# ZK EVM Research

## Klaytn Dev Ambassador Research

***Disclaimer**

본 아티클은 클레이튼의 데브 엠베서더 리서치 프로그램의 활동 결과물로, 데브 엠베서더가 작성했습니다.
아티클 내 소개되는 프로젝트들과 아티클의 저작권 및 소유는 저자로 참여한 엠버서더와, 참조한 프로젝트 또는 기관들에게 있습니다. 본 아티클의은 블록체인 프로젝트의 기술 지식을 공유하는 데 기여하는 데 그 목적이 있으며, 특정 프로젝트에 대한 투자 추천이나 조언이 아님을 알려드립니다. 본문에 대한 수정 제안은 아래의 저자 /검토자 연락처로 의견 주시면, 검토후 반영 하도록 하겠습니다.

***저자**

- 김현우 (lewis.kim@klaytn.foundation) | Klaytn Dev ambassador, Klaytn Foundation
- 송민규 (mikekks123@gmail.com) | Klaytn Dev ambassador

***검토 및 수정**

- iron.cho (iron.cho@klaytn.foundation ) | Klaytn Foundation

klaytn

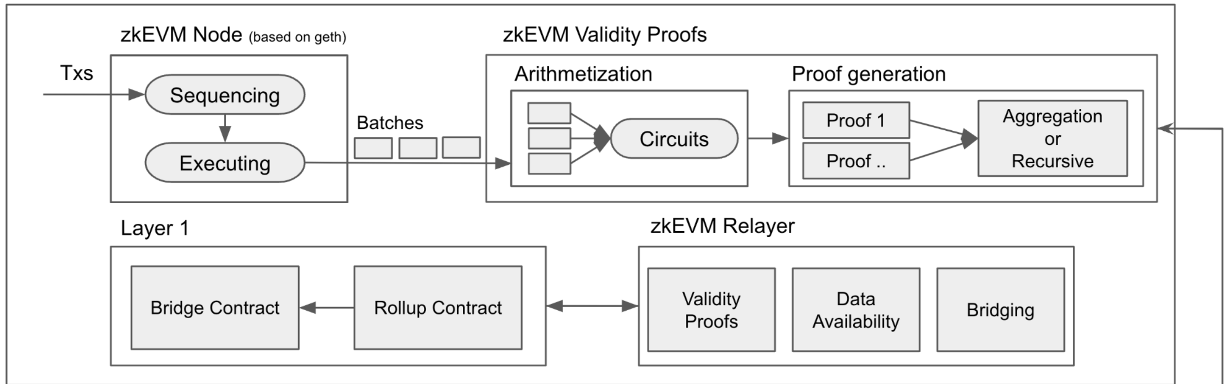
# 들어가며

블록체인의 생태계 확장과 Mass Adoption을 위해 확장성 문제 해결이 필요했고, Layer1 네트워크들이 여러 방안으로 확장성 문제를 해결하였다. 하지만 기존의 Monolithic 네트워크 구조에서는 탈중앙성, 보안성, 확장성의 문제를 동시에 해결하기에는 구조적인 한계가 있었다. 이를 해결하기 위해 블록체인의 Modular 네트워크 구조가 제시되었으며, 대표적인 Layer1인 이더리움은 Rollup Centric Model을 통해 해결을 하고자 한다. Rollup Centric이란 Layer2인 Rollup 네트워크들이 실행 계층이(Execution Layer) 되어서, Layer1를 대신하여 Layer2에서의 트랜잭션의 실행, 상태 변화들을 수행을하고, Layer1은 최종 보안 레이어(Settlement Layer)로서 Layer2에서의 실행한 상태들을 검증하고 일부 데이터 가용성(Data Availability)을 제공한다. 이러한 Rollup Centric 모델은 Layer1의 탈중앙화를 만족하면서 확장성을 해결할 수 있을 것으로 많은 커뮤니티에서 각광을 받고 있다. 이러한 Rollup Centric에서 실행 계층인 Layer2 들은 Optimism과 zk 로 분류가 되고, zk Rollup은 영지식 증명의 수학적인 검증으로 Layer1 에서의 검증 기간 단축과 검증의 완결성이라는 장점이 있다.

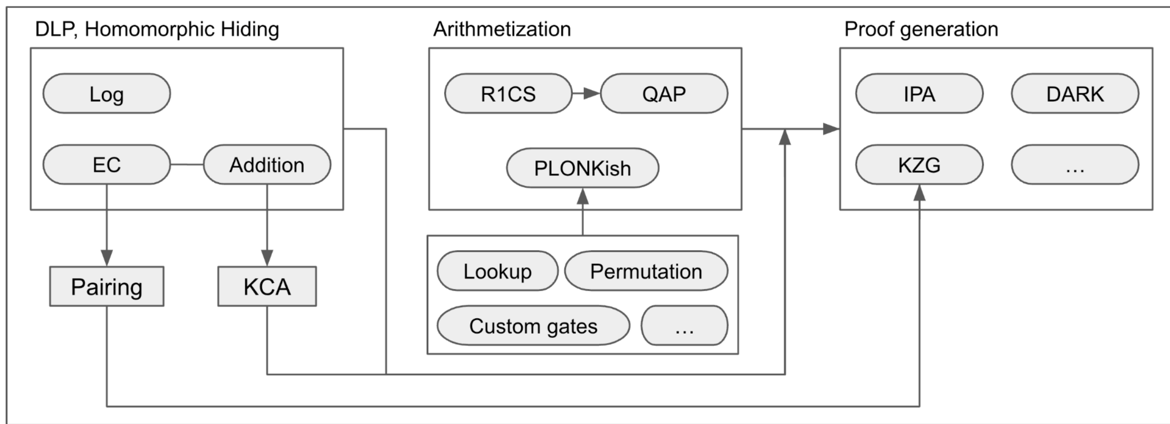
이러한 zk Rollup은 Dapp Service들을 Layer2인 실행계층에 온보딩하기 위해 EVM을 완벽히 호환하는 zkEVM을 연구개발 하고 있다. zkEVM은 Layer2에서 필요한 복잡한 zkp 연산 지원과 Layer1의 EVM을 호환하여, 기존의 Dapp 서비스들이 별도의 추가 작업없이 zkEVM에서 쉽게 실행 되는 것을 목표로 하며, zkEVM은 블록체인에서의 확장성 문제를 해결하는 하나의 커다란 게임체인저(Game Changer)로 예상된다.

본 리서치에서는 zkEVM의 기술들을 분석하는것을 목표로 하며, zkEVM기술들의 지식을 공유하는데 기여하고자 한다. zkEVM은 기술적으로 스마트 컨트랙트를 실행하는 EVM과, 실행된 트랜잭션의 Trace를 통해 산술 회로 및 증명을 생성하는 ZKP 기술로 나눌 수 있다. EVM 파트의 경우 Layer1과의 호환성을 위해 대부분 기존 Layer1의 코드를 기본으로 하며, 해시함수와 같은 일부 영지식 증명에 친화적이지 않은 연산은 지원하지 않는 프로토콜도 있다. 트랜잭션이 성공적으로 실행되면 해당 트랜잭션 내에서 발생한 연산 및 상태를 Trace의 형태로 확인할 수 있는데, 이를 기반으로 영지식 증명을 생성하게 된다. 영지식 증명의 생성을 어떤 zkp scheme을 통해 구현했는가에 따라 프로토콜의 성능 및 EVM 호환성이 결정되기 때문에, 이를 이해하는 것이 zkEVM을 이해하는 데에 필수적이라고 할 수 있다. 본 리서치에서는 이러한 zkEVM을 이해하기 위해 필요한 zkp의 기초적인 지식과 zkp scheme들을 알아보며, 마지막으로 zkEVM의 주요한 기술 스택들을 기술한다. 자세한 구성은 아래의 <그림 1>과 같다.

ZKEVM




ZKP Algebra




<그림1>

# 목차

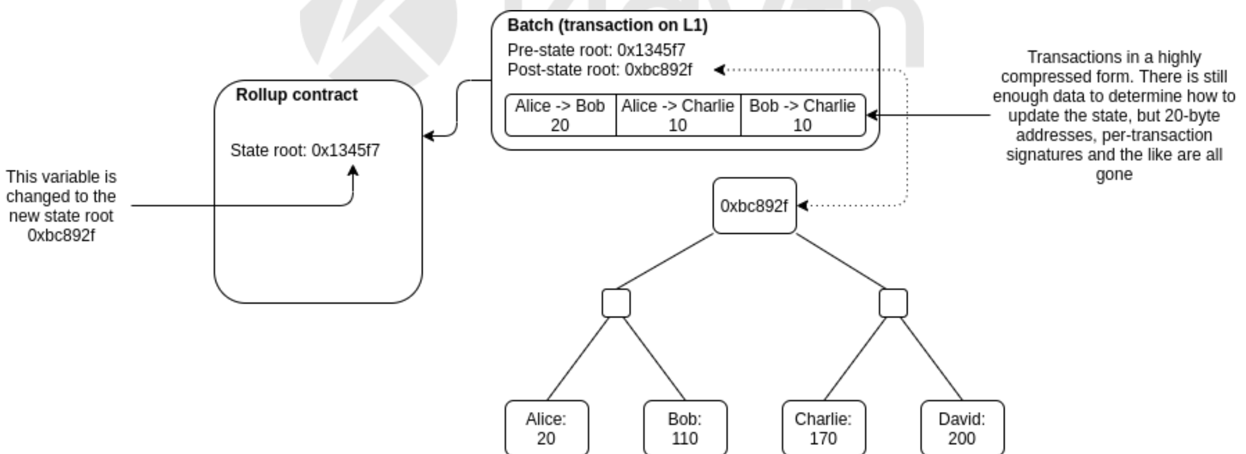
1. Layer 2:  Rollup 소개
2. zkEVM 배경 지식
    2.1. ZKP
    2.2. ZK-SNARK
    2.3. KZG Commitment
    2.4. PLONK
3. zKEVM : Scroll project Deep dive
    3.1. Halo 2
    3.2. Scroll circuit overview
    3.3. R/W ↔ Executions
    3.4. Build zkEVM circuit
    3.5. Verifying System
4. 결론

klaytn

# 1. Layer2 : Rollup 소개

Layer1 블록체인들의 확장성 문제를 해결하기 위해 다양한 리서치가 진행되고 어려 프로토콜들이 개발되어 왔다. 그 중 이더리움의 Layer 2를 기반으로 한 Rollup 중심의 모델이 커뮤니티에서 많은 긍정적인 반응을 받았다. Layer 2는 Layer 1과는 독립적인 네트워크로 , 구현에 따라 Rollup, Side chains 등으로 나뉘지만 기본적으로 computational cost가 큰 부분은 off-chain에서 실행한 후 이에 대한 데이터의 검증을 Layer 1에서 진행한다. 우리가 살펴볼 Rollup은 트랜잭션 실행을 off-chain으로 진행한 후 이에 대한 검증은 Layer 1에서 진행하는 방식으로, Layer 1의 탈중앙화와 보안을 유지하면서 트랜잭션 비용 및 처리 시간을 획기적으로 줄일 수 있었다.

Layer 2 은 Layer 1를 대신하여 트랜잭션의 실행과 상태변화를 하기 때문에 상태관리와 함께, 트랜잭션의 실행을 통해 상태를 업데이트하게 된다. 실제로 Layer 2에서 트랜잭션이 실행되고 나면 아래와 같이 State root가 새로 계산되게 되는데, Layer 1에 있는 Rollup contract는 Layer 2로 부터 받은 트랜잭션 데이터와 업데이트 된 State root를 통해 이를 검증한다.




< 그림 2 Rollup의 구조, 이미지 출처: https://vitalik.ca/general/2021/01/05/rollup.html>

하지만 Rollup contract가 Layer 2에서 발생한 트랜잭션을 매번 실행시킨 후 state root를 검증하는 것은 매우 비효율적이며 Layer 1의 확장성 문제를 해결하지 못한다. 따라서 이를 어떻게 효율적으로 검증할 것인지에 대한 문제가 발생하는데, 이 문제를 해결하기 위해 크게 다음 두 가지 Rollup 프로토콜이 개발되고 있다.

1) **Optimistic Rollup**: Rollup contract는 Layer 2의 상태를 검증할 수 있는 데이터를 받지만 매번 검증을 진행하지 않는다. 대신 정해진 기간 (주로 Challenge period라고 불리는)안에 특정 트랜잭션 실행이 잘못되었다는 fraud-proof가 다른 주체에 의해 제기될 수 있고 제기된 proof가 Rollup contract에서 검증되며, 만약 사실이라면 체인의 상태가 롤백된다(제출된 배치의 상태를 받아 들이지 않는다.). 즉, Challenge period 내에는 Layer 2체인의 상태가 finalize되었다고 할 수 없다.
2) **ZK Rollup**: Layer 2에서 발생한 트랜잭션이 올바르게 실행되었는지를 검증할 수 있는 ZKP[1]를 생성하여 Layer 1의 Rollup contract에서 이를 검증한다. 즉, Rollup contract에서 수행되는 검증 트랜잭션이 Layer 1에서 finalize 된다면 즉시 Layer 2의 상태도 finalize된다.

Rollup 프로토콜들은 기존 이더리움 개발자 및 유저들이 빠르게 온보딩 할 수 있는 환경을 제공하고자 노력하고 있다. 기존 이더리움의 인터페이스를 쉽게 가져올 수 있는 Optimistic Rollup과 달리, zk rollup은 ZKP에 친화적이지 않은 기존의 이더리움의 EVM구조로 인해 많은 연구개발을 하고 있다. EVM의 호환성을 온전히 맞추면서 ZKP검증을 위한 여러 ZKP 연산들이 필요한데 이것을 지원하는것이 zkEVM 이다. zkEVM이 완성되면, 기존의 Dapp서비스들은 별도의 추가 작업없이 Layer2로 온보딩이 가능하며, 이는 빠른 트랜잭션 처리와 낮은 수수료로 이더리움의 네트워크 신뢰성과 사용성 효과를 얻을 수 있다.

본 아티클에서는 ZKP의 기초 개념부터 실제 zkEVM의 구현을 이해하기 위해 다양한 ZKP 프로토콜을 자세히 살펴볼 것이다. 먼저 2장에서는 기초적인 영지식 증명에 대한 배경과 개념을 알아본 후, 대표적인 영지식 증명 프로토콜인 zk-SNARK와 PLONK를 실제 예시와 함께 자세히 살펴볼 것이다. 이후 3장에서는 실제 zkEVM 프로토콜인 Scroll에서 사용하는 Halo2[2]와 함께, 실제 이더리움의 트랜잭션을 통해 zkEVM의 회로 생성 과정에 대해 알아볼 것이다. 이를 통해 다양한 영지식 증명 프로토콜과 zkEVM의 동작 원리에 대해 이해를 가질 수 있을 것으로 기대한다.

[1] Zero-knowledge proof, 영지식 증명
[2] 정확히는 ZCash에서 개발한 Halo2의 변형 버전인 Halo2-ce이다. 기존 Halo2의 증명 생성 과정을 IPA에서 KZG기반으로 교체하였다.

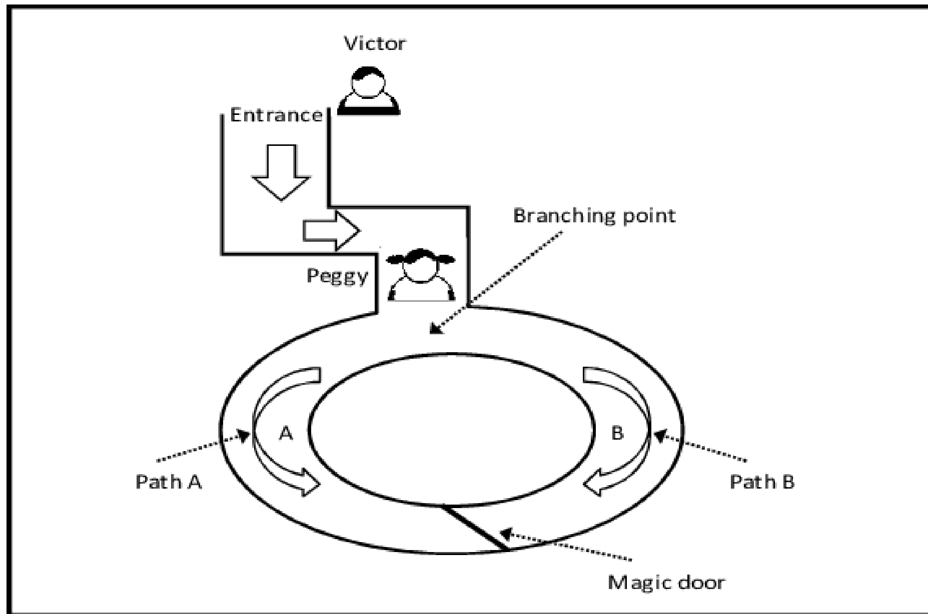
# 2. zkEVM 배경 지식

## 2.1 ZKP

암호학에서 Zero-knowledge proof(ZKP)란 주어진 문제에 대해 Prover가 답을 가지고 있음을 "답을 가지고 있다는 것 외에 어떠한 정보도 노출하지 않으면서" Verifier에게 증명하는 것을 말한다. 즉, ZKP는 3가지 다음 특성을 가진다.

1. Completeness: 만약 Prover의 진술이 참이라면, 정직한 Prover는 항상 정직한 Verifier를 설득시킬 수 있다.
2. Soundness: 만약 Prover가 악의적이라면, Prover는 어떠한 악의적인 행동으로도 Verifier를 설득시킬 수 없다.
3. Zero-knowledge: Verifier는 검증 과정에서 Prover의 진술이 참이라는 사실 외에는 어떠한 정보도 얻을 수 없다.

ZKP를 설명할 때 가장 자주 쓰이는 사고 실험인 알리바바의 동굴의 예시를 살펴보자.




< 그림 3 알리바바의 동굴. 이미지 출처 : link>

klaytn

위 상황을 요약하면 다음과 같다.

1. Victor는 Verifier이고, Peggy는 Prover이다.
2. 동굴 내에는 Magic door가 있고, 비밀키가 없다면 절대 문을 열고 반대편으로 나올 수 없다.
3. Peggy가 A 혹은 B로 들어간 후, Victor는 Peggy에게 A 혹은 B로 나오라 한다. (Challenge)
4. 충분히 많은 시도동안 Peggy가 Victor의 테스트를 모두 통과한다면, Victor는 Peggy가 비밀키를 가지고 있다는 것을 확률적으로 확신할 수 있다.
5. 한번이라도 Peggy가 테스트를 통과하지 못한다면, Peggy의 증명은 실패한다.

이 상황을 위에서 알아본 ZKP의 특성과 연결지어 보자.

1. Completeness: 만약 Peggy가 비밀키를 가지고 있다면, Peggy는 항상 Victor를 설득시킬 수 있다.
2. Soundness: 만약 Peggy에게 비밀키가 없다면, 충분히 많은 챌린지를 진행할 때 Peggy는 Victor를 설득시킬 수 없다.
3. Zero-knowledge: Victor는 Peggy가 비밀키를 가지고 있다는 정보 외에 어떠한 정보도 얻을 수 없다.

klaytn

이를 통해 Peggy와 Victor는 ZKP 프로토콜을 통해 “비밀키의 보유 여부”라는 진술에 대한 증명을 수행한 것이라는 것을 알 수 있다. 하지만 다음 상황을 가정해보자. “만약 Victor가 진행할 챌린지에 관한 정보를 Peggy가 미리 알고 있었다면 어떨까?” 비록 Peggy는 비밀키를 가지고 있지 않지만, Victor가 어느 길로 나오라고 할 지에 대한 정보를 가지고 있기 때문에 Victor를 속일 수 있다. 이러한 상황을 막기 위해 우리는 Peggy의 비밀키에 대한 정보를 Victor에게 알리지 않는 것뿐만 아니라, Victor가 진행할 챌린지에 대한 정보도 Peggy에게 알려지지 않아야 한다. 간단하지만 ZKP가 가져야할 특성을 모두 가지고 있는 알리바바의 동굴의 예시를 항상 떠올리며 ZKP 프로토콜을 분석한다면 훨씬 쉽게 이해할 수 있을 것이다.

## 2.2 ZK-SNARK

ZK-SNARK란 Zero-Knowledge Succinctness Non-interactive Arguments of Knowledges의 약어로, 이름에서 알 수 있듯 Prover와 Verifier의 소통이 없는 간결한 증명을 생성하는 프로토콜이다. Prover는 문제에 대한 해를 숨기면서도 자신의 해로 주어진 문제를 해결할 수 있다는 것을 증명해야하고, Verifier는 Prover의 증명을 검증해야한다. 우리는 ZK-SNARK에서 사용되는 **Homomorphic Hiding**과 **KCA** 라는 수학적 개념을 알아보고, 이를 바탕으로 실제 간단한 예시를 통해 ZK-SNARK의 전체 과정을 살펴볼 것이다.

### a) Homomorphic Hiding

Homomorphic Hiding(HH)은, ZKP를 이해하기 위해 가장 핵심적인 개념이다. HH는 어떤 값을 숨기면서도 증명할 수 있는 수학적 관계를 도출해내고 싶을 때 사용한다. 이 때 HH를 통해 값을 숨기는 것을 "Commit한다"라고 하고, 숨겨진 값을 Commitment라고 부른다. 우선 어떤 값을 숨긴다는 것에 집중하자. HH가 값을 숨긴다는 것은 본래의 값을 입력으로 받아 출력을 내고, 출력을 통해 입력을 알아낼 수 없다는 것을 뜻한다. 우리는 단방향 해시 함수가 이런 역할을 할 수 있다는 것을 쉽게 알아챌 수 있다. 하지만 단방향 해시 함수는 출력에 대한 수학적 관계를 도출해낼 수 없다. 예시를 살펴보자.

1) $x + y = 7$이라는 문제에 대해, Prover가 이를 만족하는 $(x', y')$를 알고 있다는 것을 증명하고 싶다.
2) Prover는 $(x', y')$를 숨기기 위해 단방향 해시 함수인 $H(x)$를 통해 Verifier에게 값을 전달한다.
3) $(H(x'), H(y'))$를 전달받은 Verifier는 본래 값을 알아내진 못하지만, 더이상 $H(x') + H(y') = H(7)$과 같은 수학적 관계를 도출해낼 수 없기 때문에 Prover가 정답을 가지고 있다는 것을 증명할 수 없다.

따라서 단방향 해시함수는 HH로 사용될 수 없다. 위 예시를 통해 우리는 HH가 가져야할 조건들을 다음과 같이 정의할 수 있다.

1) HH를 통해 계산된 출력을 통해 본래 값을 찾을 수 없어야 한다.
2) 만약 입력이 다르다면, HH를 통해 계산된 출력도 모두 달라야 한다.
3) HH를 통해 계산된 출력을 통해 우리는 증명가능한 수학적 관계식을 도출해낼 수 있어야 한다.

이제 HH로 사용될 수 있는 함수들을 살펴보자.

* 우리는 아래에서 다룰 수학적 개념들에 대한 엄밀한 증명보다 사용에 집중한다. 자세한 증명은 직접 찾아보길 추천한다.

1. Homomorphic Hiding : 이산 로그 문제 (DLP)

이산로그 문제란 충분히 큰 소수 $p$와 $p$의 원시근[3] $g$ 가 있을 때, $g^x (mod\, p) = y$에 대해서 $g^x, p, y$가 주어졌을 때 $x$를 구하는 문제로 정의되고, 이는 상수 시간내에 해결할 수 없다는 것이 증명되어 있다. 그렇다면 DLP가 어떻게 HH로 사용될 수 있는지 예시를 통해 살펴보자.

소수 $p = 5$에 대해, 원시근 3을 $g$로 택하여 $3^x (mod\, 5) = y$를 정의하고, $x$에 대해 모든 거듭제곱 값을 계산하면 다음과 같다.

| x | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| y=3^x (mod 5) | 1 | 3 | 4 | 2 | 1 | 3 | 4 |

이를 통해 $x \in \{0, p - 2\}$에 대해, $\{1, 3, 4, 2\}$가 중복되지 않은 집합을 이루며, $Z^*_5 = \{1, 2, \cdots, p - 1\}$ 집합의 모든 값을 표현한다는 것을 알 수 있다. 또한 $g^x$를 통해 $x$를(x가 충분히 큰 소수일때) 알아내기 매우 어렵다는 것이 증명되어있고, 지수의 곱셈법칙을 통해 계산된 거듭제곱끼리의 수학적 관계식[4]까지 도출해낼 수 있으므로 HH 함수로 사용될 수 있다. DLP를 위에서 살펴본 예시에 대입해보자.

1) Prover는 x값들인 $(3, 4)$를 문제의 해로 가지고 있고, 따라서 y값들인 $(2, 1)$을 Verifier에게 보낸다.
2) Verifier는 $2 * 1 = 3^{7\, mod(5-1)} (mod\, 5) = 27\, (mod\, 5) = 2$ 를 통해 Prover의 $(x', y')$ 값에 대한 정보는 알 수 없지만, 문제의 정답을 가지고 있다는 것을 증명할 수 있다.

예시를 위해 아주 작은 $p$를 선택하여 계산이 어렵지 않다고 느껴지지만, $p$를 13과 같이 조금만 더 큰 소수로 설정하여도 계산량이 매우 커지는 것을 알 수 있다.

2. Homomorphic Hiding : 타원 곡선 이산 로그 문제 (ECDLP)

ECDLP에 대해 알아보기 전에, 타원 곡선에 대해 간단히 살펴보자. 타원 곡선은 다음 방정식을 만족하는 점들의 집합으로 정의된다.

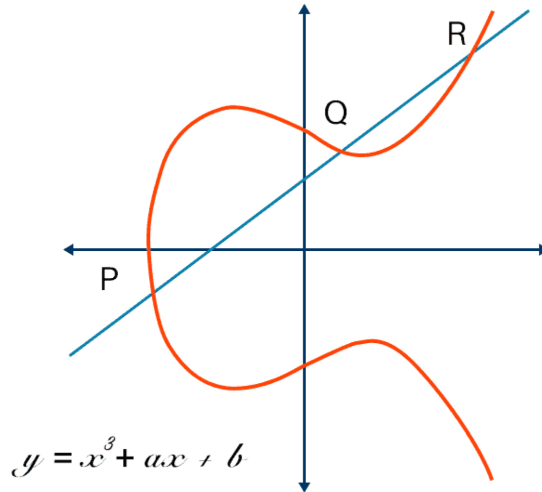
[3] 거듭제곱을 통해 Z* = {1, ..., p-1}의 모든 값을 표현할 수 있는 양의 정수.
[4] g^a * g^b = g^((a+b) (mod p-1)) (mod p)

klaytn

1. $y^2 = x^3 + ax + b$

2. $4a^3 + 27b^2 \neq 0$

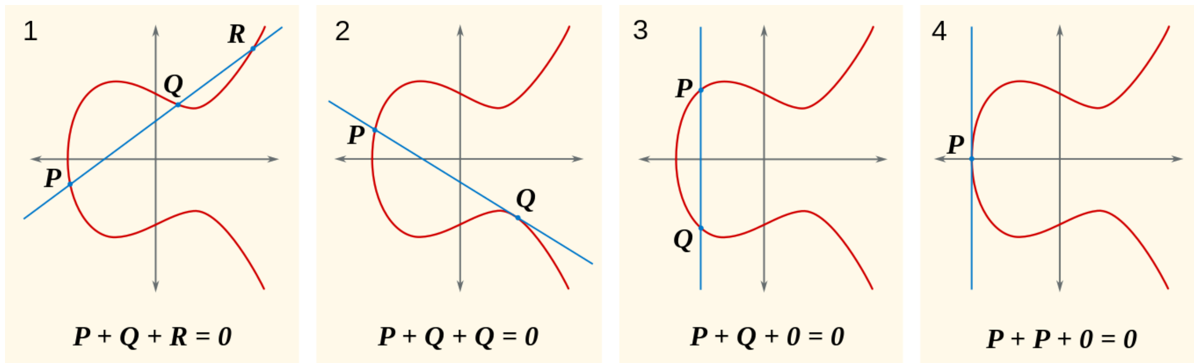
좌표평면에는 일반적으로 다음과 같이 그려진다.




<그림 4 타원 곡선. 이미지 출처: https://avinetworks.com/glossary/elliptic-curve-cryptography>

klaytn

타원 곡선 위의 점 사이에서는 다음과 같이 덧셈 연산이 정의된다: *타원 곡선 위의* $P$, $Q$ *점에 대해,* $P + Q = -R$*이다.* 이는 아래 그림을 보면 쉽게 이해할 수 있다. ($-R$은 $R$의 $x$ 축 대칭으로 정의한다.)




<그림 5 타원 곡선 위에서의 덧셈 연산. 이미지 출처 : https://en.wikipedia.org/wiki/Elliptic_curve>

DLP에서와 유사하게, 소수 $p$에 대해 타원 곡선 위의 점도 $mod\ p$를 통해 유한군의 집합으로 표현하자. 그리고 우리는 임의의 타원 곡선을 하나 선택하고, 타원 곡선 위의 한 점 $P$를 $k$번 더한 $kP$를 정의할 수 있다. 이 때 스칼라 값 $n$에 대하여 $nP = 0$, 즉 $(n + 1)P = P$가 되어 본래의 점으로 돌아오는 원소 $P$를

생성원($Generator$)라 하고, 집합 내의 원소는 다음과 같다: $Group = \{0, P, 2P, \cdots, (n-1)P\}$. 만약 그룹 내의 모든 점이 생성원이 될 수 있다면, 해당 그룹을 순환군이라고 한다.

DLP에서와 유사하게, 충분히 큰 순환군 내에서 $kP$로부터 $k$를 계산하는 것은 매우 어렵다는 것이 알려져 있다. 아래 예시를 살펴보자.

1) $y^2 = x^3 + 2x + 3 \ (mod\ 97)$에 대해 타원 곡선 위 한 점 $P = (3, 6)$을 찾을 수 있다.
2) $P$를 두 번 더한 $P + P = 2P = (80, 10)$을 계산할 수 있다.
3) $P$를 계속해서 더했을 때, 5$P$에 도달하면 $5P = 0$이 되고, $6P = 0 + P = P$가 된다.

3번을 통해 우리는 타원 곡선 위에서 순환군을 찾을 수 있고, 해당 순환군의 한 점 $P$에 대해 $nP = 0$을 만족하는 $order\ n$이 존재한다는 것을 알 수 있다. 또한 해당 순환군 내에서는 중복되는 값이 없으며 $kP$ ($k < n$)의 값으로부터 $k$를 찾는 것은 매우 어렵다는 것을 알고 있으므로 타원 곡선 위 덧셈을 HH로 사용할 수 있다. ECDLP를 위에서 살펴본 예시에 대입해보자.

1) Prover는 $(3, 4)$를 문제의 해로 가지고 있고, 따라서 $(3P, 4P)$를 Verifier에게 보낸다.
2) Verifier는 $3P + 4P = 7P = 2P$를 통해 Prover의 $(x', y')$ 값에 대한 정보는 알 수 없지만, 문제의 정답을 가지고 있다는 것을 증명할 수 있다.

지금까지 우리는 DLP, ECDLP를 통해 HH 함수가 될 수 있는 후보를 살펴보았다. 하지만 지금의 HH를 통해서는 덧셈 연산에 대한 검증만 할 수 있고 곱셈에 대해선 검증하기 까다롭다. 예를 들어 다음 상황을 가정해보자.

| 본래 값을 숨긴 $X = HH(x), Y = HH(y), Z = HH(z)$가 있을 때, Prover는 $z = xy$를 만족함을 보이고 싶다고 하자. 하지만 지금의 HH로는 곱셈 연산을 검증할 수 없으므로 이를 위한 또다른 연산이 필요하다. |
|---|

DLP와 달리, 타원 곡선에서는 Pairing을 통해 이를 쉽게 나타낼 수 있다. 타원 곡선에서 Pairing은 서로 다른 타원 곡선위의 점을 또 다른 타원 곡선 위로 매핑하며, 다음과 같은 형태를 가진다. *bilinear Pairing을 완전히 이해하기 위해서는 많은 수학적 배경 지식이 필요하니, 아래와 같이 직관적으로 이해를 하는데 초점을 준다. bilinear Pairing 은 ZKP와, 블록체인에서의 Commitement기술, BLS Sign Consensus 등에서 증명하는데 많이 사용된다.

$$e\colon\ G_1 \times G_2 \to G_T$$

만약 $G_1, G_2, G_T$가 같은 $order\ n$을 가지는 타원 곡선이라면, 다음과 같이 bilinearity를 가지는 타원 곡선 위 모든 연산은 Pairing으로 사용될 수 있다:

- $e(P,\ Q\ +\ R)\ =\ e(P,\ Q)\ *\ e(P,\ R)$
- $e(aP, bQ)\ =\ ab\ *\ e(P,\ Q)\ =\ g^{ab}$ where $g\ \in G_T$

따라서 Pairing을 통해 ECDLP를 HH로 사용할 때 위 예시를 다음과 같이 검증할 수 있다.

$$e(X,\ Y)\ =\ e(xG_1,\ yG_2)\ =\ xy\ *\ e(G_1,\ G_2)\ =\ e(zG_1,\ G_2)\ =\ z\ *\ e(G_1,\ G_2)$$

위와 같이 ECDLP를 사용하면 Pairing을 통한 곱셈 연산에 대한 검증도 쉽게 할 수 있으므로 대부분의 경우에서 ECDLP를 HH로 사용할 것이다.

### b) KCA

ZKP에서, Prover는 증명하고 싶은 문제를 다항식의 형태로 변환하여야 한다. 어떤 문제의 해를 가지고 있다는 것을 수식적으로 증명하기 위해선, 올바른 입력값을 통해 문제의 해를 도출해내고, 이 과정은 다항식의 형태로 나타나기 때문이다. 즉 모든 문제는 Prover가 올바른 다항식을 가지고 있는지를 검증하는 형태로 바뀌게 된다. Prover가 다음과 같은 다항식을 가지고 있을 때, 다항식에 대한 어떠한 정보도 Verifier에게 알리지 않고 이를 증명하고 싶다고 하자.

$$P(x)\ =\ a_0\ +\ a_1x\ +\ a_2x^2\ +\cdots+\ a_nx^n$$

다항식의 차수는 공개되어 있을 때, 이는 결국 Prover가 다항식의 모든 계수 $\{a_0, \cdots, a_n\}$을 가지고 있음을 보이는 것이다. 이 때, 앞서 배운 HH를 통해 Prover와 Verifier는 다음과 같은 과정을 통해 검증을 진행한다.

1) Verifier는 다항식 $P(x)$를 평가할 랜덤한 좌표 $s\ \in \{0,\ p\ -\ 1\}$을 선택하고, 이를 HH로 숨긴 $E(s^0),\ E(s^1),\ E(s^2), \cdots\ E(s^n)$을 Prover에게 보낸다. ($E(x)$는 $HH$이다.)

2) Prover는 다음 과정을 통해 $P(x)$를 $s$에서 평가한 후 이를 숨긴 값 $E(P(s))$를 계산할 수 있고, 이를 Verifier에게 보낸다.

$$E(P(s)) = E(a_0 + a_1 s^1 + \cdots + a_n s^n) = a_0 E(s^0) + a_1 E(s^1) + \cdots + a_n E(s^n)$$

3) Verifier는 $E(P(s))$를 확인한다.

다음 과정을 통해, Verifier는 다항식을 평가할 챌린지 $s$를 Prover에게 공개하지 않고, Prover 또한 $P(s)$를 Verifier에게 공개하지 않으면서 검증을 진행할 수 있다. 하지만 Verifier는 Prover가 사용한 $\{a_0, \cdots, a_n\}$이 정말 다항식 $P(x)$의 계수인지, 아니면 Prover가 아무렇게나 선택한 값인지 알 수 없으므로 Prover가 다항식의 계수를 사용했다는 것을 검증할 방법이 필요하다. 이 방법이 바로 KCA (Knowledge of Coefficient Assumption)이다.

KCA의 기본 아이디어는 다음과 같다. 랜덤한 $\alpha \in \{1, \cdots, p - 1\}$을 선택하자. 그리고 타원곡선 위 두 점 $(a, b)$에 대해 $b = \alpha \cdot a$를 만족하는 쌍을 $\alpha - pair$라고 부르자. 이 때 우리는 ECDLP에 의해 $(a, b)$ 값만으로는 $\alpha$를 구할 수 없다는 것을 알 수 있고, $(a, b)$를 통해 다른 $\alpha - pair$를 구하는 방법은 또 다른 상수 $\gamma$를 곱하여 $a' = \gamma \cdot a, b' = \gamma \cdot b$를 계산하는 방법 밖에 없다.

$$b' = \gamma \cdot b = \gamma \cdot \alpha \cdot a = \alpha \cdot \gamma \cdot a = \alpha \cdot a'$$

만약 Verifier가 제출한 $\alpha - pair$에 대해 Prover가 또 다른 $\alpha - pair$를 제출할 수 있다면, 우리는 Prover가 이를 만족하는 상수 $\gamma$를 가지고 있다고 확신할 수 있는 것이다. 여기서 유의해야하는 것은, 우리는 Prover의 $\gamma$가 올바른지를 검사하는 것이 아닌, Prover가 새로운 $\alpha - pair$를 만든 상수 $\gamma$를 "가지고 있다"는 것을 확인한다는 것이다.

위 단일 $\alpha - pair$를 실제 다항식에 대해 확장한 것을 d-KCA라고 부르고, 다음과 같은 과정을 통해 검증된다.

1) Verifier는 $\alpha, s$를 선택하고, 다음과 같이 $\alpha - pair$들을 계산하여 Prover에게 보낸다.

$$(E(s^0), E(\alpha s^0)), (E(s^1), E(\alpha s^1)), \cdots, (E(s^d), E(\alpha s^d))$$

2) Prover는 새로운 $\alpha - pair$: $a' = E(P(s)), b' = E(\alpha P(s))$를 계산하여 Verifier에게 보낸다.

$$a' = E(P(s)) = E(a_0 + a_1 s^1 + \cdots + a_d s^d)$$

$$b' = E(\alpha P(s)) = E(\alpha a_0 + \cdots + \alpha a_d s^d) = a_o E(\alpha) + a_1 E(\alpha s^1) + \cdots + a_d E(\alpha s^d)$$

3) Verifier는 Prover의 $(a', b')$를 검증하고, 만약 $\alpha - pair$가 맞다면 Prover가 다항식의 계수 $\{a_0, \cdots, a_d\}$를 사용하여 이를 계산했다는 것을 확신할 수 있다.

3)번 과정에 대해 여전히 Prover가 올바른 다항식을 모르더라도 $\alpha - pair$를 계산할 수 있는 것이 아니냐는 의문이 들 수 있다. 우리가 d-KCA를 통해 검증하려는 것은 Prover가 올바른 다항식을 가지고 있는 것이 아니라, 검증 과정에서 동일한 다항식의 계수 $\{a_0, \cdots, a_d\}$를 사용하는 것이다. 예를들어, 만약 Prover가 공개된 문제에 대한 해를 모를 때 임의로 다항식의 계수를 조작하여 테스트를 통과할 수 있다. 하지만, d-KCA를 사용한다면 계수의 조작을 통해 본래의 문제(a')는 통과할 수 있지만, d-KCA를(b') 통과할 수 없게 된다. 즉, d-KCA는 Prover가 공개된 문제를 수정하지 않고 사용하는지를 검증하는 과정인 것이다. 이는 다음 ZK-SNARK의 실제 검증 과정을 통해 쉽게 이해할 수 있을 것이다.

### c) ZK-SNARK with Example

알리바바의 동굴에서처럼 수천번의 정보 교환이 필요한 증명을 실제 환경에서 사용하는 것은 많은 제약이 존재한다. 즉, ZKP를 실제로 사용하기 위해서는 간결한 증명이 필요하다. 따라서 우리는 앞서 배운 개념들을 사용해 ZK-SNARK를 통해 주어진 문제를 다항식으로 바꾸는 과정 및 최종적인 증명을 생성하는 과정까지 자세히 다뤄볼 것이다.

1. Original problem

이론상으로, ZK-SNARK는 NP class[5]에 속하는 모든 문제에 적용될 수 있다. 이번 예시에서는 다음과 같이 주어진 다항식의 해를 찾는 문제를 사용할 것이다.

| 다항식 $f(x) = x^3 + x^2 + 5$가 있을 때, $f(z) = 17$이 되는 $z$를 알고 있는가. |
|---|

[5] 비결정적 튜링 머신으로 다항 시간 내에 해결할 수 있는 문제의 집합

2. Arithmetic circuit

먼저 주어진 문제를 곱셈 연산 단위로 분리하여 다음과 같이 산술 회로를 만들 수 있다.

1. Gate1: $sym_1 = x * x$
2. Gate2: $sym_2 = sym_1 * (x + 1)$
3. Gate3: $out = (sym_2 + 5) * 1$

각 Gate는 2개의 입력 wire와 하나의 출력 wire를 통해 구성된다. 즉, 우리는 $f(x)$를 3개의 곱셈 Gate와 5개의 매개변수 $\{1, x, sym_1, sym_2, out\}$로 나타낸 것이다. Prover가 올바른 해를 도출 할 수 있다면 올바른 매개변수의 값 $s = \{1, 2, 4, 12, 17\}$를 가지고 있어야 한다. 이 때 $(x + 1)$과 같은 값에 의문을 가질 수 있다. 지금은 ZK-SNARK는 곱셈 Gate만 사용하며, 다른 연산 값들은 뒤에서 살펴볼 weight matrix를 통해 설정한다고 이해하면 충분하다.

3. R1CS

원래의 문제를 다항식으로 바꾼다는 것은, 해당 다항식이 특정 제약 조건을 만족하는 해를 가진 다항식이라는 것을 뜻한다. Prover는 자신이 가지고 있는 다항식이 제약 조건을 만족시키는 것을 증명하는 것이다. ZK-SNARK에서는 다항식의 제약 조건으로 R1CS를 사용한다. R1CS (Rank-1 Constraint System)이란, solution vector $s$가 있을 때 아래와 같은 형태를 가진 제약 조건들의 세트를 말한다.Prover가 올바른 $s$를 가지고 있다면, 아래의 제약 조건을 만족해야 한다.

$$sa_i \circ sb_i - s \circ c_i = 0, \; where\; 1 \le i \le n$$

우리의 위의 산술회로 과정의 5개의 매개 변수인 solution vector $s = \{1, x, sym_1, sym_2, out\}$가 있을 때, R1CS와 산술 회로의 각 Gate가 동일한 형태인 것을 쉽게 알 수 있다. 각 게이트에 대해 좌항을 넘기면 3개의 제약 조건을 가진 R1CS를 만들 수 있다.

1. Gate1: $x * x - sym_1 = 0$
2. Gate2: $(x + 1) * sym_1 - sym_2 = 0$
3. Gate3: $1 * (sym_2 + 5) - out = 0$

klaytn

즉, Gate들을 위의 수식형태를 만족하는 행렬로 표현할수가 있다. 이를위해 solution vector s가 열의 인덱스이고, 행은 각 gate들을 나타낸다. a_i를 예를 들면, Gate1의 경우 $sa_1 = x$가 되어야 하므로 $a_1 = [0, 1, 0, 0, 0]$이 되어야한다. Gate 2의 경우 $sa_2 = x + 1$이 되어야 하므로 $a_2 = [1, 1, 0, 0, 0]$이 되어야 한다. 이처럼 각 $a_i$, $b_i$, $c_i$를 모두 계산하면 아래의 그림6과 같다.

| | 1 | x | sym_1 | sym_2 | out |
|---|---|---|---|---|---|
| a_i | 0 | 1 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 0 | 0 |
| | 1 | 0 | 0 | 0 | 0 |
| b_i | 0 | 1 | 0 | 0 | 0 |
| | 0 | 0 | 1 | 0 | 0 |
| | 5 | 0 | 0 | 1 | 0 |
| c_i | 0 | 0 | 1 | 0 | 0 |
| | 0 | 0 | 0 | 1 | 0 |
| | 0 | 0 | 0 | 0 | 1 |

<그림 6 $a_i$, $b_i$, $c_i$ 와 s행렬식 표현>

이 때 $a_i$, $b_i$, $c_i$는 각 Gate에서 solution vector $s$에 곱해지는 weight matrix라고 볼 수 있고, $(x + 1)$은 $sa_2$를 통해 나타내는 것을 확인할 수 있다. 즉 R1CS는 각 Gate에서 wire들이 가져야 하는 값을 나타내는 것이다. 실제로 $sa$를 계산해보면 $\{x, x + 1, 1\}$ 벡터를 얻을 수 있고, 만약 Prover가 올바른 $s$를 가지고 있다면 $\{2, 3, 1\}$이 된다는 것을 기억하자.

4. QAP

R1CS를 위해 나타낸 행렬은 실제 프로토콜에서는 수천개의 행과 열을 가진다. 이를 간결하게 표현하기 위해선 선형 벡터 행렬 다항식의 형태로 변환하여야 한다.

$a_i$의 첫번째 column을 확인해보면, Gate $\{1, 2, 3\}$에서 $\{0, 1, 1\}$을 가지는 것을 확인할 수 있다. 즉, 다항식의 evaluation 형태인 것이다. 대응되는 Gate를 $x$로 두고 y를 x에 해당되는 값으로 두면, 첫번째

column은 세 점 $(1, 0), (2, 1), (3, 1)$을 지나는 다항식으로 표현할 수 있다. 라그랑주 보간법(Lagrange interpolation) 덕분에 우리는 $n$개의 점이 주어졌을 때 $n-1$차 다항식을 쉽게 계산할 수 있다. 라그랑주 보간법을 통해 첫번째 열의 세점을 지나는 evaluation 형태를 다항식으로 표현을 하게 되면 $y = -2t^2 + 2.5t - 0.5$와 같다. 이 식에서 t에 1를 대입하면 0이 나오게된다. 다항식의 계수 정보를 아래의 행렬인 A[t]의 1열에 해당하는 값이다. 모든 벡터에 대해 이를 계산한 후 다항식의 계수를 행렬로 나타내면 다음과 같다.

| | 1 | x | sym_1 | sym_2 | out |
|---|---|---|---|---|---|
| A[t] | -2 | 0 | 0 | 0 | 0 |
| | 2.5 | 1.5 | 0 | 0 | 0 |
| | -0.5 | -0.5 | 0 | 0 | 0 |
| B[t] | 5 | 3 | -3 | 1 | 0 |
| | -7.5 | -2.5 | 4 | -1.5 | 0 |
| | 2.5 | 0.5 | -1 | 0.5 | 0 |
| C[t] | 0 | 0 | 3 | -3 | 1 |
| | 0 | 0 | -2.5 | 4 | -1.5 |
| | 0 | 0 | 0.5 | -1 | 0.5 |

<그림 7 $A[t], B[t], C[t]$ 행렬식 표현>

위 행렬은 Gate t($\in \{1, 2, 3\}$)에서 평가된 각 다항식의 계수를 나타낸다. 각 다항식은 여전히 R1CS를 만족해야 하므로, $A \cdot s * B \cdot s - C \cdot s = 0$을 만족해야 한다 ($A = [A[1], A[2], A[3]]$). 아직 위의 식이 어떻게 문제를 간결하게 표현하는지 헷갈릴 수 있다. $A \cdot s$를 직접 계산해보자. 만약 Prover가 올바른 $s$를 가지고 있다면 $A \cdot s = [-2, 5.5, -1.5]$가 되고, 이는 다항식 $A(t) = -1.5t^2 + 5.5t - 2$의 계수를 나타낸다. $A(t)$를 $t \in \{1, 2, 3\}$에 대해 계산해보면 $(2, 3, 1)$을 해로 얻는다. 그리고 이는 위에서 weight matrix $a_i$를 통해 계산한 값과 동일하다는 것을 알 수 있다! $B(t), C(t)$ 또한 동일하게 계산되고, 다항식의 연산 $A(t) * B(t) - C(t) = 0$을 만족해야 한다. 이제 우리는 2차 방정식 3개를 통해 Prover가 올바른 $s$를 가지고 있음을 검증할 수 있는 것이다.

$t \in \{1, 2, 3\}$에 대해서 $A(t) * B(t) - C(t) = 0$을 만족해야 한다는 것은, $A(t) * B(t) - C(t)$가 $Z(t) = (t-1)(t-2)(t-3)$으로 나눠진다는 것을 뜻한다. 즉, $\frac{A(t)*B(t)-C(t)}{Z(t)} = H(t)$를 만족하는 $H(t)$를 Prover가 계산할 수 있다는 것이다.

이를 통해 최종적으로 변환된 문제는 다음과 같다.

| ***Prover는 주어진 문제 $A[t]$, $B[t]$, $C[t]$와 $Z[t]$가 주어졌을 때, 이를 통해 $A(t)$, $B(t)$, $C(t)$, $H(t)$를 계산할 수 있는 올바른 $s$를 알고 있는가?*** |
| --- |

이를 식으로 나타내면 다음과 같다.

1. $A(t) * B(t) - C(t) = Z(t) * H(t)$를 만족하는 $H(t)$가 존재한다.
2. $A(t)$, $B(t)$, $C(t)$는 주어진 문제 $A[t]$, $B[t]$, $C[t]$를 통해 계산된다.
3. $A(t)$, $B(t)$, $C(t)$는 동일한 assignment $s$를 통해 계산된다.

1번 조건은 R1CS를 만족하는지에 관한 증명이고, 2번은 d-KCA를 사용하여 해당 다항식들이 정말 공개된 문제를 나타내는 다항식의 계수를 통해 계산되었는지, 3번은 모두 동일한 $s$가 곱해졌는가에 한 증명이다. 2, 3번 증명이 없다면 Prover는 R1CS만 통과할 수 있는 임의의 다항식과 $s$값을 조작하여 검증을 통과할 수 있다. 본격적인 검증과정에 들어가기 전에, Prover는 올바른 다항식을 알고 있다는 것에 대한 증명을 어떻게 생성할 수 있을까? Prover는 다항식에 대한 정보를 공개하고 싶지 않기 때문에, 다항식의 계수를 공개하는 방식은 사용할 수 없다. 그렇다면 다음 상황을 가정해보자: "*Prover가 잘못된 다항식을 가지고 있을 때, 충분히 넓은 필드에서 임의로 선택한 한 점에 대해 잘못된 다항식이 주어진 조건을 만족할 확률은 얼마일까?*"

서로 다른 $n$차 방정식은 최대 $n$개의 점[6]에서 만나기 때문에 Verifier가 다항식을 평가할 $x$값을 Prover에게 노출하지 않는다면, 우리는 단 한점에서의 evalutaion만 가지고 검증할 수 있다.

[6] Schwarz-Zippel lemma

klaytn

5. Proving system

모든 검증은 Pairing연산을 통해 이루어진다. 먼저 첫번째 조건에 관한 검증 과정을 살펴보자.

1. Verifier는 랜덤한 $t$ (이전까진 챌린지를 위한 값을 $s$로 나타냈지만, solution vector $s$와 분리하기 위해 $t$를 사용한다.)를 선택하고 $E(t^0),\ E(t^1), \cdots, E(t^d),\ E(A[t]),\ E(B[t]),\ E(C[t])$를 계산하여 Prover에게 보낸다.
2. Prover는 $E(A(t)),\ E(B(t)),\ E(C(t)),\ E(H(t))$를 계산하여 Verifier에게 보낸다.

$$E(A(t)) \ = \ E(s \cdot A[t]) = s \cdot E(A[t])$$

$$E(H(t)) \ = \ E(h_0 + h_1 t + h_2 t^2 + \cdots + h_d t^d) = h_0 E(t^0) + h_1 E(t^1) + \cdots + h_d E(t^d)$$

3. Verifier는 Prover가 보낸 다항식의 값이 $A(t)B(t) - C(t) = H(t)Z(t)$를 만족하는지 검증하기 위해 다음 Pairing을 계산한다.

$$\frac{e(E(A(t)), E(B(t)))}{e(E(C(t)), G)} = \ e(E(H(t)),\ E(Z(t)))$$

3번 과정의 Pairing이 $A(t)B(t) - C(t) = H(t)Z(t)$를 검증하는 과정은 다음과 같다.

$$e(E(A(t)),\ E(B(t))) \ = \ e(A(t) \cdot G,\ B(t) \cdot G) \ = \ g^{A(t)B(t)}$$

$$e(E(C(t)),\ G) \ = \ e(C(t) \cdot G,\ G) \ = \ g^{C(t)}$$

$$e(E(H(t)),\ E(Z(t))) \ = \ g^{H(t)Z(t)}$$

$$g^{A(t)B(t)-C(t)} = g^{H(t)Z(t)} \rightarrow A(t)B(t) - C(t) = H(t)Z(t)$$

첫번째 증명에서, 우리는 $E(A(t)), E(B(t)), E(C(t))$가 정말 $E(A[t]), E(B[t]), E(C[t])$로 부터 계산되었는지 검증해야 한다. 이를 위해 d-KCA를 진행하자.

1. Verifier는 랜덤한 $\alpha_A,\ \alpha_B,\ \alpha_C,\ t$를 선택하고 $\alpha - pair$인 $(E(a_1(t)),\ E(\alpha_A a_1(t))),\ \ \cdots,\ (E(c_m(t)),\ E(\alpha_C c_m(t)))$ 를 계산하여 보낸다. 이 때 $a_i(t),\ b_i(t),\ c_i(t)$는 $A[t],\ B[t],\ C[t]$의 column vector이고, $m$은 column의 개수이다.
2. Prover는 다음과 같이 새로운 $\alpha - pair$를 계산하여 Verifier에게 보낸다.

$$E(A(t)) = E(a_1(t) \cdot s + a_2(t) \cdot s + \cdots + a_m(t) \cdot s)$$

$$= s \cdot E(a_1(t)) + s \cdot E(a_2(t)) + c \cdots + s \cdot E(a_5(t))$$

$$E(\alpha_A A(t)) = E(\alpha_A a_1(t) \cdot s + \alpha_A a_2(t) \cdot s + \cdots + \alpha_A a_5(t) \cdot s)$$

$$= s \cdot E(\alpha_A a_1(t)) + \cdots + s \cdot E(\alpha_A a_5(t))$$

3. Verifier는 다음과 같이 $\alpha - pair$테스트를 진행한다.

$$e(E(A(t)), E(\alpha_A)) = e(E(\alpha_A A(t)), G)$$
$$e(E(B(t)), E(\alpha_B)) = e(E(\alpha_B B(t)), G)$$
$$e(E(C(t)), E(\alpha_C)) = e(E(\alpha_C C(t)), G)$$

마지막 검증또한 d-KCA로 진행한다.

1. Verifier는 랜덤한 $\alpha_{A+B+C}$, $t$를 선택하고 $E(\alpha_{A+B+C}(A[t] + B[t] + C[t]))$를 계산하여 보낸다. 이미 각 $E(A[t])$, $E(B[t])$, $E(C[t])$는 Prover가 알고 있다.
2. Prover는 새로운 $\alpha - pair$의 $b' = s \cdot E(\alpha_{A+B+C}(A[t] + B[t] + C[t]))$를 계산하여 Verifier에게 보낸다.
3. Verifier는 $\alpha - pair$를 확인한다. 이 때 $a' = s \cdot E(A[t] + B[t] + C[t])$값은 1번 검증 과정을 통해 Verifier가 이미 가지고 있는 값이고, Prover가 동일한 $s$를 사용하지 않는다면 동일한 값을 도출할 수 없다.

최종적으로 Prover가 Verifier에게 제출하는 증명을 정리해보자.

$$E(A(t)), E(\alpha_A A(t)), E(B(t)), E(\alpha_B B(t)), E(C(t)), E(\alpha_C C(t))$$
$$E(H(t)), E(s \cdot \alpha_{A+B+C}(A[t] + B[t] + C[t])$$

총 8개의 증명을 Verifier에게 제출한다. 증명의 개수는 문제의 따라 달라지지 않으므로 아무리 복잡한 문제로부터 시작해도 Prover는 단 8개의 증명만 Verifier에게 전송하면 검증을 진행할 수 있다.

ZK-SNARK의 전 과정이 끝난 것 같지만, 아직 우리의 검증 과정에서는 Verifier와 Prover가 서로 값을 주고 받아야 한다. 마지막으로 이를 해결하기 위한 Non-interative를 추가해볼 것이다.

6. Non-interactive

지금까지는 Verifier가 직접 챌린지 값을 선택하고, 필요한 연산을 수행한 후 Prover에게 보냈다. 하지만 이 과정에서 Verifier는 챌린지 값의 생성 및 관리를 엄격히 진행해야 하고, 다른 Verifier는 모두 다른 챌린지 값을 선택하기 때문에 같은 문제에 대해서 불필요한 연산을 수행하게 된다. Non-interactive 과정의 아이디어는 만약 검증된 Third-party가 챌린지 값을 생성하여 가지고 있다면, Prover는 Third-party에게 챌린지 정보를 받은 후 바로 Verifier에게 전송하여 검증을 진행할 수 있다는 것이다. 이 때 Third-party가 값을 생성하는 과정을 Trusted Setup이라 부르고, 생성된 챌린지 값을 Common Reference String(CRS)이라 부른다. 이 때 주의해야 할 점은 생성된 챌린지에 대한 정보는 아무에게도 알려지면 안되고, Third-party 조차도 생성된 값은 즉시 HH를 통해 숨긴 후 반드시 본래 값을 삭제해야 한다(toxic wasted). 생성된 CRS는 외부에 공개가 되어도 되며, Prover와 Verifier는 CRS를 이용하여 증명을 생성하고 검증한다.

Non-interactive한 과정을 통한 ZK-SNARK 검증은 다음과 같이 이루어진다.

1. Trusted setup을 통해 계산된 챌린지 값 $E(\alpha)$, $E(t^{i})$와 문제에 관한 $E(A[t])$, $E(B[t])$, $E(C[t])$ 등을 미리 계산해놓는다.
2. Prover는 Third-party에게 챌린지 값을 받은 후 proof를 생성하고 Verifier에게 전송한다.
3. Verifier는 Pairing을 통해 위에서 진행한 검증을 진행한다.

이를 통해 Prover와 Verifier는 상호 소통없이 증명을 전송하고 검증할 수 있다.

ZK-SNARK는 간결한 증명 및 Non-interative 특성을 바탕으로 많은 블록체인기반 영지식 프로토콜에 적용되었다. 하지만 산술 회로를 표현하는 과정에서 곱셈연산에 특화 되어있기 때문에 복잡한 해시 함수나 bitwise연산의 검증엔 사용되기 무리가 있다. 앞으로 살펴볼 PLONK와 Halo2는 ZK-SNARK의 산술화 과정을 발전시키고, Lookup 테이블이라는 새로운 개념을 도입하여 이러한 문제를 해결한다.

klaytn

## 2.3 KZG Commitment

KZG Commitment란, 특정 조건을 만족하는 다항식을 알고 있음을 한번의 Pairing 비교를 통해 검증할 수 있는 Commitment scheme이다. ZK-SNARK와 달리 PLONK는 KZG Commitment를 통해 다항식을 검증하게 된다.

### a) Original KZG-Commitment

Prover는 $p(z) = y$를 만족하는 다항식 $p(x)$를 가지고 있음을 증명하고 싶다고 가정하자. 이 때 KZG Commitment를 사용하면 다음과 같이 검증 과정이 진행된다.

1. Prover는 $p(x)$에 대한 commitment $C = [p(s)]_1 = p(s) \cdot G_1$를 계산한다. 이 때 $s$는 Trusted setup 과정을 통해 생성된 랜덤한 챌린지 값이다.
2. Prover의 $p(x)$는 다음 조건을 만족해야 한다: $p(x) - y = (x - z)q(x)$.
3. Prover는 $q(x)$에 대한 commitment $C = [q(s)]_1$를 계산하고 Verifier에게 보낸다.
4. Verifier는 Schwartz-Zippel Lemma에 따라 랜덤한 챌린지 $s$에서 $p(s) - y = (s - z)q(s)$를 만족하는지 다음 Pairing 연산을 통해 검증하면 된다.

$$e([p(s)]_1 - [y]_1, [1]_2) = e([q(s)]_1, [s]_2 - [z]_2)$$

$$\rightarrow (p(s) - y) \cdot e(G_1, G_2) = q(s)(s - z) \cdot e(G_1, G_2)$$

### b) Batch version of KZG-Commitment

실제 PLONK 프로토콜에선, 서로 다른 점에서 evaluation을 진행해야 하는 다항식들이 존재한다. 또한 하나의 다항식이 아닌 여러 다항식을 다룬다. 이를 한번에 검증하기 위해 우리는 KZG Commitment의 Batch version을 사용한다.

1. 서로 다른 두 개의 evaluation point $z, z'$에서 $s, s'$로 평가되는 두 다항식의 집합 $\{f_i\}_{i\in[t_1]}, \{f'_i\}_{i\in[t_2]}$가 있다. ($[t] = \{1, 2, \cdots, n\}$)

klaytn

2. Verifier와 Prover는 다음 Open protocol[7]을 가진다: $\{\{cm_i\}_{i\in[t_1]}, \{cm'_i\}_{i\in[t_2]}, \{z, z'\}, \{s_i, s'_i\}\}$.

   $cm_i$, $cm'_i$는 각각 $[f_i(x)]_1$, $[f'_i(x)]_1$이다.

3. RLC(Random Linear Combination)를 위한 랜덤 값 $\gamma$, $\gamma'$를 통해 Prover는 다음을 다항식을 계산한다.

$$h(X) := \sum_{i=1}^{t_1} \gamma^{i-1} \frac{f_i(X) - f_i(z)}{X - z},\ h'(X) := \sum_{i=1}^{t_2} \gamma'^{i-1} \frac{f'_i(X) - f'_i(z')}{X - z'}$$

   다소 복잡해보이지만, $h(X)$, $h'(X)$는 $q_i(X)$, $q_i'(X)$의 RLC이다.

4. Prover는 $W := [h(x)]_1$, $W' := [h'(x)]_1$을 계산하고 Verifier에게 보낸다.
5. Verifier는 RLC를 위한 랜덤 값 $r'$을 선택한다.
6. Verifier는 다음을 계산한다.

$$F := \left(\sum_{i\in[t_1]} \gamma^{i-1} cm_i - \left[\sum_{i\in[t_1]} \gamma^{i-1} s_i\right]_1\right) + r'\left(\sum_{i\in[t_2]} \gamma'^{i-1} cm'_i - \left[\sum_{i\in[t_2]} \gamma'^{i-1} s'_i\right]_1\right)$$

   $F$는 Prover가 계산한 $h(X)$의 분자의 Commitment와 동일한 것을 알 수 있다.

7. Verifier는 다음 Pairing 연산을 통해 Verifier가 reconstruction한 $h(X)$, $h'(X)$와 실제 Prover가 계산한 $h(X)$, $h'(X)$가 동일한지 확인한다.

$$e(F + z\cdot W + r'z'\cdot W', [1]_2)\cdot e(-W - r'\cdot W', [x]_2) = 1$$

$$\rightarrow e(W + r'\cdot W', [x]_2) = e(W, [x]_2)\cdot e(r'\cdot W', [x]_2)$$

$$= e(W, [x - z + z]_2)\cdot e(r'\cdot W', [x - z' + z']_2)$$

$$= e(W, [x - z]_2)\cdot e(W, [z]_2)\cdot e(r'\cdot W', [x - z']_2)\cdot e(r'\cdot W', [z']_2)$$

$$= e(W\cdot [x - z]_1 + r'\cdot W'\cdot [x - z']_2, [1]_2)\cdot e(z\cdot W + r'z'\cdot W', [1]_2)$$

$$= e(F + z\cdot W + r'z'\cdot W', [1]_2)$$

[7] Verifier와 Prover가 공유하는 public-coin 프로토콜

## 2.4 PLONK (Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge)

ZK-SNARK에서 살펴봤듯이 ZKP 생성 과정은 크게 2가지로 나눌 수 있다. 먼저 주어진 문제를 산술화하는 과정과, 이를 증명으로 만드는 과정이다. ZKP에서 산술화 과정을 프론트엔드라 부르고, 증명을 생성하는 과정을 백엔드라고 부른다. 지금부터 살펴볼 PLONK는 PLONKish 산술화라는 새로운 산술화 과정을 제안한다. 이를 통해 ZK-SNARK에서 발견한 산술화 단계의 한계를 극복하여 더욱 복잡한 문제도 효율적으로 ZKP로 변환할 수 있다. 또한 우리가 최종적으로 이해하고자 하는 Scroll의 Halo2(정확히는 Halo2-ce)또한 PLONKish 산술화로부터 발전된 형태의 프론트엔드를 사용한다. 따라서 Halo2를 알아보기 전 PLONK의 기초적인 개념들을 먼저 이해하는 것이 큰 도움이 될 것이다.

1. ZK-SNARK VS PLONK

본 리서치에서는 다루지 않지만, ZK-SNARK와 PLONK는 Trusted Setup 과정 중 증명을 위한 Proving key / Verifying Key를 생성하는 과정에서 차이가 있다. ZK-SNARK의 경우에는 위 두 Key를 생성할 때 QAP형태의 “회로"를 입력으로 받는다. 다시 말해서, ZK-SNARK는 한 쌍의 Key로는 하나의 회로만 검증이 가능하며, circuit-dependent하다는 것을 뜻한다. 이는 영지식 증명의 확장성에 매우 큰 장벽이다. 예를 들어 EVM 환경에 ZK-SNARK를 도입한다면, 서로 다른 트랜잭션들은 모두 다른 회로를 가질 것이고, 따라서 각 회로에 대한 Key를 모두 생성해야 한다는 것을 뜻한다. 이는 현실적으로 구현하기 매우 어렵다. 하지만 PLONK는 circuit-dependent한 Key가 아닌, 한번의 범용적인(Universal) Trusted Setup과정을 통해 어느 회로든 증명을 생성하고 검증할 수 있다는 것을 뜻한다. 이를 통해 기존 ZK-SNARK가 가지는 확장성 문제를 크게 개선할 수 있다.

다음으로 두 프로토콜이 가지는 회로를 살펴보자. ZK-SNARK와 PLONK 모두 주어진 문제를 산술 회로로 변환 하지만 회로의 형태는 전혀 다르다. 이는 두 프로토콜이 회로를 서로 다른 관점에서 분석하기 때문이다.

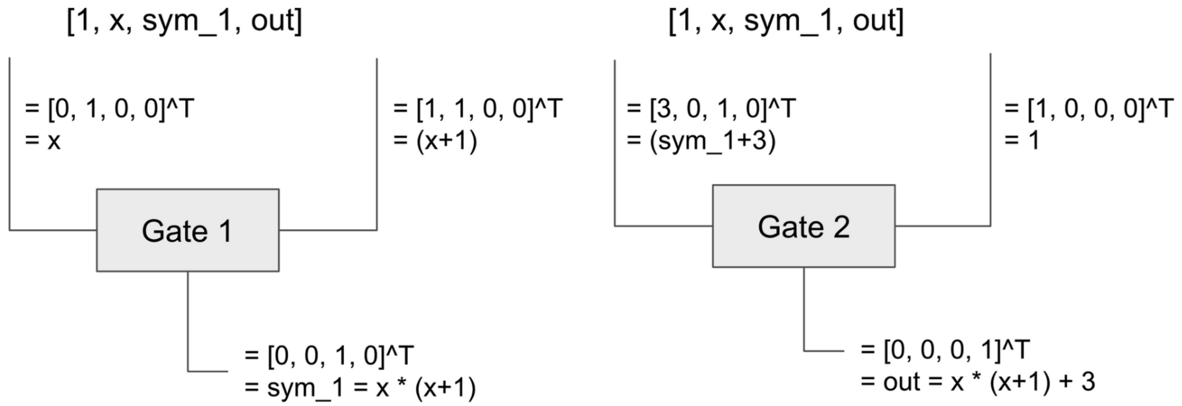
- ZK-SNARK: 회로를 분석할 때 “wire”(값)에 집중한다.
- PLONK: 회로를 분석할 때 “gate”(연산)에 집중한다.

다음은 같은 문제로부터 ZK-SNARK와 PLONK가 어떻게 산술 회로를 설계하는지를 나타낸 그림이다.

klaytn

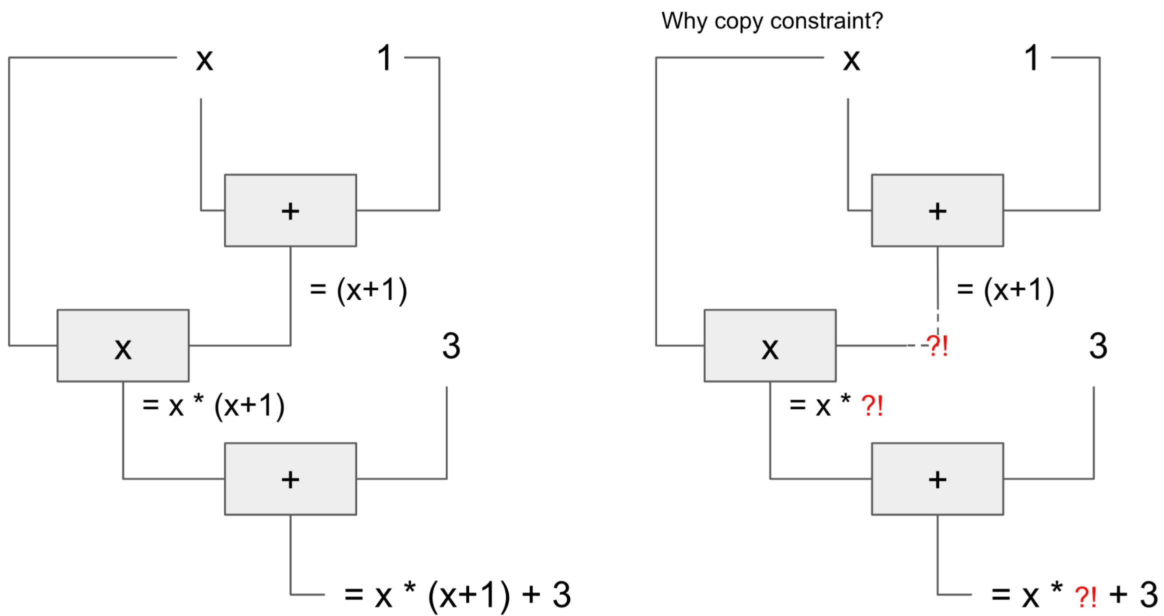
P: f(x) = x(x+1) + 3, f(a) = 9. Find a.

zk-SNARK:
Gate 1: sym_1 = x * (x+1)
Gate 2: out = (sym_1+3) * 1




<그림 8 ZK-SNARK 산술 회로>

PLONK:




<그림 9 PLONK 산술 회로-1>

일반적인 회로에서, wire는 스스로 값을 생성하는 것이 아닌, gate에 연결되어 값을 전달해준다. 즉, 최초의 입력과 최종 출력을 제외하면 내부의 모든 wire는 이전 gate의 출력과 다음 gate의 입력 값을 연결해주는 구성 요소인 것이다.

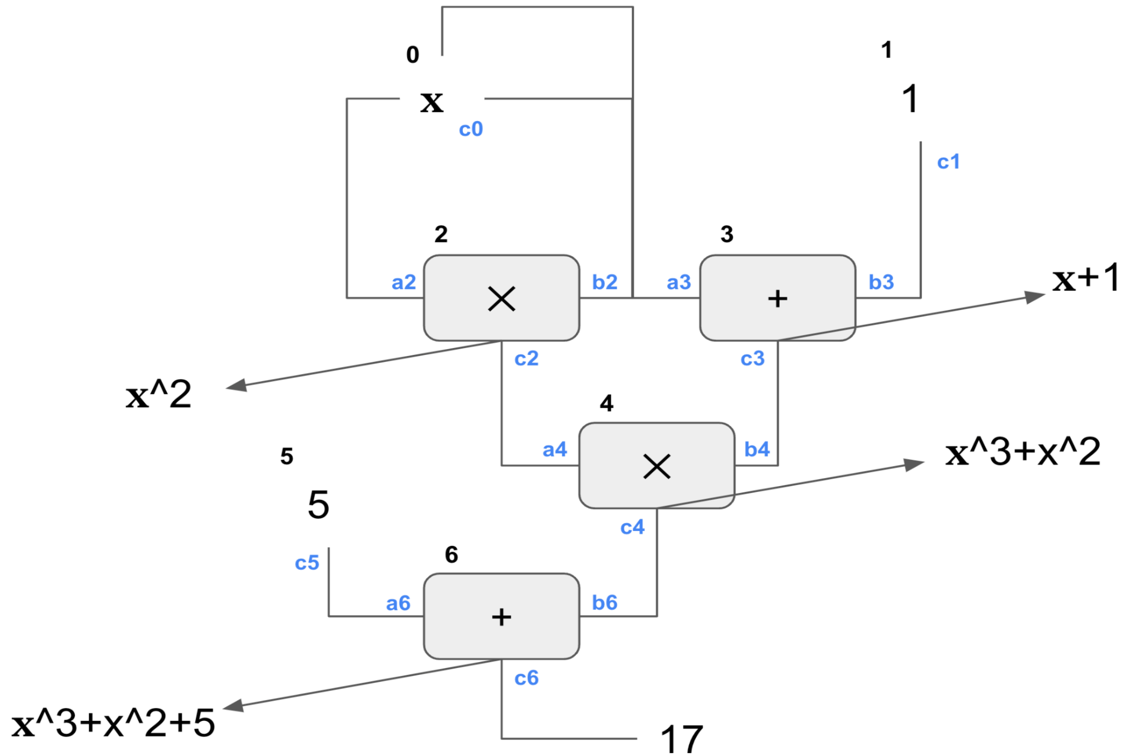
먼저 ZK-SNARK의 경우를 살펴보자. 주어진 문제를 곱셈 gate로 분리하고, 이를 바탕으로 회로를 설계한다. 하지만 그림에서와 같이, 완성된 형태의 회로가 아닌 gate 1과 gate 2가 각각의 회로를 구성하여 서로 연결되어 있지 않은 것을 알 수 있다. R1CS의 Rank-1이라는 이름에서도 알 수 있듯이 각 Gate에 대응하는 제약 조건은 독립적이기 때문이다. 또한 회로의 gate의 입력과 출력에 해당하는 wire의 값은 gate의 출력이 아닌 weight matrix를 통해 표현한다.

다음으로 PLONK는 일반적인 회로의 형태를 가지고 있다. 최초의 입력을 바탕으로 덧셈과 곱셈 gate로 하나의 완성된 회로를 구성한다. 즉, wire는 gate간의 값을 전달해주는 역할을 하고 모든 값은 gate에서 계산되는 것이다. 이 경우 다양한 연산을 지원하는 gate를 통해 회로의 구성을 더욱 유연하게 할 수 있어 ZK-SNARK에서 문제가 되었던 복잡한 연산을 효율적으로 처리할 수 있다. 예를 들어,  하지만 ZK-SNARK와 다르게 wire가 값을 전달하기 때문에 다음 gate는 자신의 입력이 정말 이전 gate의 출력인지에 대한 검증을 추가로 진행해야한다. PLONK의 오른쪽 그림을 보자. Prover는 문제의 해를 모르는 상황에서 회로를 조작하기 위해 첫번째 덧셈 gate에서 나온 $(x + 1)$을 무시하고 자신이 가지고 있는 다른 값을 넣었다고 가정하자. 만약 gate의 연산에 대한 검증만 진행하게 된다면, 두번째 곱셈 gate는 자신의 입력 값이 잘못되었는지 알 수 없기 때문에 그대로 연산을 수행하게 된다. 이를 막기 위해선 첫번째 gate의 출력과 두번째 gate의 입력이 동일한지에 대한 제약 조건이 추가되어야 한다. 즉, PLONK에서는 gate의 연산에 대한 제약 조건과 함께 하나의 wire는 동일한 값을 가지는 지에 대한 제약 조건또한 함께 고려되어야 한다. 지금부터 PLONK에서 두 제약 조건 및 이를 바탕으로 증명을 생성하는 과정을 살펴볼 것이다.

klaytn

## 2. 산술 회로

우리는 ZK-SNARK의 과정과 직접 비교할 수 있도록 같은 문제로부터 시작할 것이다. 이전 예시에서 보았듯이, 최소 연산 단위로 gate를 만들고 이를 통해 전체 회로를 설계한다.




<그림10 PLONK 산술 회로-2>

klaytn

Provers는 문제에 대한 비공개 입력과 회로 내부의 wire들의 값(witness라고 부른다.)을 알고 있음을 증명해야 한다. 이를 위해선 앞서 이야기했듯 두 가지 제약 조건을 만족함을 보여야 한다.

1) Gate 제약 조건: 모든 gate에 따라 $a_i$, $b_i$, $c_i$는 정확하게 계산되어야 한다.
2) Copy 제약 조건: 같은 wire라면 같은 값을 가진다. 예를 들어, $c_0 = a_2 = b_2 = a_3$이다.

먼저 Gate 제약 조건을 알아보자.

3. Gate 제약 조건

Gate 제약 조건은 다음 방정식의 형태를 가진다.

$$(Q_{L_i})a_i + (Q_{R_i})b_i + (Q_{O_i})c_i + (Q_{M_i})a_ib_i + Q_{C_i} = 0$$

$Q_{L_i}, Q_{R_i}, \cdots, Q_{M_i}$는 각 gate가 $\{$왼쪽 입력, 오른쪽 입력, 출력, 곱셈, 상수$\}$에 할당하는 weight를 뜻하며 이에 따라 gate의 종류가 바뀐다. 이 때 우리는 최종적인 gate 제약 조건 다항식이 0으로 evaluation되길 원하기 때문에 우변은 0으로 고정한다. 예를 들어, + gate는 다음과 같이 구성된다.

$$Q_{L_i} = Q_{R_i} = 1,\ Q_{O_i} =- 1,\ Q_{M_i} = Q_{C_i} = 0$$

$$a_i + b_i - c_i + 0 + 0 = 0 \rightarrow a_i + b_i = c_i$$

위의 회로에서, 우리는 0번 gate부터 6번 gate까지의 총 7개의 제약 조건을 가진다.

$$[(i,\ Q_{L_i}),\ (i,\ Q_{R_i}),\ (i,\ Q_{O_i}),\ (i,\ Q_{M_i}),\ (i,\ Q_{C_i})]^6_{i=0}$$

표로 각 게이트의 제약 조건과 wire값을 나타내면 다음과 같다.

| idx | a_i | b_i | c_i | qL | qR | qO | qM | qC |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | -1 | 0 | 0 |
| 1 | 4 | 3 | 5 | 1 | 1 | -1 | 0 | 0 |
| 2 | 2 | 3 | 3 | 0 | 0 | -1 | 1 | 0 |
| 3 | … | … | … | … | … | … | … | … |
| 4 | … | … | … | … | … | … | … | … |
| 5 | … | … | … | … | … | … | … | … |
| 6 | 2 | 1 | 7 | 1 | 1 | -1 | 0 | 0 |

<그림11 제약 조건과 wire 값들 2>

우리는 $n$개의 점을 가지고 있을 때 $n - 1$차 다항식을 만들 수 있다는 것을 알고 있다. 본 예시에서는 6차 다항식을 Lagrange interpolation[8]을 통해 만들 수 있다. 즉, 우리의 제약 조건은 다음과 같이 gate의 index를 입력으로 하는 방정식으로 표현할 수 있다.

[8] 후에 다루겠지만, 실제로 각 domain은 i가 아닌 w^i를 사용하기 때문에 FFT/IFFT를 통해 다항식의 형태를 변환한다.

$$Q_L(x)a(x) + Q_R(x)b(x) + Q_O(x)c(x) + Q_M(x)a(x)b(x) + Q_C(x) = 0$$

$$x \in \{0, 1, 2, 3, 4, 5, 6\}$$

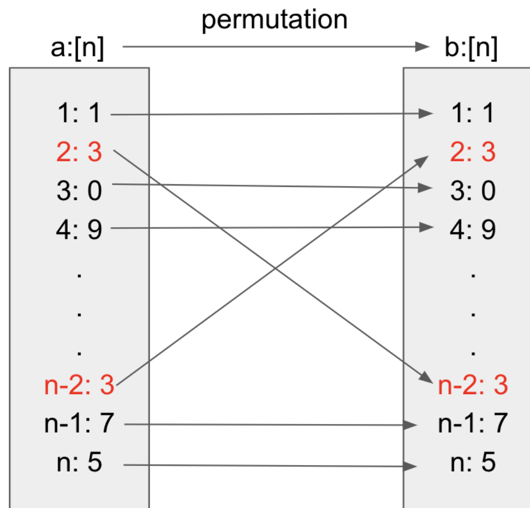
이 후 Prover는 위 gate 제약 조건 방정식을 만족하는 $a(x)$, $b(x)$, $c(x)$를 알고 있다는 증명을 생성한다.

4. Copy 제약 조건

Copy 제약 조건은 조금 복잡하다. 우리의 목표는 같은 wire는 같은 값을 가지고 있다는 것을 증명하는 것이다. 이는, 만약 우리가 두 값을 바꾸게 되더라도 결과는 동일하다는 것을 뜻한다. 우리는 이를 permutation을 통해 확인할 수 있다.

a) Permutation check

본래 집합 $a = (a_1, a_2, \cdots, a_n)$이 있을 때, $a_2$를 $b_{n-2}$로 보내고 $a_{n-2}$를 $b_2$로 보내는 permutation $\sigma: [n] \rightarrow [n]$이 있다. 우리는 $b_i = a_{\sigma(i)}$를 만족하는 다음 매핑을 가진다.




<그림 12 Permutation>

Prover가 올바른 permutation을 생성했다면 각 집합 간의 1대1 대응 관계를 만족해야 한다. 또한 우리는 같은 값을 가지는 index만 다루기 때문에 두 집합은 동일해야 한다. 이를 보이기 위해선 두 집합 내를 돌며 모든 값들을 비교해볼 수 있다. 하지만 이는 값을 비교하는 연산또한 증명을 생성해야 하기

때문에 매우 비용이 커진다. 대신, 우리는 매우 큰 집합에서 다음과 같은 방법을 통해 두 집합이 일대일 대응을 만족한다는 것을 보일 수 있다.

1. RLC를 위한 랜덤 값 β, γ를 선택한다.
2. 우리는 값 뿐만 아니라 순서또한 중요하기 때문에[9] index와 값을 함께 인코딩한다:
   $a'_i = a_i + \beta \cdot i,\ b'_i = b_i + \beta \cdot \sigma(i)$
3. 다음을 확인한다; $\prod_{i \in [n]} a'_i = \prod_{i \in [n]} b'_i$

3번 과정은 두 집합내의 모든 인코딩된 값을 곱하여 축적한다. 매우 큰 집합에 대해서 1대1 대응이 아닌 집합이 우연히 같은 축적 값을 가질 확률은 매우 낮다. 이를 이용해 단 한번의 비교를 통해 두 집합의 1대1 대응을 검증할 수 있다.

b) Copy 제약 조건

우리는 위에서 사용한 집합 간 1대1 대응을 검증하는 방법을 적용할 것이다. 이번엔 $3n$개의 점 $\{a_i, b_i, c_i\}_{i \in [n]}$에 대하여 $\sigma(2) = 2n - 2$를 만족하는 permutation[10]이 있다고 하자. $b_{n-2}$는 전체 $3n$개의 점 중 $2n - 2$번째 점이다. 이를 일렬로 나열하면 다음과 같다.

$$a_1, a_2, \cdots, a_{n-1}, a_n, b_1, b_2, \cdots, b_{n-2}, b_{n-1}, b_n, c_1, c_2, \cdots, c_n$$

$$\rightarrow a_1, b_{n-2}, \cdots, a_{n-1}, a_n, b_1, b_2, \cdots, a_2, b_{n-1}, b_n, c_1, c_2, \cdots, c_n$$

$a_2 = b_{n-2}$를 만족한다면 두 집합은 1대1 대응이고 동일하다.

이를 확인하기 위해선, 우리는 다음 방정식을 만족하는지 검증하면 된다.

$$\prod_{i \in [n]} (a_i + \beta \cdot i + \gamma)(b_i + \beta \cdot (n + i) + \gamma)(c_i + \beta \cdot (2n + i) + \gamma)$$

---

[9] 그림 9에서 서로 다른 값을 가지는 index 3과 4의 순서를 바꾸면 서로 동일하지 않은 permutation이지만 값만 인코딩할 시 제약 조건이 만족된다.
[10] 실제 프로토콜에서 같은 값을 가지는 wire들을 해시맵 형태로 저장한 후 shift 연산을 통해 permutation 매핑을 생성한다. 예를 들어 모두 1을 값으로 가지는 (1, a1), (1, b2), (1, c4)가 있을 때 a1를 b2로, b2를 c4로, c4를 a1으로 매핑한다.

$$= \prod_{i \in [n]} (a_i + \beta \cdot \sigma(i) + \gamma)(b_i + \beta \cdot \sigma(n + i) + \gamma)(c_i + \beta \cdot \sigma(2n + i) + \gamma)$$

두 집합의 index와 값을 인코딩한 값을 모두 곱하여 축적한 두 스칼라 값이 동일한지 비교하면 permutation을 적용한 집합과 이전 집합이 동일한지 검증할 수 있다. 우리는 증명을 생성하기 위해 위 연산을 다항식으로 표현해야 한다. 전체 값을 모두 축적하여 스칼라 값을 얻는 것 대신, 입력까지만 축적하는 형태로 다항식을 쉽게 표현할 수 있다. 예를 들어 $k$를 입력으로 받으면 1부터 $k$까지의 축적 값을 반환하는 다항식을 만들 수 있다. 우리는 다음과 같이 다항식 $Z$를 정의한다. 식이 너무 길어지기 때문에 인코딩한 값을 각각 $f_j$, $g_j$로 치환했다.

$$Z(X) = L_1(X) + \sum_{i=1}^{n-1} (L_{i+1}(X) \prod_{j=1}^{i} \frac{f_j}{g_j}$$

- 이 때 시작점이 $\omega^0$이 아닌 $\omega^1$인 것에 의문이 들 수 있다. 실제로 $\omega$는 1의 거듭제곱근인 순환군의 원소이기 때문에 매 점에서 정의만 만족한다면 시작점은 상관없다.

위 식에 따라 $Z$를 계산해보자.

$$Z(\omega^1) = 1$$

$$Z(\omega^2) = \frac{f_1}{g_1}$$

$$\cdots$$

$$Z(\omega^n) = \frac{f_1}{g_1} \cdot \frac{f_2}{g_2} \cdots \frac{f_{n-1}}{g_{n-1}}$$

$$\rightarrow Z(\omega^i) f_i - Z(\omega^{i+1}) g_i = 0$$

$Z(\omega^1) = 1$이 단순히 $L_1(X)$에 의해 계산되는 것에 의문이 생길 수 있다.

$$Z(\omega^n) f_n - Z(\omega^{n+1}) g_n = 0$$

$$\rightarrow \frac{f_1}{g_1} \cdot \frac{f_2}{g_2} \cdots \frac{f_{n-1}}{g_{n-1}} \cdot f_n = Z(\omega^1) g_n$$

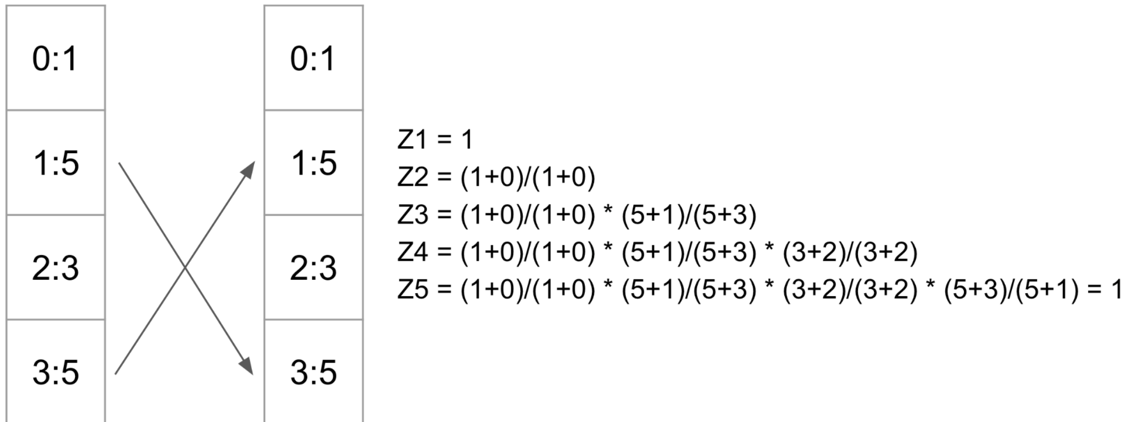
$$\rightarrow \frac{f_1}{g_1} \cdot \frac{f_2}{g_2} \cdots \frac{f_{n-1}}{g_{n-1}} \cdot \frac{f_n}{g_n} = Z(\omega^1) = 1$$

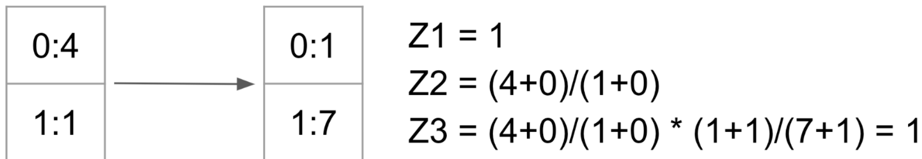
klaytn

따라서 본래의 $Z(X)$는 다음과 같은 형태인 것을 알 수 있다.

$$Z(X) = L_1(X)\prod_{k=1}^{n}\frac{f_k}{g_k} + \sum_{i=1}^{n-1}(L_{i+1}(X)\prod_{j=1}^{i}\frac{f_j}{g_j}$$

$Z$가 어떻게 permutation을 검증할 수 있는지에 대한 직관적인 예시는 다음을 통해 확인할 수 있다.




<그림 13 Permutation 검증 예시>

최종적으로 Verifier가 검증하고 싶은 제약 조건은 다음과 같다.

$$L_1(X)(Z(X) - 1) = 0$$

$$Z(X)f(X) - Z(X \cdot \omega)g(X) = 0$$

5. 제약 조건 및 개념 정리

우리는 PLONK에서 사용할 gate 제약 조건과 copy 제약 조건을 모두 알아보았다. 지금부터는 제약 조건을 최종적으로 정리한 후 메인 프로토콜의 본격적인 증명 생성과 검증 과정을 살펴볼 것이다.

<표기법 1 Plonk 표기법>

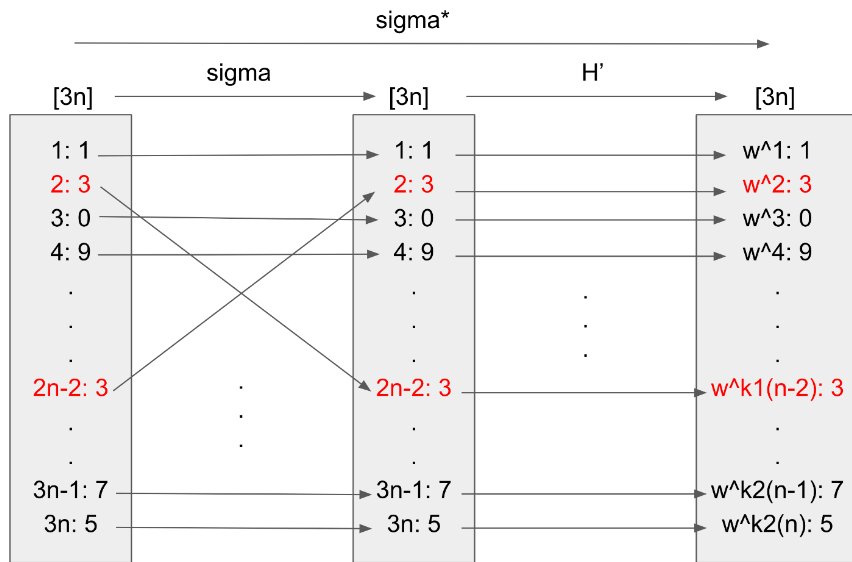
| 표기 | 설명 |
| --- | --- |
| $q_L(X)$ | Gate의 왼쪽 입력에 대한 selector 다항식 |
| $q_R(X)$ | Gate의 오른쪽 입력에 대한 selector 다항식 |
| $q_O(X)$ | Gate의 출력에 대한 selector 다항식 |
| $q_M(X)$ | Gate의 곱셈 연산에 대한 selector 다항식. |
| $q_C(X)$ | Gate의 상수 입력에 대한 selector 다항식. |
| $Z(X)$ | Copy 제약 조건 다항식. |
| $S_{ID}(X)$ | 항등 순열 다항식. |
| $\sigma(X)$ | 순열 다항식. |
| $L_i(X)$ | 라그랑주 기저 다항식. |
| $\omega$ | 유한군 내의 곱셈 집합에서 정의되는 $n$의 제곱근 ($\omega^n = 1 \, mod \, q$). FFT/IFFT를 위해 다항식의 evaluation 정의역으로 사용된다. |
| $n$ | 회로의 행 크기. 회로는 $n$개의 행을 가진다. ($\omega^0, \omega^1, \cdots, \omega^{n-1}$) |
| $x$ | 다항식 evaluation을 위해 선택된 좌표값. |

klaytn

a) 제약 조건 정리

1) $q_L(X),\ q_R(X),\ q_O(X),\ q_M(X),\ q_C(X)$: 회로 내 gate의 selector 다항식

2) $S_{ID1}(X) = X,\ S_{ID2}(X) = k_1X,\ S_{ID3}(X) = k_2X$: 항등 순열이며, $k_1,\ k_2$는 각 순열의 값이 겹치지 않도록 선택된다. 우리가 다루는 정의역이 $X \in H$일 때, $H' = H \cup (k_1H) \cup (k_2H)$라 하자. 즉, $H'$는 3개의 항등 순열로 계산되는 모든 값들의 집합이다. 이 때, $H'$는 다음과 같이 index $i$를 $\omega^i$로 매핑시킨다: $i \rightarrow \omega^i,\ n + i \rightarrow k_1\omega^i,\ 2n + i \rightarrow k_2\omega^i$. 복잡해보이지만, 사실은 앞서 설명했듯이 다항식 간의 형태 변환을 위한 FFT/IFFT를 사용하기 위해 정의역 $i \in [3n]$을 $\omega^i$로 대응을 시킨 것이다. 이 때 $[3n] \rightarrow \sigma \rightarrow H'$ 매핑을 하나의 $\sigma^*$로 묶자. 예를 들어, $\sigma(2) = 2n - 2,\ \sigma(2n - 2) = 2$를 만족하는 $\sigma$가 있을 때, $\sigma^*(2) = H'(\sigma(2)) = H'(2n - 2) = k_1\omega^{n-2}$를 만족한다.




<그림 13 Permutation 매핑 예시>

최종적으로 우리는 permutation 다항식을 다음과 같이 표현할 수 있다.

$$S_{\sigma 1}(X) = \sum_{i=1}^{n} \sigma^*(i)L_i(X)$$

$$S_{\sigma 2}(X) = \sum_{i=1}^{n} \sigma^*(n + i)L_i(X)$$

$$S_{\sigma 3}(X) = \sum_{i=1}^{n} \sigma^{*}(2n + i)L_i(X)$$

b) SNARK 증명 Relation

SNARK 증명 Relation은 다음과 같이 정의된다. 이는 $l$[11]개의 공개 입력과 $3n - l$개의 witness(비공개 값)로 이루어 진다.

$$R \subset \mathbf{F}^{l} \times \mathbf{F}^{3n-l}$$

$$x = (\omega_i)_{i \in [l]}, \omega = (\omega_i)^{3n}_{i=l+1}$$

c) 선형화 트릭

만약 동일한 ζ에서 evaluation되는 $f_i(X)_{i \in [n]}$이 있다고 하자. 이 때 $h_i(X) = f_i(X) - f_i(\zeta) = 0$을 검증하기 위해 각각의 $f_i$에 대하여 증명 및 검증을 진행하는 것보다, 랜덤 값 $\alpha$를 이용한 RLC를 통해 다음과 같이 선형화된 $H(X)$를 생성하고 이에 대한 증명 및 검증을 한번만 진행하면 더욱 효율적이다.

$$H(X) = h_1(X) + \alpha h_2(X) + \cdots + \alpha^{n} h_n(X)$$

d) Transcript[12]

ZK-SNARK에서 봤듯이, 우리는 non-interactive한 시스템을 설계해야 한다. PLONK에서 우리는 Fiat-Shamir heuristic을 통해 랜덤 값(챌린지를 포함하여)을 생성한다. 이를 위해 transcript가 존재하며 이는 common preprocessed input을 이어붙이는 방식으로 만들어 진다. 추후 transcript를 적절히 해시하여 랜덤 값을 얻는다.

---

[11] 공개 입력의 경우 gate 제약 조건에서 왼쪽 입력 값을 1로 두고 상수 값을 -1로 두는 방식으로 설정된다. 즉, 상수 값 내에 공개 입력 값의 다항식이 포함되어 있는 형태이다.
[12] PLONK의 구현에서는 주로 Merlin Transcript를 사용한다. https://merlin.cool/use/protocol.html

6. 메인 프로토콜

**Common preprocessed input :**

$$n, \quad (x \cdot [1]_1, \cdots, x^{n+5} \cdot [1]_1), \quad (q_{M_i}, q_{L_i}, q_{R_i}, q_{O_i}, q_{C_i})^n_{i=1}, \quad \sigma^*,$$

$$q_M(X) = \sum_{i=1}^{n} q_{M_i} L_i(X), \ q_L(X) = \sum_{i=1}^{n} q_{L_i} L_i(X), \ q_R(X) = \sum_{i=1}^{n} q_{R_i} L_i(X),$$

$$q_O(X) = \sum_{i=1}^{n} q_{O_i} L_i(X), \ q_C(X) = \sum_{i=1}^{n} q_{O_i} L_i(X),$$

$$S_{\sigma 1}(X) = \sum_{i=1}^{n} \sigma^*(i) L_i(X), \ S_{\sigma 2}(X) = \sum_{i=1}^{n} \sigma^*(n + i) L_i(X),$$

$$S_{\sigma 3}(X) = \sum_{i=1}^{n} \sigma^*(2n + i) L_i(X),$$

**Public input:** $l, \ (\omega_i)_{i \in [l]}$

지금부터 Prover는 총 5개의 라운드를 통해 증명을 생성하게 된다.

**Round 1 :**

Generate random blinding scalars $(b_1, \cdots, b_9) \in F$ (Field $F$)

Compute wire polynomials $a(X), \ b(X), \ c(X)$ :

$$a(X) = (b_1 X + b_2) Z_H(X) + \sum_{i=1}^{n} \omega_i L_i(X)$$

$$b(X) = (b_3 X + b_4) Z_H(X) + \sum_{i=1}^{n} \omega_{n+i} L_i(X)$$

$$c(X) = (b_5 X + b_6) Z_H(X) + \sum_{i=1}^{n} \omega_{2n+i} L_i(X)$$

klaytn

Compute $[a]_1 := [a(x)]_1$, $[b]_1 := [b(x)]_1$, $[c]_1 := [c(x)]_1$

First output of $\mathbf{P}$ is $([a]_1, [b]_1, [c]_1)$

<PLONK 증명 생성 1>

이 때 $Z_H(X) = X^n - 1$이다. 각 식을 잘 살펴보면, 입력을 다항식으로 표현한 부분과 랜덤한 blinding 값들이 선형결합되어 있다. 이 때 랜덤한 blinding 값들은 이름 그대로 각 $a(X)$, $b(X)$, $c(X)$를 숨기기 위해서 사용된다. 만약 $a(X) = \sum_{i=1}^{n} \omega_i L_i(X)$를 사용한다면 $\omega_i$에 대한 정보를 $a(X)$를 통해 온전히 알 수 있게 되기 때문에 zero-knowledge 특성을 잃게 된다. 각 입력에 대한 다항식을 모두 계산하고 나면 $x$로 evaluation한 값을 commit한다.

**Round 2 :**

Compute permutation challenges $(\beta, \gamma) \in F$ (Field $F$) :

$$\beta = \mathcal{H}(transcript, 0), \gamma = \mathcal{H}(transcript, 1)$$

Compute permutation polynomial $z(X)$ :

$$z(X) = (b_7 X^2 + b_8 X + b_9) Z_H(X)$$

$$+ L_1(X) + \sum_{i=1}^{n-1} (L_{i+1}(X) \prod_{j=1}^{i} \frac{(\omega_j + \beta\omega^j + \gamma)(\omega_{n+j} + \beta k_1 \omega^j + \gamma)(\omega_{2n+j} + \beta k_2 \omega^j + \gamma)}{(\omega_j + \sigma^*(j)\beta + \gamma)(\omega_{n+j} + \sigma^*(n+j)\beta + \gamma)(\omega_{2n+j} + \sigma^*(2n+j)\beta + \gamma)})$$

Compute $[\boldsymbol{z}]_1 := [z(x)]_1$

Second output of $\mathbf{P}$ is $([\boldsymbol{z}]_1)$

<PLONK 증명 생성 2>

두 번째 라운드도 마찬가지이다. $(b_7 X^2 + b_8 X + b_9) Z_H(X)$와의 선형 결합을 통해 본래 $Z(X)$값을 숨긴다. 뒷 부분은 위에서 계산한 $Z(X)$와 동일하다. 다항식을 계산한 후 $x$에 대하여 evaluation한 값을 commit한다.

**Round 3 :**

Compute quotient challenge $\alpha \in F\ (Field\ F)$ :

$$\alpha = \mathcal{H}(transcript)$$

Compute quotient polynomial $t(X)$ :

$t(X) =$

$(a(X)b(X)q_M(X) + a(X)q_L(X) + b(X)q_R(X) + c(X)q_O(X) + PI(X) + q_C(X))\frac{1}{Z_H(X)}$

⋯ **(1) Gate Constraint**

$+ ((a(X) + \beta X + \gamma)(b(X) + \beta k_1 X + \gamma)(c(X) + \beta k_2 X + \gamma)z(X))\frac{\alpha}{Z_H(X)}$

$- ((a(X) + \beta S_{\sigma_1}(X) + \gamma)(b(X) + \beta S_{\sigma_2}(X) + \gamma)(c(X) + \beta S_{\sigma_3}(X) + \gamma)z(X\omega))\frac{\alpha}{Z_H(X)}$

⋯ **(2) Copy Constraint - define Z**

klaytn

$+ (z(X) - 1)L_1(X)\frac{\alpha^2}{Z_H(X)}$

⋯ **(3) Copy Constraint - Initial condition of Z**

Split $t(X)$ into $degree < n$ polynomials $t'_{lo}(X)$, $t'_{mid}(X)$ and $t'_{hi}(X)$ of $degree$ at most $n + 5$, such that

$t(X) = t'_{lo}(X) + X^n t'_{mid}(X) + X^{2n} t'_{hi}(X)$

Now choose random scalars $b_{10}, b_{11} \in F$ and define

$t_{lo}(X) := t'_{lo}(X) + b_{10}X^n$, $t_{mid}(X) := t'_{mid}(X) - b_{10} + b_{11}X^n$, $t_{hi}(X) := t'_{hi}(X) - b_{11}$

Note that we have $t(X) = t_{lo}(X) + X^n t_{mid}(X) + X^{2n} t_{hi}(X)$

Compute $[t_{lo}] := [t_{lo}(x)]_1$ , $[t_{mid}]_1 := [t_{mid}(x)]_1$ , $[t_{hi}(x)]_1$

Third output of $\mathbf{P}$ is $([t_{lo}]_1, [t_{mid}]_1, [t_{hi}]_1)$

<PLONK 증명 생성 3>

다소 복잡해보이는 $t(X)$가 있다. 하지만 식의 구조를 살펴보면 α에 따라 서로 다른 제약 조건을 선형 결합한 구조인 것을 알 수 있다. α의 차수에 따라 식을 나눠서 살펴보자.

1) $\alpha^0$를 보면 gate 제약 조건이라는 것을 쉽게 확인할 수 있다. 이전에 말했듯이 $q_C(X)$는 $PI(X) + q_c(X)$로 구성되어 있는 것을 알 수 있다.
2) $\alpha^1$를 보면 copy 제약 조건의 $Z$를 나타낸 것을 알 수 있다. 위에서 살펴본 $Z(\omega^i)f_i - Z(\omega^{i+1})g_i$ 식과 동일하다.
3) $\alpha^2$는 copy 제약 조건의 $Z$의 초기 조건이다.

즉, $t(X)$는 우리가 만든 gate, copy 제약 조건을 모두 선형 결합한 형태이다. KZG commitment에서와 유사하게, 만약 Prover가 올바른 $a(X)$, $b(X)$, $c(X)$를 가지고 있다면 모든 제약 조건 다항식은 $\omega^i$($i \in [n]$)에서 0이 되며, 이를 $Z_H(X)$로 나눈 $t(X)$가 존재해야 한다.

그런데 $t(X)$는 최대 $3n + 5$[13]의 차수를 가진다. 따라서 $t(X)$를 그래도 evaluation하기 위해선 최대 $x^{3n+5}$까지 준비해야 한다는 것을 뜻한다. 대신에, $t(X)$를 $n$차수에 따라 병렬적으로 $t_{lo}$, $t_{mid}$, $t_{hi}$로 분리하면 최대 $x^{n+5}$로 evaluation이 가능하다. Prover와 Verifier의 공통 입력에서 $x^{n+5}$까지 계산한 것이 이를 위해서이다.

**Round 4 :**

Compute evaluation challenge $\zeta \in F$ :

$$\zeta = \mathcal{H}(transcript)$$

Compute opening evaluations :

$$\bar{a} = a(\zeta),\ \bar{b} = b(\zeta),\ \bar{c} = c(\zeta),\ \overline{s_{\sigma1}} = S_{\sigma1}(\zeta),\ \overline{s_{\sigma2}} = S_{\sigma2}(\zeta),$$

[13] Z의 조건을 나타낸 다항식에서 3 * (n + 1) + (n + 2) - (n) = 3n + 5의 차수를 가진다.

klaytn

$$\overline{z_\omega} = z(\zeta\omega)$$

Fourth output of $\mathbf{P}$ is $(\bar{a}, \bar{b}, \bar{c}, \overline{s_{\sigma1}}, \overline{s_{\sigma2}}, \overline{z_\omega})$

< PLONK 증명 생성 4>

챌린지 값을 계산한 후 이를 통해 각 다항식에 대한 evaluation을 구한다. $Z$의 경우 $\zeta$를 그대로 사용할 시 $\zeta - 1$까지의 축적 값만 반영되기 때문에 한 차수 위인 $\zeta\omega$를 사용하여 evaluation하는 것을 확인하자. 여기서 왜 $S_{\sigma3}(\zeta)$가 없는 지에 대한 의문이 생길 수 있다. 논문 상에서는 계산량을 줄이기 위해 evaluation를 줄였다고 하지만, 실제 구현[14]을 살펴본 결과 $S_{\sigma3}(\zeta)$을 대부분 계산하고 있는 것을 확인할 수 있었다. 다만 본 리서치는 PLONK 논문의 구현을 기본으로 내용을 전개할 것이다.

**Round 5 :**

Compute opening challenge $\nu \in F$ :

$$\nu = \mathcal{H}(transcript)$$

Compute linearisation polynomial $r(X)$ :

$$r(X) =$$

$$[\bar{a}\bar{b} \cdot q_M(X) + \bar{a} \cdot q_L(X) + \bar{b} \cdot q_R(X) + \bar{c} \cdot q_O(X) + PI(\zeta) + q_C(X)]$$

$\cdots$ **(1) Gate Constraint**

$$+ \alpha[(\bar{a} + \beta\zeta + \gamma)(\bar{b} + \beta k_1\zeta + \gamma)(\bar{c} + \beta k_2\zeta + \gamma) \cdot z(X)$$

$$- (\bar{a} + \beta\overline{s_{\sigma1}} + \gamma)(\bar{b} + \beta\overline{s_{\sigma2}} + \gamma)(\bar{c} + \beta \cdot S_{\sigma_3}(X) + \gamma)\overline{z_\omega}]$$

$$+ \alpha^2[(z(X) - 1)L_1(\zeta)]$$

$\cdots$**(2) Define Z**

[14] https://github.com/ZK-Garage/plonk, https://github.com/dusk-network/plonk

$$- Z_H(\zeta) \cdot (t_{lo}(X) + \zeta^n t_{mid}(X) + \zeta^{2n} t_{hi}(X))$$

$$\cdots\textbf{(3)}\ t(X) \cdot Z_H(X)$$

Compute opening proof polynomial $W_\zeta(X)$ :

$$W_\zeta(X) = \frac{1}{X-\zeta}(r(X) + v(a(X) - \bar{a}) + v^2(b(X) - \bar{b}) + v^3(c(X) - \bar{c})$$

$$+ v^4(S_{\sigma_1}(X) - \overline{s_{\sigma1}}) + v^5(S_{\sigma_2}(X) - \overline{s_{\sigma2}}))$$

Compute opening proof polynomial $W_{\zeta\omega}(X)$ :

$$W_{\zeta\omega}(X) = \frac{(z(X) - \overline{z_\omega}}{X - \zeta\omega}$$

Compute $[W_\zeta]_1 := [W_\zeta(x)]_1$ , $[W_{\zeta\omega}]_1 := [W_{\zeta\omega}(x)]_1$

The fifth output of $\mathbf{P}$ is $([W_\zeta]_1, [W_{\zeta\omega}]_1)$

**Return**

$$\pi_{snark} = ([a]_1, [b]_1, [c]_1, [z]_1, [t_{lo}]_1, [t_{mid}]_1, [t_{hi}]_1, [W_\zeta]_1, [W_{\zeta\omega}]_1,$$

$$\bar{a}, \bar{b}, \bar{c}, \overline{s_{\sigma1}}, \overline{s_{\sigma2}}, \overline{z_\omega})$$

Compute multipoint evaluation challenge $u \in F$:

$$u = \mathcal{H}(transcript)$$

<PLONK 증명 생성 5>

최종적으로 증명을 생성하는 마지막 라운드이다. 선형화 다항식 $r(X)$의 형태를 살펴보면 우리가 앞서 살펴봤던 선형화 트릭을 통해 gate, copy 제약 조건과 $t(X)$를 하나로 합친 것을 알 수 있다. 유도 과정을 살펴보면, gate, copy 제약 조건에 대한 식을 $p_i(X)$라 하면 $t(X) = \frac{p_1(X)}{Z_H(X)} + \frac{p_2(X)}{Z_H(X)} + \frac{p_3(X)}{Z_H(X)}$로 나타낼 수

klaytn

있다. $p(X) = p_1(X) + p_2(X) + p_3(X)$라 하면 $Z_H(X)$를 양 변에 곱하여 $r(X) = p(X) - t(X)Z_H(X)$를 유도할 수 있다.

다음으로 최종적인 증명 $W_\zeta(X)$, $W_{\zeta\omega}(X)$를 계산한다. 이는 우리가 이미 살펴본 KZG commitment의 batch 버전과 유사한 것을 쉽게 알 수 있다. 먼저 $W_\zeta(X)$를 살펴보면 $\zeta$에서 evaluation되는 $r(X)$와 각 다항식을 RLC하여 계산했다. $W_{\zeta\omega}(X)$는 $Z$가 $\zeta\omega$에서 evaluation되어야 하므로 따로 계산되었다. 최종적으로 각 증명 다항식의 commitment를 계산하고 최종 SNARK 증명인 15개의 commitment 및 evaluation 값을 생성한다.

다음으로 Verifier가 증명을 어떻게 검증하는 지를 살펴보자.

Verifier preprocessed input :

$[q_M]_1 := q_M(x) \cdot [1]_1$, $[q_L]_1 := q_L(x) \cdot [1]_1$, $[q_R]_1 := q_R(x) \cdot [1]_1$, $[q_O]_1 := q_O(x) \cdot [1]_1$,

$[q_C]_1 := q_C(x) \cdot [1]_1$, $[s_{\sigma_1}]_1 := S_{\sigma_1}(x) \cdot [1]_1$, $[s_{\sigma_2}]_1 := S_{\sigma_2}(x) \cdot [1]_1$,

$[s_{\sigma_3}]_1 := S_{\sigma_3}(x) \cdot [1]_1$, $x \cdot [1]_2$

$V((\omega_i)_{i\in[\ell]}, \pi_{snark})$ :

1. Validate $([a]_1, [b]_1, [c]_1, [z]_1, [t_{lo}]_1, [t_{mid}]_1, [t_{hi}]_1, [W_\zeta]_1, [W_{\zeta\omega}]_1) \in \mathbb{G}_1^9$
2. Validate $(\bar{a}', \bar{b}, \bar{c}, \overline{s_{\sigma1}}, \overline{s_{\sigma2}}, \overline{z_\omega}) \in F^6$
3. Validate $(\omega_i)_{i\in[\ell]} \in F^\ell$
4. Compute challenges $\beta, \gamma, \alpha, \zeta, v, u \in F$ as in prover description, from the common inputs, public input, and elements of $\pi_{snark}$
5. Compute zero polynomial evaluation $Z_H(\zeta) = \zeta^n - 1$
6. Compute Lagrange polynomial evaluation $L_1(\zeta) = \frac{\omega(\zeta^n-1)}{n(\zeta-\omega)}$
7. Compute public input polynomial evaluation $PI(\zeta) = \Sigma_{i\in[\ell]}\omega_i L_i(\zeta)$

klaytn

8. To save a verifier scalar multiplication, we split $r$ into its constant and non-constant terms. Compute $r$'s constant term :

$$r_0 := PI(\zeta) - L_1(\zeta)\alpha^2 - \alpha(\bar{a} + \beta\overline{s_{\sigma 1}} + \gamma)(b' + \beta\overline{s_{\sigma 2}} + \gamma)(\bar{c} + \gamma)\,\overline{z_{\omega}},$$

And let $r'(X) := r(X) - r_0$

9. Compute first part of batched polynomial commitment $[D]_1 := [r']_1 + u \cdot [z]_1$ :

$$[D]_1 := \bar{a} \cdot \bar{b} \cdot [q_M]_1 + \bar{a} \cdot [q_L]_1 + \bar{b} \cdot [q_R]_1 + \bar{c} \cdot [q_O]_1 + [q_C]_1$$
$$+ ((\bar{a} + \beta\zeta + \gamma)(\bar{b} + \beta k_1\zeta + \gamma)(\bar{c} + \beta k_2\zeta + \gamma)\alpha + L_1(\zeta)\alpha^2 + u) \cdot [z]_1$$
$$- (\bar{a} + \beta\overline{s_{\sigma 1}} + \gamma)(b' + \beta\overline{s_{\sigma 2}} + \gamma)\alpha\beta\overline{z_{\omega}} \cdot [s_{\sigma_3}]_1 - Z_H(\zeta)([t_{lo}]_1 + \zeta^n \cdot [t_{mid}]_1 + \zeta^{2n} \cdot [t_{hi}]_1$$

10. Compute full batched polynomial commitment $[F]_1$ :

$$[F]_1 := [D]_1 + v \cdot [a]_1 + v^2 \cdot [b]_1 + v^3 \cdot [c]_1 + v^4 \cdot [s_{\sigma_1}]_1 + v^5 \cdot [s_{\sigma_2}]_1$$

11. Compute group-encoded batch evaluation $[E]_1$ :

$$[E]_1 := (-r_0 + v\bar{a} + v^2\bar{b} + v^3\bar{c} + v^4\overline{s_{\sigma 1}} + v^5\overline{s_{\sigma 2}} + u\overline{z_{\omega}}) \cdot [1]_1$$

12. Batch validate all evaluations:

$$e([W_{\zeta}]_1 + u \cdot [W_{\zeta\omega}]_1, [x]_2) == e(\zeta \cdot [W_{\zeta}]_1 + u\zeta\omega \cdot [W_{\zeta\omega}]_1 + [F]_1 - [E]_1, [1]_2)$$

<PLONK 증명 검증>

먼저 Verifier는 공통 입력으로부터 필요한 값을 계산한다. 그 후 라운드 1부터 4까지는 Prover가 전달한 증명이 모두 우리가 다루고 있는 범위 내에 있는지 range check를 한다. 라운드 5부터 7까지는 검증에 필요한 $Z_H(X)$, $L_1(\zeta)$, $PI(\zeta)$등을 미리 계산해 놓는다. 이 때 $L_1(\zeta)$값이 잘못 계산되어 있는데, $L_x(X) = \frac{c_x(X^n-1)}{X-x}$이기 때문에 $L_1(\zeta) = \frac{\zeta^n-1}{n(\zeta-1)}$이다. 라운드 8부터 본격으로 검증을 위한 다항식 및 값들을

생성한다. 라운드 8에선 $r(X)$의 상수항인 $r_0$를 먼저 계산하여 $r'(X) := r(X) - r_0$을 계산한다. 상수항 $r_0$은 직접 $r(X)$에서 상수항을 살펴보면 쉽게 알 수 있다. 라운드 9는 라운드 8에서 계산한 $r'$의 commitment와 copy 제약 조건 $z$의 commitment한 값의 RLC인 $[D]_1$를 계산한다. 이러한 계산을 하는 이유는 라운드 10, 11에서 알 수 있다.

라운드 10, 11을 자세히 살펴보면 $W_\zeta$, $W_{\zeta\omega}$의 분자와 유사한 값을 가짐을 알 수 있다. 특히 $[F]_1 - [E]_1$의 형태를 예상해보면 $[W_\zeta \cdot (x - \zeta) + u \cdot W_{\zeta\omega} \cdot (x - \zeta\omega)]_1$, 즉 $W_\zeta$, $W_{\zeta\omega}$의 분자의 commitment임을 알 수 있다. 마지막으로 라운드 12를 통해 batch 형태로 pairing을 통해 증명을 검증한다.

라운드 12의 좌항을 먼저 살펴보자.

$$e([W_\zeta]_1 + u \cdot [W_{\zeta\omega}]_1, [x]_2) = x(W_\zeta(x) + u \cdot W_{\zeta\omega}(x)) \cdot e(G_1, G_2)$$

다음으론 우항을 살펴보자.

$$e(\zeta \cdot [W_\zeta]_1 + u\zeta\omega \cdot [W_{\zeta\omega}]_1 + [F]_1 - [E]_1, [1]_2)$$

$$= (\zeta \cdot W_\zeta(x) + u\zeta\omega \cdot W_{\zeta\omega}(x) + F(x) - E) \cdot e(G_1, G_2)$$

$$= (\zeta \cdot W_\zeta(x) + u\zeta\omega \cdot W_{\zeta\omega}(x) + W_\zeta(x)(x - \zeta) + u \cdot W_{\zeta\omega}(x)(x - \zeta\omega)) \cdot e(G_1, G_2)$$

최종적으로 두 항을 비교해보면 다음과 같다.

$$x(W_\zeta(x) + u \cdot W_{\zeta\omega}(x))$$

$$= \zeta \cdot W_\zeta(x) + u\zeta\omega \cdot W_{\zeta\omega}(x) + W_\zeta(x)(x - \zeta) + u \cdot W_{\zeta\omega}(x)(x - \zeta\omega))$$

$$\rightarrow W_\zeta(x)(x - \zeta) + W_{\zeta\omega}(x)(ux - u\zeta\omega)$$

$$= W_\zeta(x)(x - \zeta) + W_{\zeta\omega}(x)(ux - u\zeta\omega)$$

즉 pairing을 통해 결과적으로 Verifier가 공통 입력과 commitment를 통해 reconstruction한 증명과 Prover의 증명이 동일한 지 검사하며 검증이 끝나게 된다.

우리는 지금까지 PLONK의 산술 회로부터 모든 증명 과정을 살펴보았다. PLONK는 앞으로 살펴볼 Halo2에 기본이 되는만큼 gate 및 copy 제약 조건의 아이디어를 이해하고 넘어가는 것이 중요하다.

PLONK에서는 추가로 Plookup과 같은 개념이 존재하지만, 이는 Halo2의 Lookup 테이블을 통해 알아볼 것이다.

## 3. zkEVM : Scroll project Deep dive - Tech

지금까지 우리는 ZKP의 기초 개념과 ZK-SNARK, PLONK를 살펴보았다. PLONK는 ZK-SNARK가 가지는 circuit-dependent 특성, 단순한 연산 등의 여러 단점을 보완한다. 하지만 PLONK 또한 zkEVM에 적용되기에는 개선되어야 할 부분이 많다. 실제로 이를 보완하기 위해 TurboPLONK와 UltraPLONK arithmetization 등 많은 프로토콜들이 등장하였다. 해당 프로토콜들을 통해 초기 PLONK보다 더욱 유연한 gate의 설계, 즉 높은 차수의 제약 조건까지 설계할 수 있는 custom gate의 도입 및 lookup table을 적극적으로 활용하여 영지식 증명으로 변환하기 어려운 해시함수와 같은 연산들을 효율적으로 검증할 수 있게 되었다. 여러번 설명했듯이 zkEVM은 영지식 증명 친화적이지 않은 연산들과 증명 과정 중 발생하는 많은 overhead 때문에 상용화에 어려움이 있었다. 하지만 위와 같은 프로토콜들이 등장하고 이를 기반으로 더욱 개선된 프로토콜이 개발되고 있다.

이제부터 본격적으로 Scroll에서 사용되는 ZKP scheme인 Halo2-ce를 알아보고, Scroll에서 실제로 ZKP를 생성하는 과정을 알아볼 것이다. Halo2-ce란, PLONK를 기반으로 zCash 프로토콜이 개발한 Halo2를 PSE 및 Scroll 프로토콜에서 EVM에 적용할 수 있도록 수정한 영지식 증명 프로토콜이다. Halo2와 Halo2-ce는 증명과정에서 사용하는 scheme[15]을 제외한 나머지 과정이 거의 동일하므로 본 리서치에서는 Halo2를 기반으로 설명을 진행한다. 이를 통해 실제로 이더리움을 Layer1으로 하는 zkEVM 프로토콜이 어떤 식으로 동작하는 지에 대한 깊은 이해를 가질 수 있을 것이다.

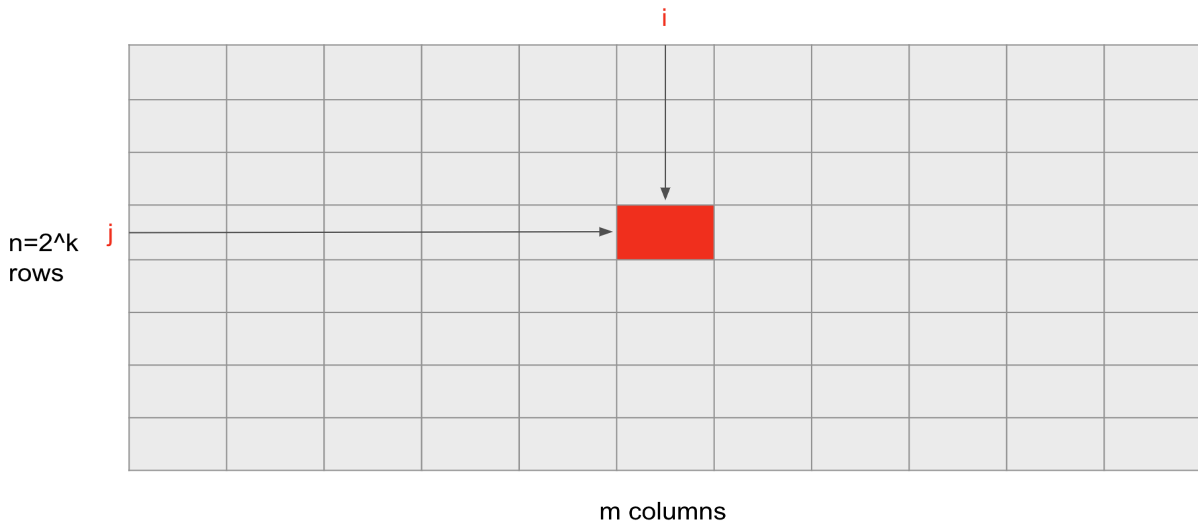
[15] 기존 Halo2는 IPA(Inner Product Arguments)를, Halo2-ce는 KZG commitment를 사용한다.

## 3.1 Halo2

EVM을 완전히 지원하기 위해선 많은 Opcode와 상태 값, 그리고 영지식 증명에 친화적이지 않은 해시함수들에 대한 연산 증명을 생성해야 한다. Halo2에서는 PLONKish 산술화를 기반으로 Custom gate 및 Lookup 테이블을 통해 이를 해결했다.

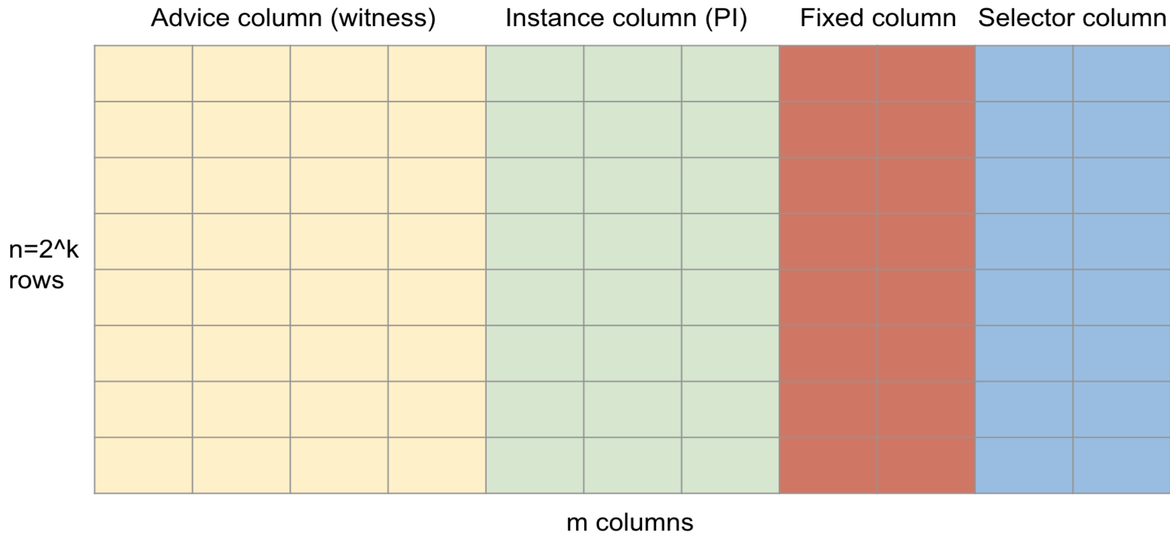
1) Halo2 circuit overview

먼저 Halo2의 회로 구성에 대해 알아보자. 이전 PLONK의 회로에서 2D 행렬에 witness값들을 위한 열과 gate의 selector값들의 열이 채워져 있었던 것처럼, Halo2에서도 여러 종류의 열이 2D 행렬로 배치되어있는 유사한 형태로 회로가 구성된다.




<그림 14 Halo2 회로 형태>

Halo2에서는 다양한 종류 열이 있으며, 각각의 cell에는 열의 종류에 맞는 값이 채워지게 된다. 위의 테이블에서 $(i, j)$ 좌표의 cell을 빨간색으로 표시해두었다. 이를 Halo2에서 우리가 사용할 표기법으로 나타내면 $A_{i,j} = a_i(\omega^j)$과 같다. 이 때 $A_{i,j}$는 $(i, j)$ 좌표의 값을 뜻하고 $a_i(x)$는 $i^{th}$열의 값을 Lagrange 다항식으로 표현한 형태이다. PLONK에서와 동일하게 FFT/IFFT를 위해 evaluation 도메인을 $\{\omega^k\}_{k\in n-1}$로 사용하는 것을 확인하자. 다음으로, 회로를 구성하는 열의 종류에 대해 알아보자.




<그림 15 Halo2 열의 종류>

Halo2의 회로에는 4가지의 주요 열이 있다.

1. Advice 열: Advice 열에 속하는 값은 비밀 입력과 witness 값들로 구성되어 있다. (Prover)
2. Instance 열: Instance 열에 속하는 값은 공개 입력으로 구성되어 있다. (Prover & Verifier)
3. Fixed 열: Fixed 열에 속하는 값은 회로를 구성할 때 미리 계산되어 고정되고, 이후 변하지 않는다.
4. Selector 열: Selector 열에 속하는 값은 바이너리로, 각 행에서 제약 조건의 적용 여부를 나타낸다.

우리는 이러한 종류의 열을 통해 쉽게 제약 조건을 생성할 수 있다. 예를 들어, 같은 행의 Advice열에서 연속된 3개의 값을 곱한 값과 같은 행의 Fixed 열에 속하는 값이 동일하다는 제약 조건을 3번째 행을 제외한 모든 행에 추가하고 싶다고 가정하자. 이는 다음과 같이 매우 간단하게 구현할 수 있다.

$$S(\omega^j) * (A_1(\omega^j) \cdot A_2(\omega^j) \cdot A_3(\omega^j) - F_1(\omega^j)) = 0$$

$$S(\omega^3) = 0$$

첫번째 식은 연산에 대한 제약 조건이다. Selector 값과 연산 값이 곱해져 있고, Selector 값이 1이라면 해당 연산을 만족해야 한다.(이 때 3번째 행에서는 Selector 값이 0이므로 해당 제약 조건은 결과적으로 모든 행에서 만족하게 된다.) 두번째 식은 Selector 값에 대한 제약 조건이다. 3번째 행에서는 해당 제약 조건이 적용되지 않기 때문에, Selector 값이 0이어야 한다. 위 두 식이 뜻하는 바를 쉽게 이해할 수 있을 것이다.

Halo2에서는, 거의 모든 것이 Lookup 테이블과 테이블의 증명을 위한 보조 회로들로 이루어져 있다. 그리고 Lookup 테이블은 주로 Fixed 열[16]에 속한다. Lookup 테이블은 영지식 친화적이지 않은 계산에 대해 이를 따로 계산하여 Lookup 테이블에 채워넣고, 회로에서는 단순히 Lookup 테이블의 값을 조회하는 형태로 구성된다. 예를 들어 4-bit의 bitwise XOR 연산을 살펴보자.

Advice column (witness) PI(Omitted) Fixed column(LU table) Selector column

n=2^k rows

| Advice column (witness) | | | Fixed column(LU table) | | | Selector column |
|---|---|---|---|---|---|---|
| a_0 | b_0 | c_0 | 0000 | 0000 | 0000 | 1 |
| a_1 | b_1 | c_1 | 0000 | 0001 | 0001 | 1 |
| a_2 | b_2 | c_2 | 0000 | 0010 | 0010 | 1 |
| … | … | … | … | … | … | 1 |
| a_(n-4) | b_(n-4) | c_(n-4) | 1111 | 1100 | 0011 | 1 |
| a_(n-3) | b_(n-3) | c_(n-3) | 1111 | 1101 | 0010 | 1 |
| a_(n-2) | b_(n-2) | c_(n-2) | 1111 | 1110 | 0001 | 1 |
| a_(n-1) | b_(n-1) | c_(n-1) | 1111 | 1111 | 0000 | 1 |

Lookup!

<그림 16 Halo2 Lookup 테이블 예시>

ZKP에서 bitwise연산은 각 비트의 range check와 비트의 크기만큼의 연산을 수행해야 하므로 많은 비용이 발생한다. Halo2에서는 미리 bitwise 연산에 대한 Lookup 테이블을 만들고 연산의 결과만을 조회할 수 있다. 본 예시에서는 $a \oplus b = c$의 제약 조건이 있을 때, selector가 1이면 해당 행에서 $a, b$에 맞는 연산 결과를 Lookup 테이블에서 조회하는 것을 확인할 수 있다. 참고로 모든 열의 요소의 개수 (행의 개수)는 같아야 하며, 이를 맞추기 위해 패딩이 추가될 수 있다.

[16] 실제로는 이름처럼 고정되어 있지는 않다. 우리가 다루는 굉장히 넓은 범위 내에서 모든 경우의 수를 미리 계산하는 것은 불가능하기 때문이다. 초기 버전에서는 주어진 범위 내에서 모든 경우의 수를 미리 계산하여 고정시키는(Fixed) 구현도 있지만, 대부분의 경우 입력을 받은 후 필요한 값들만 채워넣는다.

### 2) Halo2: Custom gate

Halo2에서는 회로의 디자인을 훨씬 유연하게 만들어 주는 custom gate가 있다. 이를 통해 우리는 ZK-SNARK의 2차원 제약 조건보다 훨씬 높은 차수의 제약 조건을 생성할 수 있다.

| Advice column (witness) | | | | Instance column (PI) | | | Fixed column | | Selector column | |
|---|---|---|---|---|---|---|---|---|---|---|
| … | … | … | … | … | … | … | … | … | 1 | … |
| … | … | … | … | … | … | … | … | … | 1 | … |
| A0_2 | A1_2 | A2_2 | A3_2 | … | … | … | … | … | 1 | … |
| … | … | … | A3_3 | … | … | … | … | … | 1 | … |
| … | A1_4 | … | A3_4 | … | … | … | … | … | 1 | … |
| … | … | … | … | … | … | … | … | … | 1 | … |
| A0_6 | … | … | … | … | … | … | … | … | 1 | … |
| … | … | … | … | … | … | … | … | … | 1 | … |

<그림 17 Halo2 Custom gate 예시>

위 예시에서 우리는 다음과 같은 제약 조건을 정의한 것이다.

$$S(\omega^j) \cdot (A_{1,2} \times A_{2,2} \times A_{3,2} - A_{3,3} \times A_{3,4} \times A_{1,4}) = 0$$

동일한 방식으로, 우리는 수 천개의 제약 조건을 자유롭게 설계할 수 있다. 하지만 각 제약 조건에는 selector 열이 같이 사용되기 때문에 회로의 크기가 비효율적으로 커질 수 있다. 이를 해결하기 위해 Halo2에서는 간단한 Selector 결합을 사용한다. 이는 만약 서로 다른 두 selector 열이 겹치지 않을 경우 두 selector 열을 하나로 합칠 수 있다는 단순한 최적화[17]를 통해 selector 열의 수를 줄일 수 있다.

### 3) Halo2: Lookup Argument

Prover는, Lookup 테이블에서 값을 조회할 때 이를 임의로 추가하거나 삭제할 수 없다. 즉, Lookup 테이블의 값을 조회하는 열을 $A$라고 하고 Lookup 테이블 열을 $Z$라고 하면, $A \subset Z$ 를 만족해야 한다는 것이다. 이를 위한 제약 조건을 생성해보자.

[17] 이후 알아보겠지만, Halo2에서는 다양한 회로를 하나로 배치하기 위해 Floor Planner를 사용한다. 쉽게 예상할 수 있듯 selector 열의 최적화는 Floor Planner가 어떻게 회로를 배치하냐에 따라 그 성능이 달라질 수 있다.

a) Without zero-knowledge

Advice column (witness) | Fixed column

n=2^k rows

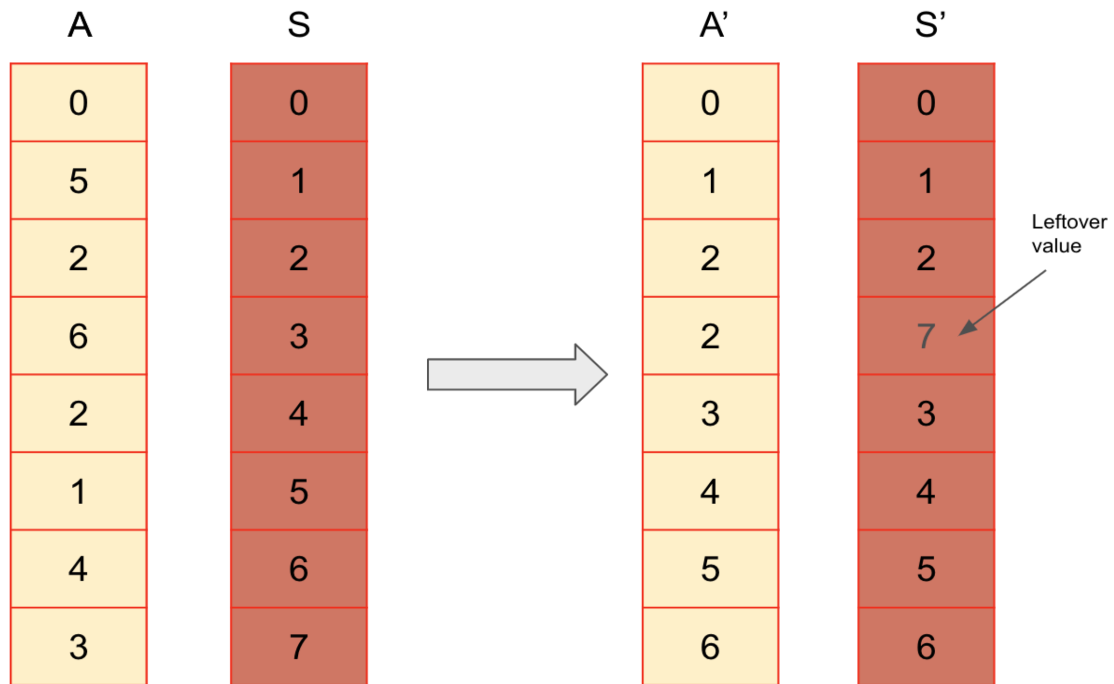
| a_0 | … | 0 | … | … | 0 | … | … |
|---|---|---|---|---|---|---|---|
| a_1 | … | 5 | … | … | 1 | … | … |
| a_2 | … | 2 | … | … | 2 | … | … |
| a_3 | … | 6 | … | … | 3 | … | … |
| … | … | 2 | … | … | 4 | … | … |
| a_n-3 | … | 1 | … | … | 5 | … | … |
| a_n-2 | … | 4 | … | … | 6 | … | … |
| a_n-1 | … | 3 | … | … | 7 | … | … |

Lookup

klaytn

<그림 18 Lookup Argument - Lookup 예시>

위의 예시에서 Advice 열의 Lookup 테이블을 참조하는 열을 $A_2$, 그리고 Lookup 테이블을 $S_0$라 부르자. 앞서 말했듯 $S_0$는 특수한 경우 (좁은 범위에 한해) 모든 경우의 수가 미리 계산되어 있을 수도 있고, 대부분은 입력을 받은 후 필요한 값들만 계산하여 구성한다. 이 때 어느 경우에서든 위 예시와 같이 $A_2 \subset S_0$를 만족해야 한다. 이를 증명하기 위해 Prover는 두 열 사이의 permutation을 제안한다.




<그림 19 Lookup Argument - Permutation>

먼저 $A'$는 $A$를 정렬한 배열이다. 다음으로 $S'$는, 첫 번째 행에서는 $A'[0]$의 값을 가져오고, 이후로는 $A'[i]$와 $A'[i-1]$가 다를 때는 $A'[i]$, 같을 때는 현재 $i$를 저장해두었다가(위 예시에선 3) 이후 남은 값(위 예시에선 7)을 채운다. 이렇게 생성된 $A'$, $S'$은 올바르게 계산되었다면 $A'[i] = S'[i] \,||\, A'[i] = A'[i-1]$ 를 반드시 만족해야 한다. 우리는 이를 다음 방정식으로 표현할 수 있다.

$$(A'(X) - S'(X)) \cdot (A'(X) - A'(\omega^{-1}X)) = 0$$

$$l_o(X) \cdot (A'(X) - S'(X)) = 0$$

그리고 Prover는 $A'$, $S'$가 각각 $A$, $S$로 부터 계산되었다는 것 또한 증명해야 한다. 이는 $A$, $S$와 $A'$, $S'$가 이전에 살펴보았던 copy 제약 조건을 만족하는 permutation을 통해 계산되었다는 것이고, 우리는 이미 이를 증명하기 위해 $Z(X)$를 다음과 같이 정의해야한다는 것을 알고 있다.

$$Z(\omega^i) \cdot (A(\omega^i) + \beta)(S(\omega^i) + \gamma) - Z(\omega^{i+1}) \cdot (A'(\omega^i) + \beta)(S'(\omega^i) + \gamma) = 0$$

$$Z(\omega^0) = 1$$

즉, Verifier는 다음과 같이 $Z$를 검증하게 된다.

$$Z(\omega X) \cdot (A'(X) + \beta)(S'(X) + \gamma) - Z(X) \cdot (A(X) + \beta)(S(X) + \gamma) = 0$$

$$l_0(X) \cdot (1 - Z(X)) = 0$$

b) With zero-knowledge

PLONK에서는 다항식을 숨기기 위해 $Z_H(X)$를 통한 RLC를 계산했다. Halo2에서는, 각 배열을 숨기기 위해 배열 자체에 랜덤 값을 추가할 것이다. 만약 값을 숨기지 않는다면, Verifier는 이론상으로 다항식을 통해 본래 Advice 열의 값을 추측할 수 있다.

| idx | A | S | | A' | S' | | q_last | q_blind |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | | 0 | 0 | | 0 | 0 |
| 1 | 5 | 1 | | 1 | 1 | | 0 | 0 |
| 2 | 2 | 2 | | 2 | 2 | | 0 | 0 |
| 3 | 6 | 3 | | 2 | 7 | | 0 | 0 |
| 4 | 2 | 4 | | 3 | 3 | | 0 | 0 |
| 5 | 1 | 5 | Permutation | 4 | 4 | + | 0 | 0 |
| 6 | 4 | 6 | | 5 | 5 | | 0 | 0 |
| 7 | 3 | 7 | | 6 | 6 | | 0 | 0 |
| 8 | r | r | | r | r | | 1 | 0 |
| … | … | … | | … | … | | … | … |
| 15 | r | r | | r | r | | 0 | 1 |

usable_rows = u (rows 0–7)

blinding_rows = t + 1 (rows 8–15)

<그림 20 Lookup Argument - blinding row>

위 그림에서 알 수 있듯이 우리는 배열 마지막에 랜덤 값 $r$를 추가하였다. 추가된 랜덤 값들은 기존의 제약 조건을 만족하지 않기 때문에 우리는 제약 조건을 수정해야 한다. 이를 위해 두 개의 selector $q_{last}$, $q_{blind}$를 추가할 것이다. 사용할 수 있는 기존의 행이 $u$까지 라면, $q_{last}$는 $u + 1$를 표시해주고 $q_{blind}$는 $u + 2$부터 마지막 행까지를 표시해준다. 이를 통해 수정된 제약 조건 다항식은 다음과 같다.

$$(1 - (q_{last}(X) + q_{blind}(X))) \cdot (A'(X) - S'(X)) \cdot (A'(X) - A'(\omega^{-1}X)) = 0$$

$$(1 - (q_{last}(X) + q_{blind}(X))) \cdot$$

$$(Z(\omega X) \cdot (A'(X) + \beta)(S'(X) + \gamma) - Z(X) \cdot (A(X) + \beta)(S(X) + \gamma) = 0$$

$$l_o(X) \cdot (A'(X) - S'(X)) = 0$$

$$l_0(X) \cdot (1 - Z(X)) = 0$$

각 제약 조건에 $q_{last}$, $q_{blind}$를 고려해주었다. 다만 한가지 더 고려해주어야 하는 부분이 있다. 이상적으로 $Z(\omega^u) = 1$을 만족하지만, 아주 작은 확률로 $k < u$에서 $(A + \beta)$와 같은 값이 0이 될 수도 있다. 즉, $q_{last}$에서 $Z$가 1이 아닌 0이 되는 경우도 고려를 하여, 다음 제약 조건까지 추가해준다.

$$q_{last} \cdot (Z(X)^2 - Z(X)) = 0$$

위 과정을 통해 우리는 검증 가능한 zero-knowledge lookup 테이블을 만들 수 있다.

klaytn

4) Halo2: Permutation Argument

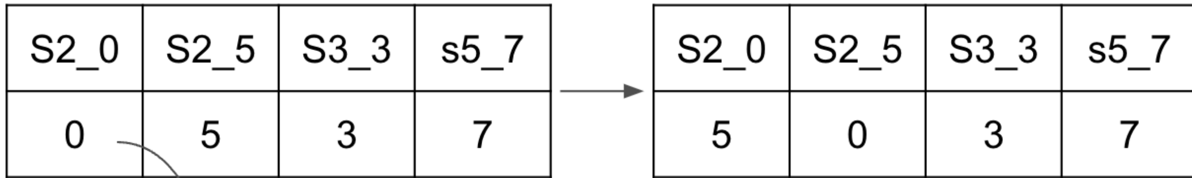
PLONK와 똑같이, Halo2에서도 같은 wire는 같은 값을 가진다는 copy 제약 조건을 추가해주어야 한다.

| | | A2 | A3 | | A5 | | |
|---|---|---|---|---|---|---|---|
| … | … | 1 | 4 | … | 0 | … | … |
| … | … | 5 | 7 | … | 3 | … | … |
| … | … | 3 | 8 | … | 2 | … | … |
| … | … | 4 | 1 | … | 8 | … | … |
| … | … | 2 | 2 | … | 8 | … | … |
| … | … | 1 | 4 | … | 5 | … | … |
| … | … | 2 | 9 | … | 0 | … | … |
| … | … | 3 | 0 | … | 1 | … | … |

| A2_0 | A2_5 | A3_3 | A5_7 |
|---|---|---|---|
| 1 | 1 | 1 | 1 |

<그림 21 Permutation Argument - 회로 예시>

위 회로에서 $A_2(\omega^0)$, $A_2(\omega^5)$, $A_3(\omega^3)$, $A_5(\omega^7)$은 동일한 wire로 연결되어 있을 수도[18] 있다. 앞서 살펴봤듯이, 동일한 wire에 연결되어 있다는 것은 서로 값을 바꾸더라도 회로가 바뀌지 않는다는 것을 뜻한다. 앞서 PLONK에서와 유사하게 우리는 permutation을 사용하여 이를 구현할 것이다. 먼저 $A_2(\omega^0)$, $A_2(\omega^5)$에 대해 permutation을 추가해보자. Permutation $S$는 다음과 같이 구성된다.



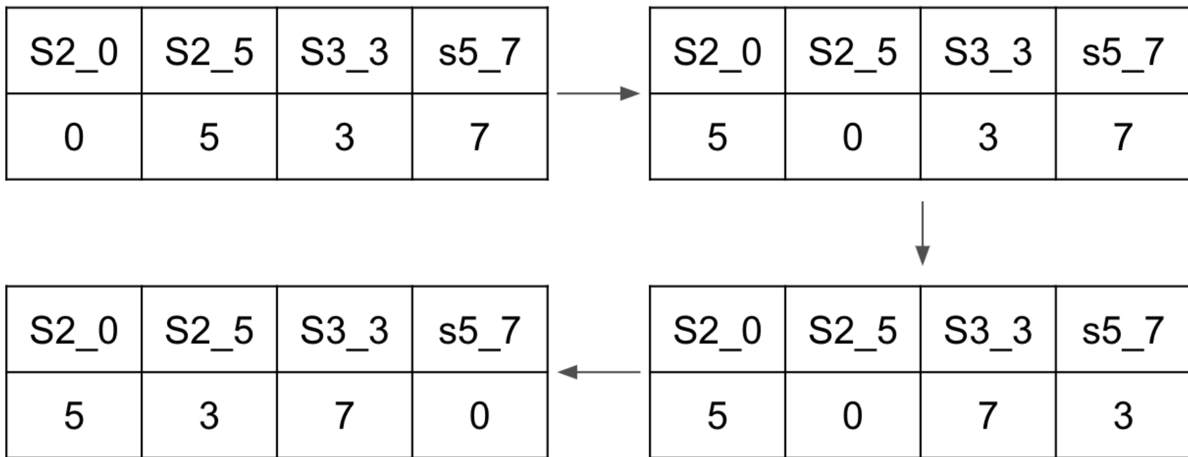
| S2_0 | S2_5 | S3_3 | s5_7 |
|---|---|---|---|
| 0 | 5 | 3 | 7 |

→

| S2_0 | S2_5 | S3_3 | s5_7 |
|---|---|---|---|
| 5 | 0 | 3 | 7 |



위 표에 적어놓은 것 처럼 실제로는 열의 순서 또한 함께 고려해야 한다. PLONK에서 언급했듯이 permutation은 shift 연산을 통해 계산된다. 나머지 wire에 대해서도 permutation을 완성해보자.



| S2_0 | S2_5 | S3_3 | s5_7 |
|---|---|---|---|
| 0 | 5 | 3 | 7 |

→

| S2_0 | S2_5 | S3_3 | s5_7 |
|---|---|---|---|
| 5 | 0 | 3 | 7 |

↓

| S2_0 | S2_5 | S3_3 | s5_7 |
|---|---|---|---|
| 5 | 0 | 7 | 3 |

←

| S2_0 | S2_5 | S3_3 | s5_7 |
|---|---|---|---|
| 5 | 3 | 7 | 0 |

처음과 마지막 permutation을 확인해보면, left shift가 적용되었다. 본 예시에서는 이해를 위해 중간 과정을 모두 표현했지만, 실제 구현에서는 같은 값을 가지는 wire를 모아놓고 한번의 shift 연산을 통해 빠르게 permutation을 생성한다.

[18] 동일한 wire가 아니더라도 같은 값을 가질 수 있다. 하지만 같은 값을 가지는 wire를 연결하는 것은 회로를 증명하는 데에 문제를 발생시키지 않으므로 이를 모두 포함하여 계산한다.

이제 이렇게 생성된 permutation을 다항식 형태로 표현해야 한다. 우리는 열의 개수보다 큰 $T$에 대해 $\delta^T = 1$이 되는 $\delta$를 통해 열의 순서를 표현한다. 제약 조건 다항식을 위해 필요한 식을 정리하면 다음과 같다.

1. $v_i(\omega^j)$: $i^{th}$열의 $j^{th}$행에 해당하는 값
2. $\beta, \gamma$: RLC를 위한 랜덤 값
3. $ID_i(\omega^j) = \delta^i \cdot \omega^j$: 항등 순열의 값
4. $S_i(\omega^j) = \delta^{i'} \cdot \omega^{j'}$: 매핑된 순열의 값

PLONK에서 이미 보았듯이, $m$이 전체 열의 개수일 때 우리는 다음과 같은 제약 조건을 만족하는 $Z$를 가진다.

$$Z_p(\omega X) \prod_{i=0}^{m-1} (v_i(X) + \beta S_i(X) + \gamma) - Z_p(X) \prod_{i=0}^{m-1} (v_i(X) + \beta \delta^i X + \gamma) = 0$$

$$l_0(X) \cdot (1 - Z_p(X)) = 0$$

왜냐하면 전체 축적 값에 대해 다음을 만족하기 때문이다.

$$\prod_{i=0}^{m-1} \prod_{j=0}^{n-1} \left(\frac{v_i(\omega^j) + \beta \delta^i \omega^j + \gamma}{v_i(\omega^j) + \beta S_i(\omega^j) + \gamma}\right) = 1$$

우리는 Lookup Argument에서와 마찬가지로 zero-knowledge를 추가해주어야 한다. 이를 위해 각 열의 마지막에 랜덤 값을 추가하고, 제약 조건을 다음과 같이 수정한다.

$$(1 - (q_{last}(X) + q_{blind}(X))) \cdot$$

$$(Z_p(\omega X) \prod_{i=0}^{m-1} (v_i(X) + \beta S_i(X) + \gamma) - Z_p(X) \prod_{i=0}^{m-1} (v_i(X) + \beta \delta^i X + \gamma)) = 0$$

$$l_0(X) \cdot (1 - Z_p(X)) = 0$$

$$q_{last}(X) \cdot (Z_p(X)^2 - Z_p(X)) = 0$$

5) Halo2: Quotient Argument

이제, 우리는 증명을 생성하기 위한 모든 제약 조건들을 정의했다. 하지만 우리는 수천개의 custom gate등과 함께 많은 제약 조건에 대한 증명을 생성해야 한다. PLONK에서 선형화 트릭을 사용했던 것처럼, Halo2에서는 제약 조건들을 RLC하여 하나의 Quotient Argument를 생성한다. 우리는 다음 5개의 제약 조건 다항식을 가지고 있다고 가정하자: $\phi_0(X), \cdots, \phi_4(X)$. 이 때 $Z(X) = X^n - 1$에 대해 우리는 제약 조건을 하나의 Quotient Argument로 합칠 수 있다.

$$h(X) = \frac{\phi_0(X) + \gamma \cdot \phi_1(X) + \gamma^2 \cdot \phi_2(X) + \cdots + r^4 \cdot \phi_4(X)}{Z(X)}$$

PLONK에서와 같이 $h(X)$ 또한 지나치게 높은 차수를 가질 수도 있다. Halo2는 commitment를 위해 최대 $n - 1$까지의 차수를 지원하므로 우리는 제약 조건의 최대 차수가 $d(n - 1)$일 때 다음과 같이 $h(X)$를 나눌 수 있다.

$$h(X) = h_0(X) + X^n h_1(X) + \cdots + X^{(d-1)n} h_{d-1}(X)$$

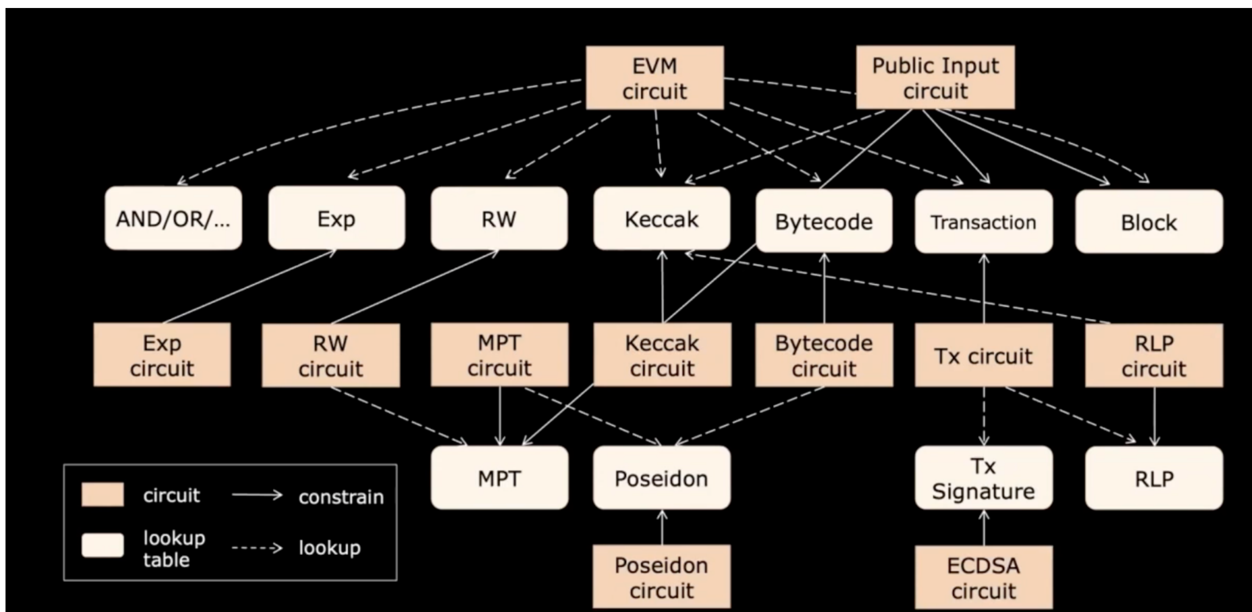
Prover는 이제 각각의 $h_i(X)$를 evaluation하고 commitment를 생성한 후 Verifier에게 보내게 된다.

지금까지 Halo2의 회로와 증명을 생성하기 위한 제약 조건들을 살펴보았다. PLONK의 개념과 공통되는 부분이 많았기 때문에 빠르게 이해할 수 있었다. Halo2에서 Custom gate의 확장으로 더욱 유연한 회로 설계가 가능해졌다는 것과, Lookup 테이블의 사용으로 기존에 영지식 증명에서 쉽게 다루지 못했던 다양한 연산에 대한 증명을 생성할 수 있게 되어 EVM 호환성을 크게 높일 수 있었다는 것에 집중하면 Scroll 프로토콜이 어떻게 동작하는 지도 쉽게 이해할 수 있을 것이다.

## 3.2 Scroll circuit overview

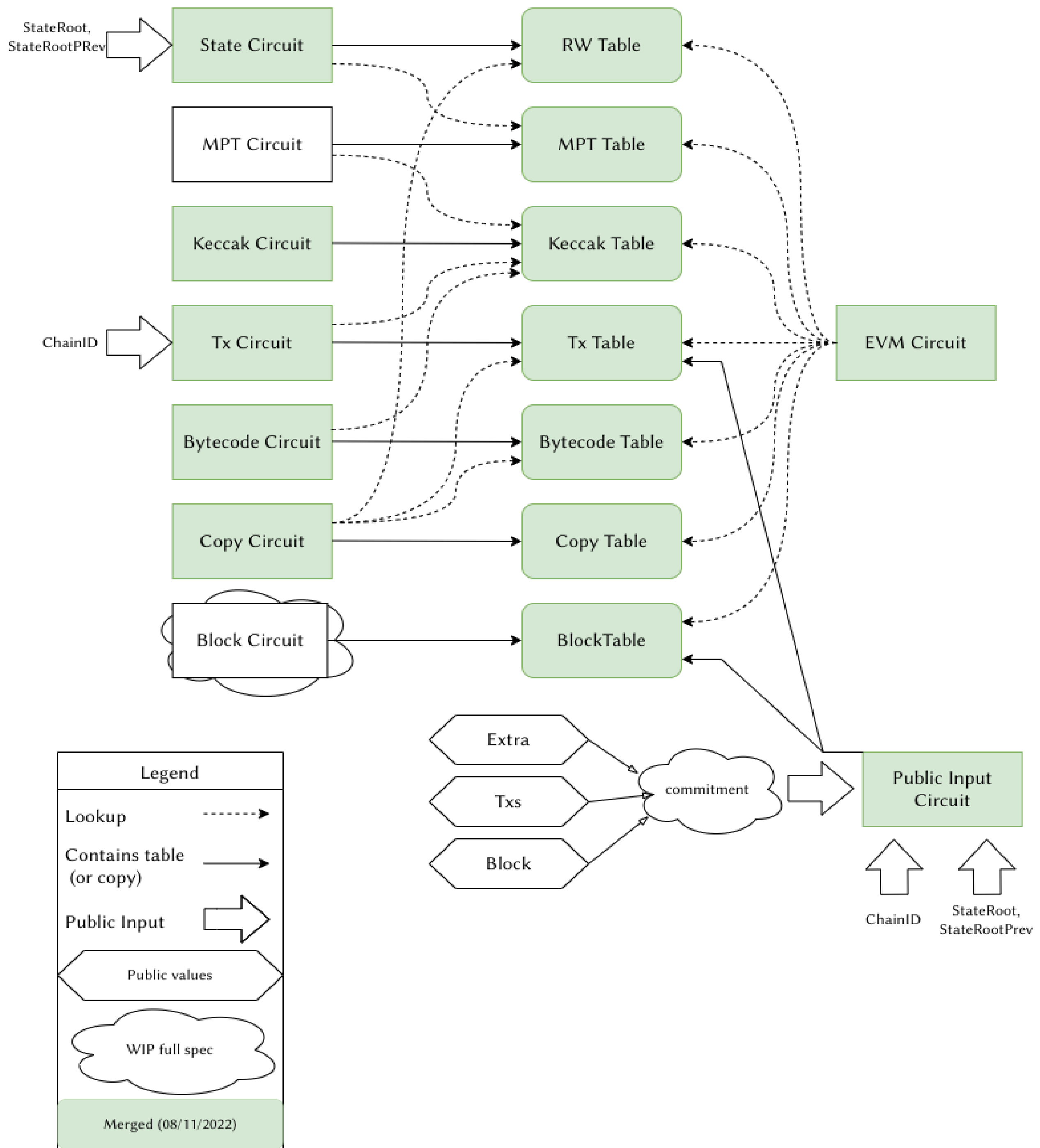
먼저, zkEVM에 대해 간단한 복습을 하고 넘어가자. 블록체인은 상태 머신이고, 트랜잭션을 실행시킴으로써 상태를 업데이트한다. 따라서 우리는 트랜잭션이 올바르게 실행되었는지와 상태가 올바르게 업데이트 되었는지에 대한 증명을 생성하고 싶은 것이다.

Scroll은 이더리움-동등한 zkEVM을 개발 중이며, 이는 기존 이더리움과 완전히 동일한 개발 환경 및 API를 지원한다는 것이다. 앞서 살펴보았듯이 이는 기술적으로 매우 어려우며, 현재까지도 활발히 연구 중에 있다.




<그림 22. Scroll 프로토콜의 zkEVM 회로 구조. 이미지 출처: https://www.youtube.com/watch?v=60lkR8DZKUA&t=1860s>

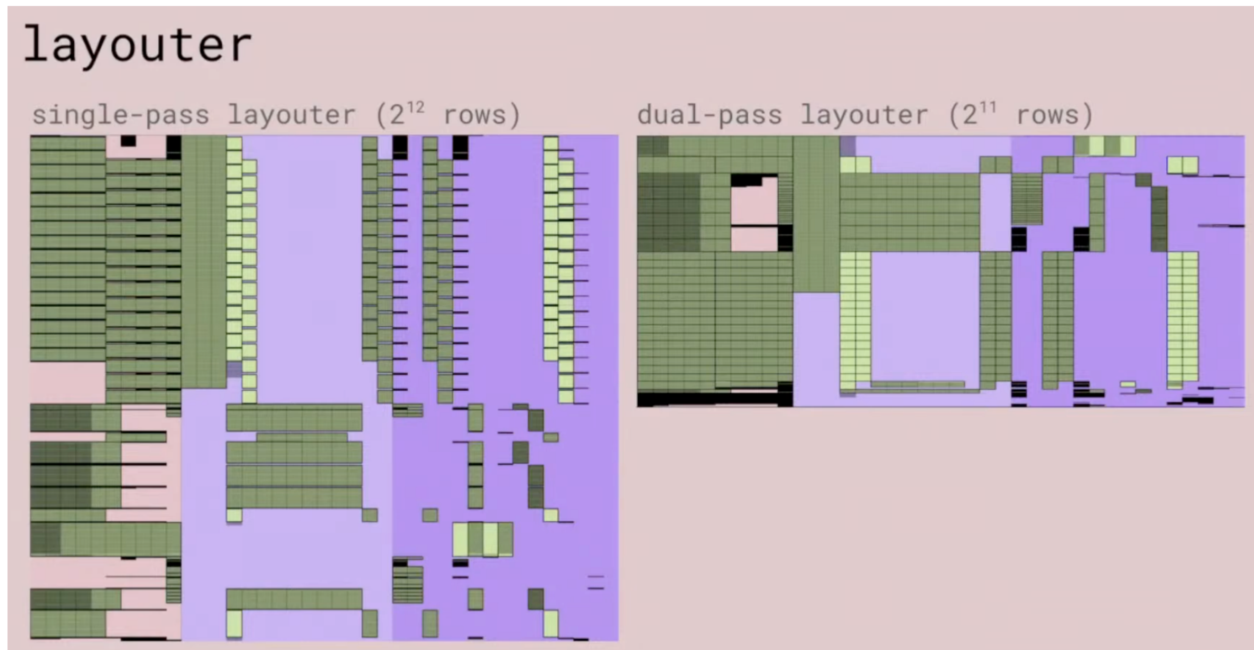
이를 극복하기 위해서 Scroll은 Lookup 테이블을 매우 적극적으로 사용한다. 실제 Opcode의 연산을 확인하는 EVM Circuit은 연산에 필요한 대부분의 값을 미리 계산된 Lookup 테이블로부터 가져옴으로써, 기존에 영지식 증명으로 생성하기 어려웠던 연산들을 사용할 수 있게 되었고 증명 생성 과정에서 발생하는 비효율성도 크게 줄일 수 있었다. 위 그림을 살펴보자. EVM Circuit은 최종적인 Opcode의 연산 및 State를 검증하게 된다. 이 때, EVM Circuit은 연산을 하기 위한 입력들과 연산의 결과 값을 비교하기 위한 결과, 그리고 영지식 증명으로 변환하기 어려운 연산들을 직접 진행하지 않는다. 그 대신 연산에 필요한 구성 요소를 모두 lookup table에서 가져오고, 각 lookup table을 검증하기 위한 sub-circuit들을 생성하게 된다. (EVM circuit또한 sub-circuit이 된다.) 뒤에서 자세히 살펴보겠지만, 이를 통해 우리는 회로 내의 Overhead를 줄이고 효율적인 회로 설계를 할 수 있게 된다.




<그림 23 Halo2-ce의 super-circuit 구조 이미지 출처 : Link >

하지만 그림 24에서 확인할 수 있듯 이러한 구현은 많은 sub-circuit들이 필요하게 되고, 이를 어떻게 관리할 지에 대해 의문이 들 수 있다. 이때,. 각각의 회로에 대한 증명을 모두 따로 생성하는 것보다 구현 상 하나의 회로에 모든 회로를 담아 하나의 증명을 생성하는 것이 더욱 직관적이고 간단할 것이다.

Halo2-ce에서는 이를 super-circuit이라고 부른다. Super-circuit은 위 그림과 같은 포함 관계[19]를 가진다. 초록색으로 표시된 회로는 모두 super-circuit 내에 함께 구성된다. 하나의 큰 super-circuit안에 다양한 회로를 가지고 있고, 이를 잘 배치해야한다. 배치에 따라 전체 회로의 크기가 줄어들기 때문에 이를 위해 FloorPlanner와 Layouter 모듈이 존재한다.




<그림 24 Halo2-ce의 Layouter 동작. 이미지 출처 : https://www.youtube.com/watch?v=VMLx0j5JuG0&t=5948s>

FloorPlanner와 Layouter 모듈의 동작은 위 그림으로 쉽게 이해할 수 있다. 첫번째 단계는 그저 모든 region을 가능한대로 나열한다. Region이란, 동일한 offset이 필요한 파트를 묶어놓은 파트이다. Custom gate를 설계할 때를 생각해보면 우리는 각 행에서 해당 제약 조건의 활성 여부를 selector 열을 통해 표시했다. 이 때, 어떤 행에 접근할 것인지에 대한 offset이 필요하다. 만약 이를 임의로 수정한다면 회로의 정상 동작을 보장할 수 없으므로 region으로 관리하는 것이다. 두번째 단계에서는 region을 유지하면서 회로를 수직으로 압축한다. 이 때 큰 region을 우선적으로 배치한다. FloorPlanner-Layouter 모듈 덕분에, 우리는 동일한 회로를 더 적은 사이즈로 압축할 수 있다.

[19] zkTrie를 위한 MPT 회로또한 최근 super-circuit에 포함되었지만 해당 그림에는 업데이트 되지 않았다.

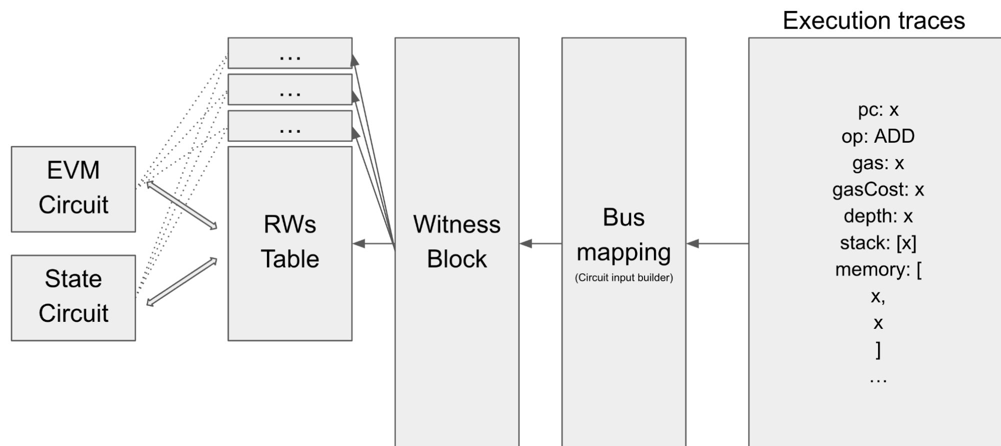
## 3.3 R/W ↔ Execution

우리는 지금까지 ZK-SNARK, PLONK, Halo2 등 다양한 ZKP 프로토콜들을 살펴보았다. 하지만 이것들이 어떻게 zkEVM과 연결되어 있는지 아직 와닿진 않을 것 같다. 따라서, 우리는 실제 이더리움의 트랜잭션 trace를 통해 간단한 State 및 EVM 회로를 생성해볼 것이다.

트랜잭션이 실행되는 과정을 떠올려보자. 스택이나 메모리, 스토리지 등의 state에서 데이터를 가져온 후, 이를 연산한 후 다시 state에 저장한다. 여기서 우리는 트랜잭션의 실행을 크게 State에 대한 읽기/쓰기와 데이터의 연산으로 나눌 수 있다. 실제 Halo2-ce에서도, 이에 맞추어 증명을 두 파트로 나눈다.

1. State 증명: Stack/Memory/Storage 실행이 올바르게 수행되었는지를 검증한다.
2. EVM 증명: 올바른 Opcode가 올바른 순서에 호출되었는지 검증한다. 예를 들어 덧셈 연산이 실행될 때의 PC와 SP 값들을 검증한다. 또한 각 Opcode의 연산과 State 증명의 연산이 올바른지도 검증한다.

EVM 증명에서, 우리는 Opcode의 실행을 검증할 값이 필요하다. 이를 위해서, EVM 회로는 트랜잭션의 trace로부터 값을 가져와야 한다. 이 때 State 증명이 검증한 데이터를 Lookup 테이블로 저장해놓고, EVM 회로가 이를 조회하는 방식을 사용한다. 후에 실제 회로가 어떻게 생성되는 지를 보면 쉽게 이해할 수 있을 것이다. 아래 그림에서 circuit input builder와 witness block은, Lookup 테이블에게 트랜잭션 trace를 정리하여 전달해주는 역할이라고 이해하면 된다. Circuit input builder와 witness block은 구현 상 입력으로 받은 trace들을 잘 정리하여 각 circuit들이 필요한 형태로 제공하는 역할을 한다.




<그림 25 트랜잭션 trace부터 EVM/State Circuit 생성 과정>

klaytn

실제로 트랜잭션의 trace로부터 circuit input builder와 witness 블록을 생성하는 예시를 살펴보자. 구현은 단순하다. 입력 trace에서 값을 찾아 복사한 후, 우리가 원하는 형태로 만든 후 저장하는 방식이다. 예를 들어 체인 id는 다음과 같이 처리된다.

```
let chain_ids = block_traces
    .iter()
    .map(|block_trace| block_trace.chain_id)
    .collect::<Vec<U256>>();
```

<그림 26 트랜잭션 trace에서 체인 id를 가져오는 코드>

위의 과정을 통해 witness block을 생성한 후에는 본격적으로 회로를 생성하게 된다.

## 3.4 Build zkEVM circuit

1) State 회로 및 RW 테이블

klaytn

우리는 아주 간단한 스마트 컨트랙트를 사용할 것이다.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract zkEVMExample {
    function simpleExample() public pure {
        assembly {
            let ptr := mload(0x40) // Load the word located at 0x40
            mstore(ptr, 0xabc) // Load 0xabc at ptr
            mstore(ptr, add(mload(ptr), 0x123)) // mem[ptr] = mem[ptr] + 0x123
            mstore(add(ptr, 0x40), 0xdef) // mem[ptr + 0x40] = 0xdef
        }
    }
}
```

<그림 27 스마트 컨트랙트 예시>

State 값들이 어떻게 변하는지 직관적으로 이해할 수 있도록 어셈블리를 사용하여 코드를 작성했다. simpleExample() 함수를 실행하면, 다음과 같은 로그를 확인할 수 있다.

```
pc  op           stack                         memory
--  -----------  --------------------------    ----------------------------------
...                                            {0x80 at 0x40 loaded in advance}
53  JUMPDEST     [   ,      ,       ,    ]     {40: 80, 80:        , c0:      }
54  PUSH1 40     [   ,      ,       ,  40]     {40: 80, 80:        , c0:      }
56  MLOAD        [   ,      ,       ,  80]     {40: 80, 80:        , c0:      }
57  PUSH4 abc    [   ,      ,    abc,  80]     {40: 80, 80:        , c0:      }
62  DUP2         [   ,    80,    abc,  80]     {40: 80, 80:        , c0:      }
63  MSTORE       [   ,      ,       ,  80]     {40: 80, 80:     abc, c0:      }
64  PUSH4 123    [   ,      ,    123,  80]     {40: 80, 80:     abc, c0:      }
69  DUP2         [   ,    80,    123,  80]     {40: 80, 80:     abc, c0:      }
70  MLOAD        [   ,   abc,    123,  80]     {40: 80, 80:     abc, c0:      }
71  ADD          [   ,      ,    bdf,  80]     {40: 80, 80:     abc, c0:      }
72  DUP2         [   ,    80,    bdf,  80]     {40: 80, 80:     abc, c0:      }
73  MSTORE       [   ,      ,       ,  80]     {40: 80, 80:     bdf, c0:      }
74  PUSH4 def    [   ,      ,    def,  80]     {40: 80, 80:     bdf, c0:      }
79  PUSH1 40     [   ,    40,    def,  80]     {40: 80, 80:     bdf, c0:      }
81  DUP3         [ 80,    40,    def,  80]     {40: 80, 80:     bdf, c0:      }
82  ADD          [   ,    c0,    def,  80]     {40: 80, 80:     bdf, c0:      }
83  MSTORE       [   ,      ,       ,  80]     {40: 80, 80:     bdf, c0:   def}
84  POP          [   ,      ,       ,    ]     {40: 80, 80:     bdf, c0:   def}
...
```

<그림 28 simpleExample() 함수 실행 로그>

이를 기반으로 Memory와 Stack 회로를 만들어볼 것이다. (Storage 회로또한 유사한 형태이다.)

2) Memory 회로 및 테이블

먼저 회로와 테이블의 개념에 대해 생각해보자. Memory 테이블에는 Memory 연산과 관련있는 MLOAD와 MSTORE과 같은 Opcode가 저장될 것이다. Memory 회로는 테이블에 채워진 Memory 연산이 올바른지에 대한 검증을 해야한다. 먼저 Memory 테이블을 검증하기 위해선 어떤 제약 조건이 필요할지 생각해보자.

- 올바르게 초기화되었는가?
- 올바른 타입의 값이 채워졌는가?
- 올바른 R/W 연산인가?
- …

위 조건을 확인하기 위해서, 우리는 다음과 같은 제약 조건을 가진다.

| Condition | Constraint | Note |
|---|---|---|
| Initialization | rw == Write && val == 0 | First row of table |
| * | field_tag == 0 && storage_key == 0 | ***field_tag*** isn't used in memory, which means 0. |
| * | Rw ∊ [0, 1] | Read or Write |
| * | key ≥ key_prev | Non-strict monotonic |
| * | val ∊ [0, 255] | Byte value |
| * | state_root == prev_state_root | Memory can't change state root |
| key ≠ key_prev | rw == Write && val == 0 | Initialized to 0 |
| key == key_prev | gc > gc_prev | Strict monotonic |
| rw == Read | val == val_prev | Previous written value |

위 제약조건에서 gc(global counter)가 생소할 수 있다. 우리가 State 테이블을 최종적으로 생성할 때, Stack/Memory/Storage로 나누어 생성한 테이블을 순서대로 호출할 수 있어야 한다. 이 때 트랜잭션 내에서 발생한 모든 RW 연산에 대한 순서를 매긴 것이 gc이다. Global counter는 현재 다루고 있는 state의 종류에 상관없이 RW연산이 발생할 때마다 증가한다.

위 제약 조건을 통해서 우리는 최종적으로 아래와 같은 Memory 테이블을 생성할 수 있다.

| Key | Val | rw | gc | Note |
|---|---|---|---|---|
| 0x40 | 0 | Write | - | Initialization |

| 0x40 | 0x80 | Write | ? (1) | Since 0x80 was loaded in advance, we don't care gc for this example. |
|---|---|---|---|---|
| 0x40 | 0x80 | Read | 4 | 56 MLOAD |
| - | | | | |
| 0x80 | 0 | Write | - | Initialization |
| 0x80 | 0xabc | Write | 10 | 63 MSTORE |
| 0x80 | 0xabc | Read | 16 | 70 MLOAD |
| 0x80 | 0xbdf | Write | 24 | 73 MSTORE |
| - | | | | |
| 0xc0 | 0 | Write | - | Initialization |
| 0xc0 | 0xdef | Write | 34 | 83 MSTORE |

klaytn

3) Stack 회로 및 테이블

Stack 회로 및 테이블도 Memory와 유사하다. 다음과 같은 제약 조건을 가진다.

| **Condition** | **Constraint** | **Note** |
|---|---|---|
| Initialization | rw == Write && val == 0 | First row of table |
| * | field_tag == 0 && storage_key == 0 | ***field_tag*** isn't used in stack, which means 0. |
| * | rw ∊ [0, 1] | Read or Write |

| * | key - key_prev ∊ [0, 1023] | Non-strict monotonic |
|---|---|---|
| * | key ∊ [0, 1023] | Stack pointer < 1024 |
| * | state_root == prev_state_root | Stack can't change state root |
| key ≠ key_prev | rw == Write && val == 0 | Initialized to 0 |
| key == key_prev | gc > gc_prev | Strict monotonic |
| rw == Read | val == val_prev | Previous written value |

위 제약 조건을 통해서 최종적으로 다음과 같은 Stack 테이블을 생성할 수 있다.

| **Key** | **Val** | **rw** | **gc** | **Note** |
|---|---|---|---|---|
| 1020 | 0 | Write | - | INIT |
| 1020 | 0x80 | Write | 28 | 81 DUP3 |
| 1020 | 0x80 | Read | 29 | 82 ADD.POP |
| - | | | | |
| 1021 | 0 | Write | - | INIT |
| 1021 | 0x80 | Write | 7 | 62 DUP2 |
| 1021 | 0x80 | Read | 8 | 63 MSTORE.POP |
| 1021 | 0x80 | Write | 13 | 69 DUP2 |
| 1021 | 0x80 | Read | 14 | 70 MLOAD.POP |

| 1021 | 0xabc | Write | 15 | 70 MLOAD.PUSH |
|---|---|---|---|---|
| 1021 | 0xabc | Read | 17 | 71 ADD.POP |
| 1021 | 0x80 | Write | 21 | 72 DUP2 |
| 1021 | 0x80 | Read | 22 | 73 MSTORE.POP |
| 1021 | 0x40 | Write | 26 | 79 PUSH1 |
| 1021 | 0x40 | Read | 30 | 82 ADD.POP |
| 1021 | 0xc0 | Write | 31 | 82 ADD.PUSH |
| 1021 | 0xc0 | Read | 32 | 83 MSTORE.POP |
| - | | | | |
| 1022 | 0 | Write | - | INIT |
| 1022 | 0xabc | Write | 5 | 57 PUSH4 |
| 1022 | 0xabc | Read | 9 | 63 MSTORE.POP |
| 1022 | 0x123 | Write | 11 | 64 PUSH4 |
| 1022 | 0x123 | Read | 18 | 71 ADD.POP |
| 1022 | 0xbdf | Write | 19 | 71 ADD.PUSH |
| 1022 | 0xbdf | Read | 23 | 73 MSTORE.POP |
| 1022 | 0xdef | Write | 25 | 74 PUSH4 |
| 1022 | 0xdef | Read | 33 | 83 MSTORE.POP |
| - | | | | |

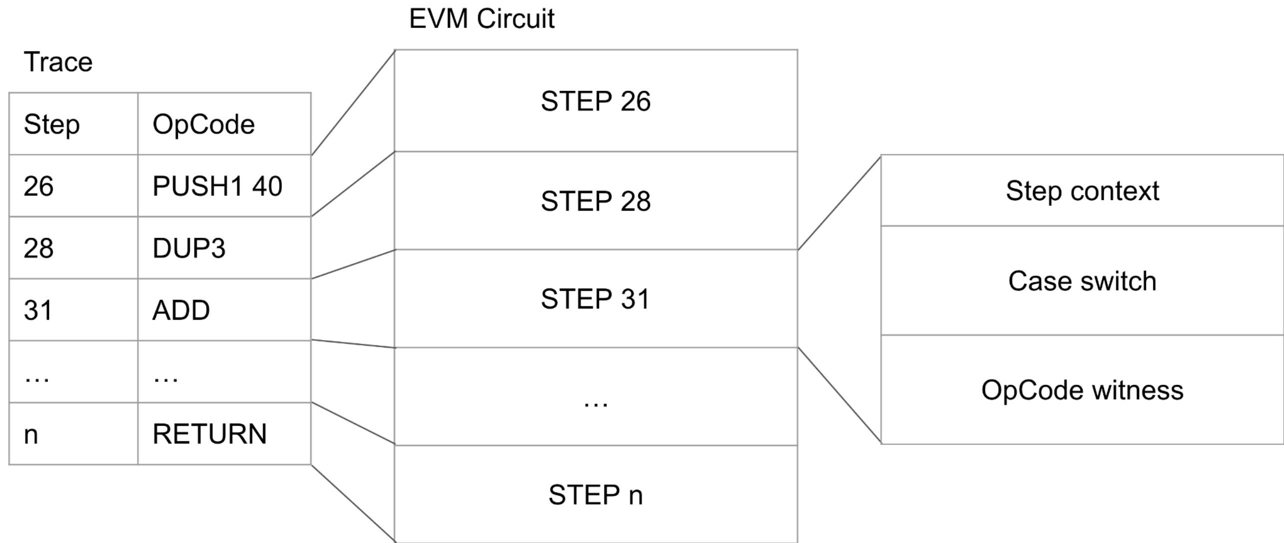
| | | | | |
|---|---|---|---|---|
| 1023 | 0 | Write | - | INIT |
| 1023 | 0x40 | Write | 1 | 54 PUSH1 40 |
| 1023 | 0x40 | Read | 2 | 56 MLOAD.POP |
| 1023 | 0x80 | Write | 3 | 56 MLOAD.PUSH |
| 1023 | 0x80 | Read | 6 | 62 DUP2 |
| 1023 | 0x80 | Read | 12 | 69 DUP2 |
| 1023 | 0x80 | Read | 20 | 72 DUP2 |
| 1023 | 0x80 | Read | 27 | 81 DUP3 |

다소 복잡해보이지만 실제로는 Stack의 RW 연산을 제약 조건을 만족하도록 같은 포인터끼리 정렬한 것이다.

Storage 회로는 다루진 않겠지만 기본적인 개념은 Memory, Stack 회로와 동일하다. Storage에 접근하는 SSTORE, SLOAD와 같은 Opcode를 모두 모은 후 제약 조건에 맞추어 정리한다. 각 회로의 자세한 스펙은 여기서 확인할 수 있다.

4) EVM 회로

State 회로를 통해 우리는 EVM 회로가 쉽게 조회할 수 있는 State 값들의 테이블을 만들었다. 이제는 EVM 회로를 통해 이러한 연산들이 올바르게 수행되었는지를 검증해보자. 먼저 EVM 회로는 다음과 같은 구조를 가진다.




<그림 29 EVM 회로 구조>

Trace를 순회하며 각 Opcode의 검증을 수행하는 행으로 구성되어 있다. 각 행에는 Opcode에 맞는 Step context, Case switch, Opcode witness 값으로 채워져있다. 각 구성 요소를 살펴보자.

1) Step context

Step context는 다음과 같이 각 스텝의 Opcode를 검증하는 데에 필요한 값과 현재 실행 환경에 대한 정보를 가지고 있다.

- rw_counter: Stack/Memory/Storage 연산의 Global counter
- call_id: 현재 호출을 구분하기 위한 unique ID
    - 한 트랜잭션내에서 많은 컨트랙트를 호출할 때 각 호출을 구분하기 위해 사용된다.
- is_root: root 호출임을 나타내는 boolean
- is_create
- code_hash: 현재 실행 중인 컨트랙트의 code hash (EOA일 경우 비어있다.)
- pc: Program counter
- sp: Stack 포인터
- gas_left: 현재 트랜잭션의 남은 가스량
- memory_word_size: EVM의 word로 계산된 메모리 사이즈
    - 메모리 사용을 위한 가스 비용을 계산하기 위해 필요하다.

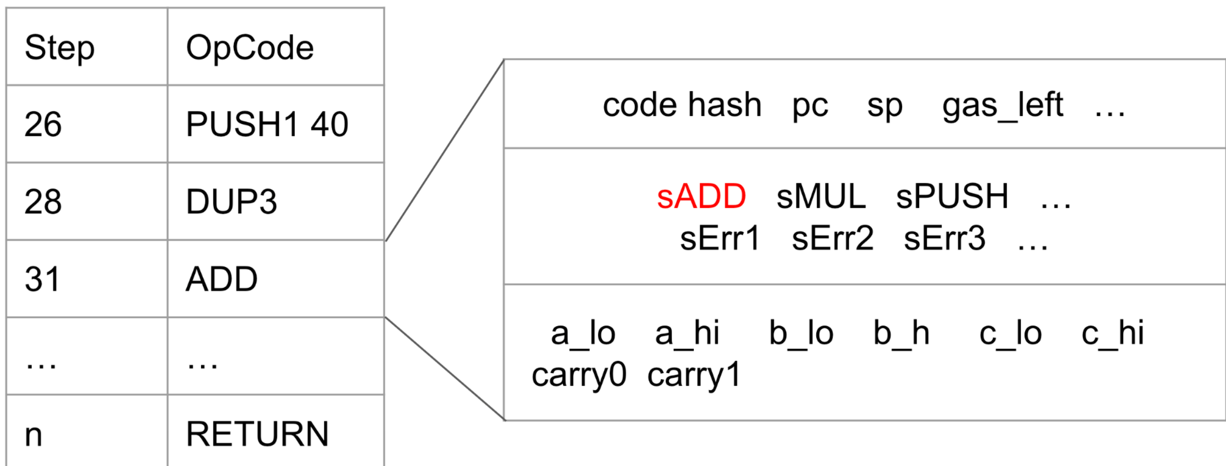
- reversible_write_counter: revert가 발생했을 때를 대비하여 rw_counter의 스냅샷값을 저장
- Log_id

2) Case switch

Case switch는 sADD, sPUSH, sErr1, sErr2과 같은 Opcode의 selector를 가지고 있다. 오직 하나의 switch만 1이어야 하며, 나머지는 모두 0이어야 한다.

3) Opcode witness

Opcode witness에는 실제로 Opcode를 실행할 값들이 담겨있다. 예를 들어, ADD는 2개의 입력과 1개의 출력과 자리올림 값(Carry)이 필요하다. 이제 간단한 ADD opcode에 대한 검증을 수행하는 실제 EVM 회로를 살펴보자.




<그림 30 ADD Opcode를 검증하는 EVM 회로>

먼저, 우리는 ADD Opcode에 대한 제약 조건이 필요하다.

$$sADD * (a_{lo} + b_{lo} - c_{lo} - carry0 * 2^{128}) = 0$$

$$sADD * (a_{hi} + b_{hi} + carry0 - c_{hi} - carry1 * 2^{128}) = 0$$

이는 우리가 배운 custom gate를 통해 매우 쉽게 만들 수 있다.

다음으로는 case switch에 대한 제약 조건이 필요하다. 단 하나의 switch만 1이어야 하며, 바이너리 값을 가져야 한다. 즉, ADD Opcode에 대해선 다음과 같은 제약 조건을 만족해야 한다.

$$sADD + sMUL + \cdots + sErr_n = 1$$

$$sADD * (1 - sADD) = 0$$

$$sMUL * (1 - sMUL) = 0$$

$$\cdots$$

$$sErr * (1 - sErr_n) = 0$$

$$sADD = 1$$

마지막으로 step context에 대한 제약 조건을 알아보자. Step context는 다음 step context가 현재의 Opcode의 실행으로 인해 올바르게 업데이트 되었는지를 확인한다. 우리가 다루고 있는 ADD Opcode의 경우 3의 gas를 소모하며, 2번의 POP, 1번의 PUSH를 진행하니 sp는 1, rw는 3이 늘어나야 한다. 또한 ADD Opcode는 트랜잭션의 종료를 뜻하지 않으므로 code hash는 동일해야 한다. 이 같은 제약 조건들은 다음과 같은 식으로 표현할 수 있다.

$$sADD * (rwcounter_{next} - rwcounter - 3) = 0$$

$$sADD * (codehash_{next} - codehash) = 0$$

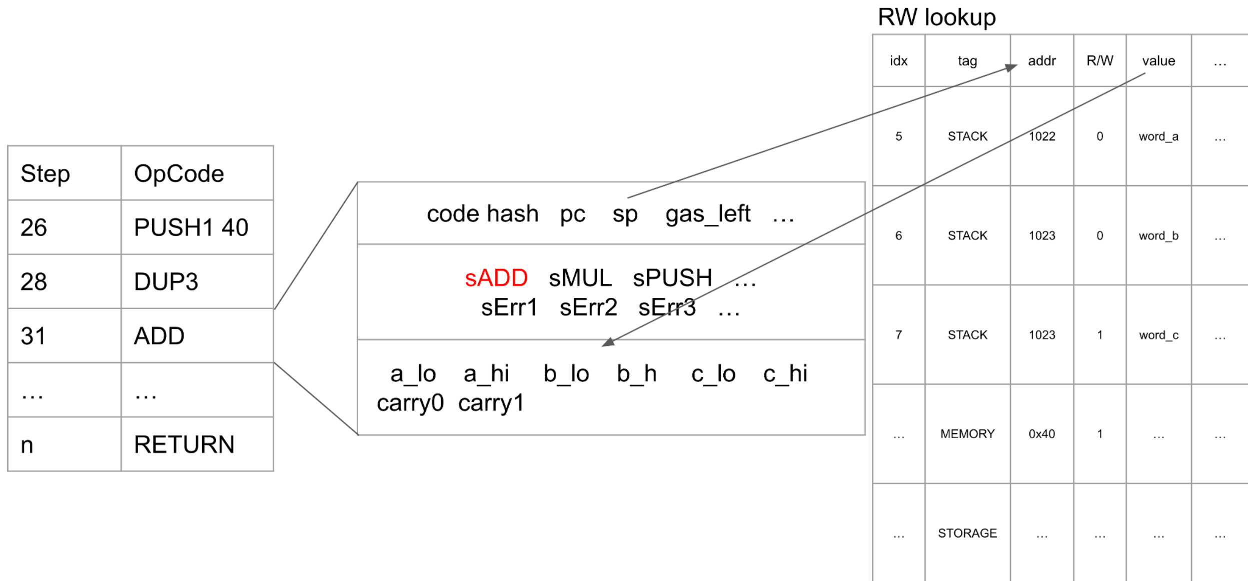
$$sADD * (pc_{next} - pc - 1) = 0$$

$$sADD * (sp_{next} - sp - 1) = 0$$

$$sADD * (gasleft_{next} + 3 - gasleft) = 0$$

지금까지 EVM 회로의 동작과 제약 조건을 살펴보았다. 마지막으로 EVM 회로에서 연산에 필요한 $a_{lo}$, $a_{hi}$와 같은 값을 RW Lookup 테이블에서 어떻게 가져오는지 알아보자. 먼저 ADD Opcode는, 두 입력을 Stack에서 가져온다. (Stack.POP) 이 후 값을 더한 후 Stack에 저장한다. (Stack.PUSH) 이를 확인하기 위해서 우리는 step context에 있는 stack 포인터를 통해 Lookup 테이블에 있는 값을 조회한 후, 이를 토대로 연산에 대한 검증을 진행하는 것이다. 아래 그림을 통해 이 과정을 확인할 수 있다.

klaytn




<그림 31 EVM회로와 RW lookup 테이블>

만약 ADD Opcode를 통해 계산된 $c_{lo}$, $c_{hi}$가 Lookup 테이블의 $word_c$와 동일하지 않다면, 우리가 생성한 RW Lookup 테이블에 문제가 있다는 것이고 증명 생성에 실패하게 된다.

지금까지 우리는 실제 스마트 컨트랙트의 함수를 호출한 트랜잭션을 통해 zkEVM 회로를 설계해보았다. 실제 zkEVM은 Bytecode, PI, MPT 등 다른 많은 회로들과 함께 이루어져 있지만, 기초적인 zkEVM 회로의 동작을 이해하는 데에 필수적인 State, EVM 회로를 살펴보았으니 다른 회로들은 직접 찾아보면 쉽게 이해할 수 있을 것이다.

klaytn

## 3.5 Verifying system

우리는 지금까지 zkEVM을 위한 회로들과 제약 조건들을 설정했다. 또한 gate, lookup, permutation 제약 조건의 계산과 검증까지 자세히 알아보았다. Halo2는 우리가 앞서 자세히 살펴본 PLONK 기반의 verifying scheme을 가지기 때문에, 다음 과정을 통해 최종적인 증명 및 검증이 진행된다.

1. 검증을 위한 다양한 값들을 evaulation하고 commitment를 생성하게 된다. 이 때, 어떤 다항식 $p(x)$에 대해 evaluation, commitment를[20] 생성하고, $z$를 challenge 값이라고 할 때 다음과 같은 데이터의 쌍을 가진다.

$$(z,\ p(z),\ [p(z)]_1)$$

2. 다항식에 대한 evaluation, commitment 값, 공개 입력 (Keccak 해시), 증명에 담긴 블록의 개수 등을 모아 최종적인 SNARK 증명을 생성하게 된다.
3. 위에서 계산한 값들을 transcript를 통해 Verifier에게 전달한다. 이 때, EVM을 위해 Keccak 해시함수를 통해 Fiat-Sharmir를 사용한다.
4. Verifier는 검증에 필요한 값들을 transcript를 통해 얻은 후, 이를 기반으로 최종적인 검증을 진행한다. PLONK에서 보았듯이 다음과 같은 형태의 pairing 연산이 진행된다.

$$e(W_x,\ [x]_2)\ =\ e(W_e,\ [1]_2)$$

결국 우리가 생성한 증명은 회로의 제약 조건을 표현한 다항식들과 이에 대한 여러 계산 결과이다. 그렇다면, 여러 증명을 하나로 모을 수는 없을까? 실제로 위에서 Verifier가 진행하는 검증의 pairing에서, 서로 다른 $W_x,\ W_e$를 RLC하여 다음과 같이 표현할 수 있다.

$$e(\sum_i \lambda^{i-1} W_x[i],\ [x]_2)\ =\ e(\sum_i \lambda^{i-1} W_e[i],\ [1]_2)$$

이를 통해 우리는 zkEVM에 대한 단일 증명과, 이를 Aggregation하여 더욱 효율적으로 검증을 진행할 수 있다.

본 리서치에서는 Verifying system에 대해 짧게 소개했지만, 실제 프로토콜에서 다항식의 evaluation 및 commitment를 통해 증명을 생성하는 과정은 구현 상 디테일이 매우 많고 복잡하다. 하지만 큰

[20] 이 때, PCS로 GWC19와 SHPLONK 중 하나를 선택하여 사용한다.

틀에서 다양한 제약 조건들을 생성하고 증명 및 검증하는 과정은 앞서 자세히 살펴보았다. 자세한 구현이 궁금하다면 Scroll 프로토콜의 깃허브[21]에서 모든 코드를 확인할 수 있다.

## 4. Conclusion

지금까지 우리는 영지식 증명의 수학적 기초부터 다양한 영지식 증명 프로토콜까지 살펴보고, 이를 통해 zkEVM 프로토콜의 실제 동작 방식까지 알아보았다. 이 과정에서 수많은 수학적 개념들과 복잡한 수식들을 이해해야 하기 때문에 쉽지 않았을 것이다. 하지만 영지식 증명은 zkEVM 뿐만 아니라 프라이버시 솔루션 및 머신러닝[22]까지 매우 다양한 분야에 도입되고 있는 유망한 기술이다.

현재는 zkEVM의 기술적 어려움 때문에 Optimistic Rollup 프로토콜들이 L2 시장을 선점하고 있다. 하지만 앞서 살펴본 Scroll 뿐만 아니라 Polygon, Taiko 등 많은 프로젝트들이 EVM 호환성을 가진 zkEVM 프로토콜을 개발하기 위해 노력하고 있다. 또한 영지식 증명을 통한 빠른 블록 완결성 및 보완은 분명 zkEVM 프로토콜이 가지고 있는 장점이며, 이를 통해 앞으로 zkEVM 프로토콜또한 시장에서 큰 관심을 받을 것이라 기대된다. 앞으로 출시될 다양한 영지식 증명 프로토콜을 이해하는데에 본 리서치가 도움이 되길 바란다.

klaytn

[21] https://github.com/scroll-tech
[22] https://www.youtube.com/watch?v=tr1TAq5-HUM

## 5.저자 소개

김 현 우

김현우(Lewis)는 현재 클레이튼의 개발자로서 core 개발을 하고 있으며, 이전에는 클레이튼 데브 엠베서더 활동을 하였다. 2020년부터 블록체인 관련 프로젝트들을 수행해왔으며, DeFi 프로토콜을 개발 해왔었다. 블록체인 분야 중 특히 영지식증명, 컨센서스, 거버넌스 구조등에 큰 관심이 있다. 또한 현재 PDAO(POSTECH-Oriented DAO & Open-Source Software Foundation)의 멤버로서 블록체인의 발전을 위해 여러 활동등을 하고 있다.

송 민 규

송민규(Mingyu Song)는 현재 광운대학교 재학중이며, 클레이튼 데브 엠베서더로서 활동을 했었다. 개인정보와 SSI에 관심이 있어 관련 프로젝트들을 여러 진행해왔다. 또한, 블록체인과 기존의 web2 서비스와의 연결을 지향하여 블록체인뿐만 아니라 일반적인 web2 기술에도 관심이 많다. 현재는 Web2 기술과 web3.0 을 연결하기 위한 프로젝트들을 진행하고 있다.

# 6. 참조 문헌 및 프로젝트


- [BCC+16] J.Bootle, A.Cerulli, P.Chaidos, J.Groth, and C.Petit. Efficient zero knowledge arguments for arithmetic circuits in the discrete log setting.pages 327–357, 2016.
- [Gro16] Groth, J.: On the size of pairing-based non-interactive arguments. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9666, pp. 305–326. Springer, Heidelberg (2016).
- [GWC19] Gabizon, A., Williamson, Z.J., Ciobotaru, O.: PLONK: permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Report 2019/953 (2019). https://eprint.iacr.org/2019/953
- [KZG10] Kate, A., Zaverucha, G.M., Goldberg, I.: Constant-size commitments to polynomials and their applications. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 177–194. Springer, Heidelberg (2010).
- Chiesa, A., Hu, Y., Maller, M., Mishra, P., Vesely, N., Ward, N.: Marlin: preprocessing zkSNARKs with universal and updatable SRS. Cryptology ePrint Archive, Report 2019/1047 (2019). https://eprint.iacr.org/2019/1047
- Boneh, D., Drake, J., Fisch, B., Gabizon, A.: Efficient polynomial commitment schemes for multiple points and polynomials. Cryptology ePrint Archive, Report 2020/081. Last revised January 31, 2020, https://eprint.iacr.org/2020/081
- https://github.com/ZK-Garage/plonk
- https://github.com/dusk-network/plonk
- https://github.com/zcash/halo2
- https://github.com/privacy-scaling-explorations
- https://github.com/scroll-tech
- https://www.alchemy.com/overviews/zkevm
- https://vitalik.ca/general/2021/01/05/rollup.html
- https://scroll.io/blog