24.09.03 Light Client and Mithril BLS Signature

Xun Zhang Wuyun Siqin Bingsheng Zhang
Zhejiang University, CHN
22221024@zju.edu.cn 3210101763@zju.edu.cn bingsheng@zju.edu.cn

September 3 2024

1 Mithril Light Client

The basic methodology of our zero-knowledge bridge it to use ZK-SNARK to generate a proof, the target chain can be convinced by verifying the proof.

The ZK-SNARK aims to prove the correctness of a(or a batch of) block header. So the smart contract on the target chain will directly verify the proof, instead of verifying the block header(like a light client).

Thus, the approach of zero-knowledge bridge is to write a circuit of light client process firstly. We need to find out the logic of Mithril(as well as Ouroboros) validating a certificate(or a block header).

1.1 Mithril Validation Code

The code is developed by **Cardano Foundation**, and it's used to communicate with IBC, or **Cosmos** blockchain. The so-callled **Cardano IBC Incubator** is a project working towards a bridge implementation to allow exchange of information from a Cardano blockchain to Cosmos SDK based blockchains.[https://github.com/cardano-foundation/cardano-ibc-incubator] In the directory './cosmos' , containg all Cosmos SDK related source code including the Cardano light client (or thin client) implementation running on the Cosmos chain. The folder was scaffolded via Ignite CLI with Cosmos SDK 0.50.

We show the Mithril light client code following(actually it is not a light clinet, because Mithril always verify the whole certificate):

```go
func (v *MithrilCertificateVerifier) VerifyStandardCertificate(
	certificate *Certificate,
	signature *entities.ProtocolMultiSignature,
) (*Certificate, error) {
	if err := v.VerifyMultiSignature(
		[]byte(certificate.SignedMessage),
		signature,
		certificate.AggregateVerificationKey,
		certificate.Metadata.ProtocolParameters); err != nil {
		return nil, err
	}

	previousCertificate, err := v.CertificateRetriever.GetCertificateDetails(certificate.PreviousHash)
	if err != nil {
		return nil, err
	}

	if previousCertificate.Hash != certificate.PreviousHash {
		return nil, fmt.Errorf("certificate chain previous hash unmatch")
	}

	currentCertificateAVK, err := certificate.AggregateVerificationKey.ToJsonHex()
	if err != nil {
		return nil, err
	}
	previousCertificateAVK, err := certificate.AggregateVerificationKey.ToJsonHex()
	if err != nil {
		return nil, err
	}

	validCertificateHasDifferentEpochAsPrevious := func(nextAggregateVerificationKey string) bool {
		return nextAggregateVerificationKey == currentCertificateAVK && previousCertificate.Epoch != certificate.Epoch
	}
	validCertificateHasSameEpochAsPrevious := func() bool {
		return previousCertificateAVK == currentCertificateAVK && previousCertificate.Epoch == certificate.Epoch
	}

	nextAggregateVerificationKey, ok := previousCertificate.ProtocolMessage.GetMessagePart("next_aggregate_verification_key")
	if ok {
		nextAvk, err := FromAvkProto(string(nextAggregateVerificationKey))
		if err != nil {
			return nil, err
		}
		nextAvkJson, err := nextAvk.ToJsonHex()
		if err != nil {
			return nil, err
		}

		if validCertificateHasDifferentEpochAsPrevious(nextAvkJson) {
			return previousCertificate, nil
		}
		if validCertificateHasSameEpochAsPrevious() {
			return previousCertificate, nil
		}
		return nil, errorsmod.Wrapf(ErrInvalidCertificate, "currentAvk and nextAvk are not match")
	}
	return nil, errorsmod.Wrapf(ErrInvalidCertificate, "can not get nextAvk from previous certificate")
}
```

If we implement the SNARK-based Mithril , then the first check, verifying the multi-signature can be replaced by the **verifying a halo2 proof**.

So Besides the mulitisignature, there are other checks to do to accept a certificate.

Assuming that we have two certificates, one has been confirmed and stored in memory, and the other one remains to be validated. We call them previousCertificate and currentCertificate.

So this is a simple scenario of client, because it does not need to validate the genesis certificate, or a chain of certificates, just need to check one "new" certificate.

Following are other checks client need to do:

1. check if previousCertificate.Hash = certificate.PreviousHash
2. if two certificates in same epoch:
   - previousCertificateAVK = currentCertificateAVK
   - previousCertificate.Epoch = certificate.Epoch
3. if two certificates in different epoch:
   - previousCertificate.next_aggregate_verification_key = currentCertificateAVK
   - previousCertificate.Epoch! = certificate.Epoch

1.2 Cardano Validation Code

The validation logic is very complex in original code, we extract some important parts of code to see, how we verify a cardano block data in Cosmos chain.

There is a concept names "client state" in this code. Briefly speaking, a client state stores the information of a cardano light client, and the light client uses it to check the new block, and then updates its "client state"(if pass).

The following code shows how a client verifying a block's data:

```
// verifyBlockData returns an error if:
// - signature is not valid
// - vrf key hash is not in SPO list
// - header timestamp is past the trusting period in relation to the consensus state
func (cs *ClientState) verifyBlockData(
	ctx sdk.Context, clientStore storetypes.KVStore, cdc codec.BinaryCodec,
	blockData *BlockData,
) error {
	verifyError, isValid, vrfHex, blockNo, slotNo := ledger.VerifyBlock(ledger.BlockHexCbor{
		HeaderCbor:    blockData.HeaderCbor,
		Eta0:          blockData.EpochNonce,
		Spk:           int(cs.SlotPerKesPeriod),
		BlockBodyCbor: blockData.BodyCbor,
	})

	if verifyError != nil {
		return errorsmod.Wrapf(ErrInvalidBlockData, "Verify: Invalid block data, data not valid, %v", verifyError.Error())
	}

	if !isValid {
		return errorsmod.Wrap(ErrInvalidBlockData, "Verify: Invalid block data, signature not valid")
	}

	if slotNo != blockData.Slot || blockNo != blockData.Height.RevisionHeight {
		return errorsmod.Wrap(ErrInvalidBlockData, "Verify: Invalid block data, slot or block not valid")
	}

```

```
	// check, calculate and store validator set for new epoch
	if cs.CurrentEpoch != blockData.EpochNo {
		newValidatorSet := CalValidatorsNewEpoch(clientStore, cs.CurrentEpoch, blockData.EpochNo)

		// verify
		if !newValidatorSetIsValid(newValidatorSet, vrfHex) {
			return errorsmod.Wrap(ErrInvalidSPOsNewEpoch, "Verify: Invalid signature")
		}

		// store
		setClientSPOs(clientStore, newValidatorSet, blockData.EpochNo)
	} else {
		oldValidatorSetBytes := clientStore.Get(ClientSPOsKey(cs.CurrentEpoch))
		oldValidatorSet := MustUnmarshalClientSPOs(oldValidatorSetBytes)
		// verify
		if !newValidatorSetIsValid(oldValidatorSet, vrfHex) {
			return errorsmod.Wrap(ErrInvalidSPOsNewEpoch, "Verify: Invalid signature")
		}
	}

	return nil
}
```

This code uses methods in other files, so we just simply introduce what have been done in the checking process. The detailed explanation is coming soon.

- VerifyBlock function:
  1. Check is KES valid.
  2. Check is VRF valid.
  3. Check if block data valid.
- Check slot and height.
- Check if it is a new epoch:
  1. Yes. Verify the signature of new validators set and store.
  2. No. Verify the signature of old validators set.

Above is the rough logic of a Cardano light client.

2 Mithril BLS Signature and Benchmark

In Mithril paper, the pairing check of BLS signature is like following:

$\mathsf{MSP.Ver}(msg, mvk, \sigma)$:Return 1 if $e(\sigma, g2) = e(\mathrm{H}_{\mathbb{G}_1}(''M''||msg), mvk)$. Otherwise return 0.

But in halo2-lib code, the signature σ and message hash is in $\mathbb{G}_2$ and the verification key is in $\mathbb{G}_1$.

So we modify the code of BLS signature verification and benchmark it. Following are our new results.

degree	advice	lookup	lookup_bits	limb_bits	proof_time	proof_size	verify_time
14	211	27	13	91	34.5397s	95808	42.6186ms
15	105	14	14	90	31.7353s	48000	25.6577ms
16	50	6	15	90	26.1168s	22752	19.5484ms
17	25	3	16	88	24.6446s	11520	12.2392ms
18	13	2	17	88	25.1638s	6080	10.6350ms
19	6	1	18	90	25.0424	3072	7.4521ms
20	3	1	19	88	35.0907s	1920	12.2006ms
21	2	1	20	88	54.0728s	1344	5.5069ms
22	1	1	21	88	89.5040s	960	10.0790ms

Table 1: Bn254 BLS Signature(σ on $\mathbb{G}_1$)

Compare to the original BLS signature(built in halo2-lib code), the Mithril BLS signature is slower(~15s vs ~25s)

2.1 Combine New BLS Signature with Merkle Tree

In the previous code, we use Poseidon hash function as the building block of merkle tree, the rate of Poseidon is set to 2. Thus we use a **ZERO** padding. But since we switch to a verification key in $\mathbb{G}_2$, it is natural use a $\mathbb{G}_2$ element as a hash input, because two 'coefficients' of $\mathbb{G}_2$ element is in $\mathbb{F}_p$:

$$\text{Original: Poseidon.input} = [pk.x, F : ZERO]$$

$$\text{New: Poseidon.input} = [pk.x.c0, pk.x.c1]$$

However, in our development schedule, we should add the signer's stake(or the ϕ evaluation of stake as a optimization), so the rate of Poseidon will be 3, and the construction will be:

$$\text{Plan: Poseidon.input} = [pk.x.c0, pk.x.c1, \phi(\mathsf{stake_i})]$$

$$\text{Plan: Merkle\_Path.input} = [\mathsf{leaf}, \mathsf{path_i}, F : ZERO]$$