

# Hey SNARKs 🖐️

## You Had me at Go

Gentle Intro to gnark; a fast and easy to use open-source zk-SNARKs library in Golang

Sigrid Jin [@sigridjineth](https://twitter.com/sigridjineth) / Golang Korea  
Go To Busan 2023 | 2023-06-03 SAT

# gnark



# Speaker

Golang Korea

[@sigridjineth](https://twitter.com/sigridjineth)



Sigrid

## Sigrid Jin / Jin Hyung Park

- Software Engineer at DSRV
- POSTECH Simperby Project Committer
- GDG Golang Korea Organizer
- THE NEW ZK STUDY Lab Lead @ 모두의연구소
- Distinguished Graduate from Harmony ZKU
- Yet Another Undergraduate math noob
- Unwavering support for @thisissigrid (> 7 yrs)
- Former) Researcher at Naver Connect

# Index



- No-brainer's intro to Zero-Knowledge Proof (no math!)
- A brief intro to gnarks API workflow
- Simple Example using gnarks API
- Limitations & Further implementations
- Conclusion & Q&A



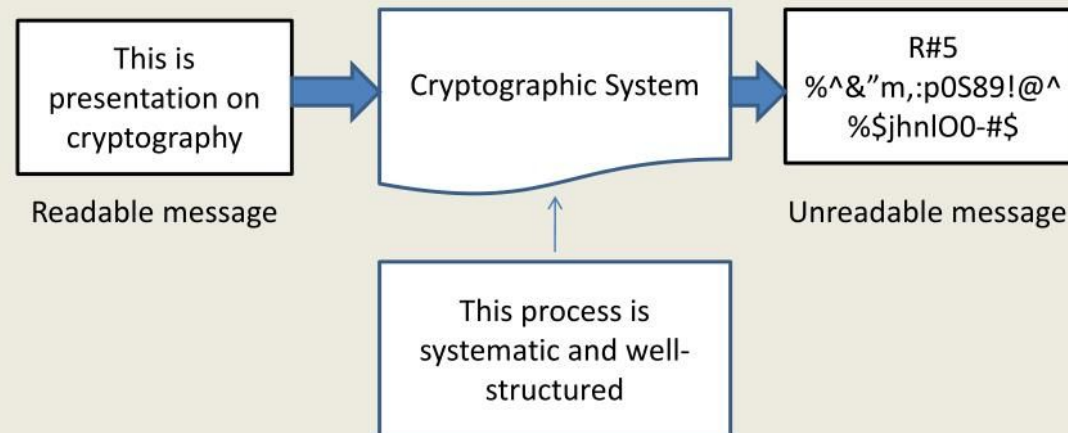
# No-brainer intro to ZKPs



# Cryptography is the tech for verifying the computation; ZKP is the cutting-edge variant of cryptography

## What is cryptography?

- **Definition :** Cryptography is the art and science of achieving security by encoding message to make them non-readable.



# Yet another Alice/Bob Example

Find this guy



In this scene



- Alice: I know where Waldo is!
- Bob: Alice, do you know what a liar is?
- Alice: I can prove to you where he is without revealing his location.
- To defend her integrity, Alice devises two solutions to prove her knowledge.

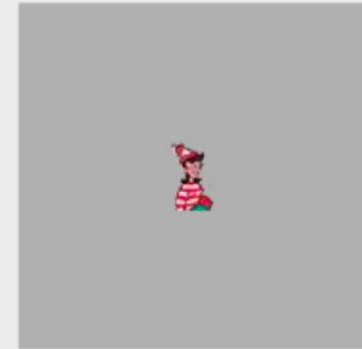
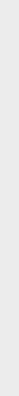
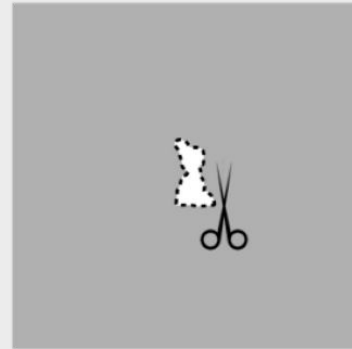
# Yet another Alice/Bob Example

- Alice cuts out Waldo from her scene and only shows Bob the Waldo snippet.
- To ensure that Alice didn't just print out a new picture of Waldo, Bob can watermark the back of Alice's scene page.



# Yet another Alice/Bob Example

- Alice cuts a hole in a very large, opaque sheet of cardboard. She places the cardboard cutout on top of the original scene. In this solution, only Waldo is shown.
- His coordinates relative to the rest of the scene is still unknown.
- Later, Alice can reproduce the scene underneath to prove that she used the original puzzle.



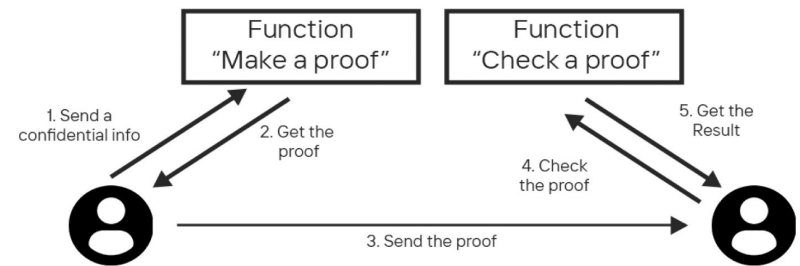


# What is Zero-Knowledge Proof?

The ability to prove computation without revealing inputs

To understand zero knowledge proof, it is first necessary to define the 2 actors involved in the process and their roles:

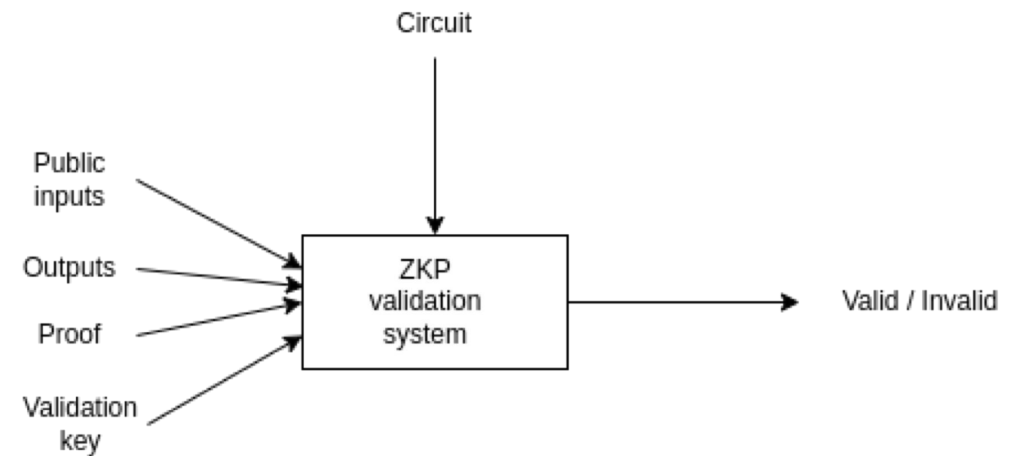
1. A Prover, who executes a computation and wants to prove to any third party that the computation was valid.
2. A Verifier, whose role is to verify that the computation done by someone else was valid.



# What is Zero-Knowledge Proof?

The ability to prove computation without revealing inputs

A computation is any deterministic program that gets inputs and returns outputs. The naive way for a verifier to verify that a computation done by a third party was valid would be to run the same program with the same inputs and check if the output is the same.



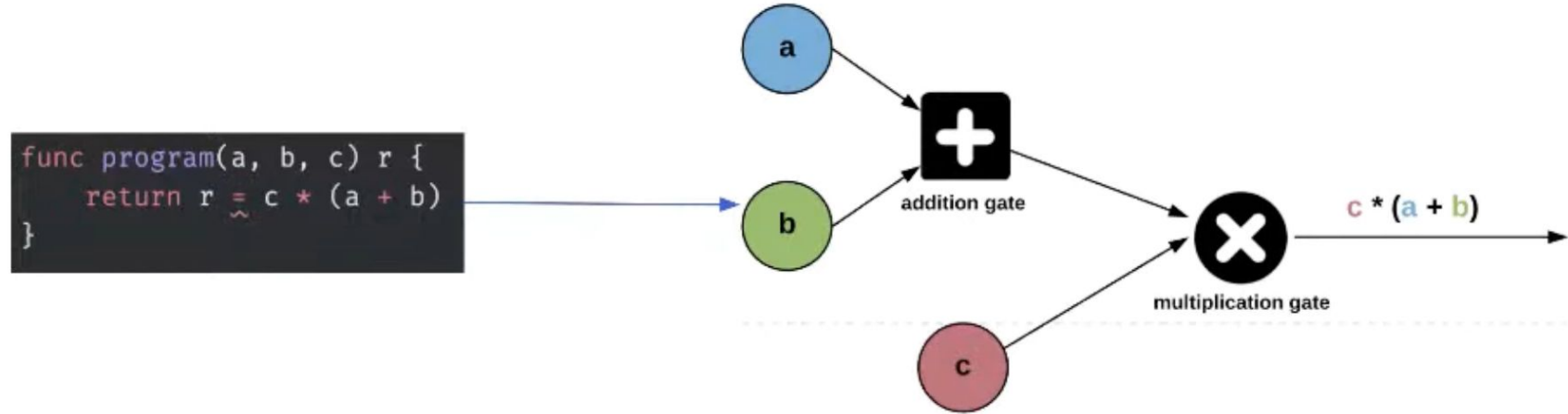
# How does ZK proof work?

The ability to prove computation without revealing inputs



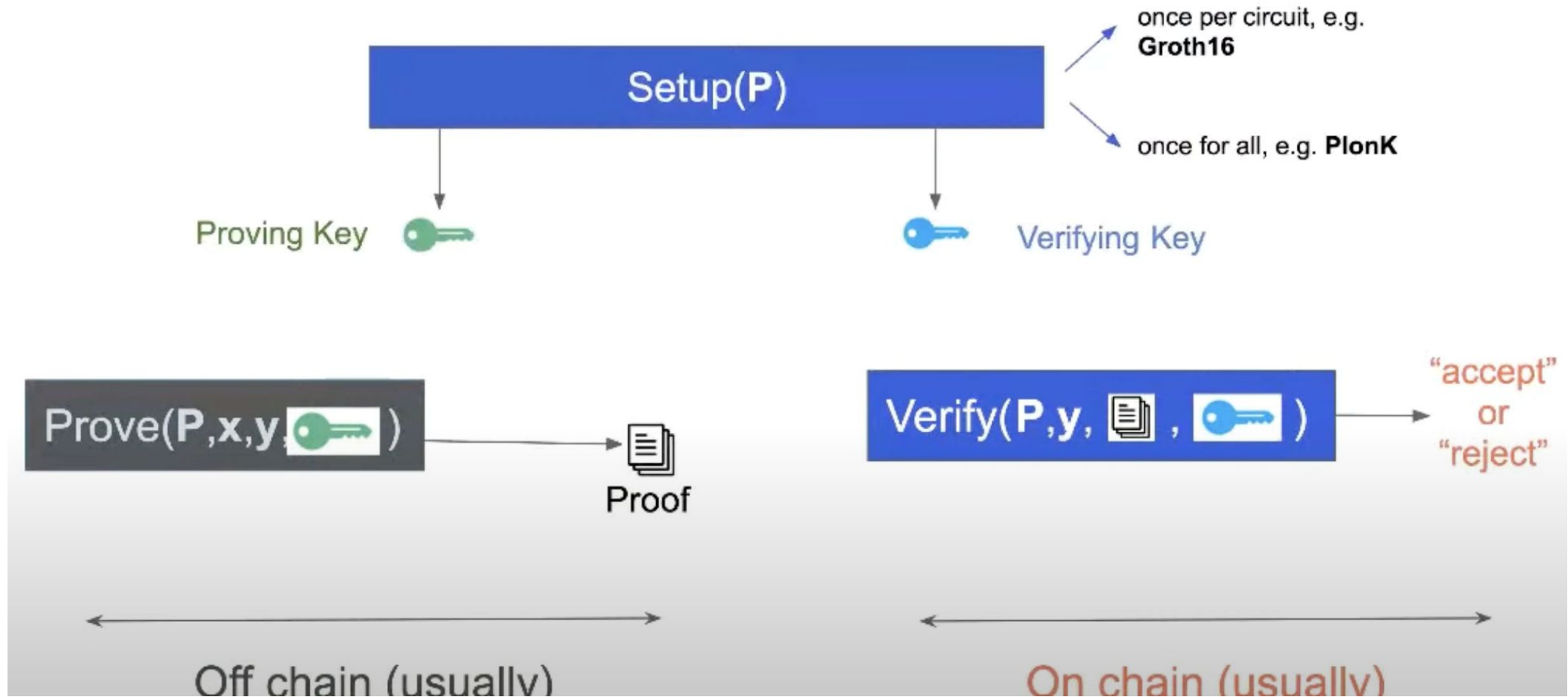
- It all starts with having a deterministic program, circuit
- **The prover executes the computation and computes the output of the program.** The prover, starting from the circuit and the output, computes a proof of his/her computation and give it to the verifier.  
**The verifier can now run a more lightweight computation starting from the proof and verify that the prover did the entire computation correctly.** The verifier doesn't need to know the whole set of inputs to verify the correctness of the computation.

# SNARKs workflow in a high-level



A program  $P$  is encoded into an arithmetic form to reason with matrices, polynomials and numbers. Please note that the SNARKs are working on the finite field (modulo  $p$ ). The prover solves the arithmetic form and encodes the results into a Proof. The verifier checks the Proof and convinces that the Prover executed the program  $P$ .

# SNARKs workflow in a high-level



[https://www.youtube.com/watch?v=Eva2IHFI\\_rl&t](https://www.youtube.com/watch?v=Eva2IHFI_rl&t)

# SNARKs workflow in a high-level

## Computational integrity

Alice proves to Bob correct execution of a program  $P$  on input  $x$ , with output  $y \leftarrow P(x)$ .

- **The cost to Bob must be far smaller than the cost of simply executing  $P$  himself.**
- Example: zk-rollup for Ethereum.

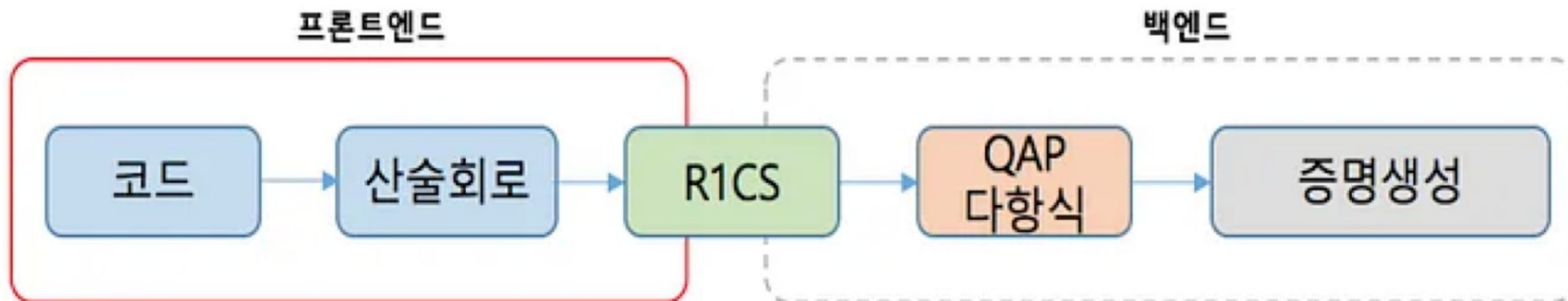
## Privacy

Alice proves to Bob that she knows secret input  $x$  with  $y \leftarrow P(x)$ .

- **Bob learns nothing about  $x$ .**
- Example: private transactions in ZCash.

[https://www.youtube.com/watch?v=Eva2IHFI\\_rl&t](https://www.youtube.com/watch?v=Eva2IHFI_rl&t)

# SNARKs workflow in a high-level



**프론트엔드(Frontend):** 함수에 대해 산술 회로식을 만드는 과정으로, 함수에 대해서 **Circuit**을 만들고 이를 산술 회로식(**polynomial**)으로 바꾸는 과정이다.


**백엔드(Backend):** **Circuit Proof**를 생성하는 과정이다. **ZKP**에서는 산술회로식의 유효성을 증명하기 위해 증명자가 검증자에게 식 전체를 직접 보내지 않는다. 커밋과 같은 과정을 통해서 산술 회로식을 짧은 값의 **commitment**로 만들고 이를 증명값으로 검증자에게 전송한다.



# Use cases: ZKML

gubsheep.eth

### Six Moonshot ZK Applications

 gubsheep

Oct 28, 2021

9

2

In the last three years, we have seen an explosion in application-level innovation on platforms like Ethereum. However, recent work in DeFi and decentralized governance has often focused on iterating on or permuting existing mechanics—in the most uninspiring cases, as Vitalik put it, "giving people tokens, and letting people earn yield on their tokens, and tokenizing the yield, and earning yield on the yield tokens...".


While novel project ownership models, improvements on wallet or dapp UX, and experiments with new community incentivization mechanisms are important for pushing adoption forwards, they are less likely to move the bottom line on what is *fundamentally* possible with decentralized tech. To raise the ceiling on what decentralized applications are even capable of doing, we need to leverage *new technology*.

We believe that new tech like zero-knowledge cryptography will play a key role in a next generation of Ethereum applications. ZK crypto is a powerful and general-purpose tool enabling a wide variety of scalability and privacy primitives. Those who have read our essay on crypto-native gaming will find this thesis familiar: **the next step function in Ethereum apps won't come from doing existing stuff better—it will come from being able to do new things that weren't possible before.**

purpose tool enabling a wide variety of scalability and privacy primitives. Those who have read our essay on crypto-native gaming will find this thesis familiar: **the next step function in Ethereum apps won't come from doing existing stuff better—it will come from being able to do new things that weren't possible before.**

Here are six ZK projects we're excited about, ranging from "prototype in production" to "sci-fi moonshot with a 10% chance of working." Much of this builds off of the excellen  
Foundat  
contract

## Open-sourcing zkml: Trustless Machine Learning for All

 Daniel Kang · Following

5 min read · Apr 4

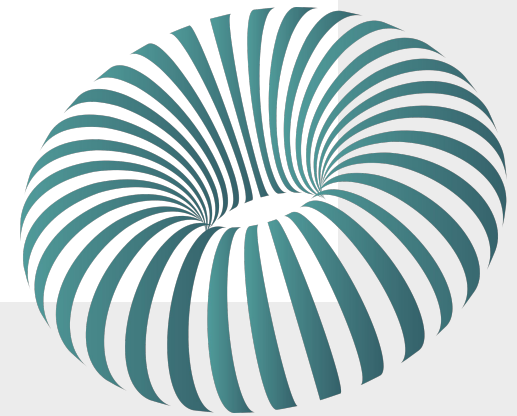
82

We're excited to announce the open-source release of zkml, our framework for producing zero-knowledge proofs of ML model execution. zkml builds on our earlier paper on scaling zero-knowledge proofs to ImageNet models but contains many improvements for usability, functionality, and scalability. With our improvements, we can verify execution of models that achieve 92.4% accuracy on ImageNet, a 13% improvement compared to our initial work! zkml can also prove an MNIST model with 99% accuracy **in four seconds**.

<https://gubsheep.substack.com/p/six-moonshot-zk-applications>



# A brief intro to gnark API



# What is gnarks?

gnark is a fast, open-source zk-SNARKs library written in Golang with Apache 2.0 license which supports the Groth16 and PlonK-algorithm implemented zk-SNARK API.

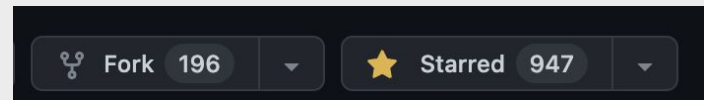
Users can write circuits using the Simple Golang API which gnark takes care of the rest; and the library exposes zk-SNARKs function (Setup/Prove and Verify) in a simple way.

<https://github.com/ConsenSys/gnark>

<https://docs.gnark.consensys.net/>

<https://github.com/ConsenSys/gnark-crypto>

<https://abit.ly/gnark-intro-korean>



# gnark workflow in a high-level

- PLONK
- Groth16

Frontend  
(or, writing a circuit)

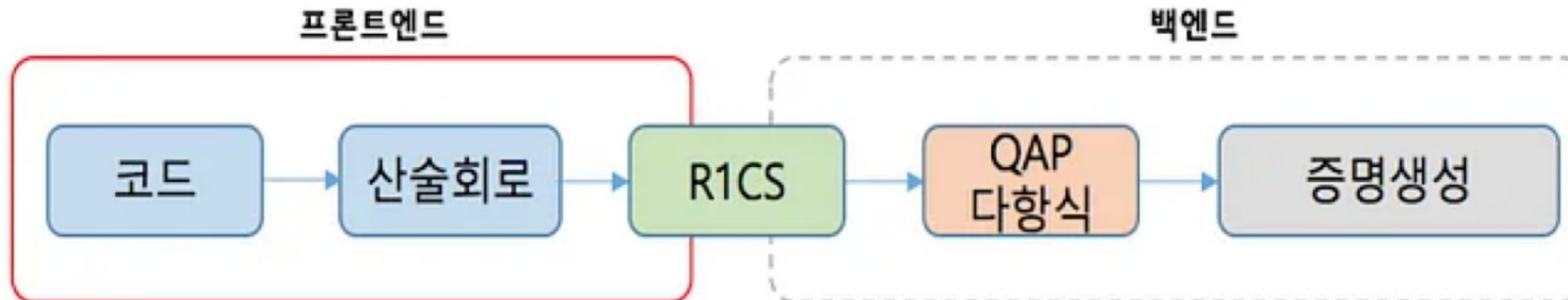
Backend  
(proof generation/verification)

- KZG
- FRI
- Plookup

- BN254
- BLS12

Pairing and elliptic  
curve cryptography

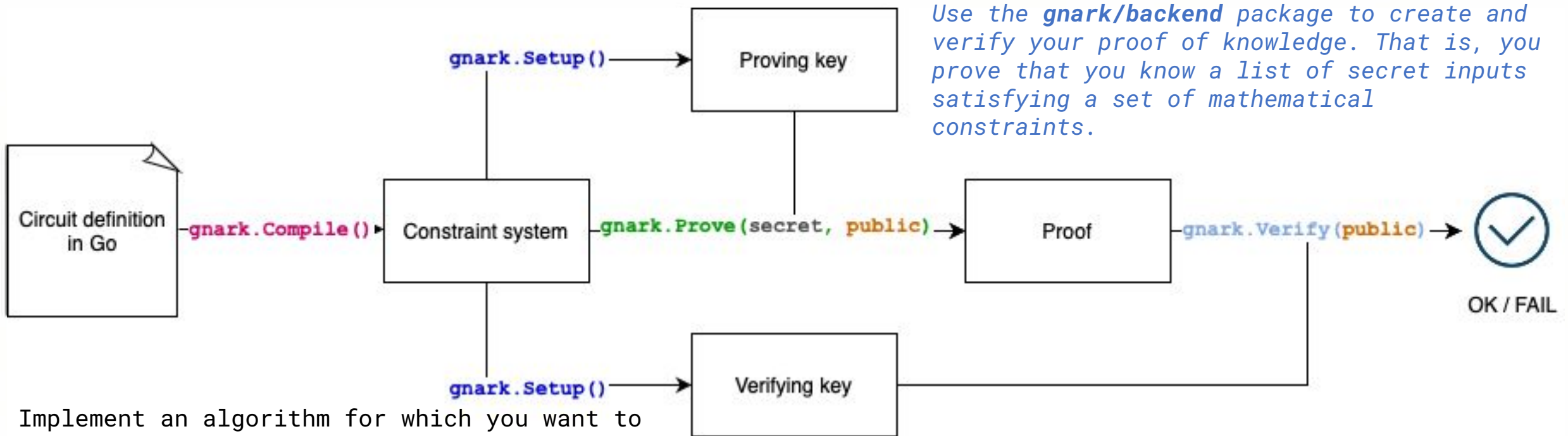
Finite Field Arithmetic  
(with big integer library)



# gnark workflow in a high-level

Use the *gnark/frontend* package to translate the algorithm into a set of mathematical constraints.

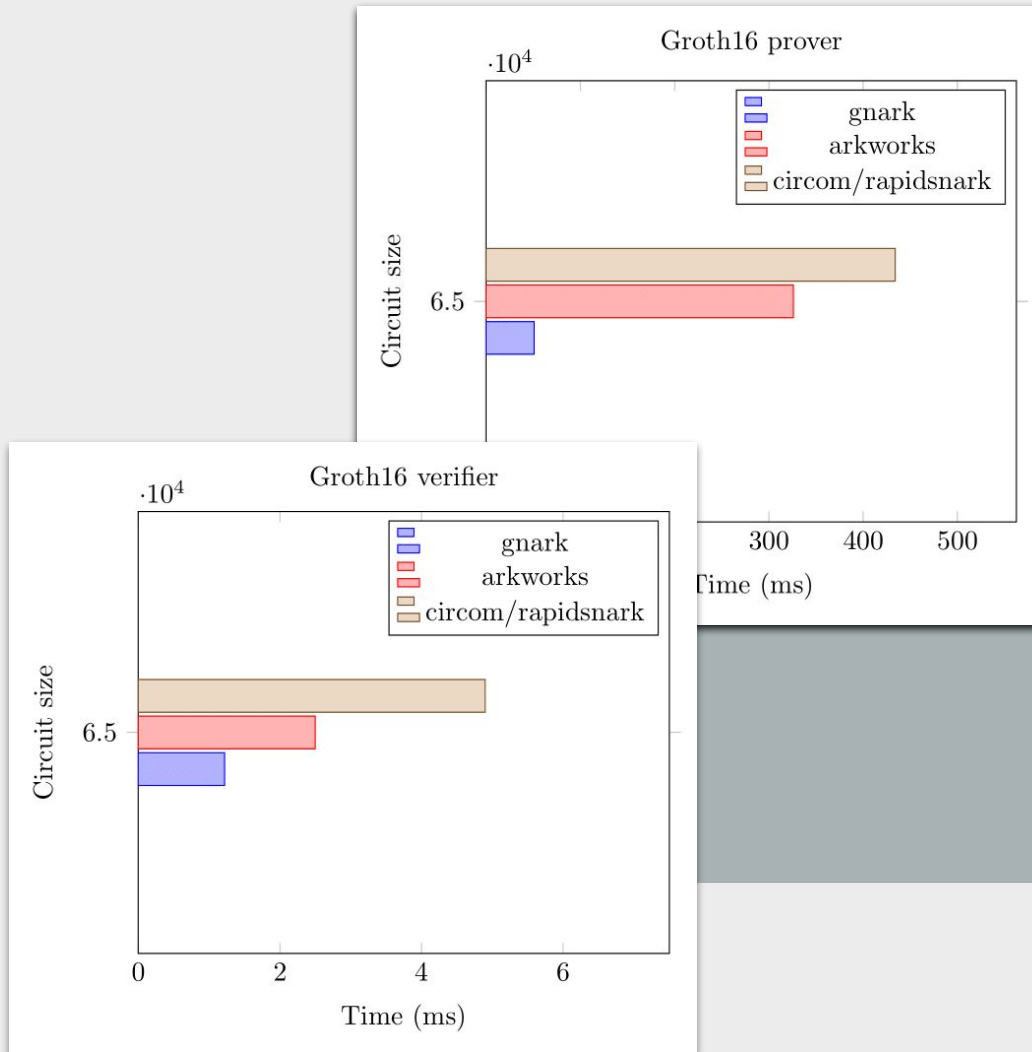
```
pk, vk, err := groth16.Setup(ccs)
proof, err := groth16.Prove(ccs, pk, witness)
err := groth16.Verify(proof, vk, publicWitness)
```



Implement an algorithm for which you want to prove and verify execution.

```
ccs, err = frontend.Compile(ecc.BN254.ScalarField(), r1cs.NewBuilder, &c)
ccs, err = frontend.Compile(ecc.BLS12_381.ScalarField(), scs.NewBuilder, &c)
```

# Pro side of gnark...

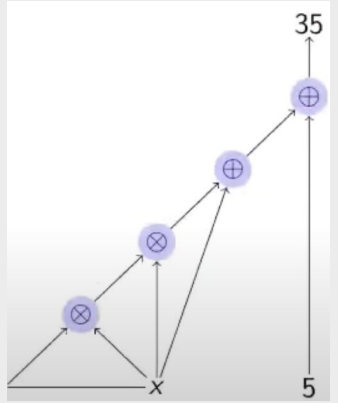


- ✓ No DSL, plain Go, no other third party dependencies. Integrating golang applications with the wizard of ZKP technologies!
- ✓ Compiles large size of circuit in seconds  
Write circuit once, use different curves and backends
- ✓ Fast performance than Arkworks(Rust) or Circom DSL
- ✓ Would able to debug and test as they would with any other Go program!

<https://www.youtube.com/watch?v=jL0vRNBBFns>

# Writing a simple circuit

$x^3 + x + 5 = 35$  (where  $x = 3$ )



```
type MyCircuit struct {  
    X frontend.Variable `gnark:"x"`  
    Y frontend.Variable `gnark:",public"`  
}
```

```
func (circuit *MyCircuit) Define(api frontend.API) error {  
    x3 := cs.Mul(circuit.x, circuit.X, circuit.X)  
    cs.AssertIsEqual(circuit.Y, cs.Add(x3, circuit.X, 5))  
    return nil  
}
```

$$x^3 + x + 5 = y$$

# Use that circuit, off-chain to create a proof

On ec.BN254 + Groth16, gnark can export the *groth16.VerifyingKey* as a solidity contract for Ethereum.

```
var cubicCircuit Circuit

// compiles our circuit into a R1CS
r1cs, _ := frontend.Compile(ecc.BN254, backend.GROTH16, &cubicCircuit)
sparseR1CS, _ := frontend.Compile(ecc.BLS12_381, backend.PLONK, &cubicCircuit)

// in test files:
assert := groth16.NewAssert(t)

var witness Circuit
witness.X.Assign(3)
witness.Y.Assign(35)
assert.ProverSucceeded(r1cs, &witness)

// in main:
pk, vk, _ := groth16.Setup(r1cs)
proof, _ := groth16.Prove(r1cs, pk, &witness)
err := groth16.Verify(proof, vk, &witness)
```

for loop to use the conventional one; while Select statement is here for if-else divergence (flattened!)

```
// Select if b is true, yields i1 else yields i2
func (cs *ConstraintSystem) Select(b
Variable, i1, i2 interface{}) Variable {
    ...
}
```



# Use Gadgets like any other Golang dependencies: Easy to debug!

```
func (circuit *mimcCircuit) Define(api frontend.API) error {  
    // ...  
    hFunc, _ := mimc.NewMiMC(api.Curve())  
    computedHash := hFunc.Hash(cs, circuit.Data)  
    // ...  
}
```

# Creating a proof from witnesses

```
type Circuit struct {
    X frontend.Variable
    Y frontend.Variable `gnark:",public"`
}

assignment := &Circuit {
    X: 3,
    Y: 35,
}

witness, _ := frontend.NewWitness(assignment, ecc.BN254)

// use the witness directly in zk-SNARK backend APIs
groth16.Prove(cs, pk, witness)

// test file --> assert.ProverSucceeded(cs, &witness)
```

## ... or creating directly for Solidity 😄

```
// 1. Compile (Groth16 + BN254)
cs, err := frontend.Compile(ecc.BN254, r1cs.NewBuilder, &myCircuit)
```

```
// 2. Setup
pk, vk, err := groth16.Setup(cs)
```

```
// 3. Write solidity smart contract into a file
err = vk.ExportSolidity(f)
```

```
// 1. one time setup
publicData, _ := plonk.Setup(cs, ...) // WIP
```

```
// 2. Proof creation
proof, err := plonk.Prove(r1cs, publicData, witness)
```

```
// 3. Proof verification
err := plonk.Verify(proof, publicData, publicWitness)
```

# Test in ~~Prod~~ Playground

The gnark playground

```
1 // Welcome to the gnark playground!
2 package main
3
4 import "github.com/consensys/gnark/frontend"
5
6 // gnark is a zk-SNARK library written in Go. Circuits are regular structs.
7 // The inputs must be of type frontend.Variable and make up the witness.
8 // The witness has a
9 //   - secret part --> known to the prover only
10 //   - public part --> known to the prover and the verifier
11 type Circuit struct {
12     X frontend.Variable `gnark:"x"` // x --> secret visibility (default)
13     Y frontend.Variable `gnark:",public"` // Y --> public visibility
14 }
15
16 // Define declares the circuit logic. The compiler then produces a list of constraints
17 // which must be satisfied (valid witness) in order to create a valid zk-SNARK
18 func (circuit *Circuit) Define(api frontend.API) error {
19     // compute x**3 and store it in the local variable x3.
20     x3 := api.Mul(circuit.X, circuit.X, circuit.X)
21
22     // compute x**3 + x + 5 and store it in the local variable res
23     res := api.Add(x3, circuit.X, 5)
24
25     // assert that the statement x**3 + x + 5 == y is true.
26     api.AssertIsEqual(circuit.Y, res)
27     return nil
28 }
29 -- witness.json --
30 {
31     "x": 3,
32     "Y": 35
33 }
```

Run gnark [[github](#)|[docs](#)] in your browser.

<https://play.gnark.io/>

# Examples: verifying any integers' exponentiate operations

```
package exponentiate

import (
    "testing"

    "github.com/consensys/gnark/test"
)

func TestExponentiateGroth16(t *testing.T) {
    assert := test.NewAssert(t)

    var expCircuit Circuit

    assert.ProverFailed(&expCircuit, &Circuit{
        X: 2,
        E: 12,
        Y: 4095,
    })

    assert.ProverSucceeded(&expCircuit, &Circuit{
        X: 2,
        E: 12,
        Y: 4096,
    })
}
```

## Examples: verifying any integers' exponentiate operations

```
package exponentiate

import (
    "github.com/consensys/gnark/frontend"
    "github.com/consensys/gnark/std/math/bits"
)

// Circuit  $y = x^e$ 
// only the bitSize least significant bits of e are used
type Circuit struct {
    // tagging a variable is optional
    // default uses variable name and secret visibility.
    X frontend.Variable `gnark:",public"`
    Y frontend.Variable `gnark:",public"`

    E frontend.Variable
}
```

## Examples: verifying any integers' exponentiate operations

```
// Define declares the circuit's constraints
// y == x**e
func (circuit *Circuit) Define(api frontend.API) error {

    // A constant bitSize is defined and set to 8.
    // This would be used to represent the bit size of the exponent e.
    const bitSize = 8

    output := frontend.Variable(1)
    // The bits variable is assigned to the binary representation of the
    exponent e,
    // represented in bitSize bits. The bits.ToBinary method does this
    conversion.
    bits := bits.ToBinary(api, circuit.E, bits.WithNbDigits(bitSize))

    ...
}
```

## Examples: verifying any integers' exponentiate operations

...

```
for i := 0; i < len(bits); i++ {
    if i != 0 {
        // If it is not the first iteration, the current output is
        squared by multiplying it with itself.
        output = api.Mul(output, output)
    }
    multiply := api.Mul(output, circuit.X)
    // The output variable is updated using a ternary-style
    operation. It is set to multiply if the current bit is 1 and remains the
    same if the current bit is 0. This corresponds to the method of
    exponentiation by squaring.
    output = api.Select(bits[len(bits)-1-i], multiply, output)
}

api.AssertIsEqual(circuit.Y, output)

return nil
}
```





# Limitations & Further Expectations

# Limitations of gnark

**The algorithm on gnark-crypto manipulate millions (if not billions) of field elements.**

Interface indirection at this level, plus garbage collection indexing takes a heavy toll on perf.

Need to derive (mostly) identical code for various moduli and curves, with consistent APIs.

Generics can't prove the solution since despite providing great flexibility and reusability in code it introduce significant performance overhead. This can be a limitation for high-performance computing requirements such as cryptographic operations in gnark.

**The lack of support for floating-point arithmetic:** The SNARK arithmetic usually happens in the scalar field of the underlying elliptic curve. The library have emulated emulation for non-native fields (using field emulation with a good modulus and doing fixed-point arithmetic.), but not for floats. [would be good contribution point? :D](#)

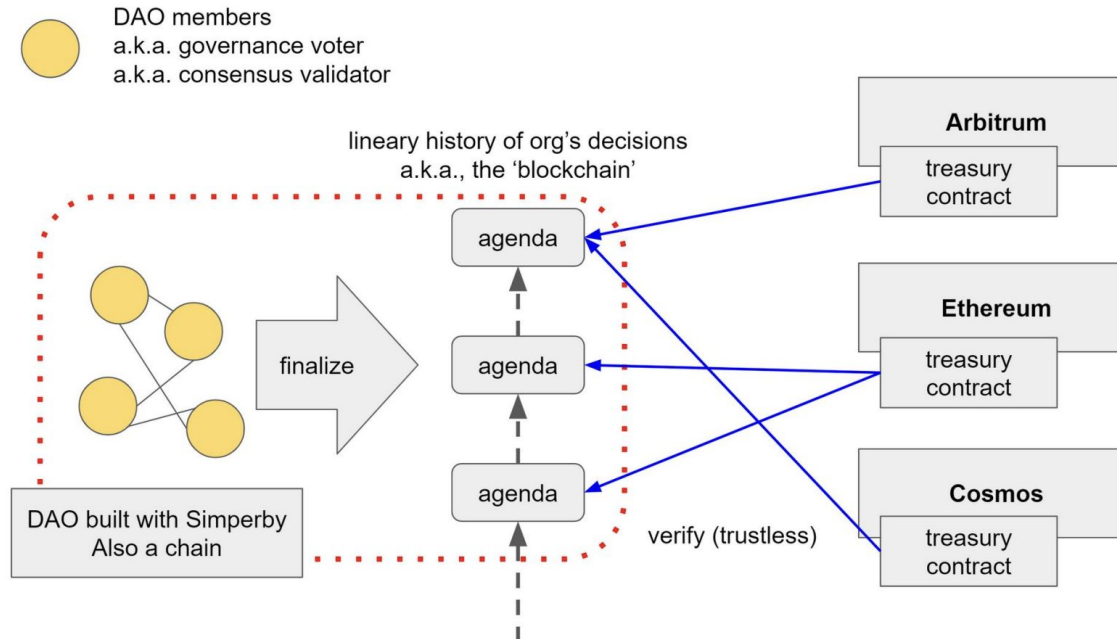
**Array Size Constraints:** In the context of defining and compiling circuits in gnark, array sizes need to be known at compile time.



# Q & A

Yet Nice Explanations in Korean:  
<https://abit.ly/gnark-intro-korean>

# One More Thing... Simperby



2023 오픈소스 컨트리뷰션 아카데미에서 **Simperby** 프로젝트가 멘티를 찾고 있습니다!  
컨센서스, **P2P** 네트워크, 분산처리, **Rust** 비동기 프로그래밍에 관심있는 분  
환영합니다.

링크:

<https://www.oss.kr/notice/show/7b08b07d-b566-42cb-9193-da299ea4197c>



**E.O.D.**  
**Thank you!**