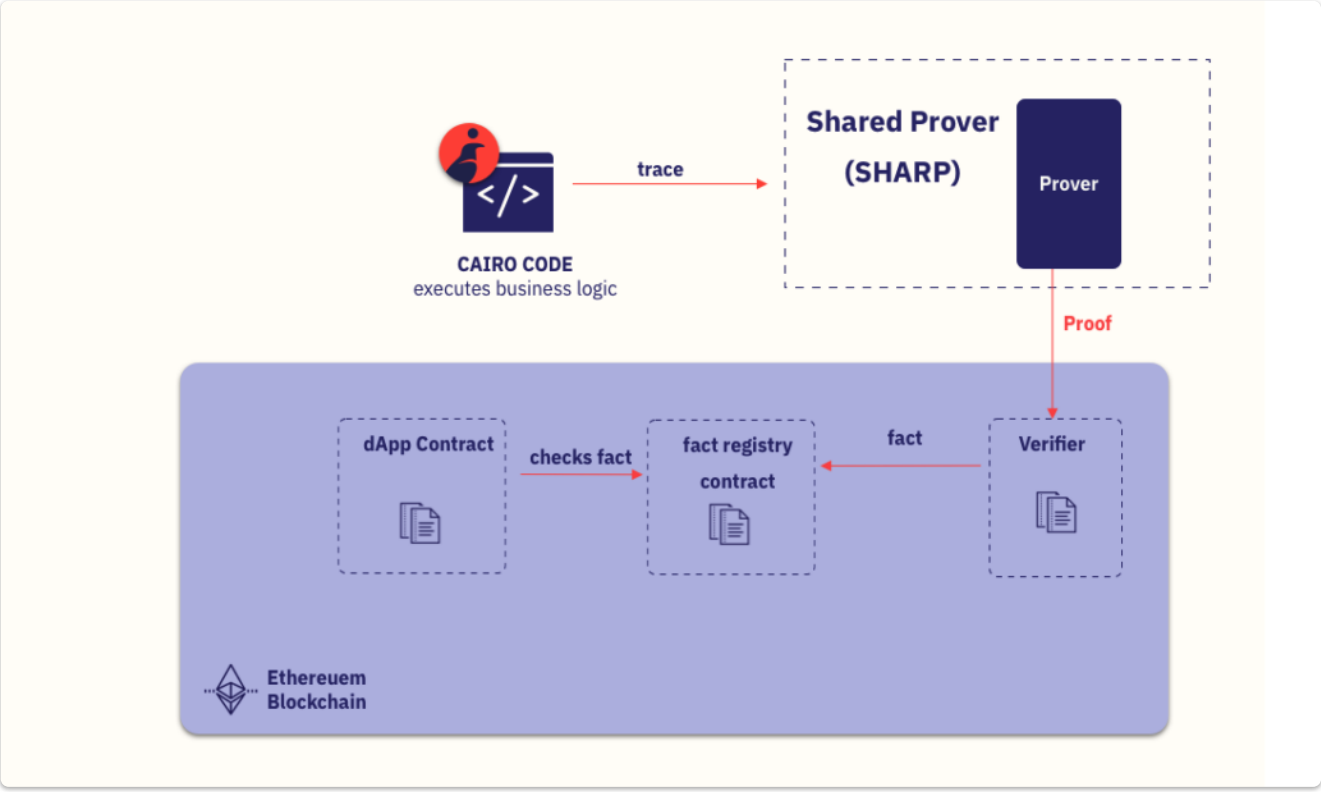
Lesson 15 - Cryptography Alternatives

Useful Videos / Podcasts

- ZK Whiteboard sessions, this week has a session from Mary Maller about lookup tables
- Stark Struck Podcast

Using SHARP the shared prover

See documentation



If you want to send to mainnet you can use the CLI
see installation instructions here

```
cairo-sharp submit --source sharp_test.cairo \
    --program_input input.json
```

Voting Systems

A Problem with rewarding users in voting systems

Imagine a system where users can vote with tokens (which they retain) and are rewarded for the number of votes they receive in a period.

Suppose that some wealthy user acquires some quantity N of tokens, and as a result each of the user's k votes gives the recipient a reward of $N \cdot q$ (q here probably being a very small number, eg. think $q = 0.000001$).
The user simply upvotes their own sockpuppet accounts, giving themselves the reward of $N \cdot k \cdot q$. Then each user has an "interest rate" of $k \cdot q$ per period.

See collusion article and Governance by Vitalik

Bribery in voting Systems

Suppose Alice can vote for a project to receive a grant.
If Charlie has a candidate project he may want to bribe Alice to vote for his project, he could do this via a side channel, and it would be unknown to the voting system.

A partial way around this is to encrypt the votes, so that if Alice's vote is seen in the system, we cannot tell which project she voted for.
To do this Alice could use some key, however if the encrypted vote is public, Alice could send the details of how she voted to Charlie, who could verify it , and Alice could claim her bribe.
A further refinement is then to allow Alice to vote multiple times, revoking the previous key she used, effectively invalidating the previous vote.
In this case Charlie loses the confidence that he has in the information that Alice sends him, as he knows she could have accepted his bribe, then later voted for someone else (and maybe get a bribe from them etc.)
MACI uses this approach, plus ZKPs to create an infrastructure that mitigates the effect of collusion.

Minimal Anti-Collusion Infrastructure

See Repo
Discussion
Zero Knowledge Podcast

The process of implementing this in a smart contract
From Introduction

Whitelisted voters named Alice, Bob, and Charlie register to vote by sending their public key to a smart contract. Additionally, there is a central coordinator Dave, whose public key is known to all.

When Alice casts her vote, she signs her vote with her private key, encrypts her signature with Dave's public key, and submits the result to the smart contract.

Each voter may change her keypair at any time. To do this, she creates and signs a key-change command, encrypts it, and sends it to the smart contract. This makes it impossible for a briber to ever be sure that their bribe has any effect on the bribee's vote.

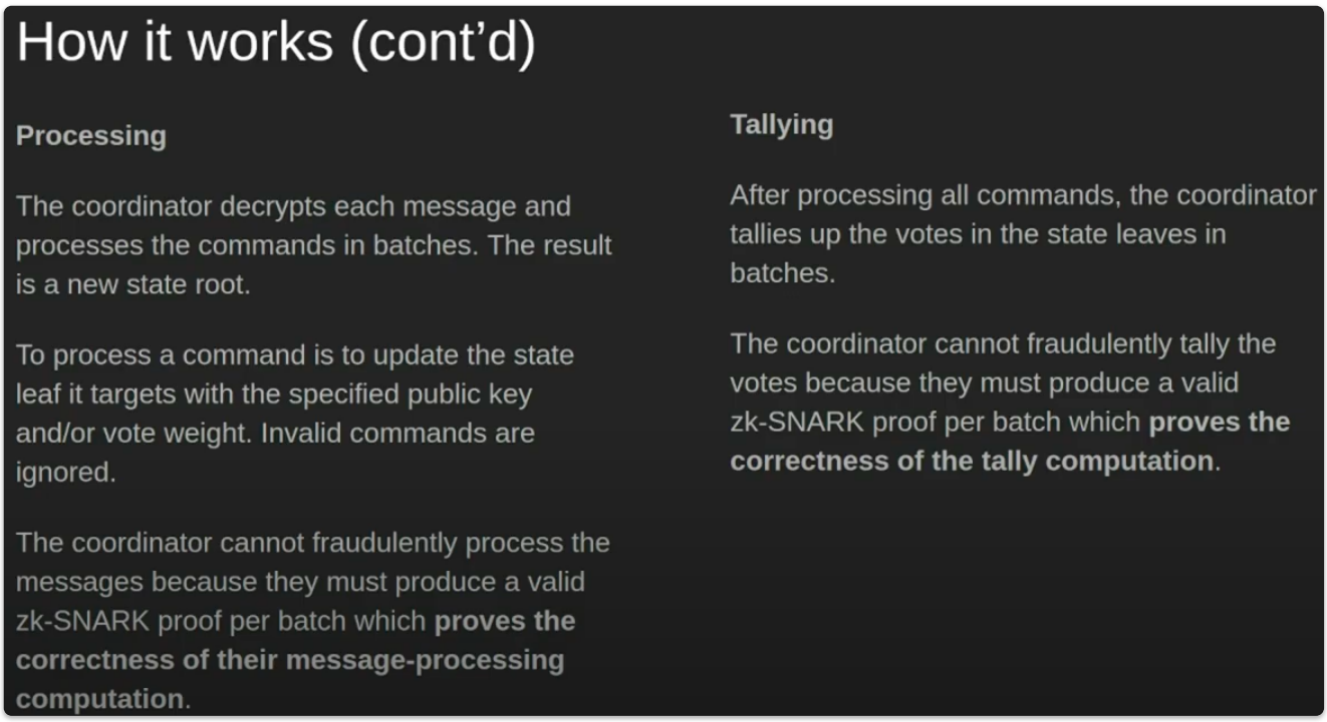
If Bob, for instance, bribes Alice to vote a certain way, she can simply use the first public key she had registered — which is now void — to cast a vote. Since said vote is encrypted, as was the key-changing message which Alice had previously sent to Dave, Bob has no way to tell if Alice had indeed voted the way he wanted her to.

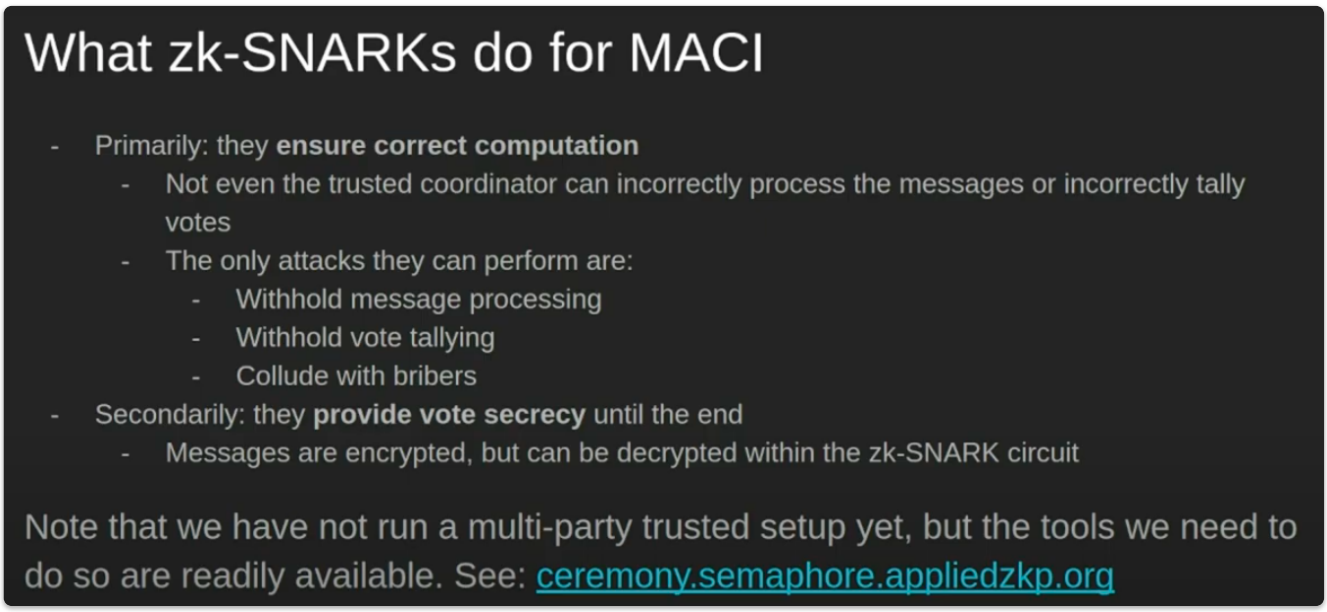
Even if Alice reveals the cleartext of her vote to Bob, she just needs to not show him the updated key command that she previously used to invalidate that key. In short, as long as she had submitted a single encrypted command before her vote, there is no way to tell if said vote is valid or not.

ZKPs used in MACI

From the MACI documentation :

There are two zk-SNARK circuits in MACI: one which allows the coordinator to prove the correctness of each state root transition, and the other which proves that they have correctly tallied all the votes.





Quadratic Voting

Quadratic voting works by allowing users to "pay" for additional votes on a given matter to express their support for given issues more strongly, resulting in voting outcomes that are aligned with the highest willingness to pay outcome, rather than just the outcome preferred by the majority regardless of the intensity of individual preferences

A general drawback to these ideas occurs where the number of votes cast is small.

Blind signatures

The notion of blind signatures was introduced by Chaum in 1982
There are two properties which any blind signature scheme must satisfy:
Blindness and Untraceability.

- Blindness means the content of a message should be blind to the signer.
- Untraceability is satisfied if, whenever a blind signature is revealed to the public, the signer will be unable to know who the owner of the signature is.

An often-used analogy to the cryptographic blind signature is the physical act of a voter enclosing a completed anonymous ballot in a special carbon paper lined envelope that has the voter's credentials pre-printed on the outside.
An official verifies the credentials and signs the envelope, thereby transferring his signature to the ballot inside via the carbon paper.
Once signed, the package is given back to the voter, who transfers the now signed ballot to a new unmarked normal envelope.
Thus, the signer does not view the message content, but a third party can later verify the signature and know that the signature is valid within the limitations of the underlying signature scheme.

Blind signature is a kind of digital signature in which the message is blinded before it is signed. Therefore, the signer will not learn the message content. Then the signed message will be unblinded.
At this moment, it is similar to a normal digital signature, and it can be publicly checked against the original message.
Blind signature can be implemented using a number of public-key encryption schemes.

Here, we only introduce the simplest one, which is based on RSA encryption.
The signer has a public key (n, e) and a secret key d.
Suppose a party A wants to have a message m signed using the blind signature.
She should execute the protocol with the signer S as follows:
A first randomly chooses a value k, which satisfies $0 \leq k \leq n-1$ and $gcd(n, k) = 1$.
For the message m, A computes $m* = mk^e$ (mod n) and sends $m*$ to S.
When S receives $m*$,
S computes $s* = (m*)d$ (mod n) and sends $s*$ back to A.
A computes $s = s*/k$ (mod n).
Now s is S's signature on the message m.

Confidential transactions

This work was originally proposed by Adam Back on Bitcointalk in his 2013 thread “bitcoins with homomorphic value”.

"To build CT I had to implement several new cryptosystems which work in concert, and invented a generalization of ring signatures and several novel optimizations to make the result reasonably efficient."

The basic tool that CT is based on is a Pedersen commitment.

A commitment scheme lets you keep a piece of data secret but commit to it so that you cannot change it later. A simple commitment scheme can be constructed using a cryptographic hash:

```
commitment = SHA256( blinding_factor || data )
```

If you tell someone only the commitment then they cannot determine what data you are committing to (given certain assumptions about the properties of the hash), but you can later reveal both the data and the blinding factor and they can run the hash and verify that the data you committed to matches.

The blinding factor is present because without one, someone could try guessing at the data; if your data is small and simple, it might be easy to just guess it and compare the guess to the commitment.

A Pedersen commitment works like the above but with an additional property: commitments can be added, and the sum of a set of commitments is the same as a commitment to the sum of the data (with a blinding key set as the sum of the blinding keys):

```
C(BF1, data1) + C(BF2, data2) == C(BF1 + BF2, data1 + data2)
C(BF1, data1) - C(BF1, data1) == 0
```

In other words, the commitment preserves addition and the commutative property applies.

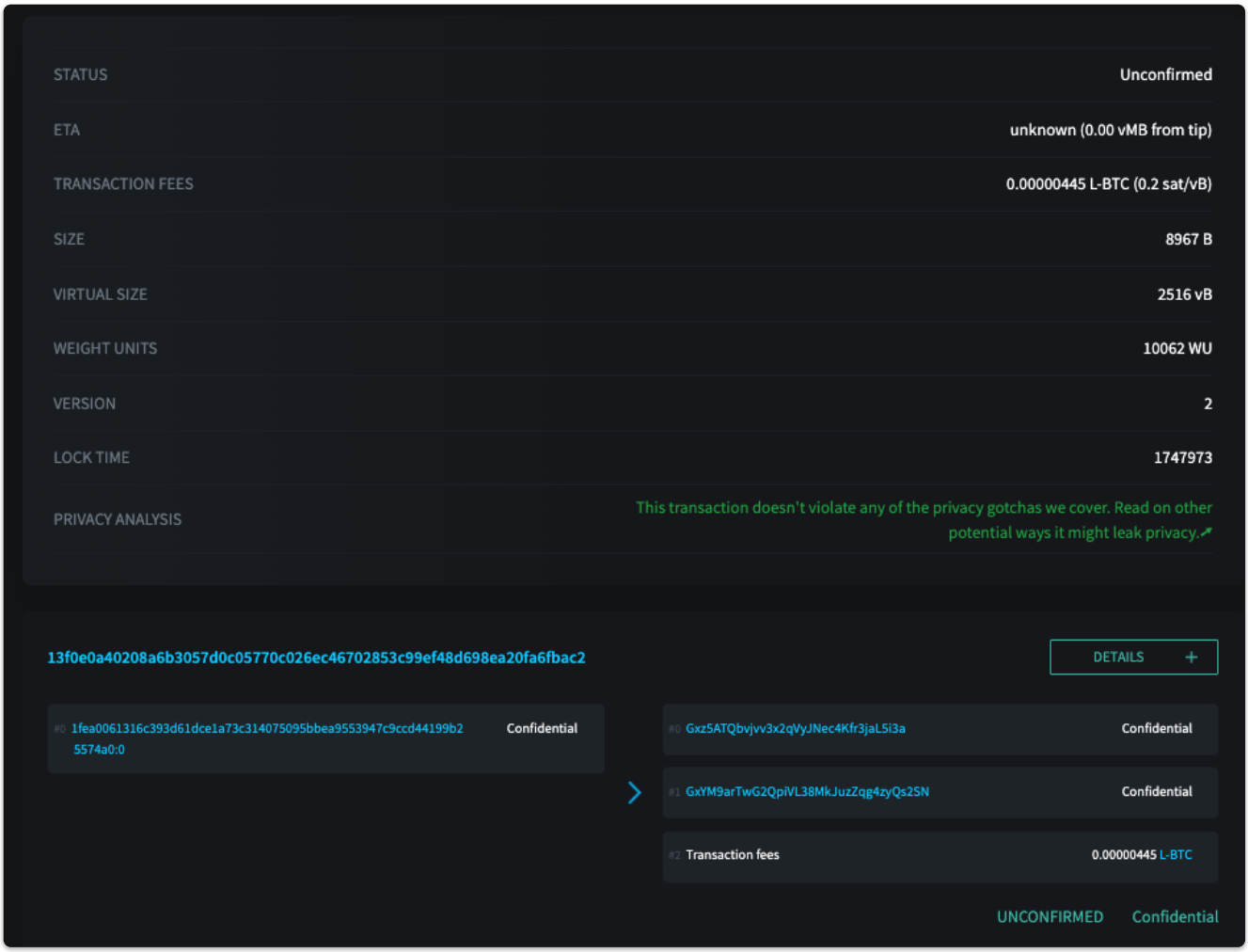
If $data_n = 1, 1, 2$ and $BF_n = 5, 10, 15$ then:

```
C(BF1, data1) + C(BF2, data2) - C(BF3, data3) == 0
```

For a workable cryptocurrency systems, range proofs would also be needed

See the Liquid Network which uses confidential transactions by default.

You can see details on their block explorer


STATUS
Unconfirmed
ETA
unknown (0.00 vMB from tip)
TRANSACTION FEES
0.00000445 L-BTC (0.2 sat/vB)
SIZE
8967 B
VIRTUAL SIZE
2516 vB
WEIGHT UNITS
10062 WU
VERSION
2
LOCK TIME
1747973
PRIVACY ANALYSIS
This transaction doesn't violate any of the privacy gotchas we cover. Read on other potential ways it might leak privacy.
13f0e0a40208a6b3057d0c05770c026ec46702853c99ef48d698ea20fa6fbac2
DETAILS
#0 1fea0061316c393d61dce1a73c314075095bbea9553947c9ccd44199b25574a0:0
Confidential
#0 Gxz5ATQbvjvv3x2qVyJNec4Kfr3jaL5i3a
Confidential
#1 GxYM9arTwG2QpiVL38MkJuzZqg4zyQs2SN
Confidential
#2 Transaction fees
0.00000445 L-BTC
UNCONFIRMED
Confidential

Threshold Cryptosystems / Secret Sharing / Multiparty Computation

The goal is to divide secret S into n pieces of data $S_i . . S_n$ in such a way that:

Knowledge of any k or more S_i pieces makes S easy to compute. That is, the complete secret S can be reconstructed from any combination of k pieces of data.
Knowledge of any $k-1$ or fewer S_i pieces leaves S completely undetermined, in the sense that the possible values for S seem as likely as with knowledge of 0 pieces.

A naive splitting of a key would just make a brute force attack easier.

Shamir Secret Sharing

Based on the fact the k points are required to define a polynomial of degree $k-1$
With our points being elements in a finite field $\mathbb{F}$ of size P where $0 < klen < P; S < P$ and P is a prime number.

Choose at random $k-1$ positive integers $a_1..a_{k-1}$ with $a_i < P$ and let $a_0 = S$

The person splitting the secret builds a polynomial where the secret is the constant term a_0

$f(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{k-1}x^{k-1}$

Let us construct any n points out of it, for instance set $i = 1..n$ to retrieve $(i, f(i))$.
Every participant is given a point (a non-zero integer input to the polynomial, and the corresponding integer output) along with the prime which defines the finite field to use.
Given any subset of k of these pairs, we can find the coefficients of the polynomial using interpolation. The secret is the constant term a_0

Properties of Shamir's (k, n) threshold scheme are:

- Secure: Information theoretic security.
- Minimal: The size of each piece does not exceed the size of the original data.
- Extensible: When k is kept fixed, D_i pieces can be dynamically added or deleted without affecting the other pieces.
- Dynamic: Security can be easily enhanced without changing the secret, but by changing the polynomial occasionally (keeping the same free term) and constructing new shares to the participants.
- Flexible: In organizations where hierarchy is important, we can supply each participant different number of pieces according to their importance inside the organization. For instance, the president can unlock the safe alone, whereas 3 subordinates are required together to unlock it.

Additive Secret Sharing

Given a secret $s \in F$, the dealer D selects $n-1$ random integers
$R = r_1, r_2, r_{n-1}$ uniformly from F.

D then computes

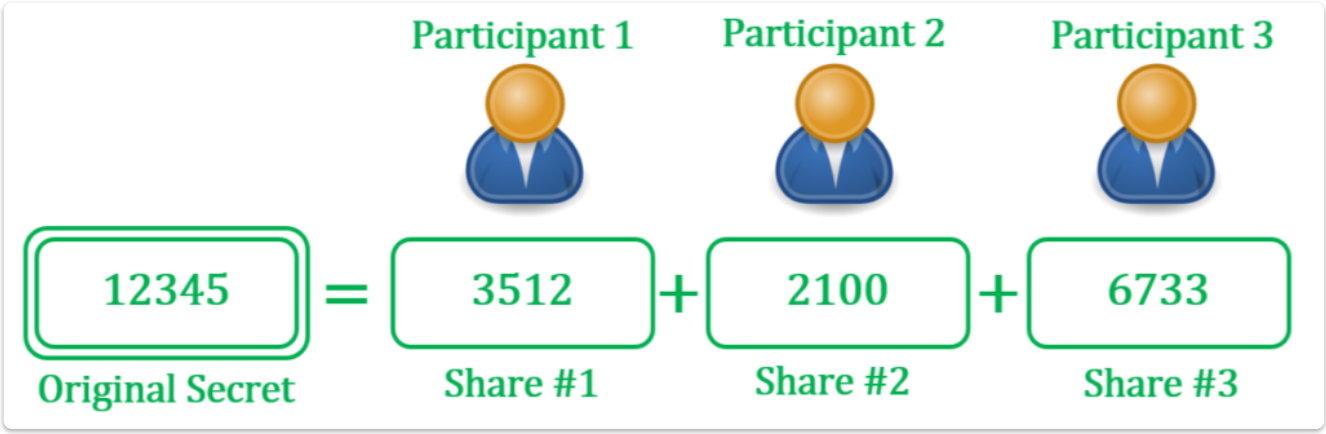
$s_n = s - \sum_{i=1}^{n-1} r_i \ mod \ F$

D sends each player P_i $1 \leq i \leq n-1$ the share $s_i = r_i$, and the share s_n is sent to P_n.

The reconstruction of secret $s \in F$ is trivial; simply add all of the shares together:

$s = \sum_{i=1}^{n} s_i \ mod \ F$

The above additive secret sharing scheme requires all participants to contribute their shares in order to reconstruct the secret.
If one or more of the participants are missing, no information about the original secret can be recovered; such a scheme is known as a perfect secret sharing scheme.



Multiparty computation overview

A key point to understand is that MPC is not a single protocol but rather a growing class of solutions that differ with respect to properties and performance.
However, common for most MPC systems are the three basic roles:

- The Input Parties delivering sensitive data to the confidential computation.
- The Result Parties receiving results or partial results from the confidential computation.
- The Computing Parties jointly computing the confidential computation

In an multi party computation,
a given number of participants, $p_1, p_2, \ldots p_n$,
each have private data, respectively $d_1, d_2, \ldots d_n$.

Participants want to compute the value of a public function on that private data:
$f(d_1, d_2 \ldots d_n)$ while keeping their own inputs secret.

Most MPC protocols make use of a secret sharing scheme such as Shamir Secret Sharing. The function $f(d_1, d_2 \ldots d_n)$ is used to define an arithmetic circuit over a finite field which consists of addition and multiplication gates.

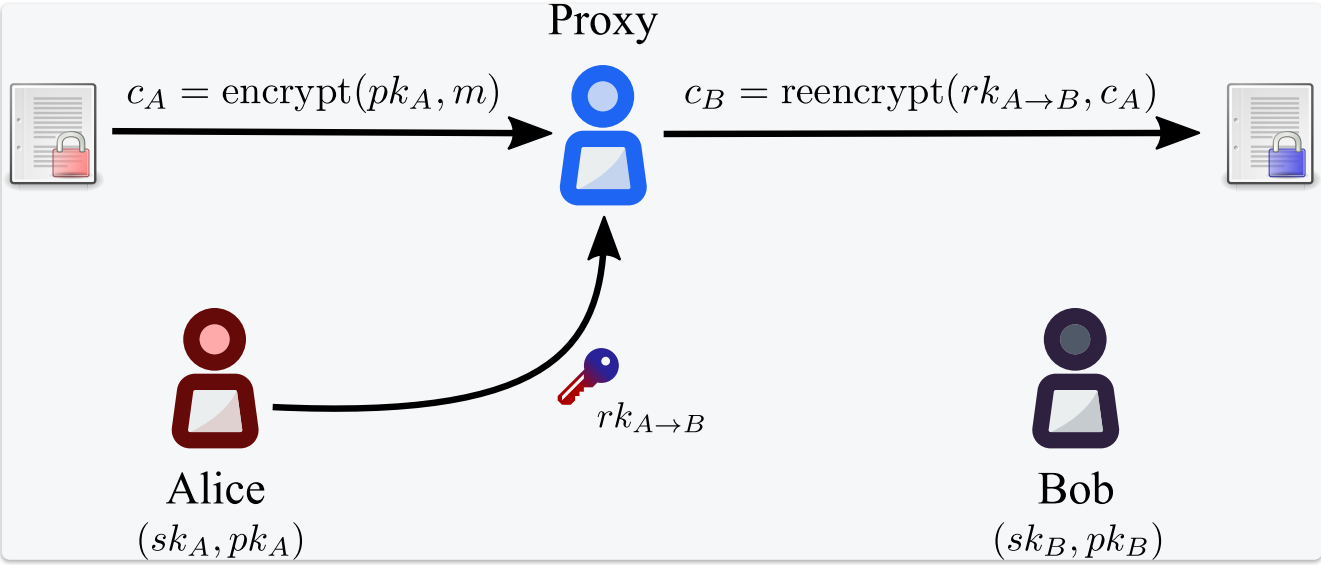
In the secret sharing based methods, the parties do not play special roles. Instead, the data associated with each wire is shared amongst the parties, and a protocol is then used to evaluate each gate.

Two types of secret sharing schemes are commonly used;

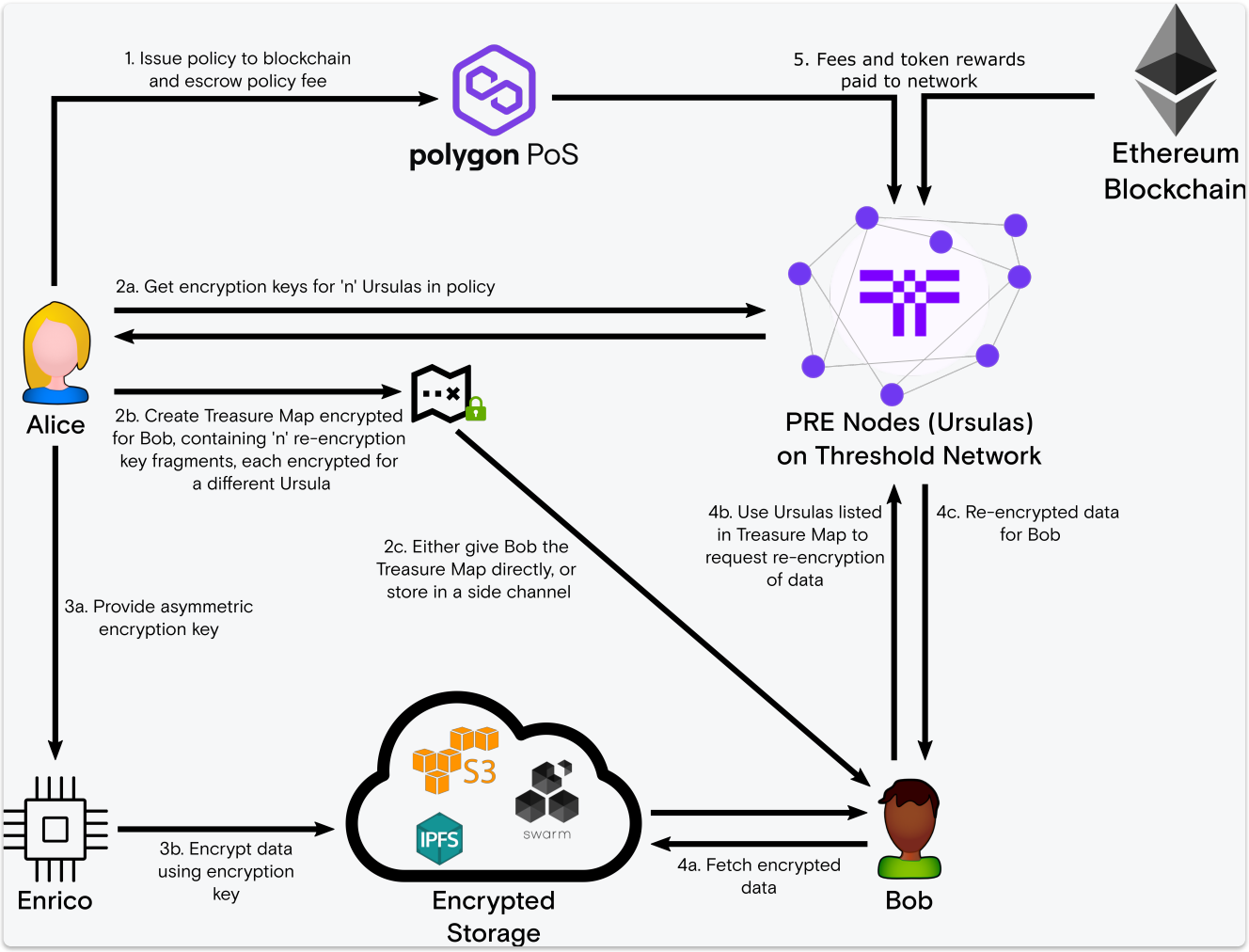
1. Shamir secret sharing
2. Additive secret sharing
   In both cases the shares are random elements of a finite field that add up to the secret in the field; intuitively, security is achieved because any non-qualifying set of shares looks randomly distributed.

Proxy Re encryption

See Nucypher documentation



THRESHOLD PRE



1. Alice, the data owner, grants access to her encrypted data to anyone she wants by creating a policy and uploading it to the NuCypher network.
2. A group of Ursulas, which are nodes on the NuCypher network, receive information about the policy, called a PolicyArrangement that include a re-encryption key share.

The Ursulas stand ready to re-encrypt data in exchange for payment in fees and token rewards. Thanks to the use of proxy re-encryption, Ursulas and the storage layer never have access to Alice's plaintext data.

3. Each policy created by Alice has an associated encryption key, which can be used by any entity (Enrico) to encrypt data on Alice's behalf. This entity could be an IoT device in her car, a collaborator assigned the task of writing data to her policy, or even a third-party creating data that belongs to her – for example, a lab analyzing medical tests. The resulting encrypted data can be uploaded to IPFS, Swarm, S3, or any other storage layer.
4. Bob, a data recipient, obtains the encrypted data from the storage layer and sends an access request to the NuCypher network. If the policy is satisfied, the data is re-encrypted to his public key and he can decrypt it with his private key.
5. Ursulas earn fees and token rewards for performing re-encryption operations.

Sigma Protocols

See article

A Sigma protocol follows these 3 steps:

1. **Commitment**: The prover generates a random number, creates a commitment to that randomness and sends the commitment to the verifier.
2. **Challenge**: After getting the commitment, the verifier generates a random number as a challenge and sends it to the prover. It is important that the verifier does not send the challenge before getting the commitment or else the prover can cheat.
3. **Response**: The prover takes the challenge and creates a response using the random number chosen in step 1, the challenge and the witness. The prover will then send the response to the verifier who will do some computation and will or will not be convinced of the knowledge of the witness.

Example

To prove knowledge of x in $g^x = y$ without revealing x,

1. The prover generates a random number r, creates a commitment $t = g^r$ and sends t to the verifier.
2. The verifier stores t, generates a random challenge c and sends it to the prover.
3. The prover on receiving the challenge creates a response $s = r + x.c$. The prover sends this response to the verifier.
4. The verifier can then check that g^s equals $y^c.t$.

The process can be made non interactive with the Fiat Shamir heuristic as follows

1. The prover generates a random number r and creates a commitment $t = g^r$.
   The prover hashes g, t and y to get challenge c : $c = Hash(g, y, t)$.
2. The prover creates a response to the challenge as $s = r + c.x$. The prover sends tuple (t, s) to the verifier.

The verifier now generates the same challenge c as $Hash(g, y, t)$
and again checks if g^s equals $y^c.t$.

Sigma protocols are efficient at proving algebraic statements such as the discrete log problem, but for more general statements we use SNARKS / STARKS

Lurk Language

See blog
Lurk is a Turing-complete language produced by Filecoin to allow creation
of recursive zk-SNARKs.
It is a statically scoped dialect of Lisp
Features

- Verifiable computation
- Succinct proofs
- Zero knowledge
- Turing-completeness
- Arbitrary traversal of content-addressable data
- Higher-order functions (e.g. functions as public inputs to computations, with proof)
- Content-addressable data for natural integration with IPFS and IPLD

Circom

See repo and documentation
Circom is DSL that can be used to create circuits, which can then be used in SNARK proofs.

You can define contraints in this way

```
pragma circom 2.0.0;

/*This circuit template checks that c is the multiplication of a and b.*/

template Multiplier2 () {

   // Declaration of signals.
   signal input a;
   signal input b;
   signal output c;

   // Constraints.
   c <== a * b;
}
```

You then compile this which gives you

1. An R1CS file
2. A C++ file needed to compute the witness
   This produced a `witness.wtns` file.

You can then use the `snarkjs` tool to generate and validate a proof for our input.

- Firstly you need to do the trusted setup
- You can then create proving and verification keys
- With the proving key you can generate a proof
  This has two files, the proof, and the public inputs
- With the verification key you can create a smart contract verifier.

One of the advantages of Circom is that it has templates for common circuits to allow you to quickly build more complex circuits.