

# DeFi 취약점 탐지와 정보보호(1) : Ethereum Basics

2025. 9. 27. 성균관대학교 법학관



# 강사 소개

- **Education**

- 경찰대학 법학사(2019)
- KITRI BoB(2016)
- Upside Academy(2025)

- **Experience**

- 제주해안경비단(2019 - 2021)
- 경기도남부경찰청(2021 - 2024)
- 경찰대학(2024 -)

- **Awards**

- 화이트햇 컨퍼런스(보스턴대, 2024)
- 폴-사이버 챌린지(2022 - 2024)



# 아래 키워드를 알고 있나요?

## Blockchain

- ☐ Hash Function
- ☐ Mempool

- ☐ Node(Full/Archive)
- ☐ RPC

## Consensus Mechanism

- ☐ PoS(Proof of Stake)

- ☐ PoW(Proof of Work)

## Ethereum

- ☐ World State
- ☐ Memory/Storage
- ☐ ECDSA
- ☐ EVM(Ethereum Virtual Machine)
- ☐ EOA(Externally Owned Account)

- ☐ Merkle Tree(Merkle-Patricia Trie)
- ☐ Transaction
- ☐ Nonce
- ☐ Gas
- ☐ CA(Contract Account)

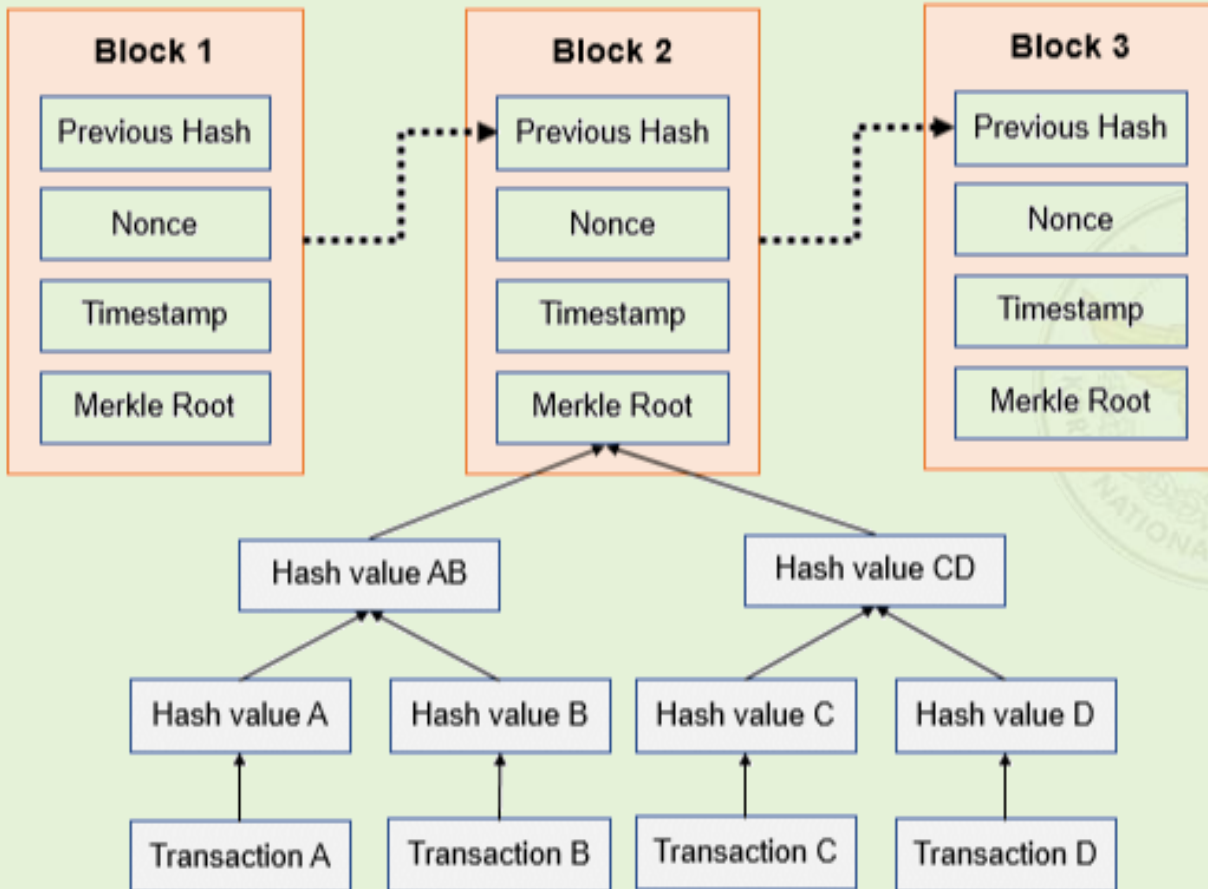
# Table Of Contents

1. 블록체인
2. 이더리움에 대하여
3. EVM
4. Solidity와 개발 패턴

# 1. 블록체인



# 블록체인이란?

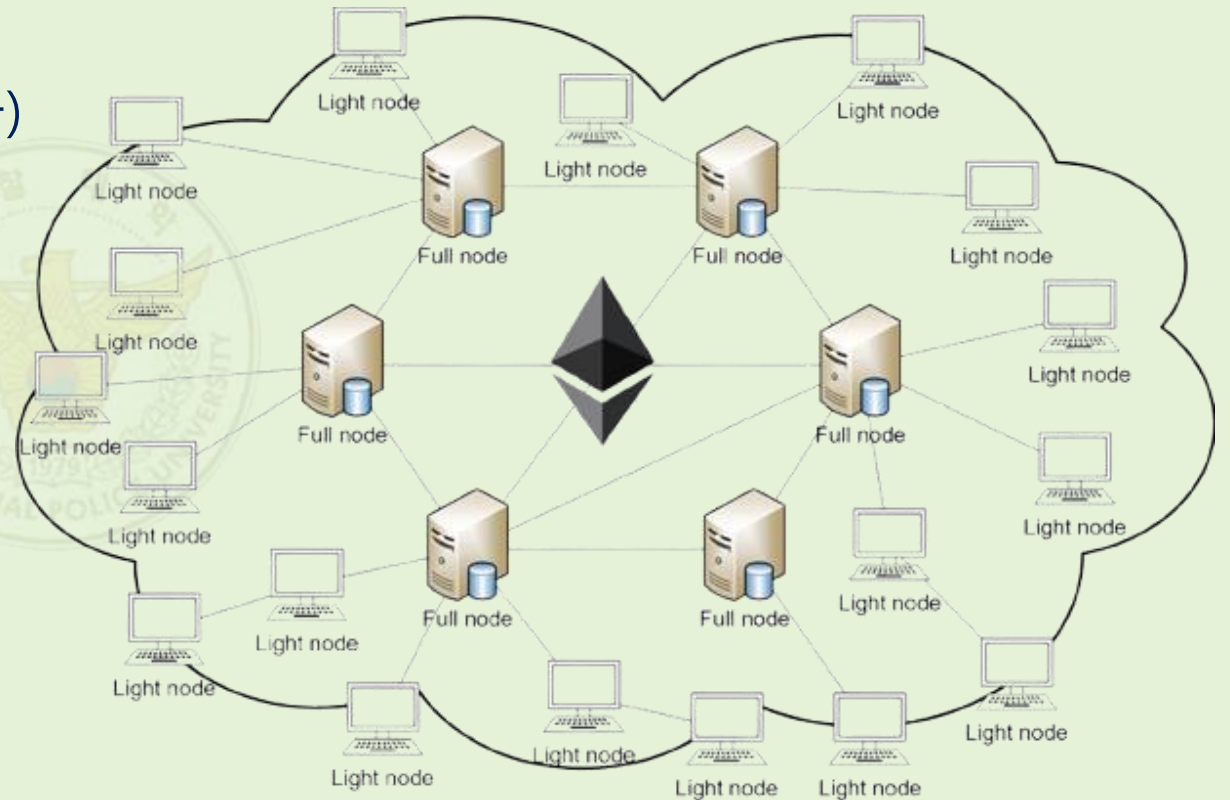


- 데이터 묶음인 **블록**을 연결한 것
- 이전 블록의 Hash를 포함하고 있음  
=> T(N) 시점의 해시를 검증하면 T(0)...T(N-1)는 보지 않아도 됨
- 블록이 추가되는 속도는 **체인마다 상이**
- Nonce는 채굴자가 작업증명을 위해 열심히 찾아낸 값
- Timestamp는 채굴자의 UNIX시간 (정확할 필요 X)
- 그 외에 다양한 정보들(대부분 RootHash)



# 블록체인 네트워크

- 여러 컴퓨터(노드)들이 서로 P2P로 연결된 네트워크
  - **Light node**: 일부 블록만 저장하거나 일부 기능만을 수행(트랜잭션 제출 등)
  - **Full node**: 트랜잭션 제출, 검증, 실행 등 모든 기능을 수행함
- 노드들은 체인에서 정한 규칙을 따름
  - **PoW**(작업 증명)
  - **PoS**(지분 증명)
- 서로 블록 정보를 주고받아 같은 상태로 유지
  - 서로 다른 블록을 정당한 블록이라고 주장하는 경우 **네트워크에 분열**이 발생(Fork)



## 2. 이더리움에 대하여





# 비트코인과 이더리움 비교

 VS 	Bitcoin	Ethereum
Founder	Satoshi Nakamoto	Vitalik Buterin
Release Date	9 Jan 2008	30 July 2015
Release Method	Genesis Block Mined	Presale
Blockchain	Proof of work	Proof of work (Planning for POS)
Useage	Digital Currency	Smart Contracts Digital Currency
Cryptocurrency Used	Bitcoin(Satoshi)	Ether
Algorithm	SHA-256	Ethash
Blocks Time	10 Mintues	12-14 Seconds
Mining	ASIC miners	GPUs
Scalable	Not now	Yes

# 비트코인 트랜잭션

- [BT](#)



# 비트코인 트랜잭션



I own  
3 bitcoins



I own some  
UTXOs that  
allow me to  
spend 3 bitcoins




# 이더리움 트랜잭션

- ETH 트랜잭션  
루어짐

Summary 1 | Contracts 2 | Events 3 | State 4 | Gas Profiler 5


### State Changes

 **USDC** ERC20 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48


mapping (address => uint256) balanceAndBlacklistStates

0x685b7c9a85b8f1e1a168892fb2dd35399bfdcb99	1226199000000 → 0
0xa9d1e08c7793af67e9d92fe308d5697fb81d3e43	69183574089267 → 69306193989267

✓ Show raw state changes (2)

 **Address** 0x4838B106FCe9647Bdf1E7877BF73cE8B0BAD5f97

Balance 16018290146987152445 → 16018330518987152445

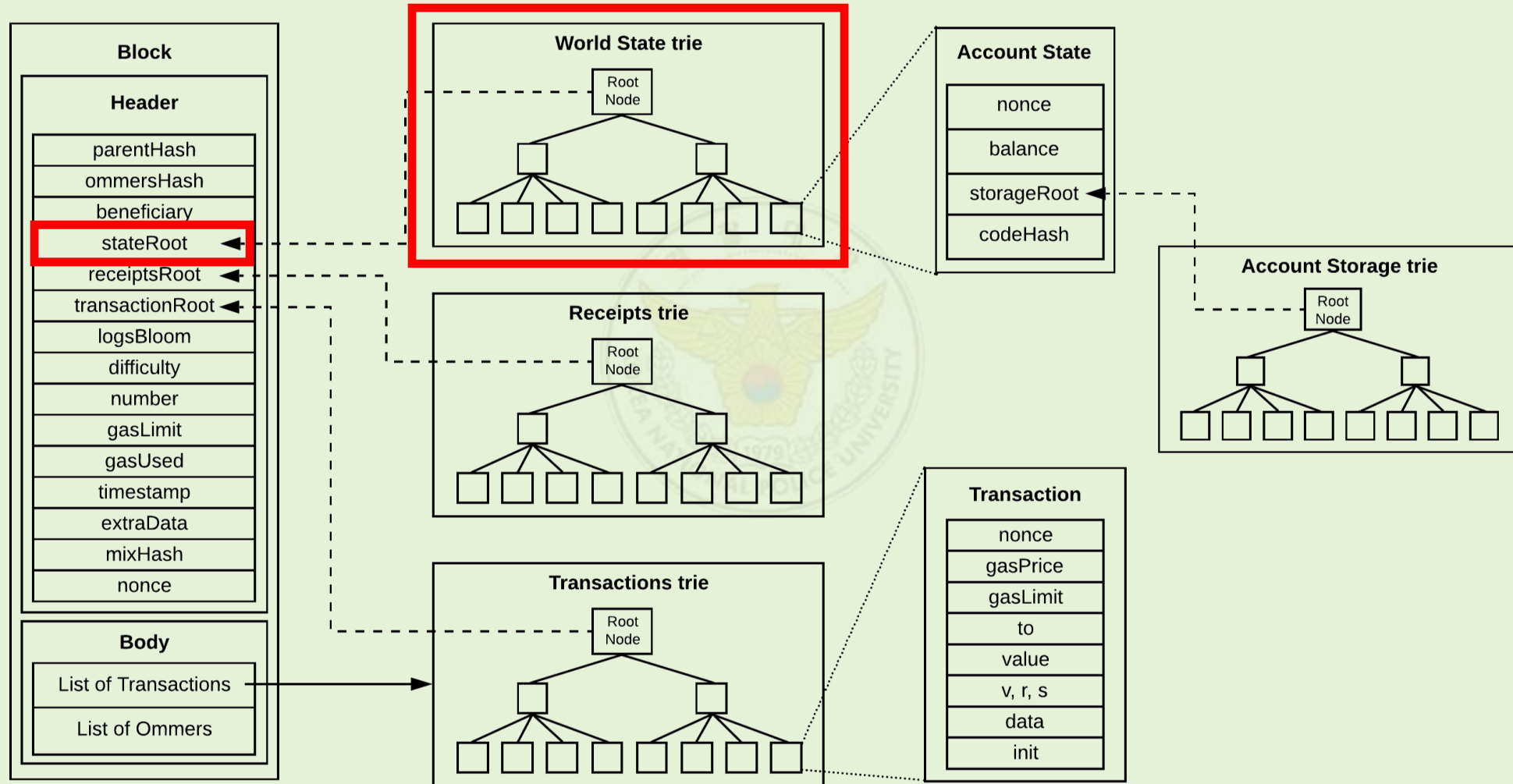
 **Address** 0x685b7c9a85b8f1e1a168892fb2DD35399bFDCB99

Balance 80097500000000 → 34708568089452

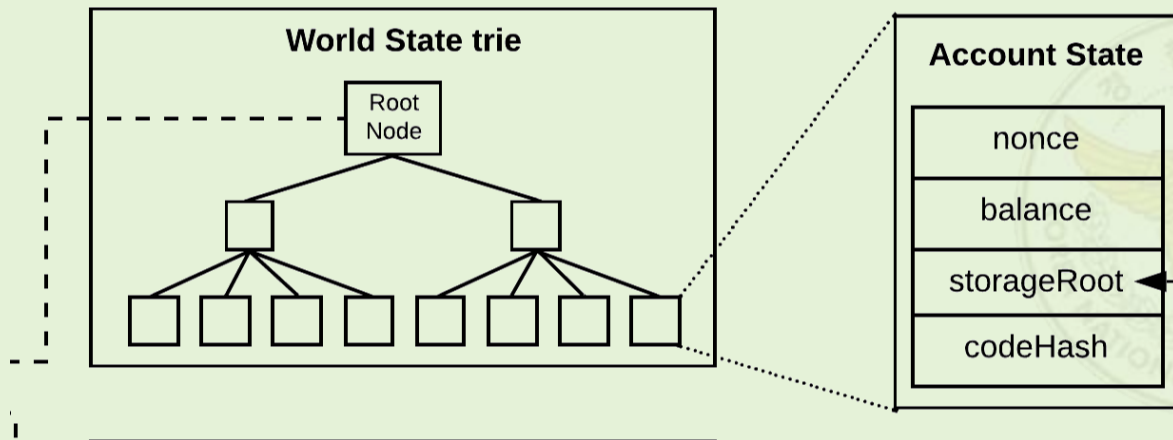
Nonce 17426 → 17427

변화가 이

# What is World State?



# State Trie(1)

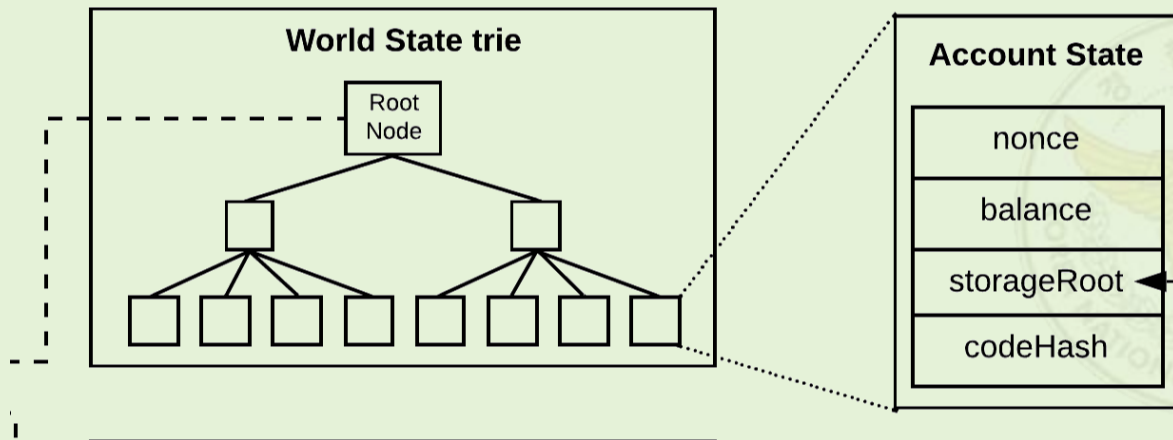


- Merkle Patricia Trie(MPT) 구조
- **Key-Value** 형태로 자료를 저장함
  - Key - H(지갑 주소)
  - Value – Account State  
RLP([nonce, balance, storageRoot, codeHash])
- 기본값은 0 or H(NULL)

\* 이후로 H(x)는 keccak256 digest을 의미합니다.

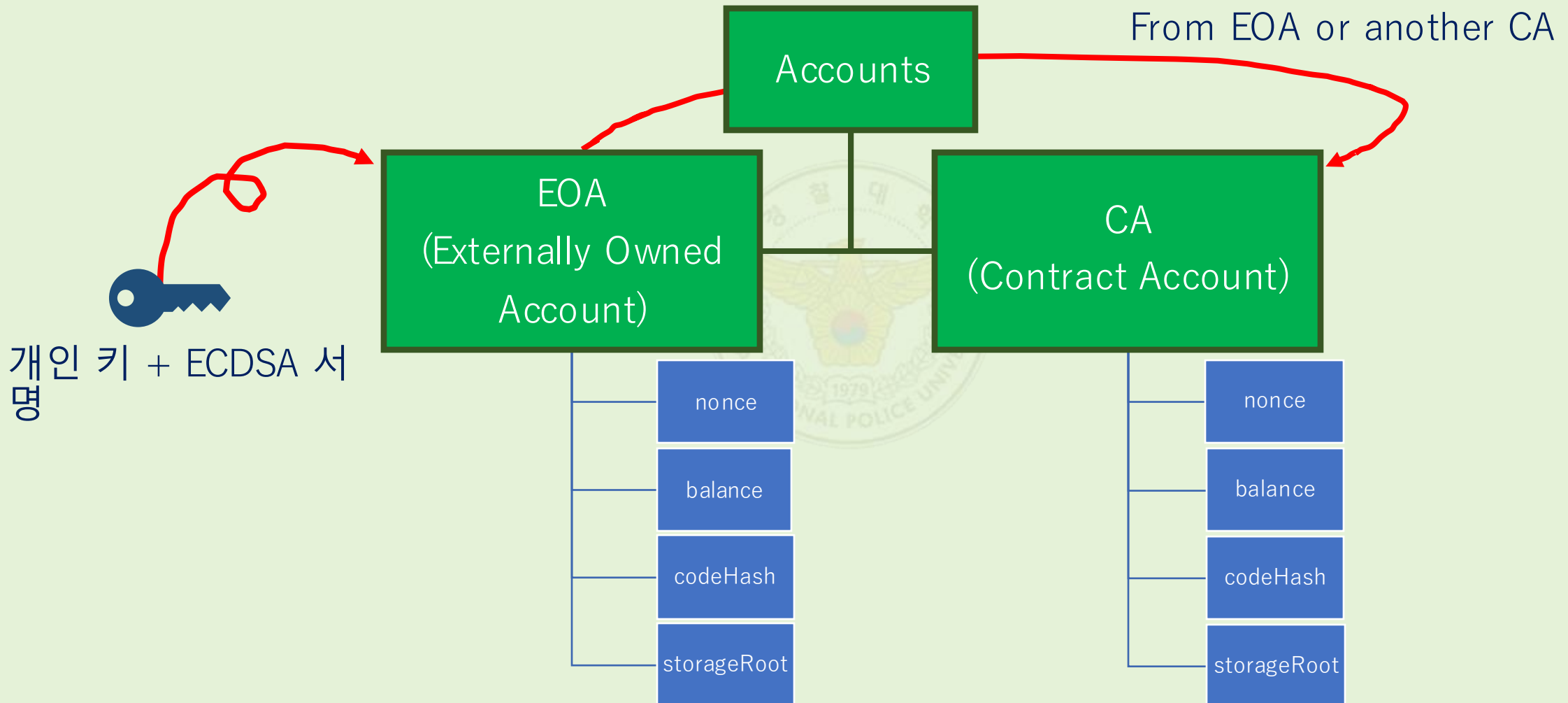


# State Trie(2)



- nonce: 이중지불 방지 장치
  - 0으로 시작, 트랜잭션당 1씩 증가
  - 이전 트랜잭션이 처리되지 않을 경우 무제한 교착상태에 빠짐
- balance: 이더리움 잔액(native ETH)
  - $10^{18}$ 의 고정 소수점으로 표현
- storageRoot
  - Storage trie의 최상단 노드의 해시값
- codeHash
  - 컨트랙트 코드의 해시값

# Ethereum Accounts



# Ethereum Transaction(1)

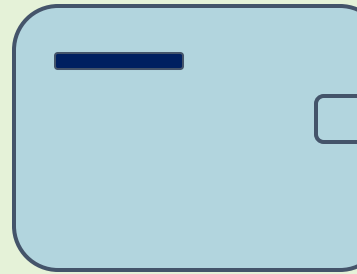


트랜잭션은 **EOA만 시작**할 수 있으며, 이더리움의 World State를 변경  
함

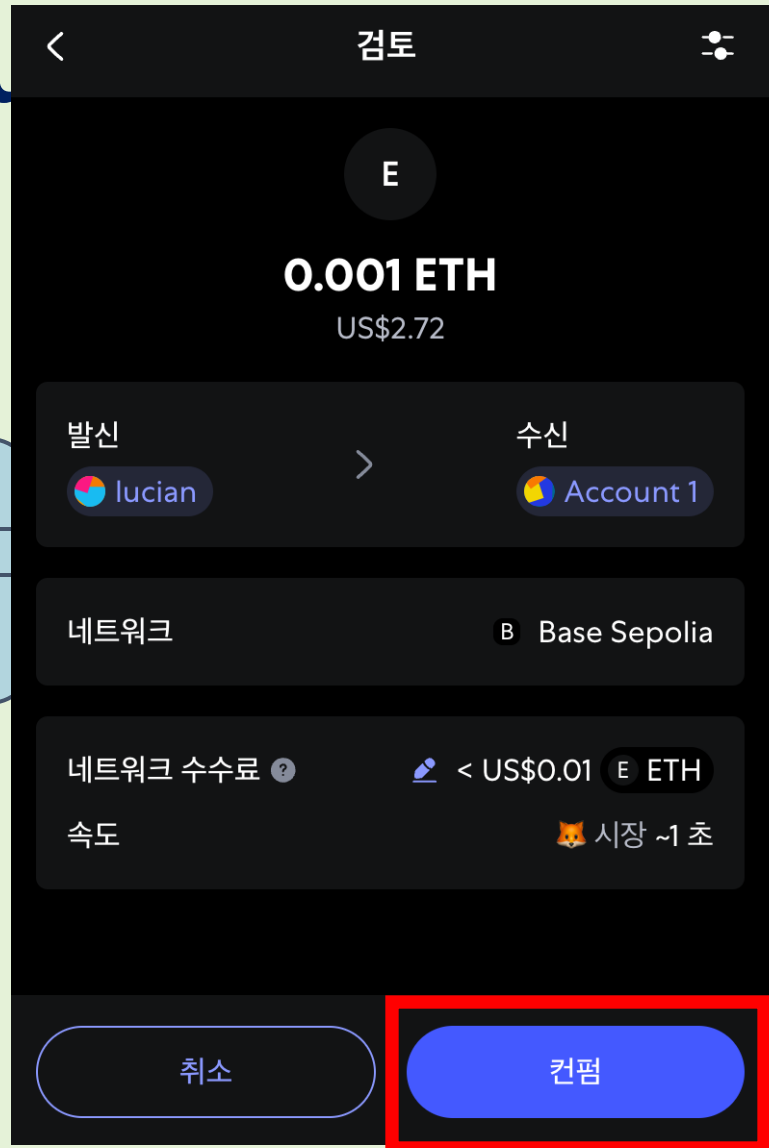


# Ethereu

# Flow(1)

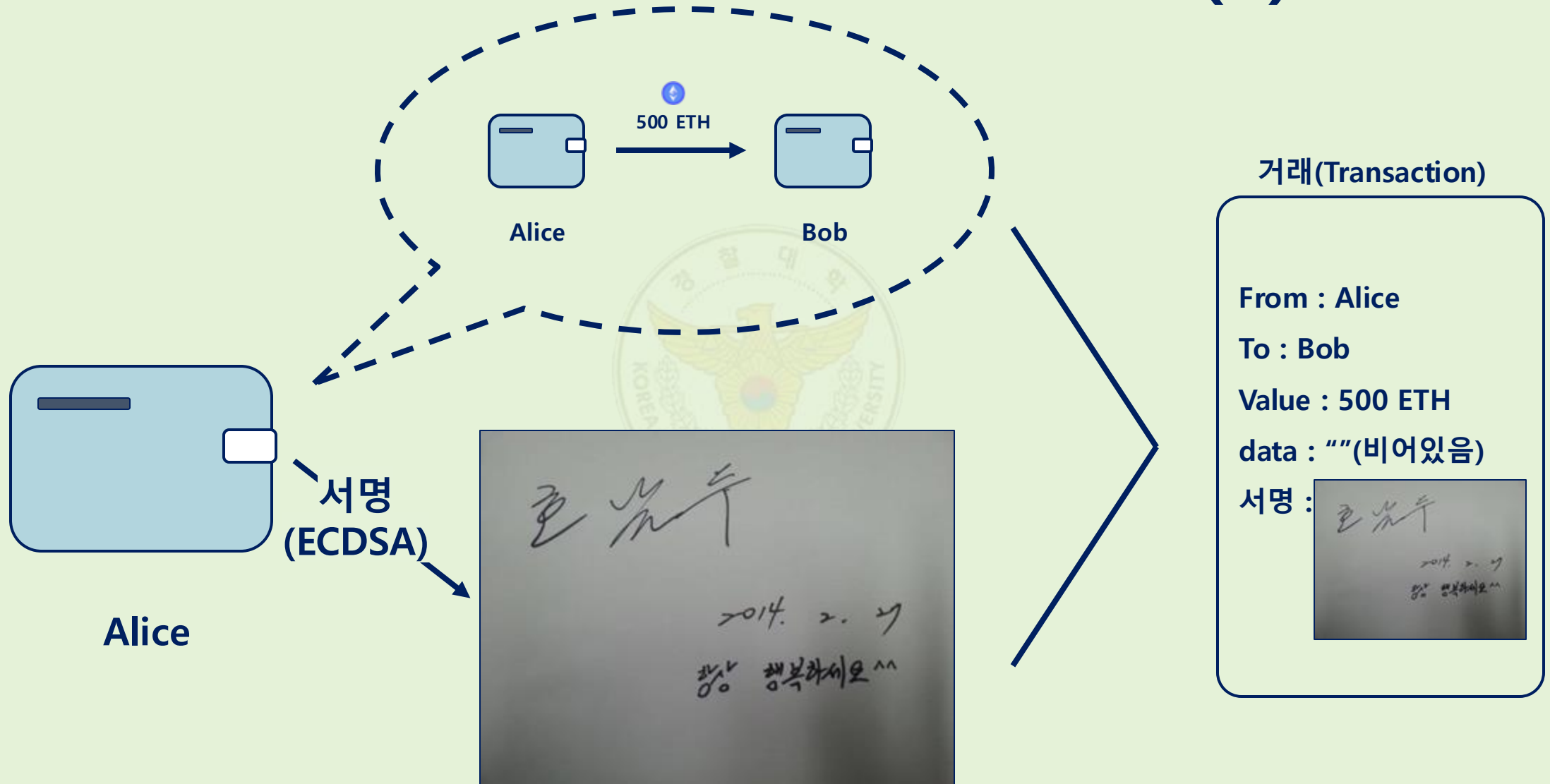


Alice

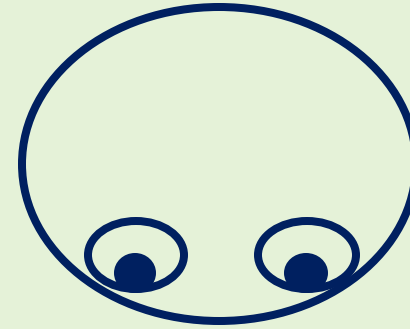


Bob

# Ethereum Transaction Flow(2)



# Ethereum Transaction Flow(3)



제안자  
(Proposer)

거래(Transaction)

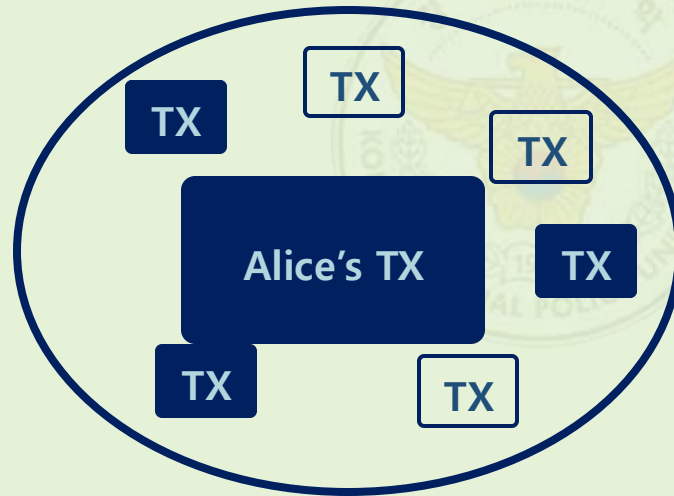
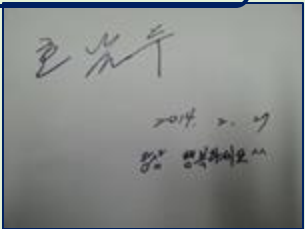
From : Alice

To : Bob

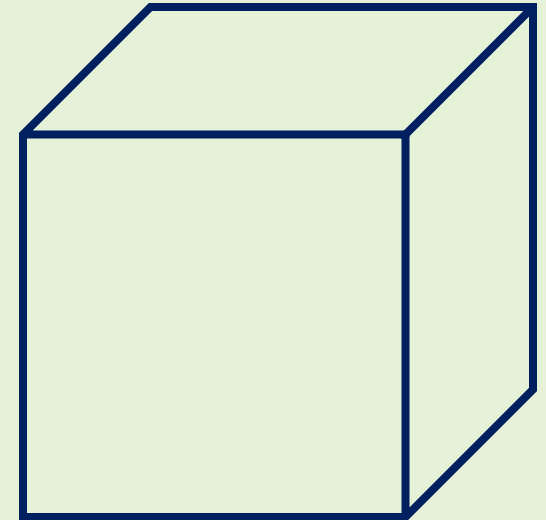
Value : 500 ETH

data : "" (비어있음)

서명 :



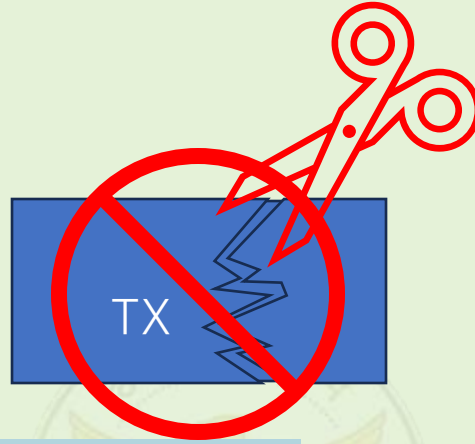
멤풀  
(Mempool)



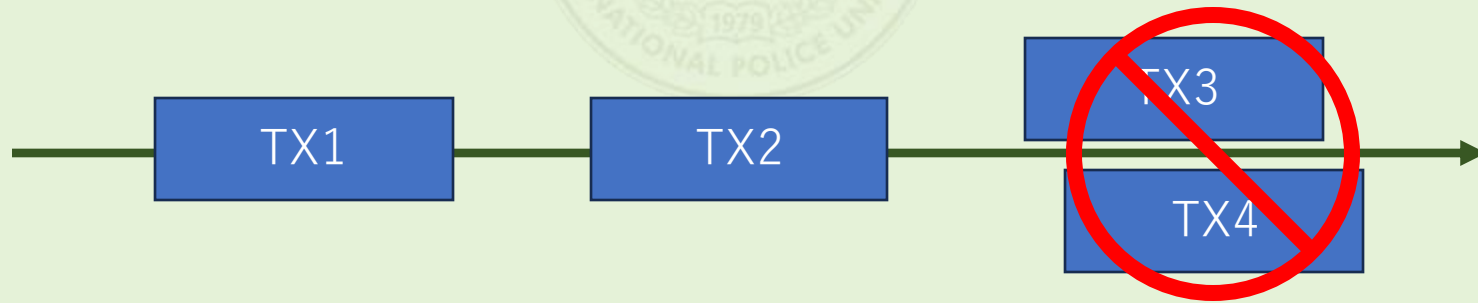
새로운 블록  
(New block)



## Ethereum Transaction(2)

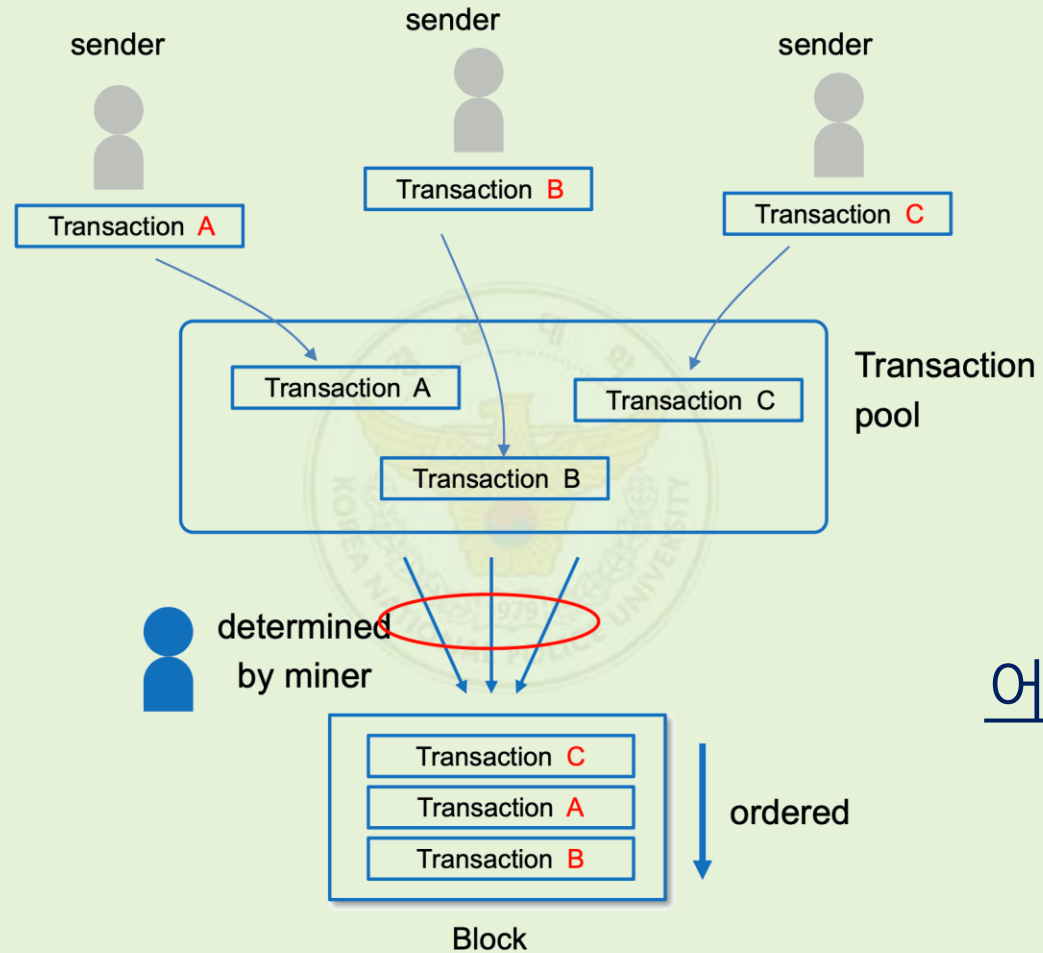


트랜잭션은 **원자적(atomic)**으로 실행되어 일부만 실행되는 경우는 없음



트랜잭션은 **순서(Order)**에 따라 실행되어 겹쳐 실행되지 않음

# Ethereum Transaction(3)



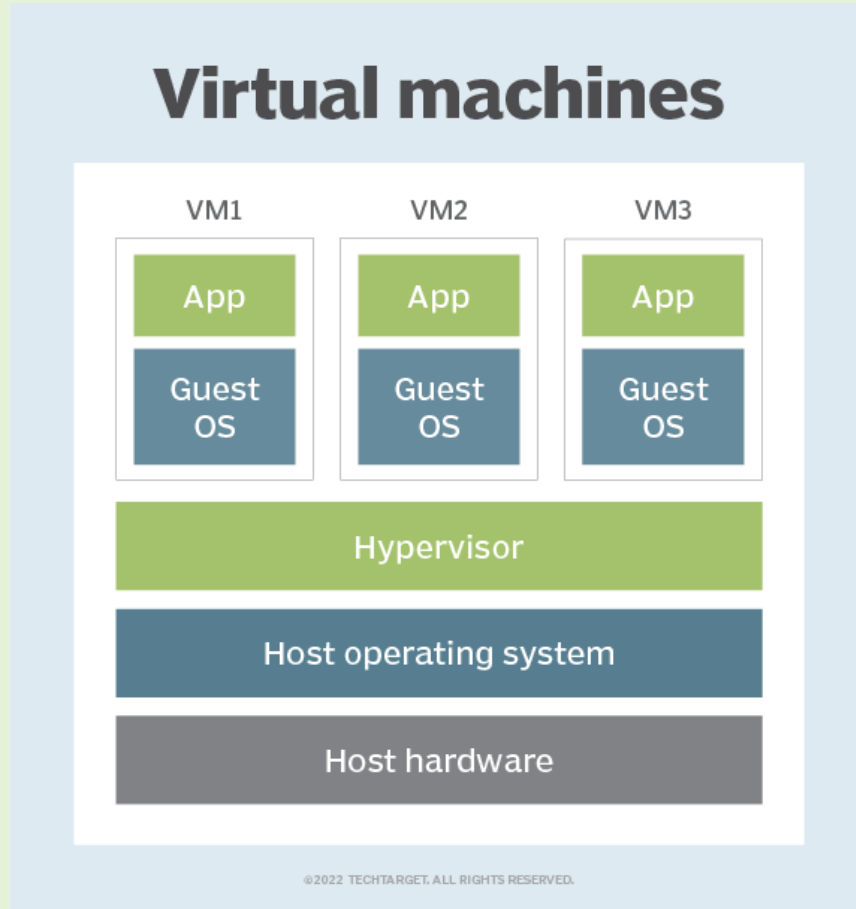
어떤 것이 가능할까요?

블록 트랜잭션의 순서는 **블록 생성자 맘대로** 결정함

### 3. EVM



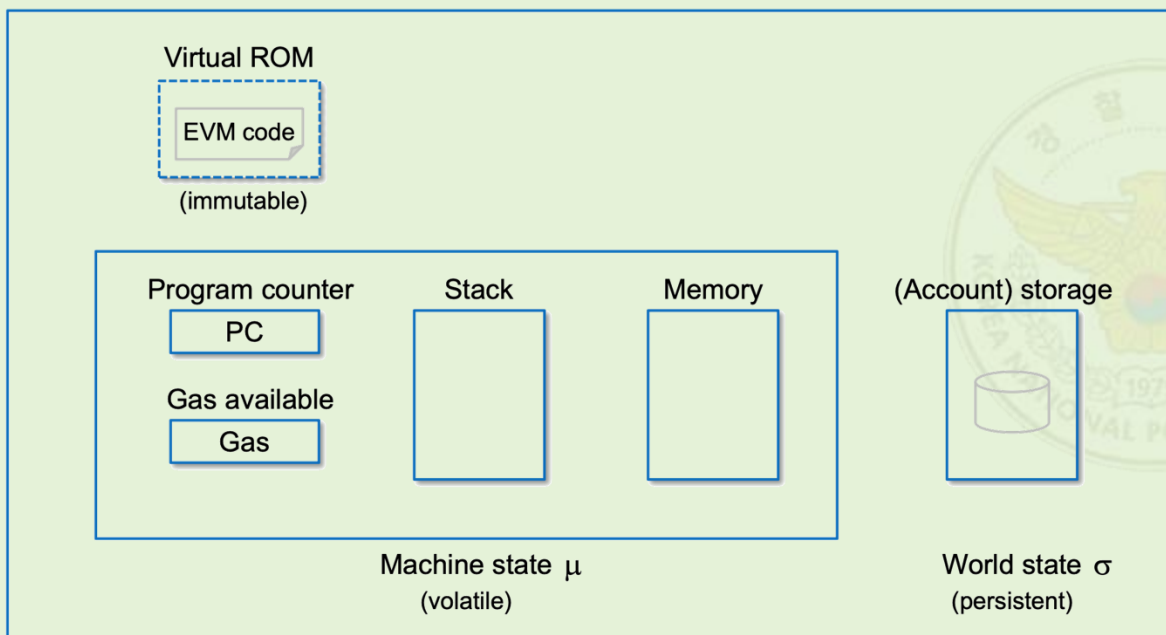
# 가상 머신(VM)



- 실제 컴퓨터 위에서 돌아가는 "가상의 컴퓨터"
- 하드웨어(CPU, 메모리, 저장공간 등)를 소프트웨어로 시뮬레이션
- 호스트 컴퓨터의 아키텍처나 운영체제와 관계없이 원하는 시스템을 구동
  - ex) Windows에서 Linux 실행
  - ex) x86에서 ARM 코드 실행
- 가상 머신 내에서의 활동이 호스트에 영향을 주지 않음

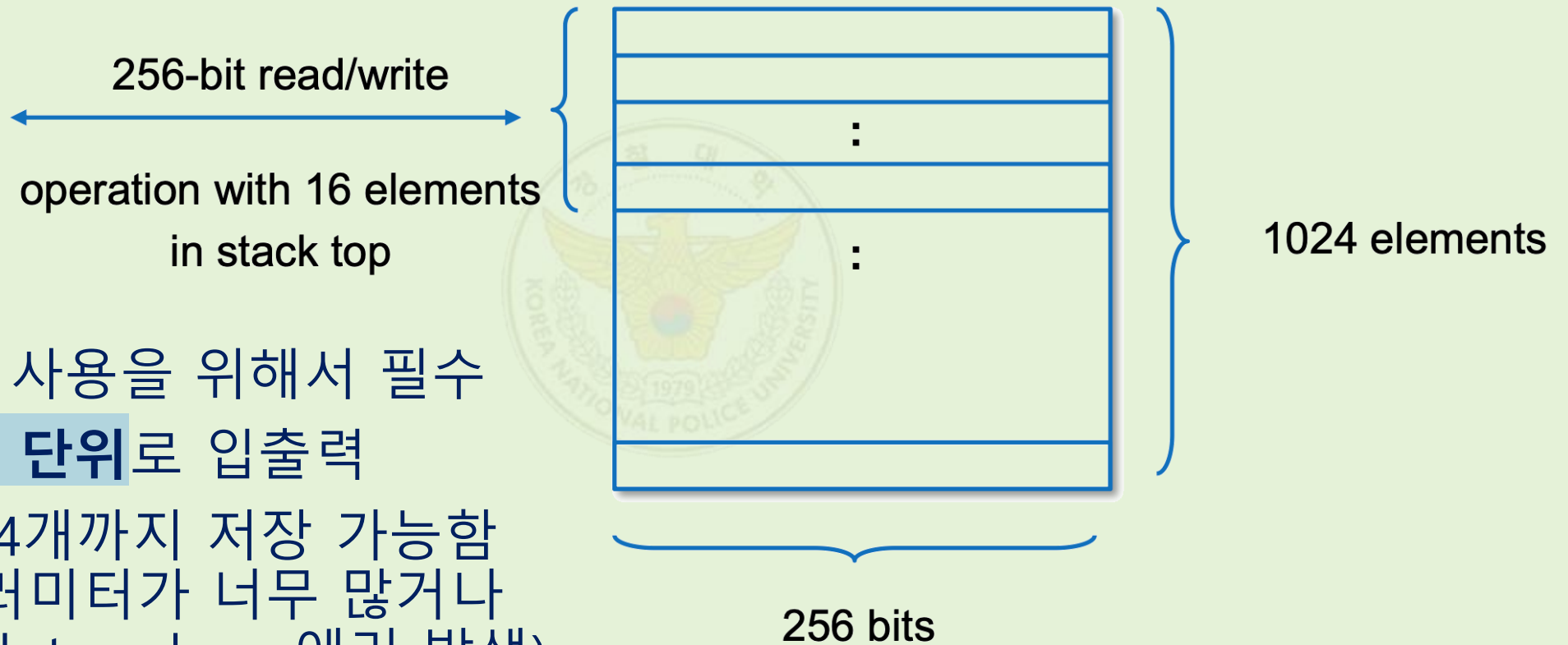


# EVM(1)



- 이더리움 노드가 **바이트코드**를 실행하기 위한 가상 머신  
=> 노드의 스펙과 관계없이 **동일한 실행 결과를 보장**
- 스택 머신(레지스터 X)
- 유저에게 저장 공간은 크게 2가지임
  - Memory(휘발성, 저렴)
  - Storage(비휘발성, 비쌈)

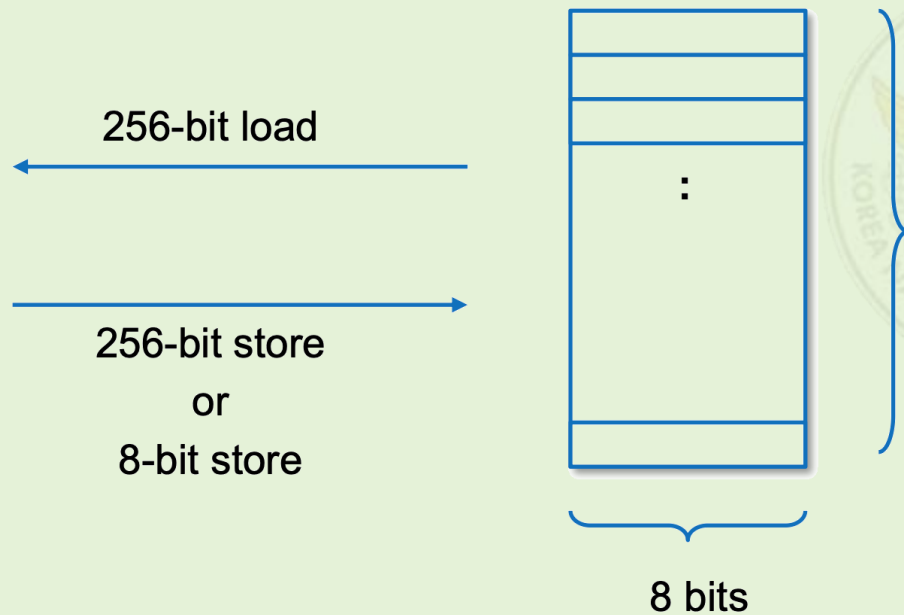
# EVM(1): 스택



- OPCODE 사용을 위해서 필수
- **32바이트 단위**로 입출력
- 최대 1024개까지 저장 가능함  
(함수 패러미터가 너무 많거나  
하면 stack too deep 에러 발생)

# EVM(1): 메모리

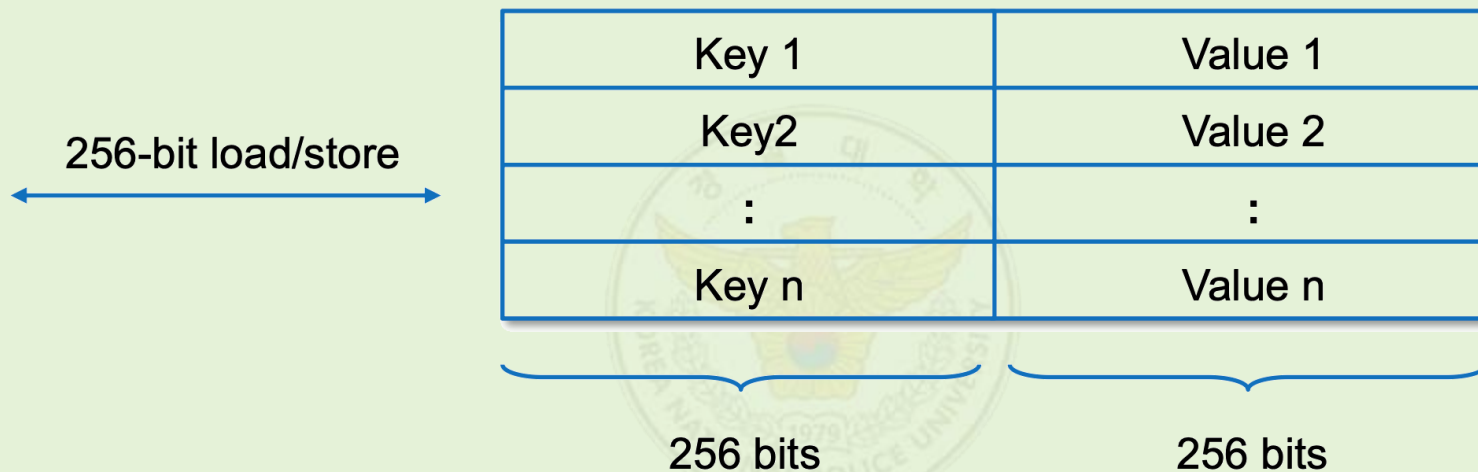
Memory



- 휘발성 영역(함수 종료 시 소멸)
- **1 or 32바이트 단위** 입출력 (offset)
- 기본값은 0(따라서 전통적인 memory corruption 류의 취약점은 발생하지 않음)

# EVM(1): 스토리지






























(Account) storage



- 비휘발성 영역(함수 종료 시 소멸)
- **32바이트 단위 입출력**(key-value)
- 주소마다 독립적이며, 갯수가 엄청 많음(key 공간이  $2^{256}$ )
- 스토리지 저장은 특히 비싸며(gas), 암호 등 비공개 자료 저장 지양



# EVM(2): OPCODEs

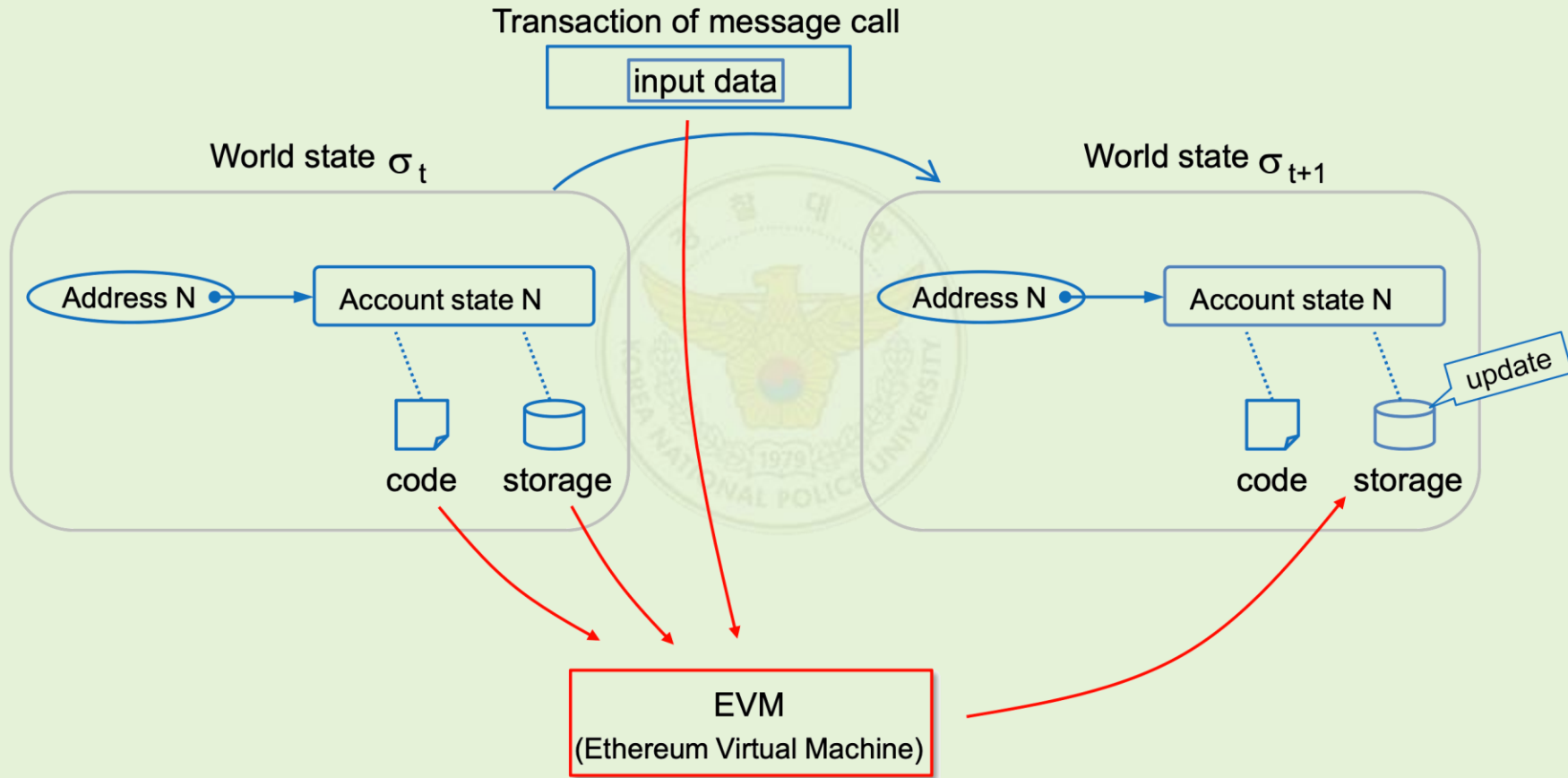
evm  codes	OpCodes	Precompiled Contracts	Contract Viewer	Playground	About the EVM	GitHub	 K	 Cancun 	
<h2>An Ethereum Virtual Machine Opcodes Interactive Reference</h2>									
Instructions <span>CANCUN</span>		Search by <span>Name</span>  <input type="text" value="Enter keyword..."/>  <span>Alt+K</span>							
OPCODE	NAME	MINIMUM GAS	STACK INPUT		STACK OUTPUT	DESCRIPTION	Expand 		
 00	STOP	0				Halts execution			
 01	ADD	3				Addition operation			
 02	MUL	5				Multiplication operation			
 03	SUB	3				Subtraction operation			
 04	DIV	5				Integer division operation			
 05	SDIV	5				Signed integer division operation (truncated)			

# EVM(2): OPCODEs

51	MLOAD	3	offset	value	Load word from memory
52	MSTORE	3	offset	value	Save word to memory
53	MSTORE8	3	offset	value	Save byte to memory
54	SLOAD	100	key	value	Load word from storage
55	SSTORE	100	key	value	Save word to storage

FA	STATICCALL	100	gas	address	argsOffset	argsSize	retOffset	retSize	success	Static message-call into an account
FD	REVERT	0	offset	size						Halt execution reverting state changes but returning data and remaining gas
FE	INVALID	NaN	150개 이외의 OPCODE 정의							Designated invalid instruction
FF	SELFDESTRUCT	5000	address							Halt execution and register account for later deletion or send all Ether to address (post-Cancun)

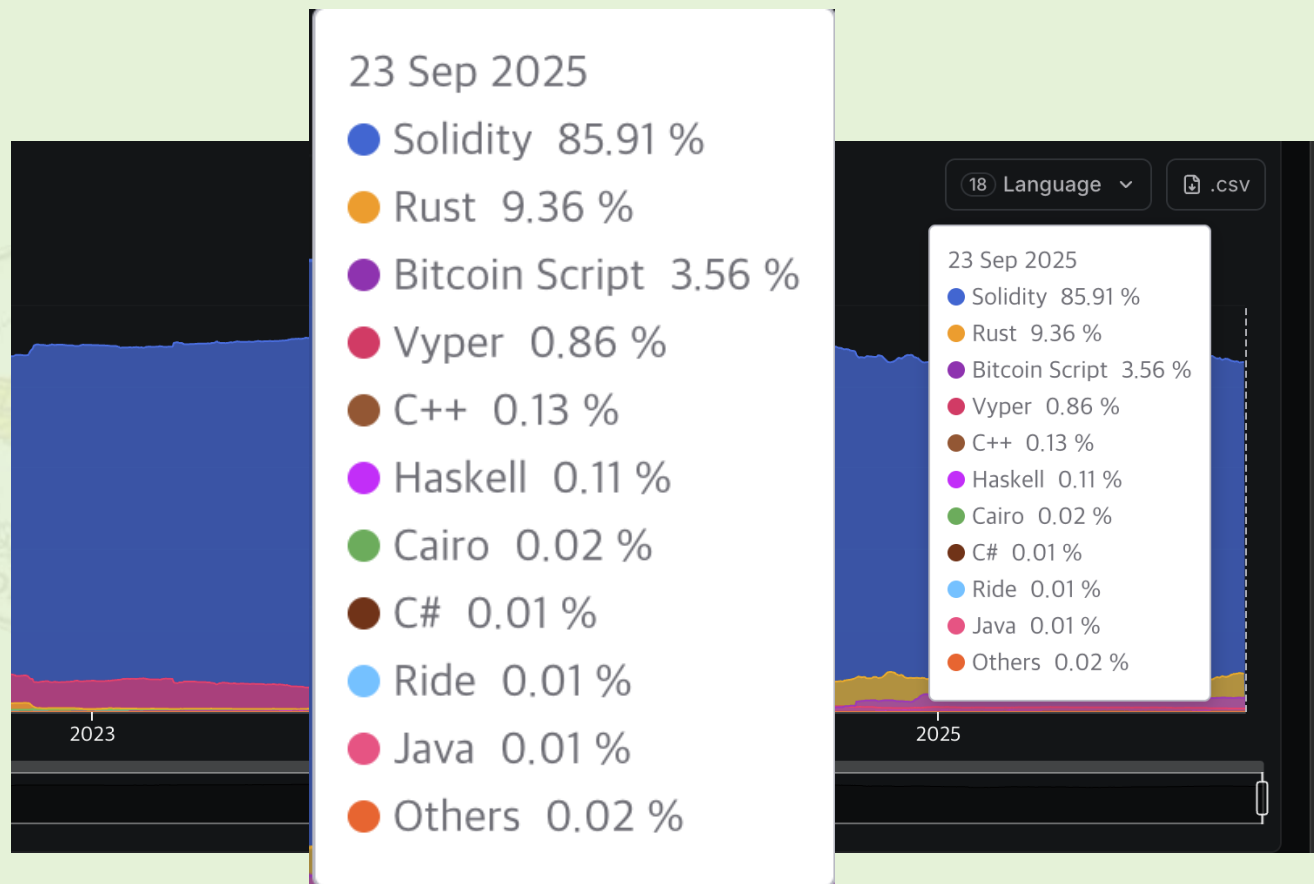
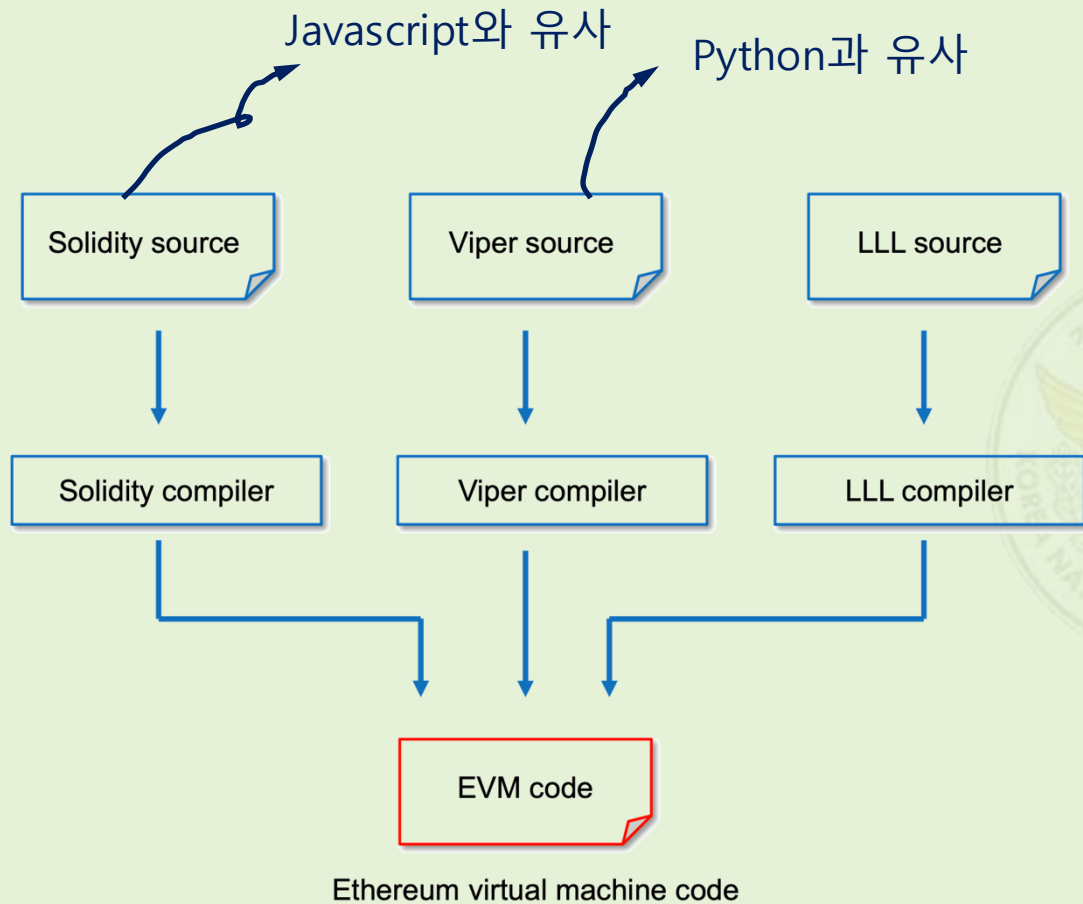
# EVM(3): EVM이 개입하는 트랜잭션



## EVM(4): Gas

- TX Fee = Gas Used(자연수) × Gas Price(ETH)
- Gas는 EVM에서 연산 비용을 측정하기 위해서 만들어낸 단위
  - 노드에 부하를 줄만한 동작일수록 비싸다.  
(무한루프 등의 트롤링 행위 차단)
  - 하드포크를 통해 가격을 올리기도 함([EIP-2929](#))
- Gas Price는 시장 상황에 따라 변동(네트워크 혼잡 시 증가)([EIP-1559](#))
- 트랜잭션 도중 Gas 부족하면 **전체가 실패처리(out of gas)**
  - 상태 변화는 롤백되지만, 실패 직전까지 사용한 Gas는 롤백 X
  - (포렌식 관점) Gas 양으로 공격 의도나 긴급도 판별

# EVM(5): EVM 코드 생성





## 4. Solidity와 개발 패턴



# What is Solidity?(1)

- Gavin Wood에 의해 제안되고 이후 Christian Reitwiessner 및 그의 팀에 의하여 개발됨
- JavaScript 및 C++와 유사한 구문/접근 제어자를 가지고 있음
- 정적 타입, 상속, 라이브러리 지원



# Solidity 코드 형태

```
// SPDX-License-Identifier: GPL-3.0
```

```
pragma solidity >=0.4.16 <0.9.0;
```

Solidity 버전 0.8.0 산술 언더/오버플로 자동 체크 (2020)

```
contract SimpleStorage {
```

```
    uint storedData;
```

함수 바깥에서 변수 선언  
=> 스토리지 변수(기본

컨트랙트 객체 정의

```
    function set(uint x) public {
```

```
        storedData = x;
```

값?)  
가시성(visibility) – 4 종류(public, private, internal, external)

```
    function get() public view returns (uint) {
```

```
        return storedData;
```

읽기 전용(스토리지 읽기 O, 쓰기 X)

```
    }
```

```
}
```

# Solidity Types(1): 정수 자료형

- 부호 없는 정수 uint (*더 많이 씀*)
  - uint8
  - uint16
  - uint24
  - ....
  - **uint256(=uint)**
- 부호 있는 정수 int
  - int8
  - int16
  - int24
  - ....
  - **int256(=int)**

32바이트 정수의 최대값(type(uint256).max)은 상당히 큰 숫자입니다.  
따라서 고정 소수점(6 ~ 27자리)을 사용해도 overflow 상황은 잘 일어나지 않습니다.

# Solidity Types(2): 주소 자료형

- 이더리움의 20바이트 주소를 저장하기 위한 자료형
  - **address** : 기본 주소
  - **payable** : 이더를 보낼 주소(오류 방지 목적)
- **address(this)** => 자신(컨트랙트)의 주소
- 변수의 초기화 시 [EIP-55](#)에 따라 체크섬 처리된 주소만 가능
- 주로 사용되는 메서드 2가지
  - **transfer** : 이더 전송(2300 가스 고정)
  - **call** : 이더 + 데이터 전송(가스 지정 가능)

```
address payable x = payable(0x123);  
address myAddress = address(this);
```

```
x.transfer(10);
```

```
address(nameReg).call{gas: 1000000}(abi.encodeWithSignature("register(string)", "MyName"))
```

# Solidity Types(3): 기타 자료형

- bool : true/false 저장
- bytes1 ~ bytes32 : 바이트 자료형(H(x) 를 담기 위한 32바이트를 많이 사용)
- string : 문자열 자료형(보통 토큰 이름, 티커 등 저장할 때 사용)



# Solidity Types(4): 매핑 자료형

- 가장 많이, 유용하게 사용되는 형태의 자료형

- Key => Value 형태로 자료 저장

- ex) address => uint256

```
mapping(address => uint) public balances;
```

- ex) address => address => uint256 (이중 매핑)

```
mapping (address => bool) public isBlackListed;
```

```
mapping (address => mapping (address => uint)) public allowed;
```

- H(abi.encode(key, slot)) 형태로 스토리지 번호를 얻어 입출력 하므로 O(1) 시간에 실행

```
function balanceOf(address _owner) public constant returns (uint balance) {  
    return balances[_owner];  
}
```

# Solidity Global Variables

- **tx.origin** : 트랜잭션을 서명한 주소(address 형)
- **msg.sender** : 해당 컨트랙트를 호출한 직전 주소 => EOA, CA 둘 다 가능
- **msg.value** : 해당 메시지와 함께 전달된 네이티브 토큰의 양
- **block.number** : 블록 번호
- **block.timestamp** : 블록 시간

# Solidity Expressions(1): if ~ else ~

- 해당 트랜잭션을 취소시키는 표현
- 모든 자산 이전이 취소되나, revert 직전까지 사용된 가스비는 환불되지  
않음

```
function tmp_thisFuncWontWork() {  
    revert("hehehe");  
}
```

# Solidity Expressions(2): if ~ else ~

- 여타 다른 언어와 동일한 사용법을 가짐

```
function tmp_if(uint256 age){  
    if (age < 18) {  
        (bool s, bytes memory r) = msg.sender.call(""){value: 10000};  
    }  
    else if (age > 60) {  
        (bool s, bytes memory r) = msg.sender.call(""){value: 50000};  
    }  
    else {  
        revert("go away");  
    }  
}
```

# Solidity Expressions(3): require

- `require(cond)` : 전달되는 조건이 만족하지 않으면 `revert()`  
= `if (!cond) { revert(); }` 을 축약한 버전으로 볼 수 있으며, 훨씬 더 자주  
쓰임

```
function tmp_revert_gimchi() {  
    require(block.number > 10_000_000, "it's not good yet");  
}
```

# Solidity Calls(1): call

- 가장 기본적인 저수준 호출 방식

```
function tmp{
    address targetContractAddress;
    address payable targetEOA;
    (bool s, bytes memory r) = targetContractAddress.call(abi.encodeWithSignature("targetFunc()")){value: 1};
    (s, r) = targetEOA.call(""){value: 1};
}
```

- EOA로의 ETH 전송시 calldata는 비워둘 수 있음
- 컨트랙트가 이더 수신 시 자동으로 receive() 함수를 실행하게 되며, 자금 수신 시 동작을 설계하기 위한 함수지만 추후 공격 목적으로 요긴하게 사용됨

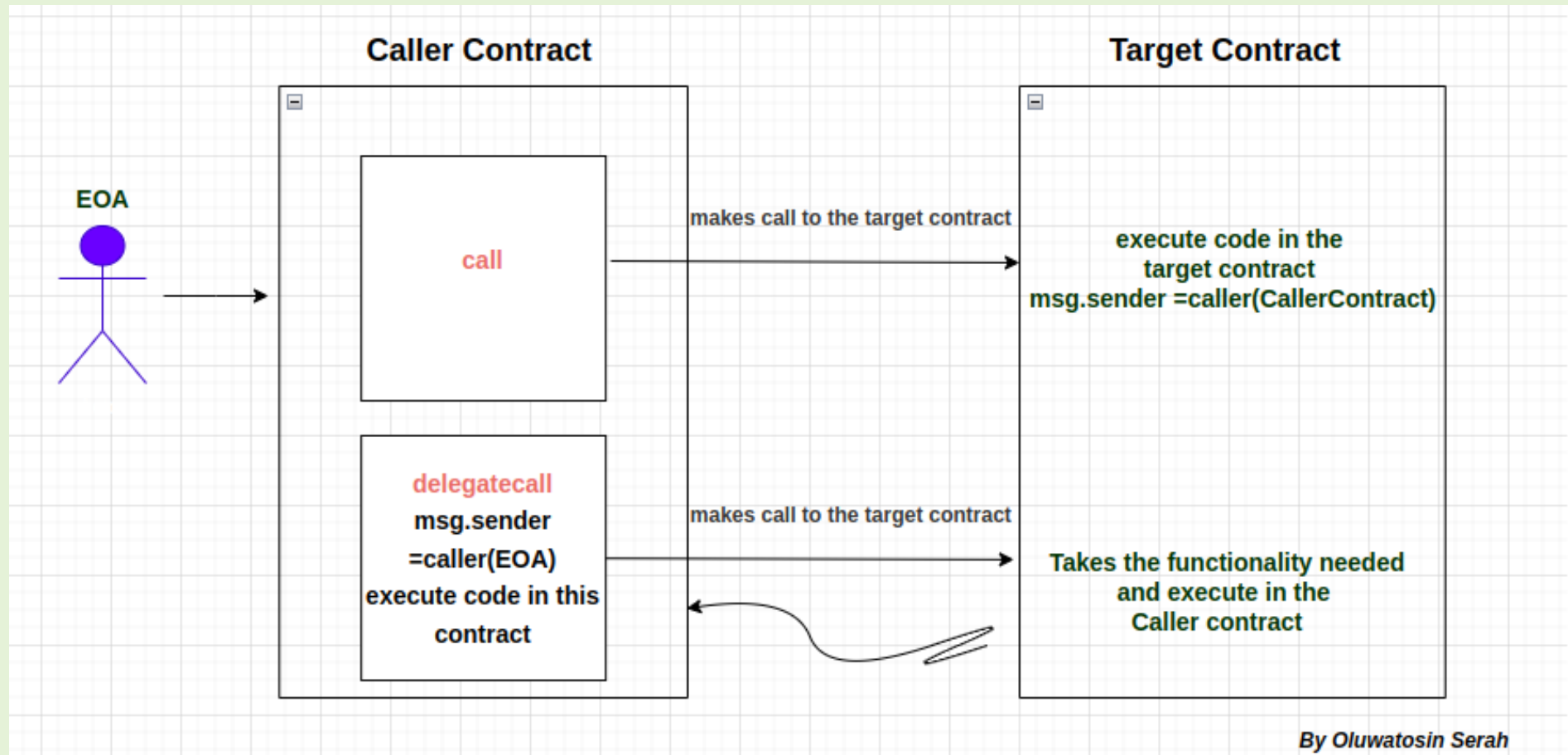
# Solidity Calls(2): staticCall

- call과 동일하나 world state를 변경할 수 없음
  - storage 수정 불가
  - ETH 전송 불가
  - 이벤트 발생 불가
  - child call 불가
- view/pure 태그가 달려있는 함수 호출 시 작동
- 풀 비율 조회, 환율(Oracle) 조회, 혹은 각종 계산함수에 사용됨

```
function tmp_staticCall() {  
    address oracleContractAddress;  
    (bool s, bytes memory r) = oracleContractAddress.staticCall(abi.encodeWithSignature("checkPrice()"));  
}
```



# Solidity Calls(3): delegateCall




# Solidity Calls(3): delegateCall


- 타겟 주소의 “코드만(EVM OPCODEs)” 가져와서 내 컨트랙트 환경에서 실행
- address(this) 기준으로 주소, address storage, 네이티브 토큰
- 이게 도대체 왜 있을까요?

```
function tmp_delegateCall() {  
    address targetContractAddress;  
    (bool s, bytes memory r) = targetContractAddress.delegateCall(abi.encodeWithSignature("delegateFunc()"));  
}
```

# Solidity 개발 패턴: Proxy

- 컨트랙트는 한번 배포하는데 gas가 상당히 많이 든다.

 F0 CREATE 32000 ?

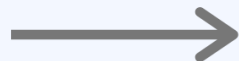
 F5 CREATE2 32000 ?

- ex) 입금/출금 기능을 지원하는 모임통장 컨트랙트를 배포한다고 생각해보자
  - 우리는 살면서 많은 모임이 있는데, 모임이 생길때마다 컨트랙트를 매번 배포할 것인가?
  - 기존 컨트랙트 A에 오류가 있어 수정본 A'를 배포하고 싶다.
  - A'를 새롭게 배포하면 A에서 자산을 옮겨와야 한다. 자산을 옮기는 함수 가지고 있어야 하고 이를 호출하는 트랜잭션도 수회 실행해야 한다.
- 어떻게 해결할 수 있을까요??

# Solidity 개발 패턴: Proxy

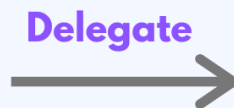


**Users**



**Proxy**

stores the  
contract state



Delegate



**Implementation**

stores the  
contract logic

**How to write upgradable smart-contracts?**

# Solidity 개발 패턴: Proxy

```
contract SimpleProxy {  
    address public implementation;  
  
    fallback() external payable {  
        (bool success, bytes memory data) = implementation.delegatecall(msg.data);  
        require(success, "Delegatecall failed");  
  
        assembly {  
            return(add(data, 0x20), mload(data))  
        }  
    }  
}
```

# Solidity 개발 패턴: Upgradable Proxy(1)

- 구현 컨트랙트(implementation)을 변경하고 싶어서 만들어진 기능
- 구현 컨트랙트(address)를 변경할 수 있는 함수를 두고, 해당 함수에 ACL(접근 제어 목록)을 두어 해당 기능을 구현할 수 있음

```
contract SimpleUpgradableProxy {
    address admin;
    address implementation;

    constructor() {
        admin = msg.sender;
    }

    function upgradeImpl(address _newImplementation) {
        require(msg.sender == admin, "hehe you can't");
        implementation = _newImplementation;
    }

    fallback() external payable {
        (bool success, bytes memory data) = implementation.delegatecall(msg.data);
        require(success, "Delegatecall failed");

        assembly {
            return(add(data, 0x20), mload(data))
        }
    }
}
```

# Solidity 개발 패턴: Upgradable Proxy(Q)

```
contract SimpleUpgradableProxy {
    address admin;
    address implementation;

    constructor() {
        admin = msg.sender;
    }

    function upgradeImpl(address _newImplementation) {
        require(msg.sender == admin, "hehe you can't");
        implementation = _newImplementation;
    }

    fallback() external payable {
        (bool success, bytes memory data) = implementation.delegatecall(msg.data);
        require(success, "Delegatecall failed");

        assembly {
            return(add(data, 0x20), mload(data))
        }
    }
}
```



# Solidity 개발 패턴: Upgradable Proxy(2)

- 구현 컨트랙트가 저장되는 슬롯이 0번, 1번 등 너무 낮은 번호대의 슬롯을 사용할 경우 충돌의 우려가 있음
- 이에 대응하여 구현 컨트랙트 주소 저장 슬롯을 고정하자는 아이디어가 제기됨([ERC-1967](#))
  - `bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1)` 을 구현 컨트랙트 전용 슬롯으로 사용
  - `0x360894a13ba1a3210667c828492db98dca3e2076cc3735a920a3ca505d382bbc`

```
contract SimpleUpgradableERC1967Proxy {
    bytes32 internal constant _IMPLEMENTATION_SLOT = bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1);

    function currentImpl() internal view returns (address) {
        return StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value;
    }

    function upgradeImpl(address newImplementation) private {
        require(Address.isContract(newImplementation), "ERC1967: new implementation is not a contract");
        StorageSlot.getAddressSlot(_IMPLEMENTATION_SLOT).value = newImplementation;
    }
}
```

# Solidity 개발 패턴: Upgradable Proxy(Q)

- CA1 : 0x4d5F47FA6A74757f35C14fD3a6Ef8E3C9BC514E8
  - CA2: 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2
  - CA3: 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48
- 
- 위 컨트랙트들은 각각 프록시 컨트랙트인가요?
  - 프록시 컨트랙트라면 어떤 종류의 프록시 표준을 사용하고 있나요?
    - 구현 컨트랙트 슬롯은 몇 번을 사용하고 있나요?
  - 업그레이드 가능한 프록시인가요?



수고 하셨습니다!!