Gnark Cheat Sheet

Getting started

— Installing Gnark —

```sh
go get github.com/consensys/
gnark@latest
```

- `frontend.Variable` is abbreviated as `Var`
- □ is in-circuit code, □ is out-circuit code

— Define Circuit —

```
import "github.com/consensys/gnark/
frontend"
type Circuit struct {
    PreImage Var `gnark:",secret"`
    Hash     Var `gnark:"hash,public"`
}
func (c *Circuit) Define(
           api frontend.API) error {
    m, _ := mimc.NewMiMC(api)
    m.Write(c.PreImage)
    api.AssertIsEqual(c.Hash, m.Sum())
}
```

— Compile —

```
var mimcCircuit Circuit
cur := ecc.BN254.ScalarField()
r1cs, err := frontend.Compile(
  cur, r1cs.NewBuilder, &mimcCircuit)
vals := &Circuit { Hash: "161...469",
PreImage: 35 }
w, _ := frontend.NewWitness(vals, cur)
pubw, _ := w.Public()
```

— Prove: Groth16 —

```
pk, vk, _ := groth16.Setup(cs)
proof, _ := groth16.Prove(cs, pk, w)
err := groth16.Verify(proof, vk, pubw)
```

— Prove: PlonK —

```
srs, lag, _ := unsafekzg.NewSRS(cs)
pk, vk, _ := plonk.Setup(cs, srs, lag)
proof, _ := plonk.Prove(cs, pk, w)
err := plonk.Verify(proof, vk, pubw)
```

API

— Assertions —

```
// fails if i1 != i2
AssertIsEqual(i1, i2 Var)
// fails if i1 == i2
AssertIsDifferent(i1, i2 Var)
// fails if v != 0 and v != 1
AssertIsBoolean(i1 Var)
// fails if v ∉ {0,1,2,3}
AssertIsCrumb(i1 Var)
// fails if v > bound.
AssertIsLessOrEqual(v Var, bound Var)
```

Made by Icer Liang, 2024-08-08.

Inspired by golang.sk

— Arithemetics —

```
// = i1 + i2 + ... in
Add(i1, i2 Var, in ...Var) Var
// a = a + (b * c)
MulAcc(a,b, c Var) Var
Neg(i1 Var) Var // -i.
// = i1 - i2 - ... in
Sub(i1, i2 Var, in ...Var) Var
// = i1 * i2 * ... in
Mul(i1, i2 Var, in ...Var) Var
// i1 /i2. =0 if i1 = i2 = 0
DivUnchecked(i1, i2 Var) Var
Div(i1, i2 Var) Var // = i1 / i2
Inverse(i1 Var) Var // = 1 / i1
```

— Binary —

```
// unpacks to binary (lsb first)
ToBinary(i1 Var, n ...int) []Var
// packs b to element (lsb first)
FromBinary(b ...Var) Var
// following a and b must be 0 or 1
Xor(a, b Var) Var // a ^ b
Or(a, b Var) Var // a | b
And(a, b Var) Var // a & b
```

— Flow —

```
// performs a 2-bit lookup
Lookup2(b0,b1 Var,i0,i1,i2,i3 Var) Var
// if b is true, yields i1 else i2
Select(b Var, i1, i2 Var) Var
// returns 1 if a is zero, 0 otherwise
IsZero(i1 Var) Var
// 1 if i1>i2, 0 if i1=i2, -1 if i1<i2
Cmp(i1, i2 Var) Var
```

— Debug —

Run the program with `-tags=debug` to display a more verbose stack trace.

```
Println(a ...Var) //like fmt.Println
```

Standard Library

— MiMC Hash —

```
import "github.com/consensys/gnark/
std/hash/mimc"
fMimc, _ := mimc.NewMiMC()
fMimc.Write(circuit.Data)
h := fMimc.Sum()
```

— EdDSA Signature —

```
import t "github.com/consensys/gnark-
crypto/ecc/twistededwards"
import te "github.com/consensys/gnark/
std/algebra/native/twistededwards"
type Circuit struct {
    pub eddsa.PublicKey
    sig eddsa.Signature
    msg frontend.Variable
}
cur, _ := te.NewEdCurve(api, t.BN254)
eddsa.Verify(cur, c.sig, c.msg, c.pub,
&fMimc)
```

— Merkle Proof —

```
import "github.com/consensys/gnark/
std/accumulator/merkle"
type Circuit struct {
  M    merkle.MerkleProof
  Leaf frontend.Variable
}
c.M.VerifyProof(api, &hFunc, c.Leaf)
```

Selector

```
import "github.com/consensys/gnark/
std/selector"
```

— Slice —

```
// out[i] = i ∈ [s, e) ? in[i] : 0
Slice(s, e Var, in []Var) []Var
// out[i] = rs ? (i ≥ p ? in[i] : 0)
//              : (i < p ? in[i] : 0)
Partition(p Var, rs bool, in []Var)
[]Var
// out[i] = i < sp ? sv : ev
stepMask(outlen int, sp, sv, ev Var)
[]Var
```

— Mux —

```
// out = in[b[0]+b[1]*2+b[2]*4+...]
BinaryMux(selBits, in []Var) Var
// out = vs[i] if ks[i] == qkey
Map(qkey Var, ks, vs []Var) Var
// out = in[sel]
Mux(sel Var, in ...Var) Var
// out[i] = ks[i] == k ? 1 : 0
KeyDecoder(k Var, ks []Var) []Var
// out[i] = i == s ? 1 : 0
Decoder(n int, sel Var) []Var
// out = a1*b1 + a2*b2 + ...
dotProduct(a, b []Var) Var
```

Concepts

— Glossary —

`cs`: constraint system, `w`: (full) witness, `pubw`: public witness, `pk`: proving key, `vk`: verifying key, `r1cs`: rank-1 constraint system, `srs`: structured reference string.

— Schemas —

Groth16: $\mathcal{L}\vec{x} \cdot \mathcal{R}\vec{x} = \mathcal{O}\vec{x}$

PlonK: $q_{l_i} a_i + q_{r_i} b_i + q_{o_i} c_i + q_{m_i} a_i b_i + q_{c_i} = 0$

SAP(Polymath): $x \cdot y = (x/2 + y/2)^2 - (x/2 - y/2)^2$

Schema	**CRS/SRS**	**Proof Size**	**Verifier Work**
Groth16	$(3n+m)\mathbb{G}_1$	$2\mathbb{G}_1 + 1\mathbb{G}_2$	$3P + \ell\,\boldsymbol{m_1}$
PlonK	$(n+a)\mathbb{G}_1 + \mathbb{G}_2$	$9\mathbb{G}_1 + 7\mathbb{F}$	$2P + 18\,\boldsymbol{m_1}$
Polymath	$(\widetilde{m} + 12\tilde{n})\mathbb{G}_1$	$3\mathbb{G}_1 + 1\mathbb{F}$	$2P + 2\,\boldsymbol{m_1} + \boldsymbol{m_2} + \tilde{\ell}\mathbb{F}$

*m =wire num, n =multiplication gates, $\widetilde{m} \approx 2m$, $\tilde{n} \approx 2n$, $\boldsymbol{m_L} = \mathbb{G}_L$ exp. a =addition gates, P =pairing, ℓ =pub inputs num, $\tilde{\ell} = O(\ell \log \ell)$ PlonK is universal setup

— Resources —

- https://docs.gnark.consensys.io/
- https://play.gnark.io/
- https://zkshanghai.xyz/

Serialization

— CS Serialize —

```
var buf bytes.Buffer
cs.WriteTo(&buf)
```

— CS Deserialize —

```
cs := groth16.NewCS(ecc.BN254)
cs.ReadFrom(&buf)
```

— Witness Serialize —

```
w, _ :=
frontend.NewWitness(&assignment,
ecc.BN254)
data, _ := w.MarshalBinary()
json, _ := w.MarshalJSON()
```

— Witness Deserialize —

```
w, _ := witness.New(ecc.BN254)
err := w.UnmarshalBinary(data)
w, _ := witness.New(ecc.BN254,
ccs.GetSchema())
err := w.UnmarshalJSON(json)
pubw, _ := witness.Public()
```

Smart Contract

— Export Solidity —

```
f, _ := os.Create("verifier.sol")
err = vk.ExportSolidity(f)
```

— Export Plonk Proof —

```
_p, _ := proof.
(interface{MarshalSolidity() []byte})
str := "0x" + hex.EncodeToString(
  _p.MarshalSolidity())
```

— Export Groth16 Proof —

```
buf := bytes.Buffer{}
_, err := proof.WriteRawTo(&buf)
b := buf.Bytes()
var p [8]string
for i := 0; i < 8; i++ {
  p[i] = new(big.Int).SetBytes(
    b[32*i : 32*(i+1)]).String()
}
str := "["+strings.Join(p[:],",")+"]"
```

Standard Library

— MiMC Hash —

```
import "github.com/consensys/gnark-
crypto/ecc/bn254/fr/mimc"
fMimc := mimc.NewMiMC()
fMimc.Write(buf)
h := fMimc.Sum(nil)
```

— EdDSA Signature —

```
import "math/rand"
import t "github.com/consensys/gnark-
crypto/ecc/twistededwards"
import "github.com/consensys/gnark-
crypto/hash"
curve := t.BN254
ht := hash.MIMC_BN254
seed := time.Now().Unix()
rnd := rand.New(rand.NewSource(seed))
s, _ := eddsa.New(curve, rnd)
sig, _ := s.Sign(msg, ht.New())
pk := s.Public()
v, _ := s.Verify(sig, msg, ht.New())
c.PublicKey.Assign(curve, pk.Bytes())
c.Signature.Assign(curve, sig)
```

— Merkle Proof —

```
import mt "github.com/consensys/gnark-
crypto/accumulator/merkletree"
depth := 5
num := uint64(2 << (depth - 1))
seg := 32
mod := ecc.BN254.ScalarField()
// Create tree by random data
mlen := len(mod.Bytes())
var buf bytes.Buffer
for i := 0; i < int(num); i++ {
  leaf, _:= rand.Int(rand.Reader, mod)
  b := leaf.Bytes()
  buf.Write(make([]byte, mlen-len(b)))
  buf.Write(b)
}
// build merkle tree proof and verify
hGo := hash.MIMC_BN254.New()
idx := uint64(1)
root, path, _, _ :=
mt.BuildReaderProof(&buf, hGo, seg,
idx)
verified := mt.VerifyProof(hGo, root,
path, idx, num)
c.Leaf = idx
c.M.RootHash = root
c.M.Path = make([]Var, depth+1)
for i := 0; i < depth+1; i++ {
  c.M.Path[i] = path[i]
}
```