

# zk-SNARK

최승주

<https://youtu.be/pP-Q5z4g2io>

# Contents

zk-SNARK

기타 추가 설명



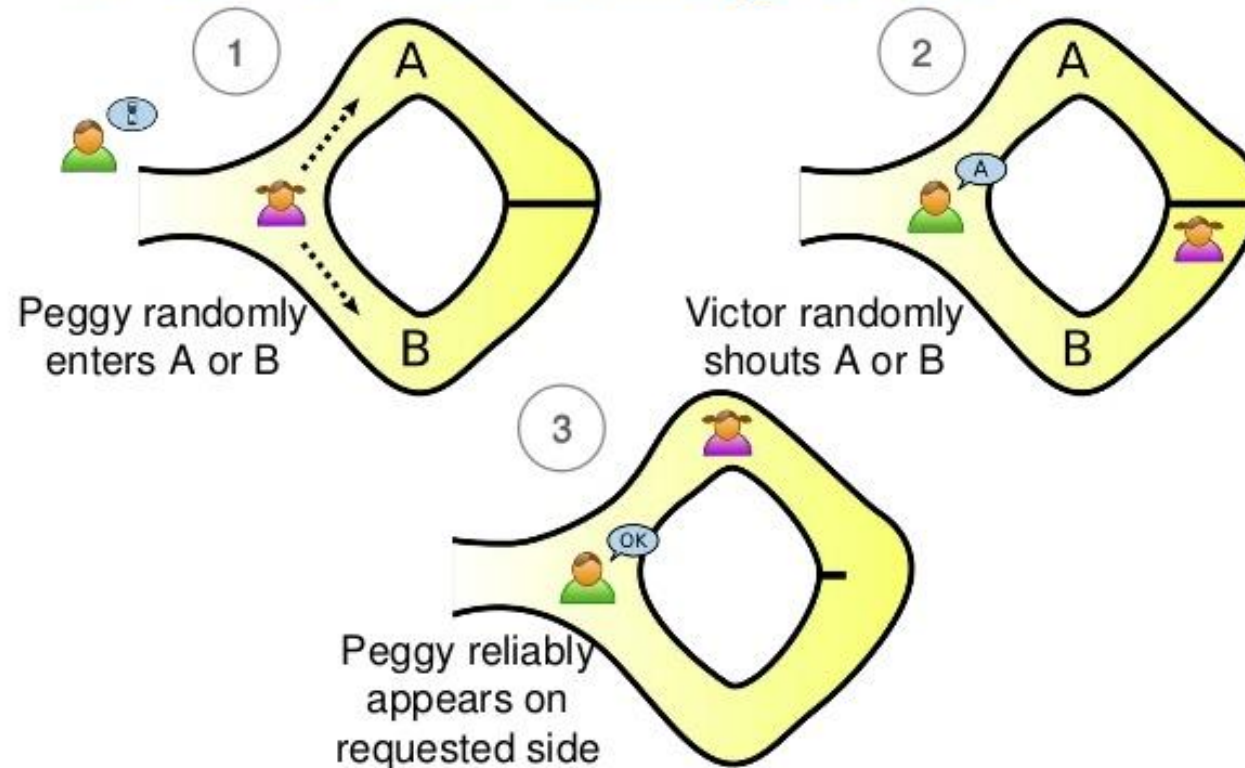
CryptoCraft LAB

# zk-SNARK

- Zero Proof Knowledge(영지식 증명)
  - 자신이 알고 있는 특정 정보는 보여주지 않으면서  
본인이 해당 정보를 알고 있다는 것을 증명하는 증명 방식

# zk-SNARK

## A cave with a magic door



출처: <https://www.slideshare.net/IgnatKorchagin/enforcing-web-security-and-privacy-with-zero-knowledge-protocols>

# zk-SNARK

- 스도쿠 예시

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

# zk-SNARK

- 4 x 4 버전 스도쿠
- 참여자: Prover, Verifier
  - Prover: 스도쿠 답을 공개하지 않고, 해당 문제가 풀렸다는 것을 증명
  - Verifier: Prover가 스도쿠 답을 알고 있다는 것을 확인하고 싶어함

# zk-SNARK

1. 해답 섞기
2. 섞는 패턴과 섞어서 나온 스도쿠 가리기
3. Verifer가 원하는 해답 공개
4. 1~3 반복

# zk-SNARK

## 0. 스도쿠 생성

1	2	3	4
4	3	2	1
2	1	4	3
3	4	1	2



# zk-SNARK

## 1. 스도쿠 섞기

1	2	3	4
4	3	2	1
2	1	4	3
3	4	1	2

섞는 규칙

1	→	4
2	→	3
3	→	1
4	→	2



4	3	1	2
2	1	3	4
3	4	2	1
1	2	4	3

# zk-SNARK

## 2. 섞는 규칙과 섞어서 나온 스도쿠 가리기

섞는 규칙

?	→	?
?	→	?
?	→	?
?	→	?

?	?	?	?
?	?	?	?
?	?	?	?
?	?	?	?

## 3. Verifier가 원하는 해답 공개


- 특정 열을 공개
- 특정 행을 공개
- 특정 구역을 공개
- 임의의 한 지점을 공개

# zk-SNARK

## 3. Verifier가 원하는 해답 공개

섞는 규칙

?	→	?
?	→	?
?	→	?
?	→	?



?	3	?	?
?	1	?	?
?	4	?	?
?	2	?	?

# zk-SNARK

- 한번 물어봤으면 또 다른 규칙으로 섞어서 공개
  - 만약 다른 방식으로 섞지 않고 똑같은 것을 계속 쓰면 해답을 유추할 위험이 있음
- Verifier는 계속해서 질문을 더 한다.
  - 우연히 해답을 한번 맞은 것일 수도 있기에 여러 번 계속해서 질문해서 Prover가 해답을 갖고 있다는 사실을 납득한다.

# zk-SNARK

- 특정 **문제**에 대해 **해답**을 알고 있다는(true) 것을 보이면 됨
- 스도쿠:
  - 섞인 스도쿠(공개)
  - 질문(해답을 아는지)
  - 질문에 대한 답변 제시(비공개 해답으로 답을 만들어 해줌)

# zk-SNARK

- 공개된 값 :  $x$
- 문제 :  $C$
- 해답 :  $w$  (witness – 비밀 정보)

# zk-SNARK

- $C(x, w) == \text{true}$  가 나오게 만들면 증명 성공
- 암호 학적으로 이러한 문제와 답을 갖는 것: Hash
- 공개되어 있는 값  $x$ 는  $w$ 의 **hash** 값

```
Function C(x,w){  
    return (sha256(w) == x);  
}
```



# zk-SNARK

## zk-SNARK 3가지 알고리즘

1.  $\text{Generator}(C_{\text{circuit}}, \lambda) \rightarrow \text{pk}(\text{proof key}), \text{vk}(\text{verifying key})$
2.  $\text{Prover}(\text{pk}, x_{\text{public input}}, W_{\text{secret input}}) \rightarrow \text{proof}$
3.  $\text{Verifier}(\text{vk}, x, \text{proof}) \rightarrow \text{true or false}$

# zk-SNARK

## zk-SNARK 3가지 알고리즘

1. Generator( $C_{\text{circuit}}$ ,  $\lambda$ ) → 증명 요구자(검증자)
2. Prover( $pk$ ,  $x_{\text{public input}}$ ,  $W_{\text{secret input}}$ ) → 증명 하려는 사람
3. Verifier( $vk$ ,  $x$ ,  $\text{proof}$ ) → 증명 요구자(검증자)

# zk-SNARK

## 1. Generator( $C$ circuit, $\lambda$ )

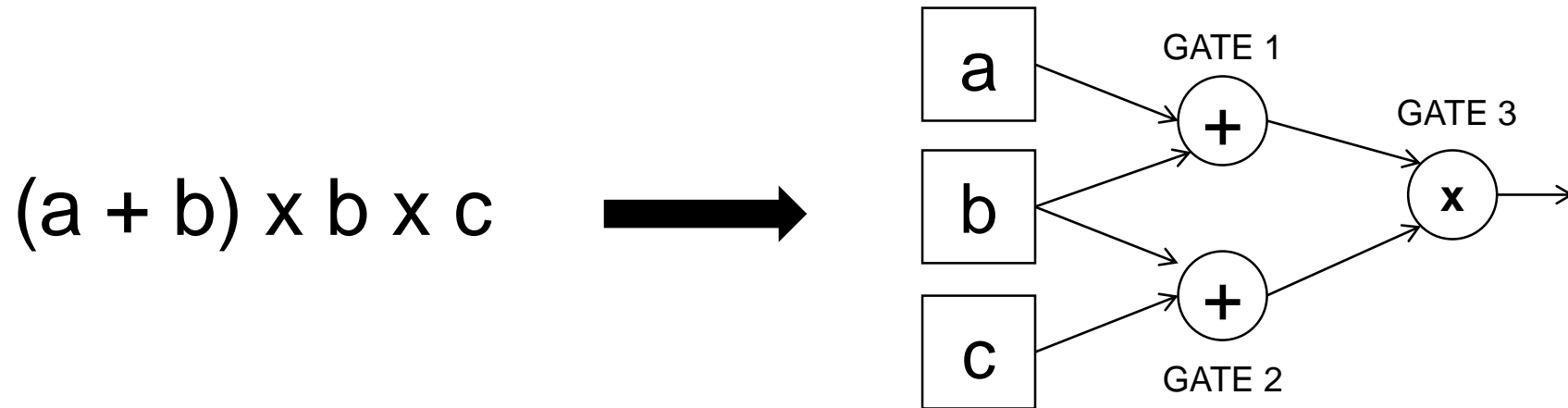
- Input 값: 문제와 랜덤 값인  $\lambda$ 를 받음 (문제는 공개,  $\lambda$ 는 절대 비공개)
- Output 값: proof 키와 verifying 키 생성 (키는 전부 공개)

C의 예시

```
Function C(x,w){  
    return (sha256(w) == x);  
}
```

# zk-SNARK

- Generator( $C_{\text{circuit}}$ ,  $\lambda$ )
- $C$ 
  - 컴퓨팅  $\rightarrow$  연산 회로  $\rightarrow$  R1CS  $\rightarrow$  QAP
  - 연산 회로: 주어진 수식을 사칙 연산 단위로 풀어 쓴 형태



# zk-SNARK

- Generator( $C$  circuit,  $\lambda$ )

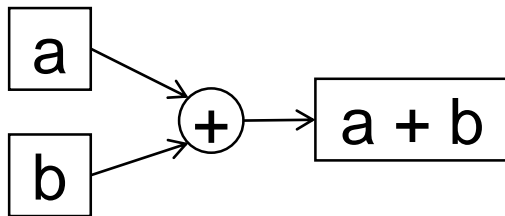
- $C$

- 컴퓨팅  $\rightarrow$  연산 회로  $\rightarrow$  **R1CS**  $\rightarrow$  QAP

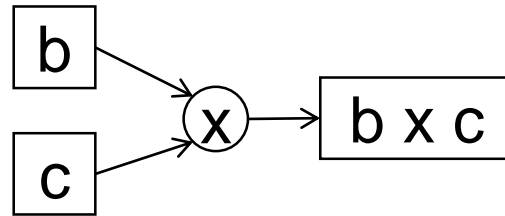
각 게이트에 대해 주어진 입출력 값이 유효한지 확인

각각의 게이트 별 제약사항의 유효성을 일일이 검사해야함

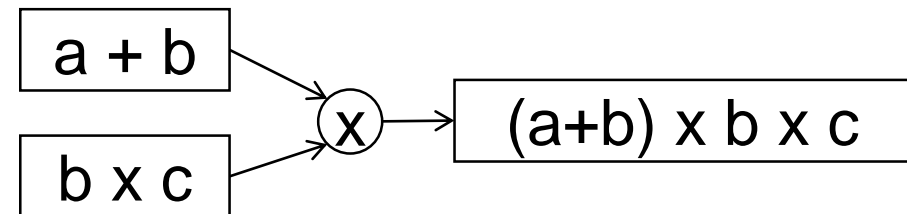
GATE 1



GATE 2



GATE 3



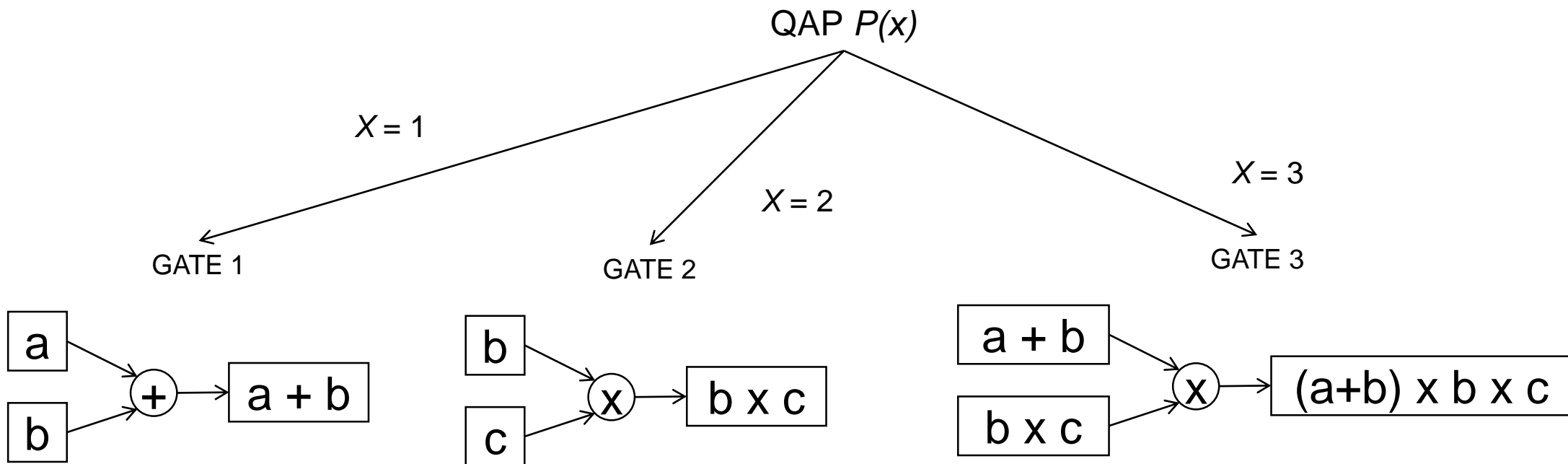
# zk-SNARK

- C
  - 컴퓨팅  $\rightarrow$  연산 회로  $\rightarrow$  R1CS  $\rightarrow$  **QAP**
  - 하나씩 확인해야 하는 비효율성을 개선하기 위해 QAP(Quadratic Arithmetic Program) 도입
  - 임의의 게이트 값을 변수에 대입  $\rightarrow$  해당 게이트 값에 해당하는 게이트의 R1CS 형태가 나오는 다항식으로 표현

# zk-SNARK

- C

- 컴퓨팅  $\rightarrow$  연산 회로  $\rightarrow$  R1CS  $\rightarrow$  **QAP**
- 임의의 게이트 값을 변수에 대입  $\rightarrow$  해당 게이트 값에 해당하는 게이트의 R1CS 형태가 나오는 다항식으로 표현



# zk-SNARK

- C
  - 컴퓨팅  $\rightarrow$  연산 회로  $\rightarrow$  R1CS  $\rightarrow$  **QAP**
  - 임의의 게이트 값을 변수에 대입  $\rightarrow$  해당 게이트 값에 해당하는 게이트의 R1CS 형태가 나오는 다항식으로 표현
- 목표 다항식(T) - P가 T로 나누어 떨어짐을 보이면 유효성 검증 가능

$$T(x) = (x - 1)(x - 2)(x - 3)$$



# zk-SNARK

- Generator

```
template<typename ppzksnark_ppT>
r1cs_ppzksnark_keypair<ppzksnark_ppT> generate_keypair()
{
    typedef Fr<ppzksnark_ppT> FieldT;
    protoboard<FieldT> pb;
    payment_in_out_gadget<FieldT> g(pb);
    g.generate_payment_in_out_constraints();
    const r1cs_constraint_system<FieldT> constraint_system = pb.get_constraint_system();
    cout << "Number of R1CS constraints: " << constraint_system.num_constraints() << endl;
    return r1cs_ppzksnark_generator<ppzksnark_ppT>(constraint_system);
}
```

# zk-SNARK

## 2. $P(pk, x, w)$

- Input 값:  $pk$ (증명 키),  $x$ (공개),  $w$ (비공개)
- Output 값:  $prf$  (증명자가  $w$ 를 안다는 것을 증명한 값 - proof)

$$prf = P(pk, x, w)$$

# zk-SNARK

## 3. $V(vk, x, prf)$

- Input 값:  $vk$ (검증 키),  $x$ (공개),  $prf$ (공개)
- Output 값: True 또는 False

$boolean = V(vk, x, prf)$

# zk-SNARK

1.  $\text{Generator}(C_{\text{circuit}}, \lambda) \rightarrow \text{pk}(\text{proof key}), \text{vk}(\text{verifying key})$
  2.  $\text{Prover}(\text{pk}, x_{\text{public input}}, W_{\text{secret input}}) \rightarrow \text{proof}$
  3.  $\text{Verifier}(\text{vk}, x, \text{proof}) \rightarrow \text{true or false}$
- Lambda: - 그리스 언어로 숫자 30
    - 파이썬 등의 프로그래밍 언어에서 함수를 정의할 때에 사용
    - 선형대수학에서 고윳값 – 변화를 취해도 변하지 않은 값

# zk-SNARK

1.  $\text{Generator}(C_{\text{circuit}}, \text{lambda}) \rightarrow \text{pk}(\text{proof key}), \text{vk}(\text{verifying key})$
  2.  $\text{Prover}(\text{pk}, x_{\text{public input}}, W_{\text{secret input}}) \rightarrow \text{proof}$
  3.  $\text{Verifier}(\text{vk}, x, \text{proof}) \rightarrow \text{true or false}$
- Lambda를 알아버리면 가짜 증명을 만들어 버릴 수 있다.
  - Lambda의 관리가 중요(한번 쓰고 제거)

# zk-SNARK

1. Generator( $C_{\text{circuit}}$ ,  $\lambda$ ) → 증명 요구자(검증자)/신뢰 기관
2. Prover( $pk$ ,  $x_{\text{public input}}$ ,  $w_{\text{secret input}}$ ) → 증명 하려는 사람
3. Verifier( $vk$ ,  $x$ ,  $\text{proof}$ ) → 증명 요구자(검증자)

# zk-SNARK 예시

## ERC20 Token

- 코인
  - 블록체인 상에서 존재하는 디지털 장부에 기록되는 숫자
- 토큰
  - 스마트 컨트랙트에서 존재하는 화폐

# zk-SNARK 예시

- 토큰으로 기능을 하는데 필요한 함수들 구현을 요청
  - 사람들이 모여서 기능 구현
  - 요청 번호 20번 – ERC20
  - 토큰을 보내고 받는 등의 매우 기본적인 함수들 구현
- 제일 잘 쓰이는 토큰: ERC20, ERC721



# zk-SNARK 예시

## ERC20 Token

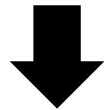
- 스마트 컨트랙트 상에서 아래와 같은 함수로 관리  
mapping (address => uint256) balances;
- uint256: address 주소가 갖고 있는 토큰의 양 기록  
A → 10 Token  
B → 15 Token...

# zk-SNARK 예시

## ERC20 Token with zk-SNARK

- 토큰을 얼마를 보내고, 얼마를 갖고 있는지를 감춘다.
  - 받는 사람과 보내는 사람은 감추지는 않음

mapping (address => uint256) balances;



mapping (address => bytes32) balanceHashes; (해시 값들은 공개)

# zk-SNARK 예시

## ERC20 Token with zk-SNARK

- 토큰을 전송할 때 확인하는 요소
  - 보내려고 하는 양이 본인이 갖고 있는 양 이하인가
  - 본인 잔고보다 초과해서 보내려고 하고 있지는 않은가

`balances[fromAddress] >= value`

# zk-SNARK 예시

## ERC20 Token with zk-SNARK

보내는 쪽

- 비밀정보( $w$ ): 갖고 있던 토큰의 양, 보내는 양
- 공개정보( $x$ ):
  - 갖고 있던 토큰의 양의 해시
  - 보내는 양의 해시
  - 보내고 난 뒤 토큰의 양 해시 값

# zk-SNARK 예시

## ERC20 Token with zk-SNARK

받는 쪽

- 비밀정보( $w$ ): 갖고 있던 토큰의 양, 받는 양
- 공개정보( $x$ ):
  - 갖고 있던 토큰의 양의 해시
  - 받는 양의 해시
  - 받고 난 뒤 토큰의 양 해시 값

# zk-SNARK 예시

## ERC20 Token with zk-SNARK

보내는 쪽

```
function senderFunction(x, w) {  
    return (  
        w.senderBalanceBefore > w.value &&  
        sha256(w.value) == x.hashValue &&  
        sha256(w.senderBalanceBefore) == x.hashSenderBalanceBefore &&  
        sha256(w.senderBalanceBefore - w.value) == x.hashSenderBalanceAfter  
    )  
}
```

# zk-SNARK 예시

## ERC20 Token with zk-SNARK

받는 쪽

```
function receiverFunction(x, w) {  
    return (  
        sha256(w.value) == x.hashValue &&  
        sha256(w.receiverBalanceBefore) == x.hashReceiverBalanceBefore &&  
        sha256(w.receiverBalanceBefore + w.value) == x.hashReceiverBalanceAfter  
    )  
}
```

Q & A

