Emotiq Yellowpaper (v1.0)

The team
info@emotiq.ch

ABSTRACT

We are nearing the future where scalability and high transaction throughput will be the baseline for blockchain technology. We are are not there yet, however, and the race for an industrial-strength blockchain foundation is still on. We present Emotiq, a decentralized public blockchain with strong consistency, fast transaction throughput, unlimited horizontal scalability, strong cryptography, transaction privacy, and natural language smart contracts.

1 Introduction

Emotiq is a public decentralized blockchain so anyone can join the network, become a validator, and earn rewards for maintaining it. Emotiq uses Proof-of-Stake (PoS) consensus, which is based on pBFT but adds strong consistency and enables all validators to agree on the validity of blocks without wasting computational power resolving inconsistencies. As a result, clients do not need to wait more than a few seconds to be certain that a submitted transaction is committed; as soon as it appears in the blockchain, the transaction can be considered confirmed.

Emotiq provides unlimited horizontal scaling and throughput of thousands of transactions per second through sharding, as well as a cross-shard commit protocol.

Emotiq uses Boneh-Lynn-Shacham (BLS, A.2) pairing-based crypto (PBC, A.1). This enables short BLS signatures on public keys, as well as hierarchical deterministic wallets with all of the advantages promoted by BIP-32 but without the security risks.

We also cloak transactions through non-interactive zero-knowledge proofs and purge old and spent transactions. The former provides increased privacy by hiding transaction amounts, and the latter drastically reduces the amount of data required to be maintained by validators.

2 Networking

The Emotiq network is composed of full nodes, called validators, and light clients, e.g., those running on mobile devices or in the browser. Each validator maintains a full copy of the blockchain and associated data structures. Validators participate in the collective signing protocol and respond to requests by light clients.

Light clients do not maintain the blockchain, but know how to talk to validators.

Emotiq maintains the core of the network by running a number of validator nodes, which both simplifies the bootstrap of new nodes and ensures continuity of the network. The addresses of the core nodes are hard-coded into each release of the Emotiq blockchain software.

Validators and light clients retain a list of addresses of the nodes they know about (peers) and add new peers to the list as they become aware of them. A new node starting will connect to one of the core nodes to fetch a list of peers. Each validator node quickly re-broadcasts received transactions to a set of its peers, after only a few light checks. This both ensures that a node cannot be DDOS-ed with transactions to validate and that it does not re-broadcast junk transactions.

To discourage bad actors, we employ a mechanism to both throttle peers and punish peers for bad transactions, etc. by blocking them from further participation in the network.

2.1 Gossip

The Emotiq blockchain uses gossip protocols to spread information without depending on fixed networks of communicating nodes. These protocols do not require every node to be reliable or always up and running and do not require every node to know about every other node. The key to a gossip protocol is that every node knows only a few other nodes (or even just one) in the network, but as long as most nodes know about at least two other nodes besides themselves, information can propagate through the network.

We use two gossip protocols: *Traditional* and *Neighborcast*. The *Traditional* gossip protocol requires every node to listen for a message and pass it on to a third (sometimes a fourth) node, but does not require every node to be aware of more than two others. *Neighborcast* is similar except it asks each node to spread the message to *all other nodes it knows about*. The two protocols are similar in that no single node knows about every other node in the system. *Neighborcast* fully exploits all the knowledge every node possesses, while *Traditional* gossip does not.

A message can be either a *Request* or a *Reply*. Some requests require no acknowledgement but others do. Replies are always sent in response to requests.

Replies can be *Direct* (i.e. sent directly to the node that originated the request) or *Indirect*, where the reply is sent back through the gossip network.

2.1.1 Light client posting a transaction

A light client creating a transaction will use *Traditional* gossip to send it to 1-2 validators who will send a reply to the light client and forward the transaction to their peers. The light client will resend the transaction to different validators if it has not received a reply within a given timeframe.

2.1.2 Forming validator groups

Validators self-organize into groups for performance reasons, e.g., for sharding or collective signing, and use gossip to do so.

Group members require a higher level of assurance that their peers are online, reliable, and fairly close to them (where close can mean either geographically nearby, or more commonly temporally close with short communication delays). They will first use traditional gossip to deduce which peers are available and which of those are close. They will then assign the closest members to be their immediate neighbors for the purpose of group communications.

Neighbors also need to be selected based on the connectivity of the network, such that each node doesn't have too many or too few neighbors; otherwise, the desirable scaling characteristics of the network can be lost. In terms of graph theory, the network needs to be an *expander* where any subset of vertices of the network has a relatively large set of neighbors. This cannot be accomplished by fully connecting the network as it would result in $O(N^2)$ messages being exchanged and a consequent loss of scalability. Group creation is kicked off by the elected leader of the current epoch. The leader uses traditional gossip to send a sound-off message to one or two validators it knows about. Each validator gossips the reply and forwards the original sound-off message to one or two peers.

Each message is timestamped at origin and keeps track of its hop count so every receiver of the reply knows how long the message took to send and how many nodes it passed through. This information is used by nodes in the group to establish their set of neighbors.

Nodes switch to neighborcast once the group is formed and every node will forward messages to all of its immediate neighbors. Every message is tagged with a unique identifier code. Nodes remember messages they have seen and ignore duplicates.

Traditional gossip and neighborcast scale communication as $O(logN)$, so they are much better for communicating within large groups.

2.2 Traditional gossip vs Neighborcast

Traditional gossip is great for lightweight, infrequent communications with few replies. Neighborcast is preferred for communications that require lots of request/reply pairs. While traditional gossip is not guaranteed to reach every node in the network, neighborcast is.

Neighborcast can scale computation as well as communication. It can be seen as a communication tree where the root node is dynamic. Intermediate nodes can coalesce the results of computations done by their subnodes, which means certain kinds of computations can be done more quickly because they're distributed.

Consider finding the maximum value for a certain key that all nodes share, for example. Each node has this key, with a different value. With neighborcast, nodes will send the request for the maximum value of the key to its immediate neighbors. Any node without subnodes will reply with its local value. Any node with subnodes will send the request to its child nodes, will collect the replies it receives, compute the maximum of those replies and its own value, and forward that single scalar value upstream.

The overall delay is the depth of the tree, which will be $O(logN)$ if the group was formed with judicious neighbor selection. The computation is cheap because the structure of the tree itself helps optimize the computation.

3 Randomness

3.1 Bias-resistant Distributed Randomness

Bias-resistant distributed randomness is a critical component of the Emotiq. We employ RandHerd[13], a large-scale distributed protocol that provides publicly-verifiable, unpredictable, and unbiased randomness against Byzantine adversaries by implementing an efficient and decentralized randomness beacon. RandHerd arranges participants into verifiably unbiased random secret-sharing groups, which repeatedly produce random output at predefined intervals. We use RandHerd to elect leaders during each block-signing round, as well as to form shard groups.

3.2 Cryptographic Sortition

Cryptographic sortition[6] is the process of randomly selecting members of a group based on stake weighting, e.g., for leader election. Random values for sortition probing are developed using verifiable random functions (VRF, B.2) to prove fairness in making selections.

4 Consensus

The Emotiq consensus protocol is based on pBFT[1] but adds strong consistency, which enables all validators to agree on the validity of blocks without wasting computational power resolving inconsistencies. Clients dont need to wait more than a few seconds to be certain that a submitted transaction is committed; as soon as it appears in the blockchain, the transaction can be considered confirmed.

4.1 Collective Signing

We adopt CoSi[12], a scalable witness cosigning protocol ensuring that every authoritative statement is validated and publicly logged by a diverse group of witnesses before any client will accept it. A statement S collectively signed by

W witnesses assures clients that S has been seen, and not immediately found erroneous, by those W observers. Even if S is compromised in a fashion not readily detectable by the witnesses, CoSi still guarantees Ss exposure to public scrutiny, forcing secrecy-minded attackers to risk that the compromise will soon be detected by one of the W witnesses.

Because clients can verify collective signatures efficiently without communication, CoSi protects clients privacy, and offers the first transparency mechanism effective against persistent man-in-the-middle attackers who control a victims Internet access, the authoritys secret key, and several witnesses secret keys. CoSi builds on existing cryptographic multisignature methods, scaling them to support thousands of witnesses via signature aggregation over efficient communication trees.

The default implementation of CoSi uses Schnorr signatures, which we replace with BLS signatures for performance reasons.A.2.1

4.1.1 CoSi Multisignatures

We use CoSi trees to provide scalable, distributed multisignature generation. A CoSi tree is an n-way tree, where each node in the tree interior is a group leader over n subnodes, each of which may be group leaders over their own subtrees. We select validators using cryptographic sortition to assign N of them to a position in a Cosi tree.

When a signature is requested from a CoSi tree, the message is propagated down the tree to all participant nodes. Each node then attempts to validate the message and decides whether or not to add its BLS (A.2) signature to the collective signature formed from the sum of all signatures of its subgroup, before passing the augmented signature back up to its parent node in the tree. Accompanying that signature is a composite public key consisting of the sum of participant node public keys, and a bitmap that represents which nodes actually signed the message.

Validation of a message requires varying computation based on the type of message. For randomness generation, it means validating all publicly verifiable quantities and producing decrypted shares. For block validation, it means verifying the public keys of all signers of the block, validating all transactions, etc.

At the top of the tree the bitmap is converted into a list of public keys for all participating nodes, which is used to verify the summed public key, and to gain a census count on how many nodes actually signed the message. That census count is checked against a pBFT[1] threshold of $2f+2$, where $f = \lfloor \frac{N-2}{3} \rfloor$ is the tolerance for Byzantine failures among the nodes, to decide whether the multisignature is acceptable.

4.1.2 CoSi on top of pBFT

A single pass through a CoSi tree is insufficient to ensure pBFT guarantees. Instead, we perform two passes.

During the first pass we send a message to be validated and signed by member nodes of the CoSi tree. If the number of singatures on the reply exceeds the pBFT threshold then we will have completed a successful *prepare* phase of pBFT.

During the second pass we send out the validated message to ensure that each node has seen its collective signature. The new collective signature attests that member nodes will remember this message for future reference. This corresponds to the *commit* phase of pBFT.

Once all member nodes have signed off on the validated message, and if the number of collected signatures remains above the pBFT threshold, the caller of this consensus round can be confident that pBFT consensus has been achieved.

4.2 Leader Election

Emotiq is a public decentralized blockchain so anyone can join the network, become a validator, and earn rewards for maintaining it. Emotiq uses Proof-of-Stake (PoS) consensus and all validators need to post a security bond of a fixed number of tokens. Requiring a security bond helps prevent against Sybil attacks, by requiring a significant financial commitment from each identity.

Any validator caught cheating by their peers loses their stake, which gets shared among all validators who point out the misbehavior by posting a *slash* transaction.

A validator elected leader of the current epoch (block height) is rewarded by collecting transaction fees from the block they form.

Leader elections are stake-weighted lotteries. The larger the validator stake (security bond), the greater the chances of winning the current round of the lottery. After serving one round, the leader relinquishes his right to be re-elected until some number of rounds later.

In order to produce a stake-weighted lottery, we could imagine dividing a circle into angular sections proportional to individual stakes, with the full circumference of the circle representing the sum of all stakes. A uniformly generated random number over a finite interval acts as a spinner in a dial readout, and wherever it points after a spin, that member is selected as the next leader. Larger stakeholders occupy a larger portion of the circumference, and so have a more likely chance of winning.

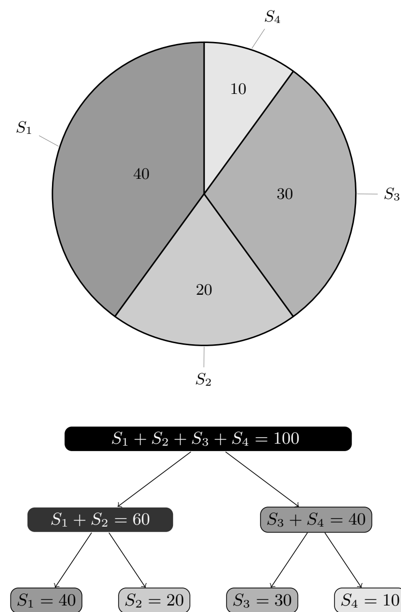

Figure 1: Imagining a spin dial as a tree. Segments are sized in proportion to the posted security bond (stake).

In practice, we form a binary tree of participants, in a consistent order, where each interior node represents the sum of all its sub-node stakes. The topmost node of the tree shows the full sum of all stakes represented in the tree, with all participants located in leafs. Branching decisions begin at the top node of the tree and descend toward leafs using the random value as a probe of the interval described by the partial sum at each node, with the division between left and right based on the relative weights of its two subtrees. Descent continues until meeting a leaf node, and declaring that node the winner of the election.

If the random probe value is converted into a fractional value of its range, and each interior node is relabeled with its division fractional value, then it is simple to choose left or right subnodes based on a single comparison between the two fractional values.

That this works can be seen by noting that every interior node of the tree serves both as denominator to nodes below, and as numerator to connections above, hence cancelling out. The final probability for any participant winning becomes the same as their escrow stake divided by the sum of all participant escrow stakes.

4.2.1 Handling leader failures

Leader elections are held on a schedule and validators recognize the signal to hold a new election.

The list of validators is publicly known and can be rebuilt by processing staking transactions starting with the genesis block. Validators keep a number of data structures in memory, including the list of other validators, and update these data structures when accepting blocks to add to the the end of the chain.

Each validator can re-run the election for themselves upon seeing the random value of the election signal, which arrives by way of authenticated randomness. All nodes should be able to agree on election outcome, but, if not, one of two things could happen.

First, if a stakeholder thinks it won the election but was mistaken, then any attempts at forming new Cosi networks by the faux leader will be silently rejected by others, and it will fail to obtain a consensus on any new requests. If that happens, the rejected leader will re-synchronize its list of validators, rerun its own election, and re-join after agreeing on the outcome.

Second, a validator may think that someone else had won the election. In that case, it will silently reject signing requests by actual election winner, and never see any requests from the presumed one. The validators will re-sync its list of other validators if this happens for two consecutive election rounds.

If the election winner is absent or comes under attack, it may not respond to its own election as leader, form the block, or broadcast the block for signing. Any validator can call for a new early election, and after $(2f+1)$ (supermajority) of such calls from different validators have been seen, a new election round will start.

4.2.2 A fault-tolerant and self-healing protocol

Immediately after a leader election, each validator should determine its role for the new epoch.

- If elected leader, then begin next block assembly.
- If the node should become a member of a RandHerd group, then start running the protocol within its group.
- If the node should become a CoSi witness node, then start a timeout timer. If that timeout ever fires, then it is time to call for a new election. Incoming messages can restart or cancel the timeout timer, though.
- Otherwise, the node is on standby for the next election cycle. Nodes on standby can ignore almost all messages, but must pay attention to election related messages.

All validators should pay attention to leader election messages:

- New Election Message - Run an election with the random value provided in the election message, and determine next role for this validator - one of leader, witness signer, RandHerd participant, or standby. Reset the counter of calls for new election.
- Call For New Election Message - Increment the count, n, of such calls seen from different validators, and if $n \geq 2f+1$, then run another election round based on the last election's random value, with a decision tree that excludes current leader. Determine new leader and next role for this validator. Reset the counter of calls for new election.

There are five different events that are relevant to a validator acting as a witness signer:

- Consensus Prepare Message - Attempt to validate the message. If valid, sign and send back to leader. Validation includes checking that the message arrived from the current leader. If not, then just ignore the message. If the message was valid, then restart the timeout timer if it was running.
- Consensus Commit Message - Check that the message arrived from the current leader. If so, sign, send back to leader, and record the message. If the message arrived from someone other than the current leader, then it either already had enough signatures to become committed. Otherwise, it failed to gain commit consensus and was dropped. Either way, we can just ignore it. Leave the timeout timer running.
- Self Timeout - Gossip a call for new elections. Do not restart timeout timer.
- New Election Message - Kill the timeout timer, then proceed as usual for a new election message.
- Call For New Election Message - Increment the count, n, of such calls seen from different validators, and if $n \geq 2f+1$, then kill the timeout timer, and proceed with an early election.

5 Transactions

UTXO was first introduced by Bitcoin and, unlike Ethereum, aggregates spent and unspent coins, available across multiple wallets, into a single balance. Not only does UTXO offer

simplicity, but also drastically increases Emotiqs scaling capability, by enabling transactions to be processed independently and in parallel.

Emotiq transactions consist, at a minimum, of the senders signature and public key, the receiver address or addresses, as well as a smart contract that governs how the transaction may be used.

Emotiq uses zero-knowledge proofs to ensure transaction amounts, among other things, are not visible in the public ledger. For example, although blockchain addresses are represented by random strings, it is currently possible for mapping software to crawl Bitcoins ledger for spent and unspent (UTXO) coins, to identify the transactions of a single private key and determine a holders total wealth in Bitcoin. While non-interactive zero-knowledge proofs do not prevent such transaction graph analyses, they prevent the tracing party from seeing the amounts being transferred.

Emotiq builds upon MimbleWimble and strong cryptography to ensure that spent transactions can be pruned and new nodes can efficiently validate the blockchain without downloading any old or spent transactions. Unlike with Proof-of-Work (PoW), where transaction pruning can lower security by lowering the amount of work needed to be redone by an attacker, the approach does present an issue for our PoS blockchain.

5.1 Transaction Privacy

For blockchain systems that don't use pairing cryptography, a simple technique for cloaking all transactions can be developed, based on showing that homomorphic sums yield a recognizable zero balance. That only works when all cryptographic computations occur in a single curve, or when parings are symmetric and both curves are the same simple Elliptic Curve group.

In our system, however, we are using asymmetric pairings and the two curve groups are completely different species. All of our public keys are developed in the G_2 group which is not a simple Elliptic Curve, unlike the G_1 group. Hence we are forced to use pairing relations to develop homomorphic proofs. The result is a little bit more complicated, but not by much.[1]

The basis for transactions is that an unspent transaction (UTX) may be used by sending portions to one or more destinations. The value of an input UTX must equal the sum of the output UTX deriving from the input:

$$val(UTX_{in}) = \sum_n val(UTX_{out,n})$$

Input and output UTX are specified with public keys of their owners.

We can use pairings to cloak the quantities involved, when necessary, and provide proof of sender identity and UTX values. Bulletproofs accompany these pairings to prove that all quantities are within acceptable range.

[1] An advantage to the pairing-based proofs is that they work for all pairing systems, whether using symmetric pairings with simple Elliptic Curve groups, or asymmetric pairings and more complex Elliptic Curve groups.

A fundamental premise of bilinear pairings states that:

$$e(A + B, C) = e(A, C)\, e(B, C)$$

So if we cloak the value of an input UTX_{in} with a blinding factor k_{rand}, we could write a pairing verification relation as:

$$\begin{aligned} e(\frac{k_{rand}\, val(UTX_{in})}{s}\, U, P) \\ &= e(\frac{k_{rand}\, \sum_n val(UTX_{out,n})}{s}\, U, P) \\ &= \prod_n e(\frac{k_{rand}\, val(UTX_{out,n})}{s}\, U, P) \end{aligned}$$

for private key, s, and public key, P, of sender.

So with this relation and the expression of a pairing of sums as the product of pairings from above, we can specify a transaction as a tuple $(UTX_{in}, UTX_{out,1}, UTX_{out,2}...)$ together with proof values $(S_{in}, P, R, O_1, O_2, ...)$, where

$$S_{in} = \frac{k_{rand}\, val(UTX_{in})}{s}\, U \in G_1$$

for the input UTX, a value for any uncloaked UTX of

$$R = k_{rand}\, V \in G_2$$

and a number of

$$O_i = \frac{k_{rand}\, val(UTX_{out,i})}{s}\, U \in G_1$$

for each cloaked output value.

Some outputs will not be cloaked, such as escrow stake amounts, and transaction fees, so that everyone can see and work with them. For those, we use the R value during proofs, as in

$$e(val(UTX_{out,u})\, U, R) \in G_T$$

The full proof of validity will be that

$$\begin{aligned} e(S_{in}, P) &= (\prod_i e(O_i, P))\, (\prod_u e(val(UTX_{out,u})\, U, R)) \\ &= e(\sum_i O_i, P)\, e((\sum_u val(UTX_{out,u}))\, U, R) \end{aligned}$$

By providing $R \in G_2$, we are not in danger of disclosing anything about the cloaked input UTX_{in} and outputs $UTX_{out,i}$, whose information is provided in group G_1. The blinding factor k_{rand} ensures that nobody can learn anything useful about any of the cloaked outputs $UTX_{out,i}$.

Use of the private and public keys, s and P, provides binding commitment and irrefutable proof of origination. Blending in the private key s, ensures that nobody could have forged the spending.

Information about the actual values of $UTX_{out,i}$ must be provided separately to each recipient, via encrypted messages. But the public can fully verify the validity of these transactions, all without knowing anything about the actual values involved.

Once a recipient receives disclosure of the amount, they can verify that

$$e(O_i, P) = e(val(UTX_{out,i})\, U, R)$$

Reporting proofs in G_1 keeps them as short as possible.

6 Blocks

Blocks consist of a header and a list of transactions. The block header contains a hash of the previous block, the collective signature, and the list of public keys of co-signers, plus the Merkle root of the transactions stored in the block.

The block generation frequency is only limited by the interval between RandHerd[13] beacon outputs (currently 6s) and the time required to propagated the block through the communication tree to collect witness signatures.

7 Future Work

We'll extend this paper with sections on smart contracts and sharding in the future.

APPENDIX

A Cryptography

A.1 Pairing-based Cryptography

Emotiq utilizes advanced bilinear pairing-based cryptography[7][8] (PBC) for user keying, Boneh-Lynn-Shacham (BLS) Signatures[9], fast multi-party signatures, and for Randomness Generation. The advantages of PBC are numerous and include short signatures, fast signature generation, safe deterministic hierarchical wallet keying, and fast multiparty randomness generation.

A bilinear pairing uses pairs of Elliptic Curves, defined over two separate groups, such that their bilinear mappings produce homomorphic encryption in a resulting composite field. If we denote the two curve groups as G_1 and G_2, their Tate pairing field G_T, and prime order finite field Z_r, then their pairing $e(G_1, G_2) \in G_T$ is such that

$$e(a\,U, V) = e(U, a\,V) = g^a$$

where $U \in G_1$, $V \in G_2$, $a \in Z_r$, and $g \in G_T$.

In our system, pairings are asymmetric. Group G_1 is always the smaller group, with the shortest representation. Pairings are ordered, and must be performed with a G_1 element as the first argument, and a G_2 element as second.

Specifically, our Z_r uses 256 bits, G_1 was chosen to have a 264-bit representation, G_2 has a 520-bit representation, G_T has a 3072-bit representation, and the prime order of the groups is $r \approx 2^{256}$, which gives us roughly 2^{128} security.

Private keys belong to the finite field Z_r with the same prime order. Public keys are generated in G_2, and signatures are generated in G_1. The embedding degree of our curves is 12, and correspond to *Type F* asymmetric pairing curves[2] in Lynn's Thesis[7]. Wherever they occur, we use compressed point representation for group elements from G_1 and G_2.

The complete specification of the Emotiq cryptosystem requires knowledge of all curve pairing parameters, plus two chosen generators $U \in G_1$, and $V \in G_2$.

[2]These curves are also known as BN Pairing Curves, for the discoveries by Barreto and Naehrig.[2]

Hash values are mapped into fields and groups in a secure manner,[3] allowing proofs of security with hashing as random oracles. We denote the hash mapping as $G(H(x)) \in G$ for mapping an item x being first hashed, $H(x)$, then mapped into the field or group, G. Hash $H(x)$ is generally SHA3/256 operating on x after its conversion to a vector of bytes.

In the following presentation we shall try to adhere to the convention that groups G_1 and G_2 are additive groups, while Z_r and G_T are both additive and multiplicative. Capitalized symbols denote fields, groups, and points from Elliptic Curves, while lower cased symbols denote elements of finite fields. Field arithmetic in Z_r is implicitly performed $(mod\, r)$.

A.2 Boneh-Lynn-Shacham (BLS) Signatures

BLS signatures are the shortest possible, and enable multisignature generation in just one pass. A BLS signature on message msg is computed as

$$Sig = s\, G_1(H(msg))$$

where $H(x)$ is the SHA3/256 hash of its argument, $s \in Z_r$ is the user's secret key value, and $G_1(H(x)) \in G_1$ is the group member that corresponds to that hash value. A signature is always accompanied by the public key of the signer, $P = s\,V$, for generator $V \in G_2$, producing a signed message as a triple

$$(msg, Sig, P)$$

Because of homomorphism we can verify a signature by noting that a valid signature exhibits the pairing relationship

$$\begin{aligned} e(Sig, V) &= e(s\, G_1(H(msg)), V) \\ &= e(G_1(H(msg)), s\, V) \\ &= e(G_1(H(msg)), P) \end{aligned}$$

And also because of homomorphism, we can easily compute a multi-party signature by simply summing the individual signatures and also summing their corresponding public keys:

$$e(\sum_i s_i\, G_1(H(msg)), V) = e(G_1(H(msg)), \sum_i P_i)$$

producing the collective triple

$$(msg, \sum_i sig_i, \sum_i P_i)$$

Therefore, during the computation of collective signatures, we need only a single pass through all participants as we gather and sum their signature components. A collective signature appears no different than a single signature.

[3]Specifically, for $z = H(x) \in Z$, the value z is mapped into Z_n fields by first absorbing its entire value, with its byte vector seen as denoting a big-endian representation. Then, if the value equals or exceeds the order of the field, n, it is successively truncated by 2 until it has value below the field order.
For Elliptic Curve groups, the value of the field is treated as an $X \in Z_q$ coordinate for a point on the curve. If that X is a valid abscissa, then its positive Y counterpart is chosen. If the X coordinate is not valid, then the curve is re-probed using $X^2 + 1$ as a new trial abscissa.

A.2.1 Comparison with Schnorr Signatures

In contrast, conventional Schnorr signatures require two signature values, forming a quadruple with message and public key. For message msg the Schnorr signature is the pair (R, u) of an Elliptic Curve point R and a field value u, where $R = r\,G$, for generator point G, and

$$r = H(k_{rand}, msg, P)$$

is chosen as a random offset. Finally

$$u = r + H(R, P, msg)\,s$$

The Schnorr signature is validated by checking that

$$u\,G = R + H(R, P, msg)\,P$$

For collective Schnorr signing, all participants are asked to compute their own commitments $R_i = r_i\,G$. Those values are collected and summed to produce a global challenge value, $c_{glb} = H(\sum_i R_i, \sum_i P_i, msg)$. Then the participants are asked to produce their u_i values against that global challenge:

$$u_i = r_i + c_{glb}\,s$$

and again the values are summed. Hence collective Schnorr signatures require two interactions with every signer of the message. Network traffic is approximately twice that required for BLS signatures, with a consequent window of opportunity for attackers to spoil the process during the second round.

A.3 Cryptographic Proofs

Cryptographic proofs can be written for almost any claim. Value range proofs can be developed to prove that a hidden quantity lies within a specific range. That will be the topic of Bulletproofs.

But in preparation for more advanced forms of cryptographic proofs, let's first examine a method for making a claim that some value is known, and can be proven to anyone as known, without revealing anything about it except that it is known – a Zero Knowledge Proof (ZKP).

ZKP's require interaction. A claim is made cryptographically. A verifier challenges the claimant for proof, and the claimant provides another one or more values that can be used by the challenger to verify the claim using a simple calculation. Everything about all math relations is known to all, so everyone can agree that the computations prove a claim.

We can convert ZKP's into Non-Interactive ZKP's (NIZKP) by using the Fiat-Shamir construction, where, instead of an interactive challenge, we simply provide a hash of the public transcript used in making the commitment to the value. It would be extraordinarily difficult for a claimant to produce a hash with a cunningly chosen value, and so a hash stands in for random challenges offered by validators.

There are two parts to any cryptographic proof. First is making a binding claim, second is a cryptographic protocol that can prove the claim. A binding claim prevents the claimant from changing their value so as to satisfy challenges. But we allow a binding claim to hide the value, making the commitment both *binding* and *hiding*. We do that with Pedersen Commitments.

A.3.1 Pedersen Commitments

For value $x \in Z_r$ that we claim to know, we choose two random generators over an Elliptic Curve, $A \in G_1$ and $B \in G_1$, with no known ECDL relationship between them. We also chose a hiding value $\gamma \in Z_r$ so that we can present the commitment without also revealing anything about the value we claim to know.

We publish our commitment and the two random generators as a triple (A, B, C), keeping x and the hiding value γ secret, and where commitment C is given as:

$$C = \gamma A + xB \in G_1$$

This is computationally *binding* on our choices of x and γ because they are applied to two independent curve generators, A and B. It is *hiding* because, even if you could perform ECDL, you still wouldn't be able to find them separately. All you would know is something about their sum, and that ranges over the entire field Z_r.

We can prove knowledge of both x and γ, for any random challenge value $z \in Z_r$, by computing and offering three values (α, L, R) where

$$\alpha = z\,\gamma + \frac{x}{z} \in Z_r$$

$$L = x\,A \in G_1$$

$$R = \gamma\,B \in G_1$$

The challenger then takes those values, and forms a new generator $G' \in G_1$

$$G' = \frac{1}{z}\,A + z\,B$$

and computes a modified commitment C' as

$$C' = \frac{1}{z^2}\,L + C + z^2\,R$$

and then sees that

$$\alpha\,G' = C'$$

For an NIZKP version of this, simply substitute

$$z = Z_r(H(A, B, C))$$

as a *nothing up my sleeve* (NUMS) challenge value.

The protocol proves knowledge of both x and hiding value γ. Values of L and R are unchanging for any challenge, and so, to allay suspicions, these could both be provided at the outset. Their constancy proves binding of x and γ.

The only thing that changes with each challenge, z, is the response, α. And since the verifier relations depend only on this response and the challenge itself, it proves that claimant knows the bound values of x and γ. This kind of proof forms the basis for Bulletproofs.

A.3.2 Pedersen Commitments with Cloaking

The problem with this kind of ZKP is that we must reveal item $L = x\,A \in G_1$. If x is known to have come from some restricted domain, then a simple brute force search, already knowing A, is all that is needed to reveal x. There is no danger facing γ because it comes from a large field where brute force search is impractical.

So to protect x, we do additional cloaking, as do Bulletproofs, using a random cloaking value $\xi \in Z_r$. Instead of

committing only to x, we now form three Pedersen commitments, one for each of ξ, $(x-\xi)$, and x. We furnish Pedersen commitment proofs on ξ and $(x-\xi)$, and an additional proof relating the three commitments.

The three commitments are

$$C_\xi = \gamma_\xi \, A + \xi \, B$$

$$C_{(x-\xi)} = \gamma_{(x-\xi)} \, A + (x-\xi) \, B$$

$$C_x = \gamma_x \, A + x \, B$$

Proof of C_ξ and $C_{(x-\xi)}$ is performed as shown above for Pedersen commitments. There is no danger to ξ and $(x-\xi)$ since they come from the entire range of field Z_r.

But to prove the C_x commitment, we publish an adjustment term, γ_{adj} in the A curve, which only someone who knows all of the γ hiding values could have created,

$$\gamma_{adj} = \gamma_x - (\gamma_\xi + \gamma_{(x-\xi)}) \in Z_r$$

Then the validator sees that

$$C_x = C_\xi + C_{(x-\xi)} + \gamma_{adj} \, A$$

This proves our knowledge since, while the commitment point addition with the B term is obvious, the point addition in A has no known relationship to ξ, x, or B.

All three commitments are computationally *binding* and *hiding.* Committing to cloaking value ξ assures the validator that we aren't just making it up as we go. And committing to $(x-\xi)$ locks in the relationship between ξ and x in these coupled Pedersen commitments. The final proof between commitments shows that we know the relationship between γ_x and the already proven γ_ξ and $\gamma_{(x-\xi)}$.

A.4 Bulletproofs

Bulletproofs[3] are used to provide range proofs on numeric values. Most useful, non-cryptographic, values come from restricted domains. We could provide range proofs on such values by providing Pedersen commitments and proofs to show that the value is equal to one of the possible values from the range. If there were only a small number of possibilities, this might seem reasonable.

But consider a value from the domain of 64-bit numbers. There are 2^{64} possible values, and it would be impractical to provide a proof over each possibility. Instead, we could provide proofs over each bit of the binary encoding of the value, to show that each one is either 0 or 1, and to prove that we know which it is. But that is still 64 or more commitments and proofs just for a single value.

Going back to the Pedersen commitment proof, notice that our proofs show that we actually know two quantities, x and hiding value γ, and all we had to do to prove that duo was to furnish one additional field value, α. That 2-to-1 reduction is the reason for the moniker *BulletProofs*, and is reminiscent of the very reason to use proofs over binary encodings of values.

By augmenting Pedersen commitment proofs to show that, not only do we know both γ and x, but that their product has a particular value, $p = \gamma \, x$, we can then develop a recursive form of Pedersen commitment proof, of size $log_2(log_2 N))$ for values in the range $0 \le x < N$, based on dot-products of vectors representing the bits of the binary encoding of x. Each stage of the recursion shrinks the bit encoding vectors by half, effectively making a binary encoding of a binary encoding. Now it becomes feasible and economical to prove value ranges of 64-bit values.

Of course, we must also use cloaking at every step, since the domains are so restricted that it would be trivial to uncover the hidden values. That complicates matters a bit, but the idea remains feasible and more economical than any other approach.

B Randomness

B.1 Fast Randomness Generation with PVSS

The use of BLS Signatures allows an abbreviated form of PVSS randomness generation. Participants in randomness generation are given a list of neighboring group nodes in the network, with whom they carry out a pBFT protocol with publicly verifiable secret sharing (PVSS).

Within each group, a sharing threshold is set at $t = \lfloor \frac{N}{3} \rfloor + 1$ for group size N. Secret random seeds are generated by each participant, then encrypted shares are formed over that secret and distributed to other group members, along with cryptographic proofs on the shares.

For sharing threshold t, a random polynomial of order $t-1$ is generated

$$p(x) = a_0 + a_1 x + ... + a_{t-1} x^{t-1}$$

with the secret value denoted by a_0. Shares are constructed by computing the value of this polynomial for each member of the group, assigned successive ordinal values, $i = 1...N$. The resulting share values, $share_i = p(i)$, are cloaked with the private key, s, and blinding factor k_{rand}, and also encrypted by multiplying the share value by the public key of each member,

$$Proof_i = \frac{k_{rand} \, p(i)}{s} \, U \in G_1$$

$$Chk_i = k_{rand} \, p(i) \, V \in G_2$$

$$E(share_i) = p(i) \, P_i \in G_2$$

$$R = k_{rand} \, U \in G_1$$

Vectors of proofs and their checks, and and a vector of encrypted shares, is generated, one element for each member of the group, and these vectors are then transmitted to each group member.

$$(Proof_1, Proof_2, ..., Proof_N)$$

$$(Chk_1, Chk_2, ..., Chk_N)$$

$$(R, E(share_1), E(share_2), ..., E(share_N))$$

Every share can be validated with

$$e(Proof_i, P) = e(U, Chk_i)$$

where P is the public key of the sender. This serves as proof of origination of the share.

Every member of the group can also verify that all shares from another group member were consistently generated

from the same sharing polynomial. To do so, we treat the share vector as a codeword from a Reed-Solomon encoding[4], compute a random polynomial of order $N-t-1$ and use that to compute a test vector from the dual-space of the original share generating polynomial:

$$f(x) = b_0 + b_1 x + ... + b_{N-t-1} x^{N-t-1}$$

$$c_\perp = (\lambda_1 f(1), \lambda_2 f(2), ..., \lambda_N f(N))$$

where weights $\lambda_i = \prod_{j \neq i} \frac{1}{i-j}$, for $i, j = 1...N$. Then the consistency of the encrypted shares is verified by checking that:

$$\sum_i c_{\perp i}\, Proof_i = G_1(0)$$

This consistency check is absolutely certain for valid sharing vectors, and has an inconsequential probability of failing to detect an improper sharing set given as $\approx 1/q$, or about 1 chance in 2^{256}. There is a greater likelihood of finding a hash collision in SHA3/256 than in seeing a failure to detect an inconsistent sharing vector.

After performing consistency checks on the sharing set from one group member, the share directed at one node can be decrypted with its secret key to produce a decrypted share,

$$\begin{aligned} Y_i &= e(\frac{1}{s_i} U, E(share_i)) \\ &= e(U, V)^{p(i)} \in G_T \end{aligned}$$

for secret key $s_i \in Z_r$. The decryption can be verified with

$$e(\frac{1}{s_i} R, E(share_i)) = e(U, Chk_i)$$

The decrypted share, Y_i, is then broadcast to all group members.

As soon as a sharing threshold number, $(n \geq t)$, of decrypted shares has been seen for any one sharing group, the secret randomness from that set can be discovered via Lagrange interpolation:

$$Y_{grp} = \prod_i Y_i^{\prod_{j \neq i} \frac{i}{i-j}}$$

Finally, after a supermajority of sharing sets has been decrypted, $(n \geq 2\lfloor \frac{N}{3} \rfloor + 1)$, their randomness is combined as a simple sum (in the exponent) in G_T, and forwarded to all clients.

$$Y_{total} = \prod_{grp} Y_{grp}$$

Final randomness results from a supermajority sum (in the exponent of the base pairing value) of randomness obtained from each group.

So the use of pairing-based cryptography shows great benefits, not only in minimizing network traffic, by making immediate commitments to portions along the way.

Timing tests show that this approach scales linearly with number of group participants, ranging from about 5 seconds for 32 group members, to about 3 minutes for 1024 group members, on an ordinary iMac with an Intel i7 processor. The timings are dominated by compute load, not network communications.

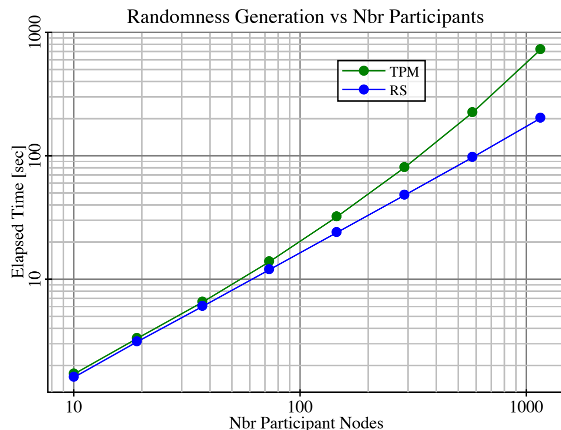

Figure 2: Comparison of performance between the TPM Method and the Reed-Solomon interpretation of proof vectors.

B.1.1 The TPM Method

There is an alternate method for generating randomness with PVSS, which we can call the TPM Method.[14][4] It performs the same share generation procedure as seen above, but instead of sending along a vector of proofs on each encrypted share, it sends along proofs on the sharing polynomial coefficients, $a_i, i = 0..t-1$ using $Aproof_i = a_i U$ for generator $U \in G_1$.

Then, at each receiving node, the shares are verified by computing the sums

$$Proof_i = \sum_{j=0..t-1} i^j\, Aproof_j, i = 1..N$$

Then the pairing relation is checked

$$e(Proof_i, P_i) = e(U, E(share_i))$$

These proof sums correspond directly to the proofs presented in the previous section. And they also directly verify that every encrypted share came from the same polynomial. One advantage of this method is that instead of transmitting $2N$ vector elements, we now only need to transmit $N+t$ elements.

However, we now also need to supply these proof sums along with any decrypted shares that we compute, and the method scales super-linearly. Timing tests have shown it to be a hair slower than the first method for $N = 32$, and about three times slower when $N = 1024$. But the method is equivalent in its information content, and provably more secure. There are no cases where an inconsistent sharing set would escape detection.

B.2 Verifiable Random Functions

A verifiable random function (VRF)[10] uses a pseudo-random function (PRF) in such a way as to furnish proof

[4]The paper has an error wherein it specifies that encrypted shares should be formed as the product of the share value and the generator of the curve. That is incorrect, as the encryption needs to incorporate information about the target node keying. It seems likely that the error crept in by way of translation to English.

that the resulting pseudo-random value came from a specific seed value mixed with the creator's secret key. And even though a secret key was involved in the generation of randomness, anyone can duplicate the calculation by knowing only the seed value and creator's public key. We follow the work of Dodis and Yampolskiy[5], to provide a PRF as follows:

For arbitrary input seeding values, form the hash $H(seed)$ using SHA3/256. This converts arbitrary objects of any length into a 256-bit random-like, but reproducible, pattern. Then map that hash value into our Z_r field, which has dimension $q = |Z_r| \approx 2^{256}$, to produce $x = Z_r(H(seed))$.

Output of the VRF is a pseudo-random value in the pairing field (3072 bits)

$$y = VRF(x, s) = e(U, V)^{1/(x+s)} \in G_T$$

for secret key $s \in Z_r$, generators $U \in G_1$, $V \in G_2$, and with proof

$$R = \frac{1}{x+s} U \in G_1$$

Output of the VRF is the quadruple $(seed, x, y, R)$, i.e., the original seed, the seed deterministically reduced to an element of Z_r, the output of the VRF computation, and the point in G_1 that represents the proof.

Verification of proof checks the pairing

$$e(R, x\,V + P) = e(U, V) \in G_T$$

for public key $P = s\,V \in G_2$, and to verify that

$$y = e(R, V)$$

and that

$$x = Z_r(H(seed))$$

B.3 Safe Hierarchical Keying

In current blockchain designs that utilize simple Elliptic Curve cryptography, the possibility of producing subkeys from a master public key is presented. But that is wholly unsafe in the event that a decryption key is also generated for a derived public key. A simple bit of finite field arithmetic is all it takes to discover the original master private key.

With PBC we can safely generate both public and private keys without exposing our master private key. This is also known as Identity-Based Encryption (IBE). But unlike conventional presentations of IBE, we do not rely on a trusted third party for the generation of our keying. Rather, we view the master key holder as the only entity that should be entitled to generate new decryption keys.

Anyone can generate new public keys at any time, based on previously known public keys. But in order to obtain a decryption key for the new public keys, you must ask the primary secret key holder for a decryption key. Doing so puts the primary key holder at no risk for exposing his or her private key.

A new public key can be generated by asking for a subkey of a given public key, using an arbitrary identity value to identify that subkey. The new public key is computed as

$$P_{id} = Z_r(H(id))\,V + P$$

for identity id, generator $V \in G_2$, public key $P \in G_2$, and where $Z_r(H(id)) \in Z_r$ is the element of the field that corresponds to the hash of the supplied identity.

You can use this public key to encrypt a message by making use of the hash of a pairing value as an XOR mask against a message

$$E(msg) = msg \oplus H(g^x)$$

where $x = Z_r(H(msg, id)) \in Z_r$, and pairing element

$$g^x = e(U, x\,V)$$

for generator $U \in G_1$, generator $V \in G_2$. The message is transmitted as the triple $(E(msg), X, id)$, with $X = x\,P_{id} \in G_2$.

In order to produce a decryption key for that new public key, the primary key holder computes

$$S_{id} = \frac{1}{s + Z_r(H(id))} U \in G_1$$

for secret key $s \in Z_r$. Producing a decryption key in G_1 ensures, by difficulty of ECDLP, that our master private key remains safe against exposure.

Homomorphism allows us to see that the pairings

$$\begin{aligned} e(S_{id}, X) &= e(\frac{1}{s + Z_r(H(id))} U, x\,(Z_r(H(id))\,V + P)) \\ &= e(U, x\,V) = g^x \end{aligned}$$

which allows us to recreate the XOR mask and decrypt to the original message

$$msg = E(msg) \oplus H(g^x)$$

Verification of the message is done by computing $x = Z_r(H(msg, id))$ and checking that

$$x\,(Z_r(H(id))\,V + P) = X \in G_2$$

In this form, a new private key cannot be used to sign messages in the same manner as BLS signatures with the master private key, because we have $P_{id} \neq s_{id}\,V$. But it does furnish a way to encrypt and decrypt messages by using the hash of the pairing result. This technique has been dubbed SAKKE by its authors Sakai-Kasahara[11]. We have extended SAKKE encryption to indefinite length by using successive SHA3 hashes on the pairing field result and an increasing index value.

3 References

[1] Miguel Castro and Barbara Liskov *Practical Byzantine Fault Tolerance*

[2] Paulo S. L. M. Barreto and Michael Naehrig *Pairing-Friendly Elliptic Curves of Prime Order*

[3] Benedikt Bunz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell *Bulletproofs: Short Proofs for Confidential Transactions and More*

[4] Ignacio Casudo and Bernardo David, *SCRAPE: Scalable Randomness Attested by Public Entities*

[5] Yevgeniy Dodis and Aleksandr Yampolskiy, *A Verifiable Random Function With Short Proofs and Keys*

[6] Yossi Gilad, Rotem Hemo, Silio Micali, Georgios Vlachos, Nickolai Zeldovich, *Algorand: Scaling Byzantine Agreements for Cryptocurrencies*

[7] Ben Lynn, PhD Thesis, *On the Implementation of Pairing-Based Cryptosystems*, June 2007

[8] Ben Lynn, PBC Library, https://crypto.stanford.edu/pbc/download.html

[9] Ben Lynn, *BLS Signatures*, https://crypto.stanford.edu/pbc/manual/ch02s01.html

[10] Silvio Micali, Michael Rabin, Salil Vadhan, *Verifiable Random Functions*

[11] Sakai-Kasahara, *IETF RFC 6508 Sakai-Kasahara Key Encryption (SAKKE)*

[12] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, Bryan Ford, *Keeping Authorities "Honest or Bust" with Decentralized Witness Cosigning*

[13] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nocals Gailly, Linus Gasser, Ismail Khoffi, Michael J. Fisher, Bryan Ford, *Scalable Bias-Resistant Distributed Randomness*

[14] Youlian Tian, Changgen Peng, Jianfeng Ma, *Publicly Verifiable Secret Sharing Scheme Using Bilinear Pairings*