei, of the gas used by this transaction.

Transactions on the Ethereum network are signed using elliptic curve cryptography or **ECC**, which further explained in section 1.4. Specifically, the elliptic curve digital signature algorithm, or ECDSA, is used. The details of this algorithm are not relevant to this report; however, they can be found here. A user may have an account on the network, which consists of a public key, a private key, and an address. The details of the relationship between these parameters are explained in section 1.5.

1.2 Pre-compiled contracts

Within Ethereum, there exist what are called **precompiled contracts,** commonly referred to as **precompiles**. These are contracts which do not exist on the chain, like user-deployed contracts; instead, they exist within the Ethereum client itself. The name is a misnomer; these contracts are never actually written and compiled for the EVM. Instead, they are simply written in the client's native language and exist as part of the client's virtual machine. In the current version of Ethereum, known as *Byzantium*, there exist eight precompiled contracts. In geth, the list of precompiled contracts is the following: [2]

```
var PrecompiledContractsByzantium = map[common.Address]PrecompiledContract{
    common.BytesToAddress([]byte{1}): &ecrecover{},
    common.BytesToAddress([]byte{2}): &sha256hash{},
    common.BytesToAddress([]byte{3}): &ripemd160hash{},
    common.BytesToAddress([]byte{4}): &dataCopy{},
    common.BytesToAddress([]byte{5}): &bigModExp{},
    common.BytesToAddress([]byte{6}): &bn256Add{},
    common.BytesToAddress([]byte{7}): &bn256ScalarMul{},
    common.BytesToAddress([]byte{8}): &bn256Pairing{},
}
```

Each precompile has a corresponding address that it lives at. Addresses `0x01` to `0x08` are reserved; when making a transaction to one of these addresses, it is one of these pre-compiles that are being called. For example, calling address `0x01` will cause `ecrecover` to be executed on the call data.

The reason for including these precompiles is that they perform commonly used operations which would be computationally expensive if they were implemented as an on-chain contract. Precompiles have a *predetermined gas cost* which is much lower than what it would be if implemented on-chain. Without a precompile, these operations would take up most of, if not exceed, the block gas limit. Thus, precompiled contracts can be used to reduce the costs of commonly-used operations on the network. An example of a precompiled contract is `ecrecover`, which, given a signed message, recovers the public key used to sign the message.

1.3 Asymmetric cryptography

Asymmetric cryptography is also known as *public-key cryptography.* One party known as Alice wishes to receive an encrypted message from another party, Bob. Alice generates some secret number known as a **private key,** denoted k. She then derives another number from it, known as a **public key**, denoted P. The main idea behind this scheme is that, given a public key P, a private key k, and some function $derive : k \to P$, it is *computationally infeasible* to perform $derive^{-1}$ where k is found from P. [Theorem 1] It is easy to get our public parameter P from our secret number k, but impossible to get our secret k from P. Thus, it is safe for Alice to share her P with Bob.

To encrypt a message for Alice, Bob uses Alice's public key P as well as the message m. He performs $Encrypt : (P, m) \to m'$ to create encrypted message m' which is sent to Alice. Alice then performs $Decrypt : (k, m') \to m$ to recreate the message from the encrypted message. Without the private key k, it is computationally infeasible to recover m from m'. [Theorem 2]

Signing a message does not hide the message, but instead, it generates a cryptographic signature linking the message with a private key. For example, say Alice wishes to send Bob a message. For Bob to be certain that the message comes from Alice, she needs to sign it using her secret key. Given a message m,

Alice performs $Sign : (k, m) \to S$, where S is the signature. $Sign$ uses Alice's private key to generate a signature for the message. She then sends m and S to Bob, who performs $Verify : (P, S, m) \to (true, false)$. $Verify$ uses Alice's public key to check if the signature came from the corresponding private key, without needing to know the private key. If the signature did in fact come from the private key k corresponding to P and was for message m, then return true; otherwise, return false. It is computa
tionally infeasible to forge a signature; that is, to generate a valid S not knowing secret key k. [Theorem 3]

For additional information, see chapters 10 - 13 of *A Graduate Course in Applied Cryptography* by Boneh and Shoup. [3]

1.4 Elliptic curves

Elliptic curve cryptography, or **ECC**, is a method of implementing asymmetric cryptography. Using elliptic curves, we can easily generate private and public keys, as well as algorithms for digital signatures. First, we will need to define **finite fields** and some related terminology.

Definition 1.1: a *finite field*, denoted F, is a set of numbers that contains a finite number of elements. The group operations, addition and multiplication, are defined over this field.

Definition 1.2: The *field order* of F is the *number of elements* in F. This is denoted $|F|$. It is also referred to as the *cardinality* of the set. A *prime finite field* is a field denoted F_p where the order is a prime number p.

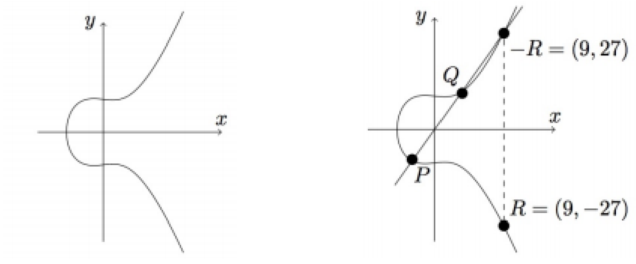
A prime field of order p may be constructed as the *integers modulo p*. In ECC, all operations on the elliptic curve are performed over the field F_p; i.e. all operations are performed *modulo p*.

Definition 1.3: An elliptic curve E is a curve of the form:

$$y^2 = x^3 + ax + b$$

This curve is defined over a prime field F_p. We write E/F_p to represent that E is defined over F_p.

Let's explore an example. Define our curve $E : y^2 = x^3 - x + 9$. This elliptic curve looks like the following:



(a) The curve (b) Adding $P = (-1, -3)$ and $Q = (1, 3)$

Figure 1.1: The elliptic curve y^2 = x^3 - x + 9

In figure (a), we see the curve plotted over $\mathcal{R}^{\in}$, the set of real numbers in two dimensions. It's easy for us to find two points that satisfy our equation $y^2 = x^3 - x + 9$; namely, $P = (-1, -3)$ and $Q = (1, 3)$. We wish to construct a method to find a third point, R, given two known points on the curve, P and Q.

As shown in figure (b), we draw a line through points P and Q. This line is defined by $y = 3x$. Then, obtain the point that is the intersection of this line and the curve. Denote this $-R = (9, 27)$. Invert the y-coordinate of $-R$ to obtain $R = (9, -27)$, our third curve point.

Note the similarity between this operation we have just defined and that of standard group addition; we take two known points to obtain a third point. Thus, define this operation as *addition*, denoted by the + operator, over an elliptic curve. The method we have used is the *tangent method*.

If we wish to add a point to itself, i.e. $P + P$, we draw a tangent to the point and find the point of intersection between the tangent and the curve, then invert its y-coordinate like above.

Definition 1.4: Denote $E(F_p)$ as the set of all points on curve E that are defined over F_p.
Definition 1.5: Define a "point at infinity", O. This point is the *identity element*, that is, for any element P in $E(F_p)$, $P + O = P$.

In the set of real numbers, zero is the identity element: say x is any real number. Then, $x + 0 = x$ for all x.

The interesting thing about addition over an elliptic curve is that if we continue to add a point to itself, i.e. $P + P + P + ...$, eventually we end up back at the same point, P. In other words, $E(F_p)$ is a finite set that can be operated over using modular arithmetic. Additionally, for every curve, there exists a point G that can create *any point on the curve* by means of addition. This can be thought of as the *smallest element* of the curve. In the set of real numbers, 1 is analogous to the generator; you can create any other real number by adding 1 to itself a certain number of times.

Definition 1.6: For a curve E/F_p, there exists a point on the curve G, the *generator*, which can be added to itself continuously to generate any point in the set $E(F_p)$.

Multiplication over the curve is repetition of addition; say we wish to find $P * 3$ for some point on our curve P. We calculate $P + P + P$ which equals $P * 3$. Thus, given our generator G, curve E/F_p and the order of our field p, we can generate any point on our curve! For each integer x in F_p, calculate $G * x$. Then, the resulting set is $E(F_p)$.

For additional information, see chapter 15 of *A Graduate Course in Applied Cryptography* by Boneh and Shoup. [3]

1.5 Elliptic curve public-key cryptography

Given the elliptic curve primitives defined in section 1.4, we can define public-private key pairs over an elliptic curve. First, let's discuss the discrete logarithm problem, the foundation of elliptic curve cryptography.

Define a curve E/F_p with generator G. Given an integer k in the field F_p, calculate a point on the curve by multiplying k with the generator to create *a point on the curve*: $P = kG$. We know from above how to do this multiplication; simply add G to itself k times.

However, say we are given this point P and generator G. Is there a way to recover k given these parameters; i.e., calculate $k = P * G^{-1}$ or $k = P/G$? Luckily for us, it is *computationally infeasible* to calculate k given P for a curve with a sufficiently large prime order. The best known discrete logarithm algorithm runs in $O(\sqrt{p})$, where p is the order of the field. For example, say we have a curve with order $\sim 2^{256}$. Then, to solve the discrete logarithm problem, this will take $\sim\sqrt{2^{256}}$ time, or $\sim 2^{128}$ time. This is infeasible to calculate, thus as long as a better algorithm to solve the discrete logarithm problem has not been found, we can say that a curve with order 2^{256} provides us 128 bits of security.

Definition 1.7: *Discrete logarithm problem.* Define a curve E/F_p with generator G. Given an integer k in the field F_p, calculate a point on the curve $P = kG$. It is *computationally infeasible* to recover k given P and G.

As an aside, the discrete logarithm problem is a somewhat misleading name in this situation; it would be more suitably called the *discrete division problem.*

Given our definition of the discrete logarithm problem, it is easy to extend it to public- and private-key cryptography. Say Alice wishes to create a public- and private-key pair for the curve E/F_p. Alice randomly picks an integer in the field F_p, denoted k; this is her *secret key.* She then multiplies k by the generator, G, to produce a point on the curve, P. P is Alice's public key. She can freely share P with anyone she wants and know that k cannot be recovered from P.

Definiton 1.8: *Public- and private-key pairs.* Define a curve E/F_p with generator G. Define $Gen\_private : F_p \rightarrow k$, which, given field F_p, randomly selects an integer from field F_p. Define $Gen\_public : (G, k) \rightarrow P$ which, given secret k and generator G, performs $P = k * G$ and returns point P.

Algorithms for digital signatures are built upon this foundation. Ethereum uses the algorithm ECDSA, the details of which are out of the scope of this report. See *Understanding Cryptography* by Paar and Pelzl for more information. **add link**

1.6 Hash functions and Ethereum accounts

Many cryptographic protocols make use of *cryptographic hash functions.* A *hash* of some data is an obfuscation of the original data. The original data may have any size; the output always has a fixed size. A hash is a one-way function; given the input, it is trivial to find the output, but given the output, it is impossible to find the input.

Definition 1.9: A cryptographic hash function is a function $H : x \rightarrow h$, which, given an input x of arbitrary size, generates an output h of fixed size. Given h, it is computationally infeasible to find x. This is referred to as *pre-image resistance.*

Additionally, changing one bit of the input should change approximately 50% of the bits of the output. It should also be computationally infeasible to find two unique inputs which map to the same output; i.e. a hash collision.

Ethereum uses the elliptic curve **secp256k1**, defined as $y^2 = x^3 + 7$ over the field F_p where $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$. [4] All public keys used on Ethereum are points on this curve; all private keys are numbers in the field F_p. Additionally, an **address** on Ethereum is the first 20 bytes of the *hash of the corresponding public key.*

1.7 Ring signatures

The signatures we have seen so far in this report are single-account signatures; that is, only one public key is included in the signature. There also exist *group signatures* in which a message is signed by a group of public keys, as opposed to just one. In the usual group signature scheme, every member of the group must sign the message.

In a ring signature scheme, we also have a group of public keys included in the signature, called a *ring.* Only one of the members of the ring is the signer of the message; however, it is impossible to determine which member of the ring is the signer. Ring signatures were originally proposed by Rivest et al. in the paper *How to leak a secret* [5], published in 2001. In this paper, the writers describe a method in which a secret could be leaked by a congress member. Given a ring consisting of the public keys of all congress members and a secret which one member wishes to leak, this member could generate then a ring signature for the message. An outsider receiving this signature will only be able to verify that one congress member signed the message, but they will be unable to determine which one. The public-key cryptography scheme used in *How to leak a secret* is RSA.

Ring signatures are a powerful scheme for producing *anonymity*. Another application of ring signatures could be voting. Someone wishes to vote but does not wish to have their vote linked to their identity; thus, generate a ring signature in which the message is the vote, and the ring is themselves plus many other voters.

There are limitations to ring signatures. Firstly, a ring signature still includes the true signer's public key. Thus, the scheme is not truly anonymous; it only offers *plausible deniability* in the face of an accusation. If the ring size is not sufficiently large, it may not offer such protection. Secondly, ring signatures are large. The size of a ring signature is determined by the size of the curve points as well as the number of public keys. To balance these two factors, a ring size of nine to twelve is suggested for both security and efficiency. For example, the ring signature-based cryptocurrency **Monero** uses a ring size of ten. [6]

The first ring signature scheme to be proposed using ECC is the *Cryptonote* whitepaper [7] by the pseudonymous Nicolas van Saberhagen in 2013. This scheme was then improved upon in the paper *Ring confidential transactions* [8] (or RingCT) released by the Monero research labs in 2015. The ring signature scheme introduced in RingCT halves the size of the resulting signature compared to that in *Cryptonote*. For the purposes of this report, we will be using the algorithm described in RingCT.

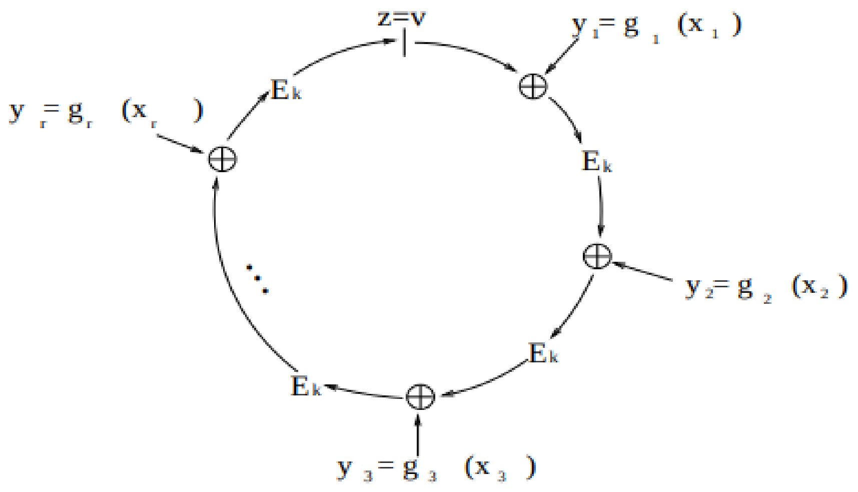


Figure 1.2: Diagram depicting ring signature algorithm [5]

This paragraph will attempt to explain the scheme at a high level shown in figure 1.2, Given a ring of public keys $\{y_1, ...y_r\}$, where y_r is the key owned by us, and a message m, start at the top of the ring by selecting a random number, v. Proceed to the next step, clockwise, by performing some sort of combinatorial logic on the public key y_1 and v. This gives as a parameter we will denote as L_1. Then, perform the obfuscation E_k on L_1 and m. Call this result c_1. Proceed around the ring, performing these operations on each successive point. Before reaching y_r, we will now have $(v, L_1, ..., L_{r-1}, c_1, ..., c_r)$. To complete the ring, we need to calculate L_r. Note that this L_r value is going *into* the initial point where we started, v. To close the ring, we then use our private key k_r to force the final value of the ring to equal our glue value, v.

This scheme returns a signature containing $(m, y_1, ..., y_r, L_1, ..., L_r, c_1, ..., c_r)$. We can then verify the signature by performing the same calculations used in signing, except in the last step, we simply check that the final value calculated is in fact equal to the value we started with.

In this scheme, we placed our secret key at index r. In reality, we wish to place our secret key at a random index. Signing can be performed in the same way as well as verification, as it does not matter which index in the ring we start from.

The original ring signature scheme consists of three functions, defined henceforth. All operations are performed on a chosen elliptic curve, E/F_p with generator G. Define our ring size, or the number of public keys in the ring, as n. Additionally, define a cryptographic hash function h.

Gen: Create a public-private key pair by selecting a random number k in F_p and calculating $P = kG$. Now, k is our private key to be used to generate the signature and P is our public key to be included in the signature. Additionally, obtain $n - 1$ other public keys, not belonging to us, to be included in the ring. Place our key in a secret index j. Output a set of public keys, $\{P_0, ..., P_{n-1}\}$ and a message, m.

Sign: Select a random *glue value*, v, an integer mod p. Also select random s_i, $i \epsilon [0, n-1]$, $i \neq j$. Start at our secret index j, compute:

$$L_j = vG$$
$$c_{j+1} = h(m, L_j)$$

Then, working successively in j mod n:

$$L_{j+1} = s_{j+1}G + c_{j+1}P_{j+1}$$
$$c_{j+2} = h(m, L_{j+1})$$
$$...$$
$$L_{j-1} = s_{j-1}G + c_{j-1}P_{j-1}$$
$$c_j = h(m, L_{j-1})$$

Now, we need to *force* $L_j = v * G$ to look like the other equations for L_i. To do this, we calculate a s_j satisfying the following:

$$s_j = v - c_j k_j \mod p$$

where k_j is our known secret key and p is the curve order. This is derived from the fact that $P = xG$:

$$L_j = vG = s_j G + c_j k_j G = s_j G + c_j P_j$$

We are able to cancel out the G in $vG = s_j G + c_j k_j G$, resulting in $v = s_j + c_j k_j$. Remember, all these operations are $\mod p$.

Output signature $\sigma = (m, P_0, ...P_{n-1}, c_0, s_0, ...s_{n-1})$

Verify: To verify signature σ, compute L'_i and c'_i for all i. Check that $c_0 = c'_0$ and check that $c'_{i+1} = h(m, L'_i)$ for all i. In particular, successively compute:

$$\begin{aligned} L'_0 &= s_0 G + c_0 P_0 \\ c'_1 &= h(m, L'_0) \\ &... \\ L'_{n-1} &= s_{n-1} G + c'_{n-1} P_{n-1} \\ c'_0 &= h(m, L'_{n-1}) \end{aligned}$$

Now, check that c'_0 we have computed is equal to c_0, the value included in the signature. If it is, return true; otherwise, return false.

The above scheme is sometimes referred to as *unique ring signatures.* This is because it is impossible to link two signatures; in other words, we cannot tell if two signatures were signed by the same secret key or not.

The RingCT protocol contains two additional features. Firstly, we introduce *key images*. A key image is an obfuscated *image* of a public key. Secondly, we introduce *linkability*. Linkability is a property allowing two or more different ring signatures to be linked to the same signer, without revealing who this signer is.

Definition 1.10: Let $H_p : x \to P$ be a hash function that returns a point on an elliptic curve E/F_p. H_p is often defined practically as $H_p(x) = h(x)G$, where h is our usual cryptographic hash function that returns a scalar.

We now modify the above protocol to include key images as well as a new function, *Link*.

Gen: Create a public-private key pair by selecting a random number k in F_p and calculating $P = kG$. k is our private key and P is our public key. Additionally, obtain $n - 1$ other public keys, not belonging to us, to be included in the ring.

Place our key in a secret index j.
Create our key image, $I = xH_p(P)$. This key image cannot be linked to P directly as it includes our private key x, essentially hiding $H_p(P)$. However, we cannot *fake* a key image by selecting some random parameters, as this will cause our signature to be malformed. We will see this in the final step of $Sign$.
Output a set of public keys, $\{P_0, ..., P_{n-1}\}$, our key image I, and a message, m.

Sign: Select a random *glue value*, v, an integer mod p. Also select random s_i, $i \in [0, n-1]$, $i \neq j$. Start at our secret index j, compute:

$$L_j = vG$$
$$R_j = vH_p(P_j)$$
$$c_{j+1} = h(m, L_j)$$

Note that we have added an R parameter. Then, working successively in j mod n:

$$L_{j+1} = s_{j+1}G + c_{j+1}P_{j+1}$$
$$R_{j+1} = s_{j+1}H_p(P_{j+1}) + c_{j+1}I$$
$$c_{j+2} = h(m, L_{j+1}, R_{j+1})$$
$$...$$
$$L_{j-1} = s_{j-1}G + c_{j-1}P_{j-1}$$
$$R_{j-1} = s_{j-1}H_p(P_{j-1}) + c_{j-1}I$$
$$c_j = h(m, L_{j-1}, R_{j-1})$$

Note that our commitment c_i now includes R_i and the calculation of R_i includes our key image I. We calculate a s_j satisfying the following:

$$s_j = v - c_j k_j \mod p$$

where k_j is our known secret key and p is the curve order. This is derived from the fact that $P = xG$:

$$L_j = vG = s_jG + c_jk_jG = s_jG + c_jP_j$$

We are able to cancel out the G in $vG = s_jG + c_jk_jG$, resulting in $v = s_j + c_jk_j$.

In addition to the above, the R values *also* result in the same equation for s_j. Using our equation for key images, $I = xH_p(P)$, we obtain:

$$R_j = vH_p(P_j) = s_jH_p(P_j) + c_jI$$
$$vH_p(P_j) = s_jH_p(P_j) + c_jI$$

$$vH_p(P_j) = s_jH_p(P_j) + c_jkH_p(P_j)$$
$$v = s_j + c_jk$$

This is the same equation derived from the L values. Note that if our key image was malformed, this step would not hold, thus we would not be able to generate a valid signature.

Output signature $\sigma = (m, P_0, ...P_{n-1}, c_0, s_0, ...s_{n-1})$

Verify: To verify signature σ, compute L'_i, R'_i and c'_i for all i. Check that $c_0 = c'_0$ and check that $c'_{i+1} = h(m, L'_i, R'_i)$ for all i. In particular, successively compute:

$$L'_0 = s_0G + c_0P_0$$
$$R'_0 = s_0H_p(P_0) + c_0I$$
$$c'_1 = h(m, L'_0, R'_0)$$
$$...$$
$$L'_{n-1} = s_{n-1}G + c'_{n-1}P_{n-1}$$
$$R'_{n-1} = s_{n-1}H_p(P_{n-1}) + c_{n-1}I$$
$$c'_0 = h(m, L'_{n-1}, R'_{n-1})$$

Now, check that c'_0 we have computed is equal to c_0, the value included in the signature. If it is, return true; otherwise, return false.

Link: Given two signatures σ_a and σ_b, return true if both contain the same I; otherwise, return false.

This completes our definition of linkable ring signatures.

1.8 Zero-knowledge protocols

possible section.

2. Problem Formulation

2.1 Motivation

The Ethereum network currently provides no privacy. All transactions made are visible to the public. Once an address is revealed to belong to a certain person, all the transactions they have made can be linked to them. However, this causes problems. For example, someone may be getting paid in ether and they don't want their salary to be public; someone may wish to deploy a smart contract to the network, but the contents of the contract may cause legal trouble; someone may wish to vote in an application on the network but does not wish to make their vote public. There are other blockchain-based networks that provide transactional privacy, such as Monero; however, they do not provide the computational capacity that Ethereum does.

A mixer is a third-party application which attempts to solve this problem. Users submit a deposit in ether and a withdraw address. The mixer then combines all the funds in a pool and sends the ether to the withdraw address, obfuscating the sender. However, a major problem with this approach is centralization. If the third-party wishes to no longer operate the mixer, they can exit with all the funds. A centralized mixer takes away the value of a decentralized system. A decentralized mixer in the form of a smart contract would solve this problem.

2.2 Formulation

To implement a decentralized mixer, ring signatures are useful. Given a ring signature, one cannot determine which member of the ring generated the signature; it can only be proven that one member of the ring generated the signature. To generate a ring signature, one needs a message, their public and private key pair, and a list of other public keys that will be included in the ring. Ethereum uses the secp256k1 elliptic curve, which has a key size of 256 bits or 32 bytes. The maximum ring size that has currently been stored on the network is five.

> The current block gas limit is approximately 6.7 million units. A ring signature, according to the RingCT protocol, is of the form $\sigma = (m, P_0, ...P_{n-1}, c_0, s_0, ...s_{n-1})$, Each public key is 64 bytes, as it consists of two 32-byte numbers ; each S parameter is 32 bytes; c_0 is 32 bytes and m is of variable length. For this purpose, let's say m is actually the *sha3* hash of a message, and is also 32 bytes. Thus, the size of a ring signature is $64n + 32n + 32 + 32 = 32(3n + 2)$

This is not sufficient for anonymity; for plausible deniability, a ring size of nine or more is needed.

The cost of storing a ring signature on the network is extremely high. Instead, we can reduce costs by moving some of the computation off-chain. By creating a pre-compiled contract which performs signing with a ring and verification of a ring-signed message, we will be able to use ring signatures on the Ethereum network. In this thesis, I would like to explore the addition of ring signatures to the EVM.

To allow for efficient computation of ring signatures on the Ethereum network, the following need to be implemented:

1. An algorithm for signing (ring-sign) and verification (ring-verify) of ring signatures using elliptic curve cryptography. The Cryptonote whitepaper proposes a one-time ring signature algorithm using ECC. [4]
2. An implementation of this algorithm, preferably in Go, as the official Ethereum client (go-ethereum) is written in Go. Additionally, it can be

implemented in Rust (for parity-ethereum), if time permits.
3. Integration of the algorithm with an Ethereum client (go-ethereum or parity-ethereum) as a pre-compiled contract.
4. A smart contract that acts as a mixer using the ring-sign and ring-verify pre-compiled contracts.
   Following these steps, a user will be able to deposit to the mixer contract and transfer their ether with the ability to obfuscate their address, adding privacy to Ethereum.

3. Literature Review

Ethereum-based:

- Mobius
- Miximus
  other:
- Monero
- zcash

4. Implementation

- explain how ring signatures are used
- how mixer will work
- expand on what I've done so far

4.x Obstacles

what issues have I had so far?

- how to send a transaction from a fresh account with no ether, without destroying privacy

5. Results and discussion

discuss:

- gas costs / $$ cost
- usability / steps needed for user

6. Conclusion

- next steps

References

[1] Ethereum Yellow Paper. https://ethereum.github.io/yellowpaper/paper.pdf
[2] go-ethereum source code. https://github.com/ethereum/go-ethereum/
[3] A Graduate Course in Applied Cryptography. https://crypto.stanford.edu/~dabo/cryptobook/BonehShoup_0_4.pdf
[4] secp256k1 curve. https://en.bitcoin.it/wiki/Secp256k1
[5] How to leak a secret. https://people.csail.mit.edu/rivest/pubs/RST01.pdf
[6] Monero ring size statistics. https://moneroblocks.info/stats/ring-size
[7] Cryptonote whitepaper. https://cryptonote.org/whitepaper.pdf

add later
[3] Implementation of a ring signature mixer in a smart contract. https://ropsten.etherscan.io/address/0x5e10d764314040b04ac7d96610b9851c8bc02815#code