ZK-EVM

23.03.30 ZK-School

Agenda

- zkEVM High-Level
- zkEVM 연구 트랜드
- zkEVM 프로젝트 현황

용어 tip

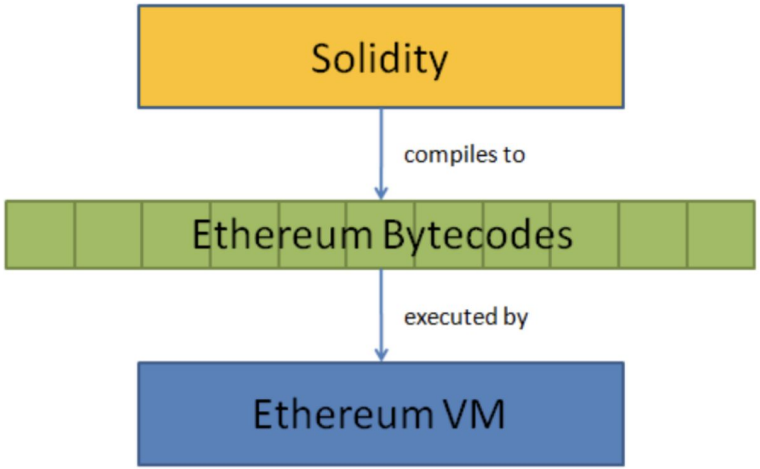
- 제약(constraints): 각 항목이 만족해야하는 조건들의 나열
- 회로(circuit): 조건을 만족하는 값을 찾기 위한 표현방식(코드화)

zkEVM High-Level

EVM(Ethereum Virtual Machine)은 블록체인 네트워크 노드들이 공유하는 하나의 가상머신이다.

EVM은 일종의 스택머신 형태로 작동하는데, 프로그래머가 solidity와 같은 하이레벨 언어로 코드를 작성하면,

그 코드가 EVM bytecode로 컴파일되어 전달되고 EVM은 그걸 실행하게 된다.



zk rollup의 역할

zkEVM은 EVM과 zk rollup의 증명인 validity proof가 호환되도록 만들어진, 영지식 증명 친화적인 가상머신이다.

1. 레이어2 네트워크에서 생성된 트랜잭션을 처리하여 상태를 계산
2. 이 상태 계산이 유효함을 증명할 수 있는 validity proof를 생성
3. validity proof를 레이어1에 전달하여 검증받은 후, 레이어2의 상태를 업데이트

현재 zk rollup에서 구동되는 app은 validity proof를 구성하는 특성으로 인해 payment와 swap으로 제한되어 있다.

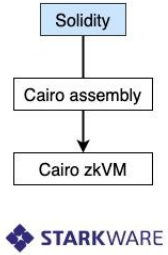
zk rollup의 역할

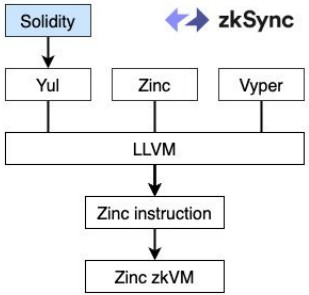
트랜잭션의 유효성을 증명하기 위해 확인해야하는 조건

- Payment 트랜잭션
  1. 사용자의 서명이 올바른가?
  2. 사용자의 잔액이 송금액보다 큰가?
  3. 전송자와 수신자가 받은 금액이 같은가?
- Swap 트랜잭션
  1. 사용자의 서명이 올바른가?
  2. 토큰 페어의 풀 유동성이 충분한가?
  3. 사용자의 잔액이 스왑액보다 큰가?
  4. x*y = k로 계산된 토큰 스왑량이 올바른가?
  5. 스왑 후 풀의 잔액이 올바르게 업데이트 되었는가?

단, circuit은 static하여 한 번 만들면 바꿀 수 없고, 다른 circuit과의 호환성도 없다.

zkEVM의 세가지 단계





Language Level

언어 단계는 코드의 기본이 되는 언어부터 영지식 친화적으로 설계하는 것을

장점

- 언어 자체를 바꾸는 솔루션의 장점은 EVM의 설계 방식에 대한 제한을 받지 않고, 영지식 친화적인 VM(zkVM)을 처음부터 자체적으로 설계할 수 있다는 것이다.

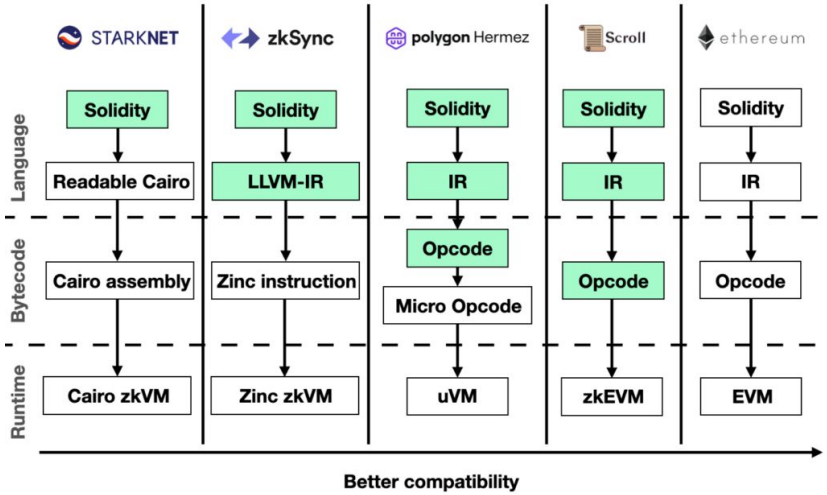
단점

- 언어를 바꾸게 되면, 기존 EVM 개발자의 경험을 해친다.
- EVM opcode를 지원하지 않는다는 특징은 EVM기반의 Layer 1 생태계를 직접 상속(inherit)하지 못하는 결과를 야기하며, 새로운 언어를 배우는 것은 블록체인 개발자가 쉽게 들어오지 못하는 장벽이 된다.

zkSync의 방향 전환

- Optimism과 Arbitrum 경쟁에서 Arbitrum이 승리한 이유도 결국 EVM과의 호환성때문이었다.
- zkSync는 이러한 시장 현황을 의식한 듯 언어 단계의 zkVM에서 바이트코드 단계의 zkEVM으로 구축 방법을 변경했다.

zkEVM의 세가지 단계



Bytecode Level

바이트코드 단계는 Solidity와의 호환성을 달성할 수 있을 뿐만 아니라 EVM의 opcode 단계와도 완전한 호환성을 달성할 수 있는 단계이다.

장점

- 개발자는 새로 익혀야하는 것 없이 이미 익숙한 Layer 1의 개발 도구에 대한 수정없이 Layer 2로 마이그레이션할 수 있다.

단점

- 기존의 opcode와 같은 EVM의 설계를 변경하지 않기 때문에 추가로 들어가는 공수가 매우 많다.

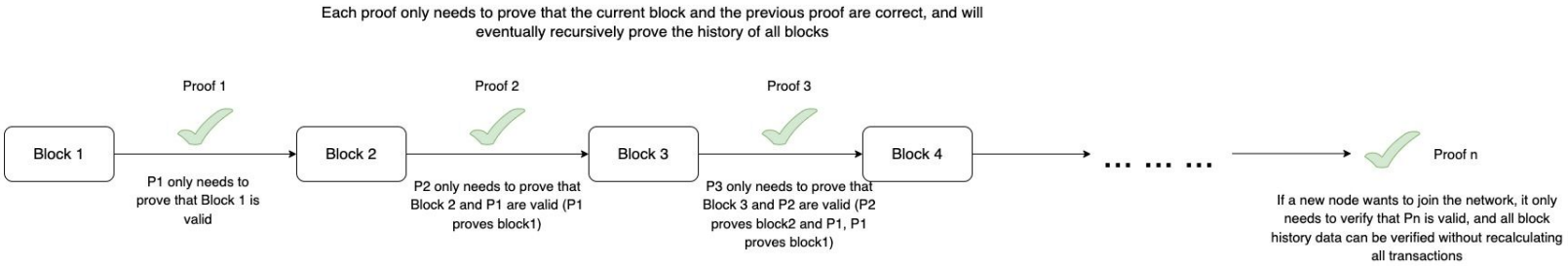
zkEVM의 세가지 단계

Consensus Level

- 컨센서스 단계는 zkEVM의 최종 목표로 컨센서스 (노드)까지 EVM과 완전한 호환성을 달성하여 지금의 이더리움을 대체할 수 있는 단계이다.

Recursive proof

- 컨센서스 단계에 recursive proof를 사용하면, 전체 블록의 유효성을 증명할 때 하나의 증명만 사용할 수도 있다.
- 이를 통해, 이더리움은 누구나 쉽게 네트워크에 합류할 수 있도록 하여 더욱 분산화되고 강건해진 네트워크가 될 것이다.



Bytecode Level의 zkEVM Logic

Understanding Ethereum Virtual Machine Logic

Layer 1이 computation integrity를 구성하는 방법은 smart contract의 재실행(re-exeuction)이며, 이를 위해 다음과 같은 로직을 따른다.

1. L1에 배포된 smart contrat의 바이트코드는 이더리움의 저장소(storage)에 저장된다.
2. 트랜잭션은 P2P 네트워크로 브로드캐스트(broadcast)된다.
3. 각 트랜잭션에 대한 풀 노드(full node)는 트랜잭션의 처리로 동일한 상태값(state)에 도달하기 위해 바이트코드를 로드(load)하고 이를 EVM에서 실행한다.

각 opcode는 다음과 같은 3가지 sub-step을 수행한다.

1. stack, memory or storage의 원소를 읽는다
2. 해당 원소에 대한 일부 계산을 수행한다.
3. stack, memory or storage의 결과를 다시 쓴다.

Bytecode Level의 zkEVM Logic

Layer 2의 zkEVM의 핵심 로직

바이트코드가 정확한 opcode를 실행하고 어떤 opcode를 빠뜨리거나 건너뛰지 않고 올바른 순서대로 실행했음을 증명해야한다. 레이어2의 EVM의 동작은 다음과 같다.

1. 레이어2의 바이트 코드를 storage에 저장한다.
2. 트랜잭션은 중앙집중식 zkEVM으로 off-chain에 전송된다.
3. zkEVM은 바이트코드의 실행뿐 아니라 트랜잭션 처리 후 계산된 상태가 올바르게 업데이트되었음을 증명하는 proof를 생성한다.
4. 레이어1 컨트랙트는 이 proof를 검증하고, 트랜잭션을 재실행하지 않고 레이어2의 state를 업데이트한다.

실행 과정의 유효성을 증명하기 위해서는 다음과 같은 항목을 증명해야한다.

1. 정확한 주소에서 로드된 바이트코드로 정확한 opcode를 실행했다.
2. 바이트코드는 어떤 opcode를 빠뜨리거나 건너뛰지 않고 올바른 순서대로 바이트 코드를 실행했다.
3. 각 opcode가 올바르게 실행되었다.

이를 위해 state에 대한 증명과 EVM 실행에 대한 증명을 별개로 분리하여 만든다: State proof, EVM proof

Bytecode Level의 zkEVM Logic

State proof가 증명하는 것

- stack, memory, storage의 상태 전환(state transition)이 정확함(correctness)를 증명함

EVM proof가 증명하는 것

- Add, Mul, Sum 등의 opcode 연산이 정확하게 실행되었음을 증명함
- 예: 정확한 시간에 올바른 opcode가 사용되었는가. opcode 자체가 유효하고 정확한가. opcode의 실행중에 발생할 수 있는 모든 비정상적인 조건(예: out_of_gas)이 존재했는가 등

zkEVM의 구축을 위한 핵심 아이디어는 state와 EVM의 실행에 대한 증명 프로세스를 분리하여 처리하는 것이다.

Heremz, zkSync2.0, Appliedzkp, Sin7Y의 zkEVM의 프레임워크는 세부적으로 다르지만, 이를 분리한다는 핵심 아이디어는 같다.

Scroll’ native zkEVM architecture

Scroll의 native zkEVM의 핵심은 constraint를 구현하는 circuit의 overhead를 줄이기 위해 연산에 필요한 sub-circuit과 sub-table을 미리 만들어 가져다 쓰는 것이다.

zkEVM(bytecode level)은 zk proof로 구현하는 opcode의 종류에 따라 native zkEVM과 custom EVM으로 나눠진다. native EVM은 EF의 AppliedZKP와 scroll이 구현하고 있고, custom EVM은 Hermez와 Matter Labs, sin7Y가 구현하고 있다.

- native zkEVM (EF에서 주로 연구 중)
  - AppliedZKP: Solidity → zkEVM
  - Sin7Y: Solidity → zkEVM
- custom zkEVM
  - Starkware: Solidity → (Comliper 예정) → Cairo → zkVM
  - (ex-)zkSync: Solidity → YUL → LLVM → zkEVM
  - Hermez: Solidity → micro Op → uVM

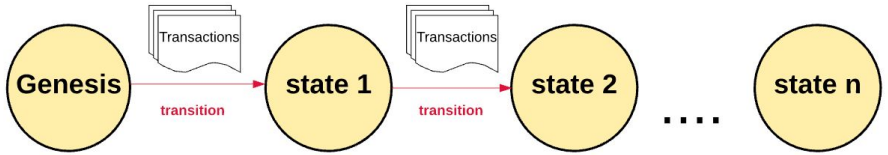
Scroll’ native zkEVM architecture

EVM의 기능은 트랜잭션을 계산하여 state 1에서 state 2로 이동시키는 상태 머신이며, 트랜잭션은 가장 작은 상태 변화를 유도하는 연산이라 이해하면 된다.

zkEVM의 기본 아이디어는 EVM을 제약하는 EVM circuit을 생성하여 EVM의 모든 실행 로직이 올바른지 증명하는 것이다. EVM circuit은 상태 변경에 사용된 트랜잭션과 그 트랜잭션에 의해 호출된 특정 opcode를 얻을 수 있다.

그 다음 circuit에서는 다음 사항들이 완전히 정확한지 증명한다.

- 상태 변경에 사용된 트랜잭션의 유효성
- Opcode가 트랜잭션에 의해 호출이 되었는가
- Opcode들의 연산 논리가 정확한가
- 연산 순서가 올바른가

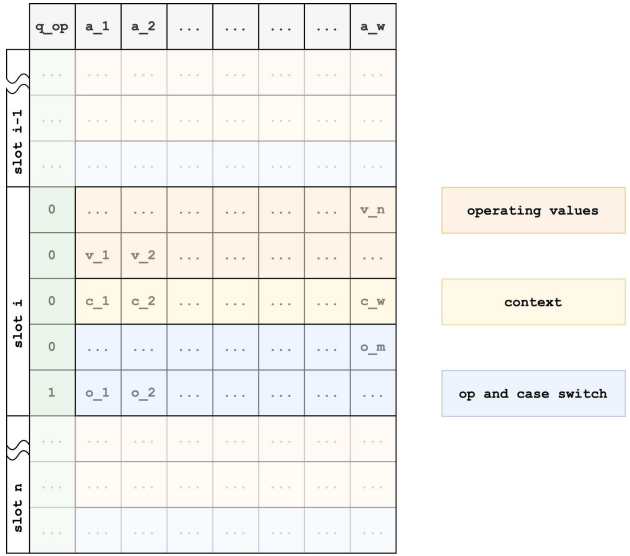


Scroll’ native zkEVM architecture

EVM에서 만족해야하는 조건들(constraint)을 단 하나의 코드(circuit)에서 구현해야한다면,

circuit은 너무 거대해져 불필요한 복잡성과 overhead가 유발된다.

이에 scroll은 EVM의 각기 다른 module을 위한 sub-circuit/table 을 설계하여 복잡성을 줄였다.



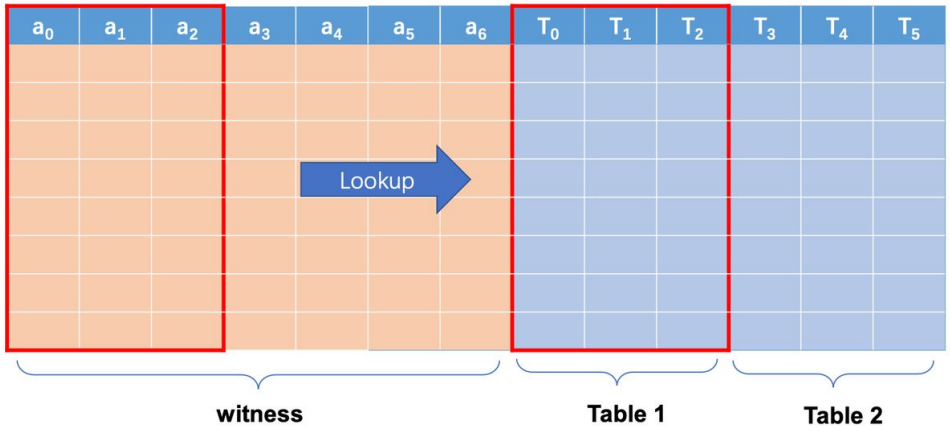
Scroll’ native zkEVM architecture

이제 EVM circuit이 proof를 생성할 때 constraint마다 다른 circuit을 구축하는게 아니라, 관련된 table을 query하면 된다.

예를 들어, memory/stack/storage의 read/write의 로직을 증명하는 경우 EVM circuit은 state table하면 된다.

state table에는 sub-circuit인 state circuit이 포함되어 있기 때문에, 이를 사용하여 proof를 생성할 수 있다.

- Opcode와 관련된 연산의 정확성을 증명하는 경우, EVM circuit은 bytecode table을 query
- Transaction block은 Tx table과 Block table을 query



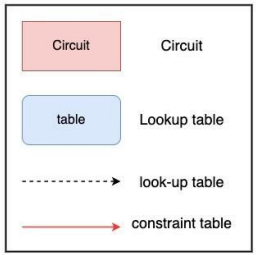
Scroll’ native zkEVM architecture

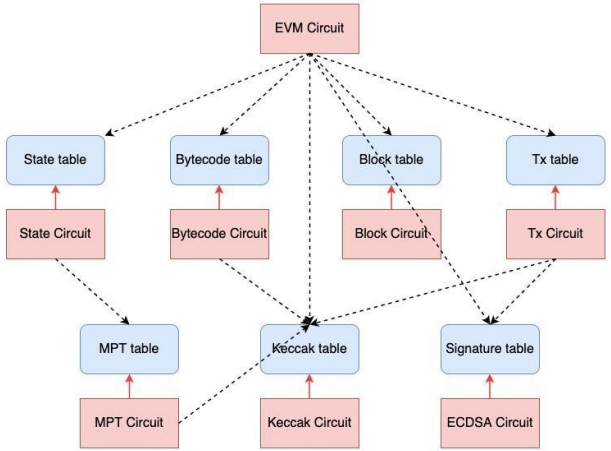
각 sub-circuit에는 constraint와 관련된 연산이 포함되어 있는 또 다른 sub-table를 query한다. State circuit은 storage와 관련된 연산을 constraint하기 위해 MPT 연산을 사용해야하기 때문에 이와 관련된 MPT table을 query한다.

또한 Tx Circuit은 Hash와 Tx sig를 검증할 때 사용해야하는 keccak table과 signature table을 query한다.

- State circuit → MPT table
- Tx circuit → Keccak table, Signature table

한편, 고정된 값이 아니기 때문에 악의적인 증명자가 유효하지 않은 table을 만들어 증명을 위조할 수 있다. 따라서, 연산뿐만이 아니라 “table 자체의 정확성"을 보장하는 것도 중요하다





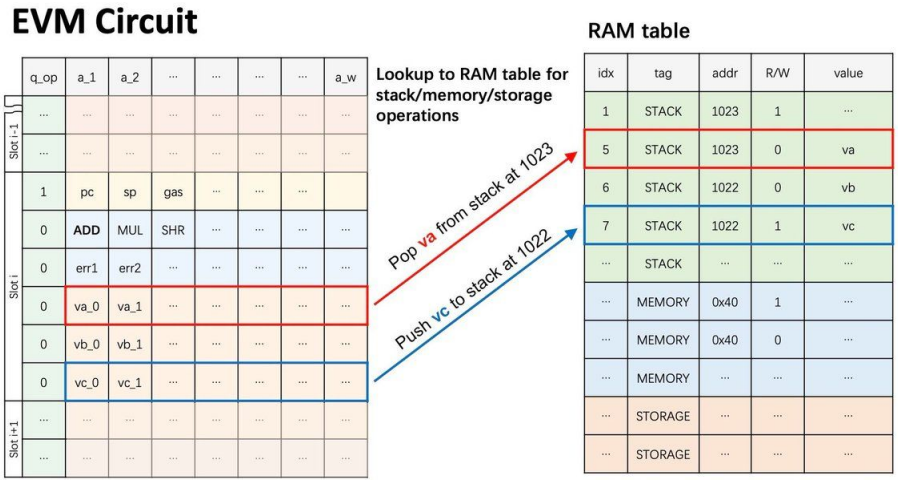
Scroll’ native zkEVM architecture

트랜잭션(trace)가 EVM circuit에 포함될 때, 모든 연산(opcode, stack/storage 등)은 재정렬된 다음 sub-circuit에 할당된다.

이 sub-circuit는 연산의 정확성을 증명하고 proof를 생성한다.

마지막으로, proof는 공개 입력값으로 aggregation circuit에 입력되고,

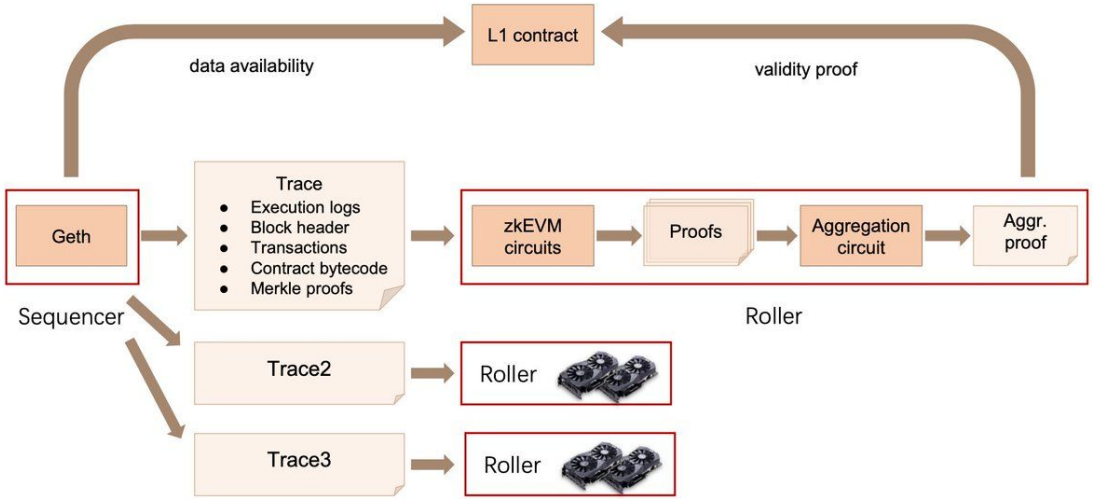
aggregation circuit은 각각의 단일 proof를 모아 aggregate proof로 만든다.

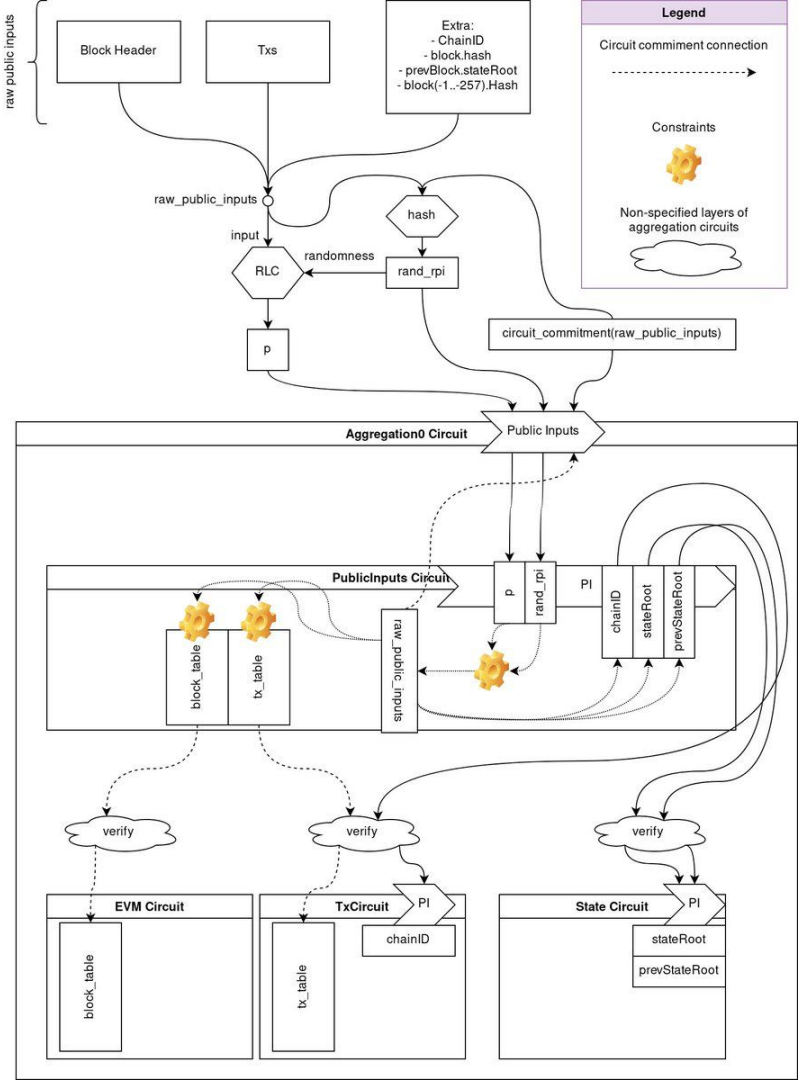


Scroll' native zkEVM architecture

이후 aggregate proof는 Layer 1 contract로 전송되어 검증된다.

아래 그림은 scroll이 만들고 있는 zkEVM의 High-Level workflow이다.





Hermez' custom zkEVM

Hermez의 가장 혁신적인 설계는 EVM instruction set을 intermediate instruction(micro opcode)로 변환하여 uVM에서 실행할 수 있도록 만드는 것이다.

여기에 많은 수의 plookup 알고리즘을 사용하여 증명 및 검증의 효율성을 향상시킨다.

uVM은 Intel의 x86와 유사하다(보통 VM을 os라고 부르는 것 같음. starkware도 그렇고).

여기에는 ROM/RAM과 같은 모듈이 있으며 Main State Machine을 사용하여 모듈간의 상태를 동기화한다. uVM은 동일한 작업을 수행하는 stack 대신 register를 사용한다

- **uVM:** A zkCircuit optimised VM with tailor made opcodes to optimise the EVM interpretation.
- **Easy to implement variable EVM opcodes:** CALL, DATACOPY, EXP, CREATE, etc.
- **Multi StateMachine architecture** to implement difficult opcodes like MULMOD, EXP or even pairings.
- **Fixed uVM assembly code** to process all transactions of a block fetching and interpreting all the opcodes.

zkEVM 연구 트렌드

zkEVM의 연구 트렌드는 **1) 테이블 2) recursive proof 3) other zk algorithm**으로 나뉜다.

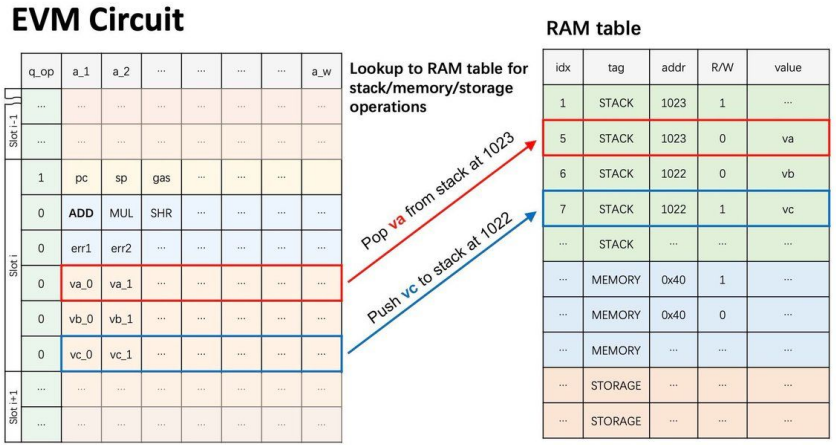
Lookup table

Lookup table의 역할은 legitimate (input, output) combinations를 미리 테이블에 계산해놓은 뒤, 증명자가 이 table에 witness value가 존재한다고 주장할 수 있는 알고리즘을 지원하는 것이다.

대표적인 것은 Plonk 알고리즘과 결합된 Plookup과 Halo2와 결합된 Lookup table이 있다.

기본적으로 Lookup table을 사용하면 다음을 만들 수 있다.

1. state Proof: 구현 절차가 올바르다는 것을 증명하는 proof
2. State proof: stack/memory/storage의 operation이 올바르다는 것을 증명하는 proof
3. EVM proof: input, output, operational logic이 올바르다는 것을 증명하는 proof



Lookup Argument in Halo2

Lookup table은 Halo2의 lookup argument기술을 사용해 random set에서 lookup을 실행할 수 있다.

Recursive Proof

zkp에서 recursion이란, Relation (circuit) 안에서 이전에 생성된 proof를 verify하는 코드를 포함하는 것을 의미한다.

원래 Verify code는 zkp가 아니라 Layer 1의 smart contract에서 실행된다. 백엔드로 kate pairing을 사용한 경우, verify를 위해서는 groth16 ECC 연산을 수행해야한다. 이 연산 그대로 zkp circuit에서 수행하면, (여러가지 이유로) 연산비용이 비싸지기 때문에 실질적인 구현에 어려움이 생긴다.

Halo는 circuit안에서 verify code를 효율적으로 수행할 수 있도록 설계된 알고리즘이다. 간단하게, Halo는 recursion의 모든 step에서 verify code를 매번 실행하지 않고, 마지막 step에서 한 번만 실행한다.

Custom Gate

Custom gate는 circuit의 gate 수를 줄이고 circuit 설계의 유연성을 높이는 방법이다.

- turboPlonk: custom gate + plonk
- Ultra Plonk: custom gate + plonk + lookup table

zkEVM circuit을 설계할 때는 많은 custom gate를 정의해야하기 때문에 binary selector가 많이 도입된다. Selector가 불필요하게 많이 있으면 증명 생성과 검증에 비용이 많이 발생하기 때문에, combined selector를 통해 selector polynomial을 최적화시키는 방법을 고안해야한다.

zkEVM 프로젝트 현황

zkVM (Language Level): StarkNet

- 언어부터 영지식 친화적으로 설계하여 자체 VM에서 코드를 실행
- 장점: EVM 설계에 제한받지 않고, 당장 돌아갈 만큼 견고한 프로덕드가 존재하며, 오버헤드가 적음
- 단점: EVM과의 호환성이 없고, 언어를 다시 배워여하기 때문에 개발 접근성이 떨어짐

Custom zkEVM (bytecode Level): PolygonHermez, zkSync, sin7Y

- 언어는 Solidity를 지원하지만, 영지식 친화적으로 설계한 VM에서 코드 실행
- Hermez: Solidity → micro Op → uVM(Intel의 x86과 유사한 os)
- 장점: 언어를 따로 배우지 않아도 되서 개발 접근성이 비교적 좋음
- 단점: 완전한 EVM 호환성이 있는 건 아님. 자체 VM에서 EVM의 연산을 지원하기 위해 추가 설계해야함

Native zkEVM (bytecode Level): Scroll

- Solidity와의 호환성 뿐만 아니라 EVM opcode와도 완전한 호환성을 달성
- 장점: 이더리움의 dApp을 그대로 Layer 2에 마이그레이션 가능
- 단점: EVM을 변형하지 않으니, 영지식을 결합하기 위해 고려해야하는 것들이 많음(circuit table 디자인 등)

Questions?!