# ZK-SNARKs in ZKBNB



# ZK-SNARK

ZK-SNARK stands for Zero-Knowledge Succinct Non-Interactive Argument of Knowledge, and is a type of zero-knowledge proof system. In `ZK-SNARK` systems, a prover must convince a verifier that they know the solution to a specific problem, without revealing the solution itself. This is achieved through the use of a proof construction (`ZK-proof`), which is a succinct, non-interactive, and argument of knowledge. The circuit is a way to describe the problem in a way that can be efficiently processed by the prover and verifier.

The circuit consists of a series of gates that represent the computations that the prover must perform to arrive at the solution to the problem. This allows the prover to demonstrate that they know the solution, without revealing the actual solution. Additionally, the use of a circuit allows for the efficient computation of ZK proofs.

Here you can find the mathematical explanation of ZK-SNARKs:

ZK-SNARKs

## ZK-SNARKs in zkBNB

Main parts that are responsible for ZK-SNARKs in ZKBNB are `zkbnb-crypto` library and `prover` service which utilizes the library:

```
https://github.com/bnb-chain/zkbnb-crypto
```

zkbnb/service/prover at master · bnb-chain/zkbnb

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or window. Reload to refresh your session. Reload to refresh your session.

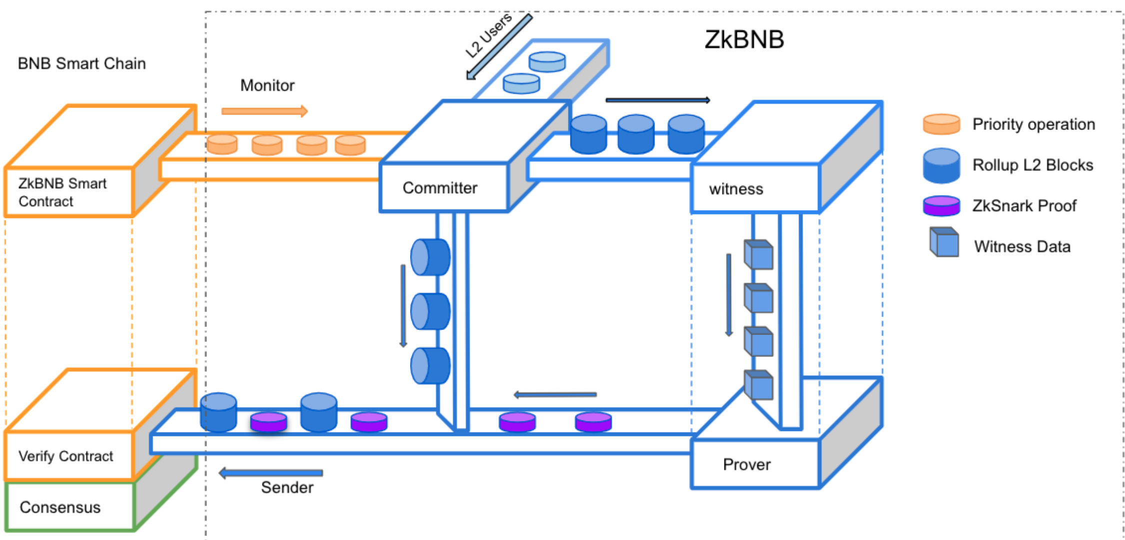
https://github.com/bnb-chain/zkbnb/tree/master/service/prover

bnb-chain/**zkbnb**

A trustless and scaling solution for BNB Smart Chain

19 Contributors · 4 Issues · 145 Stars · 72 Forks

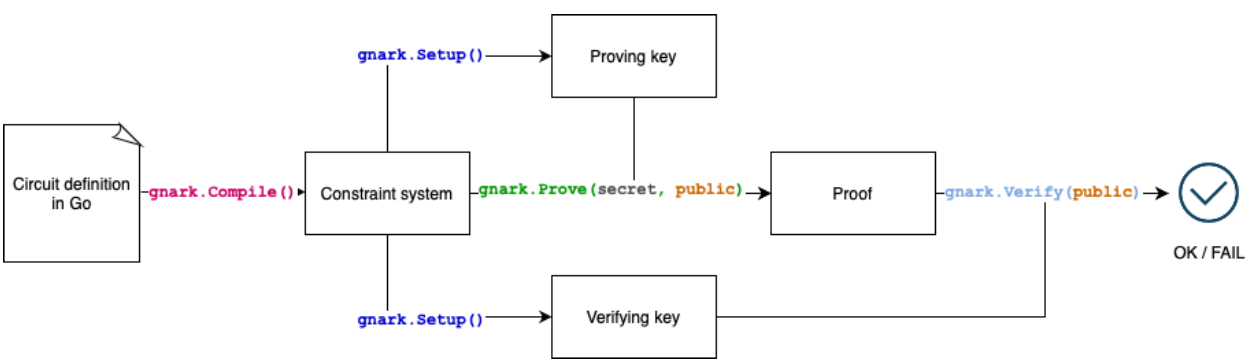
As you may see in this diagram, processed by `committer` user transactions (from L2 layer) or priority requests (from L1 layer) are then being submitted in BSC by `sender` along with `ZkSnark Proofs` created by `Prover` service.




# Gnark

`ZkBNB` uses gnark framework to create ZK-proofs, gnark is a fast zk-SNARK library that offers a high-level API to design circuits.

## gnark architecture




As you may see, `Compile` and `Setup` phases do not depend on circuit input data and created only once during zkbnb deployment. At the same time `Proof` depends on submitted blocks data and therefore should be created for each new block.

### Groth16

We use Groth16 proving system which requires circuit-specific trusted setup. It is best suited when an application needs to generate many proofs for the same circuit and performance is critical.

You can read about it in details here:

http://www.zeroknowledgeblog.com/index.php/groth16

### Montgomery form

Since Prover generates proofs and Verifier verifies them through Elliptic Curve pairings, we need a modular arithmetic. We will do it by modulos `q`, where:

```
q=21888242871839275222246405745257275088548364400416034343698204186575808495617
```

So all variables in circuit are presented in Montgomery form since it allows to do modular operations much more cheap.

> Montgomery Multiplication
>
> Unsurprisingly, a large fraction of computation in modular arithmetic is often spent on calculating the modulo operation, which is as slow as general integer division and typically takes 15-20 cycles, depending on the operand size.
>
> A https://en.algorithmica.org/hpc/number-theory/montgomery/

The basic idea is that after encoding numbers to the Montgomery form and do some transformation, we can replace modulo operation with a right shift by 32/64 which is super fast.

Downside of it is that it might be hard to debug or do logging inside circuit. You can use `zkbnb/service/prover/test/engine_test.go` to debug your circuit. Just paste witness from database inside the test and do debug:

```
var circuitBlock *circuit.Block
_ = json.Unmarshal([]byte("${your_witness}), &circuitBlock)
blockConstraints, _ := circuit.SetBlockWitness(circuitBlock)
blockConstraints.TxsCount = 10
blockConstraints.GasAssetIds = types.GasAssets[:]
blockConstraints.GasAccountIndex = types.GasAccount
if err := IsSolved(&hintCircuit{}, &blockConstraints, ecc.BN254, backend.UNKNOWN); err != nil {
  t.Fatal(err)
}
```

# Circuits in ZKBNB-crypto

As we remember, `Compile` and `Setup` phases do not depend on circuit input data and created only once during `zkbnb` deployment in deploy.sh:

```
flag=$1
if [ $flag = "new" ]; then
  echo "new crypto env"
  echo '2. start generate zkbnb.vk and zkbnb.pk'
  cd ${DEPLOY_PATH}
  cd zkbnb-crypto && go test ./circuit/solidity -timeout 99999s -run TestExportSol
  cd ${DEPLOY_PATH}
```

```
  mkdir -p $KEY_PATH
  cp -r ./zkbnb-crypto/circuit/solidity/* $KEY_PATH
fi
```

Generated files are persisted then and used for proof generation and proof submission to L1.

## Compile circuit and Setup PK/VK

We use this code to compile circuit, create `Pk` and `Vk`:

```
func TestExportSol(t *testing.T) {
  differentBlockSizes := []int{1, 10}
  exportSol(differentBlockSizes)
}

func exportSol(differentBlockSizes []int) {
  gasAssetIds := []int64{0, 1}
  gasAccountIndex := int64(1)
  sessionName := "zkbnb"
  for i := 0; i < len(differentBlockSizes); i++ {
    var blockConstraints circuit.BlockConstraints
    blockConstraints.TxsCount = differentBlockSizes[i]
    blockConstraints.Txs = make([]circuit.TxConstraints, blockConstraints.TxsCount)
    for i := 0; i < blockConstraints.TxsCount; i++ {
      blockConstraints.Txs[i] = circuit.GetZeroTxConstraint()
    }
    blockConstraints.GasAssetIds = gasAssetIds
    blockConstraints.GasAccountIndex = gasAccountIndex
    blockConstraints.Gas = circuit.GetZeroGasConstraints(gasAssetIds)
    oR1cs, err := frontend.Compile(ecc.BN254, r1cs.NewBuilder, &blockConstraints, frontend.IgnoreUnconstrainedInputs())
    fmt.Printf("Constraints num=%v\n", oR1cs.GetNbConstraints())
    if err != nil {
      panic(err)
    }

    // pk, vk, err := groth16.Setup(oR1cs)
    err = groth16.SetupLazyWithDump(oR1cs, sessionName+fmt.Sprint(differentBlockSizes[i]))
    if err != nil {
      panic(err)
    }
    {
      verifyingKey := groth16.NewVerifyingKey(ecc.BN254)
      f, _ := os.Open(sessionName + fmt.Sprint(differentBlockSizes[i]) + ".vk.save")
      _, err = verifyingKey.ReadFrom(f)
      if err != nil {
        panic(fmt.Errorf("read file error"))
      }
      f.Close()
      f, err := os.Create("ZkBNBVerifier" + fmt.Sprint(differentBlockSizes[i]) + ".sol")
      if err != nil {
        panic(err)
      }
      err = verifyingKey.ExportSolidity(f)
      if err != nil {
        panic(err)
      }
    }
  }
}
```

As you can see, we need to generate block with empty transactions and use it for circuit compilation. Also pay attention to the block size number since you can only use your pk and vk for the corresponding block size. So you can’t create pk for blockSize=300 and then during runtime use blockSize=10.

During compilation phase `Define()` method of a circuit is executed and circuit is processed. Let's take a look at it for `zkbnb-crypto` circuit:

```
func (circuit BlockConstraints) Define(api API) error {
  // mimc
  hFunc, err := mimc.NewMiMC(api)
  if err != nil {
    return err
  }

  err = VerifyBlock(api, circuit, hFunc)
  if err != nil {
    return err
  }
  return nil
}
```

In zkbnb-crypto there is only one `Define` function is executed which is provided above.

And during compile phase block with empty number of transactions will be verified and our business logic will be converted into Arithmetic Circuit form which means that millions of constraints will be created and added to the array in `oR1cs` object.

### PK

Normally `Pk` file is generated during this phase which represents coefficients of the polynomials of the original problem. In `ZkBNB` lazy setup was implemented. The main idea is to temporarily remove constraints during setup phase and then recover them during solving. This reduces the memory usage and increases prover performance during startup.

```
err = groth16.SetupLazyWithDump(oR1cs, sessionName+fmt.Sprint(differentBlockSizes[i]))
```

### VK

`VK` is generated almost the same way as `Pk`, it's not being sent anywhere and only being used to locally verify that the proof is created correctly in the prover iself.

```
verifyingKey := groth16.NewVerifyingKey(ecc.BN254)
```

### ZkBNBVerifier.sol

```
err = verifyingKey.ExportSolidity(f)
```

What is actually used - is `ZkBNBVerifier1.sol` smart contract which is generated based on `Vk` data and then deployed to the L1.

### Usage of Pk, Vk and `ZkBNBVerifier1.sol`

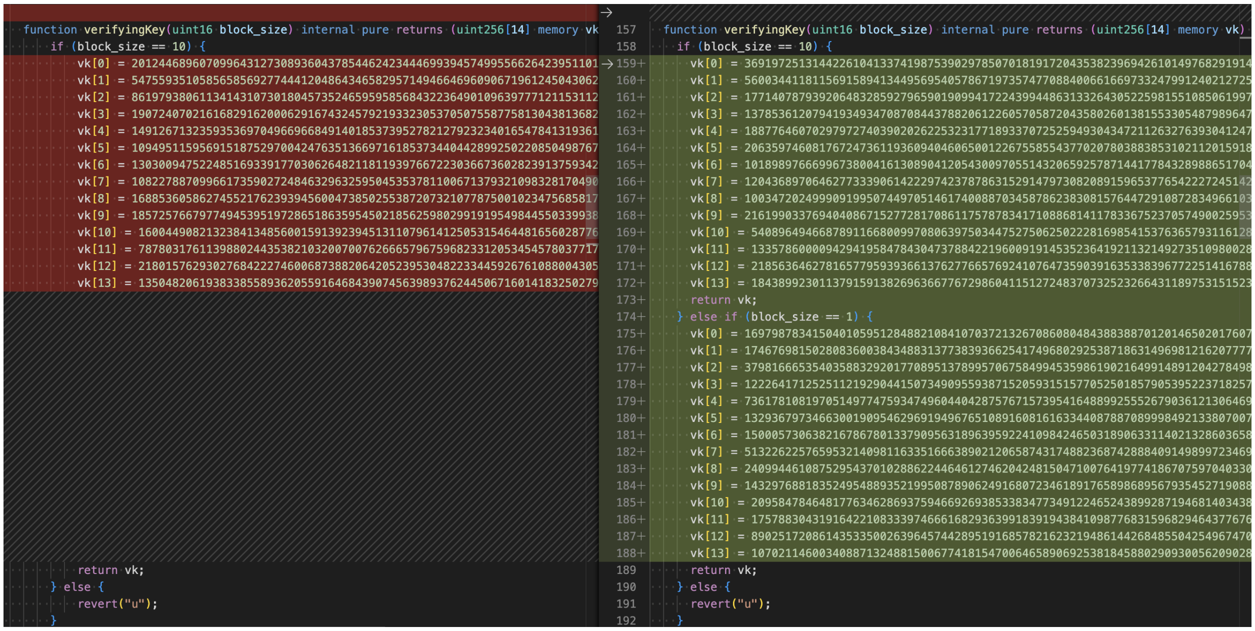
`Pk` is used during `proof` generation in Prover and it is not being sent anywhere:

```
proof, err = groth16.ProveRoll(r1cs, provingKey[0], provingKey[1], witness, session, backend.WithHints(types.PubDataToBytes))
```

`Vk` is used after `proof` generation in Prover to check that it was calculated correctly:

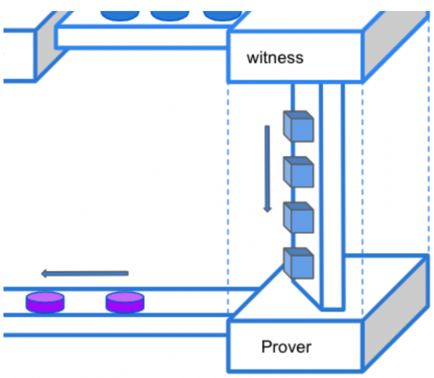
```
err = groth16.Verify(proof, verifyingKey, vWitness)
```

Exported `ZkBNBVerifier1.sol/ZkBNBVerifier1.sol` is simply used during zkbnb deployment phase in python script to update `ZkBNBVerifier.sol` 's VK data in `zkbnb-contract` and then `ZkBNBVerifier.sol` contract is used to L1 and used for proof verification.



For details about deployment of `ZkBNBVerifier.sol` to L1, check `/Users/user/IdeaProjects/zkbnb-contract/scripts/deploy-keccak256/deploy.js`

## Proof generation




As you remember, `gnark` has public and secret inputs. In case of `zkBNB` the whole block data is a secret input and only block commitment is a public data:

```
type BlockConstraints struct {
  BlockNumber     Variable
```

```
  CreatedAt       Variable
  OldStateRoot    Variable
  NewStateRoot    Variabl
  BlockCommitment Variable `gnark:",public"`
  Txs             []TxConstraints
  TxsCount        int
  Gas             GasConstraints
  GasAssetIds     []int64
  GasAccountIndex int64
}
```

So here we come to the whole idea of zkbnb-crypto. As you may see, we processed all the transactions in committer but we don't want to send any details of any transaction to L1.

> **We send to L1 only block commitment and ZK proof. And L1 smart contract can verify that all the transactions in the block are valid and therefore save a lot of calldata and increase processing speed (consider block verification happens in constant time).**

Witness is processed in a separate service, called `witness` and persisted in database. It's a separate service because witness generation takes some time and we don't want to block user requests because of it.

Here is what prover does during startup:

```
// read R1CS vectors
prover.R1cs[i], err = groth16.LoadR1CSFromFile(c.KeyPath[i])
// read proving and verifying keys generated during setup phase
prover.ProvingKeys[i], err = prove.LoadProvingKey(c.KeyPath[i])
prover.VerifyingKeys[i], err = prove.LoadVerifyingKey(c.KeyPath[i])
```

Then proof is created based on witness data fetched from DB:

```
// startup prover cron job
_, err := cronJob.AddFunc("@every 10s", func() {
    logx.Info("start prover job......")
    // cron job for receiving cryptoBlock and handling
    err := p.ProveBlock()
    if err != nil {
      logx.Severef("failed to generate proof, %v", err)
    }
  })

// inside ProveBlock() we read latest witness from db
blockWitness, err := p.BlockWitnessModel.GetLatestBlockWitness()
var cBlock *circuit.Block
json.Unmarshal([]byte(blockWitness.WitnessData), &cBlock)

// create secret witnes
blockWitness, err := circuit.SetBlockWitness(cBlock)
witness, err := frontend.NewWitness(&blockWitness, ecc.BN254)
if err != nil {
  return proof, err
}

// create public witness
var verifyWitness circuit.BlockConstraints
verifyWitness.OldStateRoot = cBlock.OldStateRoot
verifyWitness.NewStateRoot = cBlock.NewStateRoot
verifyWitness.BlockCommitment = cBlock.BlockCommitment
vWitness, err := frontend.NewWitness(&verifyWitness, ecc.BN254, frontend.PublicOnly())
if err != nil {
  return proof, err
}
```

```
// create proof based on R1CS vectors, PK created before and secret witness
proof, err = groth16.ProveRoll(r1cs, provingKey[0], provingKey[1], witness, session, backend.WithHints(types.PubDataToBytes))
if err != nil {
  return proof, err
}

// verify proof locally based on VK created before and public witness
err = groth16.Verify(proof, verifyingKey, vWitness)
```

Then proof is formatted since it has such form in for Groth16 setup (each vector is 64 bytes):

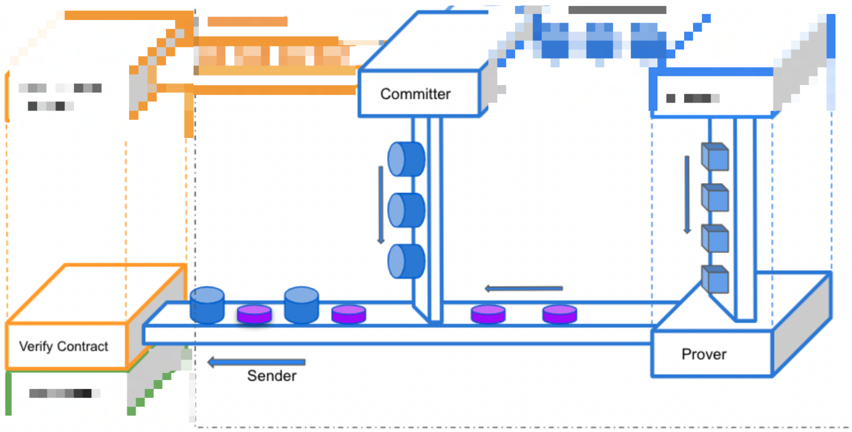
$$\pi := \left([A_r]_1, [B_s]_2, [K_{r,s}]_1\right).$$

```
type FormattedProof struct {
  A      [2]*big.Int
  B      [2][2]*big.Int
  C      [2]*big.Int
  Inputs [3]*big.Int
}

func FormatProof(oProof groth16.Proof, oldRoot, newRoot, commitment []byte) (proof *FormattedProof, err error) {
  proof = new(FormattedProof)
  const fpSize = 4 * 8
  var buf bytes.Buffer
  _, err = oProof.WriteRawTo(&buf)
  if err != nil {
    return nil, err
  }
  proofBytes := buf.Bytes()
  // proof.Ar, proof.Bs, proof.Krs
  proof.A[0] = new(big.Int).SetBytes(proofBytes[fpSize*0 : fpSize*1])
  proof.A[1] = new(big.Int).SetBytes(proofBytes[fpSize*1 : fpSize*2])
  proof.B[0][0] = new(big.Int).SetBytes(proofBytes[fpSize*2 : fpSize*3])
  proof.B[0][1] = new(big.Int).SetBytes(proofBytes[fpSize*3 : fpSize*4])
  proof.B[1][0] = new(big.Int).SetBytes(proofBytes[fpSize*4 : fpSize*5])
  proof.B[1][1] = new(big.Int).SetBytes(proofBytes[fpSize*5 : fpSize*6])
  proof.C[0] = new(big.Int).SetBytes(proofBytes[fpSize*6 : fpSize*7])
  proof.C[1] = new(big.Int).SetBytes(proofBytes[fpSize*7 : fpSize*8])

  // public witness
  proof.Inputs[0] = new(big.Int).SetBytes(oldRoot)
  proof.Inputs[1] = new(big.Int).SetBytes(newRoot)
  proof.Inputs[2] = new(big.Int).SetBytes(commitment)
  return proof, nil
}
```

Formatted proof is persisted then in database:

```
err = p.ProofModel.CreateProof(row)
```

## Proof submission to L1




During runtime execution `sender` service reads latest committed blocks that wait to be executed, fetches proofs for them and sends them to L1:

```
// reads latest committed blocks
blocks, err := s.blockModel.GetCommittedBlocksBetween(start,
  start+int64(s.config.ChainConfig.MaxBlockCount))

// some formatting
pendingVerifyAndExecuteBlocks, err := ConvertBlocksToVerifyAndExecuteBlockInfos(blocks)

// extracts proofs for these blocks
blockProofs, err := s.proofModel.GetProofsBetween(start, start+int64(len(blocks))-1)
var proofs []*big.Int
for _, bProof := range blockProofs {
  var proofInfo *prove.FormattedProof
  err = json.Unmarshal([]byte(bProof.ProofInfo), &proofInfo)
  if err != nil {
    return err
  }
  proofs = append(proofs, proofInfo.A[:]...)
  proofs = append(proofs, proofInfo.B[0][0], proofInfo.B[0][1])
  proofs = append(proofs, proofInfo.B[1][0], proofInfo.B[1][1])
  proofs = append(proofs, proofInfo.C[:]...)
}

// extracts gas price
var gasPrice *big.Int
if s.config.ChainConfig.GasPrice > 0 {
  gasPrice = big.NewInt(int64(s.config.ChainConfig.GasPrice))
} else {
  gasPrice, err = s.cli.SuggestGasPrice(context.Background())
  if err != nil {
    logx.Errorf("failed to fetch gas price: %v", err)
    return err
  }
}

// Verify blocks on-chain
txHash, err := zkbnb.VerifyAndExecuteBlocks(cli, authCli, zkbnbInstance,
  pendingVerifyAndExecuteBlocks, proofs, gasPrice, s.config.ChainConfig.GasLimit)
```

First `VerifyAndExecuteBlocks` function of `ZkBNB.sol` is executed and then `verifyBatchProofs` of `ZkBNBVerifier.sol` which contains our generated VK data during setup. And if everything is good, we update account state root and increase number of verified blocks (since blocks numbers are consecutive):

```
bool res = verifier.verifyBatchProofs(proofs, publicInputs, batchLength, block_size);
require(res, "inp");
```

```
// update account root
stateRoot = _blocks[nBlocks - 1].blockHeader.stateRoot;
totalBlocksVerified += nBlocks;
```

```
// original equation
// e(proof.A, proof.B)*e(-vk.alpha, vk.beta)*e(-vk_x, vk.gamma)*e(-proof.C, vk.delta) == 1
// accumulation of inputs
// gammaABC[0] + sum[ gammaABC[i+1]^proof_inputs[i] ]
function verifyBatchProofs(
  uint256[] memory in_proof, // proof itself, length is 8 * num_proofs
  uint256[] memory proof_inputs, // public inputs, length is num_inputs * num_proofs
  uint256 num_proofs,
  uint16 block_size
) public view returns (bool success) {
  ...
}
```

# API of ZKBNB-crypto

zkbnb-crypto also provides API to hash transactions submitted by users in L2. Let's take a look at `MintNftTx` example:

```
func (txInfo *MintNftTxInfo) Hash(hFunc hash.Hash) (msgHash []byte, err error) {
  packedFee, err := ToPackedFee(txInfo.GasFeeAssetAmount)
  if err != nil {
    log.Println("[ComputeTransferMsgHash] unable to packed amount", err.Error())
    return nil, err
  }
  msgHash = Poseidon(ChainId, TxTypeMintNft, txInfo.CreatorAccountIndex, txInfo.Nonce, txInfo.ExpiredAt,
    txInfo.GasFeeAssetId, packedFee, txInfo.ToAccountIndex, txInfo.CreatorTreasuryRate, txInfo.NftCollectionId,
    ffmath.Mod(new(big.Int).SetBytes(common.FromHex(txInfo.ToAccountNameHash)), curve.Modulus),
    ffmath.Mod(new(big.Int).SetBytes(common.FromHex(txInfo.NftContentHash)), curve.Modulus))
  return msgHash, nil

}
```

User uses then calculated hash for the message to sign transaction (using Eddsa) based on his own private key:

```
func ConstructMintNftTx(key accounts.Signer, tx *types.MintNftTxReq, ops *types.TransactOpts) (string, error) {
  ...
  msgHash, err := convertedTx.Hash(hFunc)
  signature, err := key.Sign(msgHash, hFunc)
  convertedTx.Sig = signature
  ...
}
```

In circuit message hash is calculated again based on tx data and Eddsa signature is verified to be sure transaction/user data submitted to circuit is not malicious:

```
hashValCheck = types.ComputeHashFromMintNftTx(api, tx.MintNftTxInfo, tx.Nonce, tx.ExpiredAt)
hashVal = api.Select(isMintNftTx, hashValCheck, hashVal)
...

func ComputeHashFromMintNftTx(api API, tx MintNftTxConstraints, nonce Variable, expiredAt Variable) (hashVal Variable) {
  return poseidon.Poseidon(api, ChainId, TxTypeMintNft, tx.CreatorAccountIndex, nonce, expiredAt,
    tx.GasFeeAssetId, tx.GasFeeAssetAmount, tx.ToAccountIndex,
    tx.CreatorTreasuryRate, tx.CollectionId, tx.ToAccountNameHash, tx.NftContentHash)
}

err = types.VerifyEddsaSig(
```

```
    isLayer2Tx,
    api,
    hFunc,
    hashVal,
    tx.AccountsInfoBefore[0].AccountPk,
    tx.Signature,
  )
```

# Some of the implemented tasks

▼ **Add support for Ipfs CID in NftContentHash variable**

**Problem: s**ince in gnark all variables are represented as big numbers modulo q (which is bigger than 16 bytes and smaller than 32 bytes), after new changes, Ipfs CID which occupies 32 bytes, sometimes exceeded q and therefore calculation coudn't proceed normally.

**Solution: Split NftContentHash variable into 2 16-byte variable**

https://github.com/bnb-chain/zkbnb-crypto/pull/74

▼ **Implement circuit-friendly Keccak256 hash algorithm**

Hint function of Keccak256 was used in zkbnb-crypto to get hash of block data (aka block_commitment). Since hint functions are not secure and do not generate any constraints, there was a need to have a circuit-friendly implementation of Keccak256.

https://github.com/bnb-chain/gnark/pull/12

▼ **Fix of proof generation error for Full Exit NFT Transactions**

https://github.com/bnb-chain/zkbnb-crypto/pull/72/files

## Some common questions

### When PK/VK should be regenerated?

Gary made a good table which contains data about different situations when PK/VK is changing and you can also read this article:

ZK-SNARKs

The idea is that any change in circuit code changes number of constraints and may dramatically change R1CS vectors system. And it's also up to gnark compiler whether some changes will reflect in changed R1CS or not. So it's safer to regenerate PK/VK each time you do some change in code. But definitely such things as added logging or field name modifying should not reflect at all.



| when pk vk change | | |
|---|---|---|
| | PK | VK |
| origin | | |
| add note | | |
| add field has no relation to calc | No | No |
| add field | yes | |
| change field public to private | yes | yes |
| change method name | no | no |
| change method logic | yes | yes |
| print log | no | no |
| add method | | |