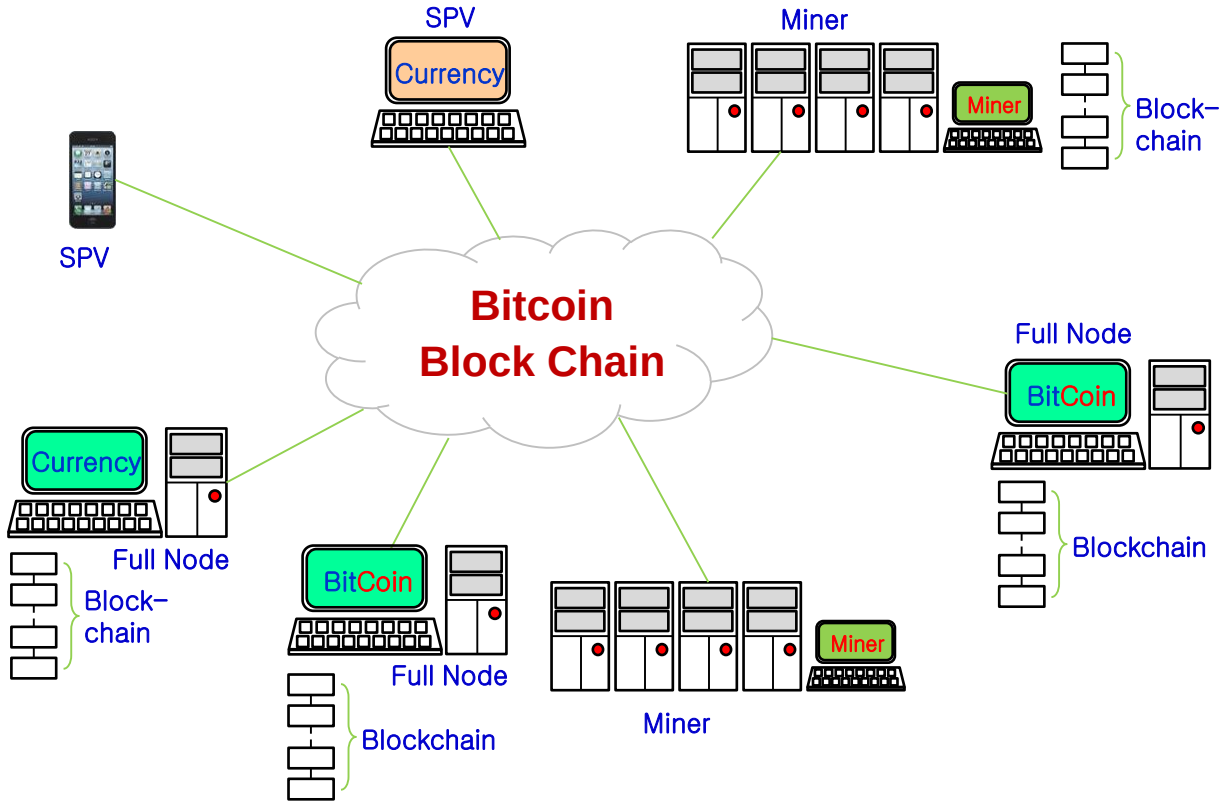
Deep dive into Bitcoin network and Blockchain



비트코인 네트워크와 블록체인 기술의 이해

(Python 실습 포함)

머 리 말

2008년 사토시 나카모토가 비트코인 (Bitcoin) 시스템을 발표한 이후, 수년 간 가상화폐 및 블록체인 기술에 대한 관심이 뜨거웠습니다. 블록체인 기술은 가상화폐 차원을 넘어 4차 산업혁명의 한 분야로 자리를 잡고 있습니다. 현재 금융, 제조, 의료 등 많은 산업 분야에서 블록체인 기술을 도입하기 위해 많은 노력을 기울이고 있습니다.

블록체인 기술은 화폐나 금융 거래 시스템에서 출발하였지만, 지금은 독립적인 기술로 분리되어 공공 블록체인, 사설 블록체인 등으로 발전, 진화하고 있습니다. 2015년 7월에는 블록체인 기반으로 스마트 컨트랙 (Smart Contract) 기능이 있는 암호화폐 (이더리움 : Ethereum)가 개발되었고, 2015년 이후 글로벌 금융 기업들은 공동 프로젝트로 R3 컨소시엄을 진행하고 있고, IBM등 거대 IT 기업들도 Hyperledger라는 프로젝트를 공동으로 진행하고 있습니다.

본 과정에서는 블록체인 기술의 기본인 비트코인 네트워크에 대해 자세히 다룹니다. 비트코인 네트워크는 화폐 거래에 한정되어 있지만, 이미 검증된 시스템이고, 기술이 많이 공개되어, 블록체인 기술을 이해하는데 매우 좋은 아이템입니다. 비트코인 네트워크를 이해하면 다른 블록체인 기술을 습득하는데 큰 도움이 될 수 있습니다.

비트코인에 관심을 갖는 그룹은 크게 3 분야로 나누어 볼 수 있습니다. 첫째, 블록체인 기술을 활용하여 신규 비즈니스를 창출하려는 그룹과 (개발자), 둘째, 가상화폐 채굴 (Mining)에 관심을 갖는 그룹 (채굴자), 그리고 셋째는, 비트코인의 거래, 투자 (Alt coin이나 ICO)에 관심이 있는 그룹 (투자자)이 있을 수 있습니다. 본 과정은 개발자, 채굴자, 투자자 모두를 대상으로 개발되었습니다. 개발자를 위해서는 Python으로 실제 기능을 일일이 확인해 보았고, 채굴자나 투자자를 위해서는 블록체인 데이터의 실제 현황들을 가급적 많이 분석해 보았습니다.

본 자료는 비트코인의 기본서인 Andreas Antonopoulos의 Mastering Bitcoin과 Imran Bashir의 Mastering Blockchain을 참고하였고, 세부 내용은 Bitcoin Developer Reference, Bitcoin Improvement Proposal (BIP) 문서와 개발자 포럼의 자료들을 주로 참조하였습니다.

1장은 개요 부분으로, 비트코인 네트워크의 전반적인 흐름에 대해 다루고, 2장은 블록체인의 핵심 기술인 암호학에 대해 다룹니다. 암호학은 다소 어려울 수 있으나, 블록체인 기술을 안전하게 만드는 가장 핵심적인 요소이므로 비교적 자세히 다루었습니다. 3장부터 5장 까지는 지갑 (Wallet), 거래 (Transaction), 채굴 (Mining) 부분으로 비트코인 네트워크의 주요 기능에 대해 자세히 다룹니다. 6장은 비트코인 P2P 네트워크의 프로토콜 부분으로, 각 노드들이 메시지를 주고 받으면서 네트워크가 자율적으로 운영되는 원리에 대해 다룹니다. 마지막으로 7장에서는 오픈 소스인 비트코인 코어를 설치해서 실제 Full 노드를 구축하고 블록체인 데이터를 탐색하는 방법을 다룹니다. 모든 내용은 이론에 그치지 않고 Python 언어를 이용하여 실제 구현하였습니다. 약 50여 개의 예제 프로그램을 통해 실제 기능을 확인해 볼 수 있도록 하였습니다.

2018년 5월 9일 조성현

목 차

1. 비트코인 네트워크 개요

2. 암호학 (Cryptography)

3. 지갑 (Wallet)

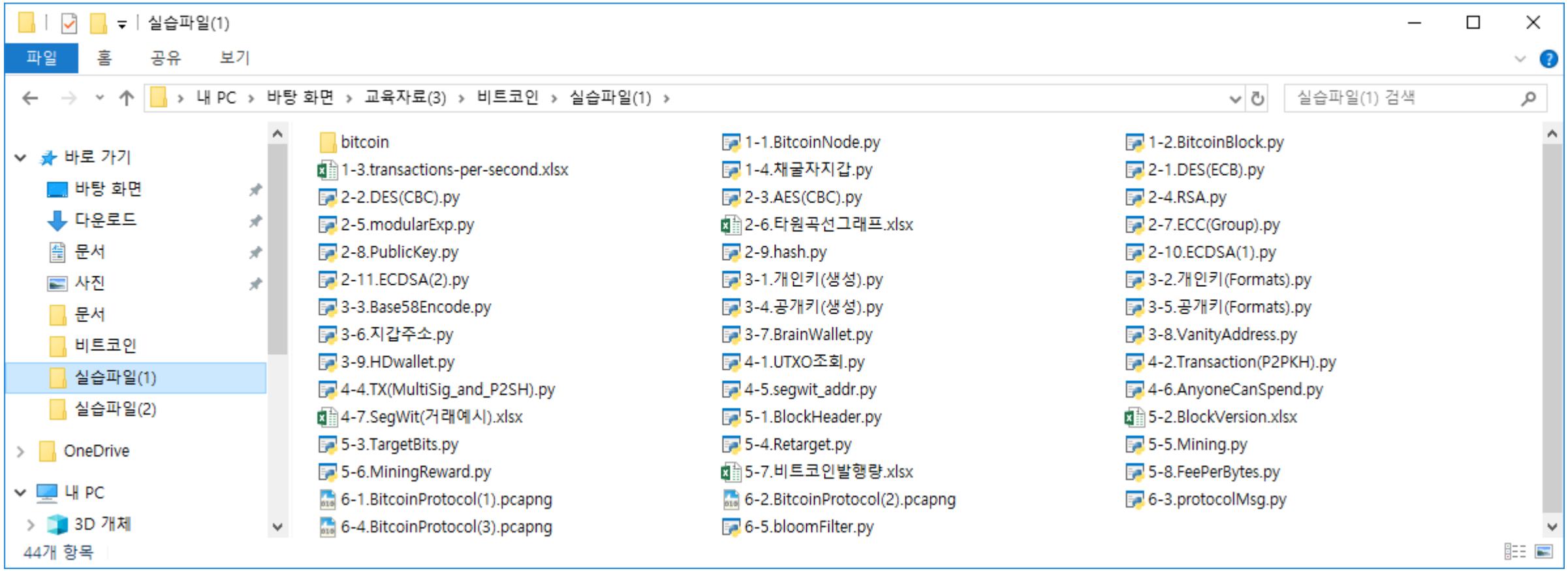
4. 거래 (Transaction)

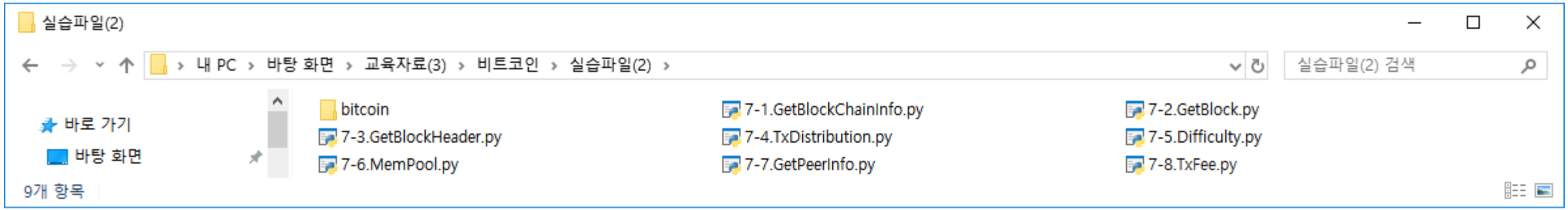
5. 채굴 (Mining)

6. 비트코인 P2P 프로토콜

7. Bitcoin Core 설치 및 블록체인 데이터 탐색

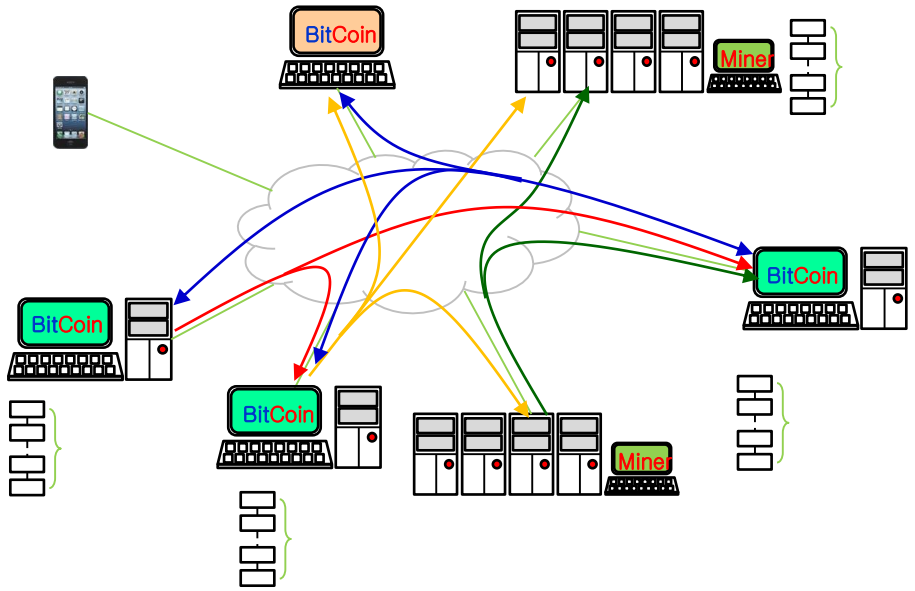
실 습 파 일





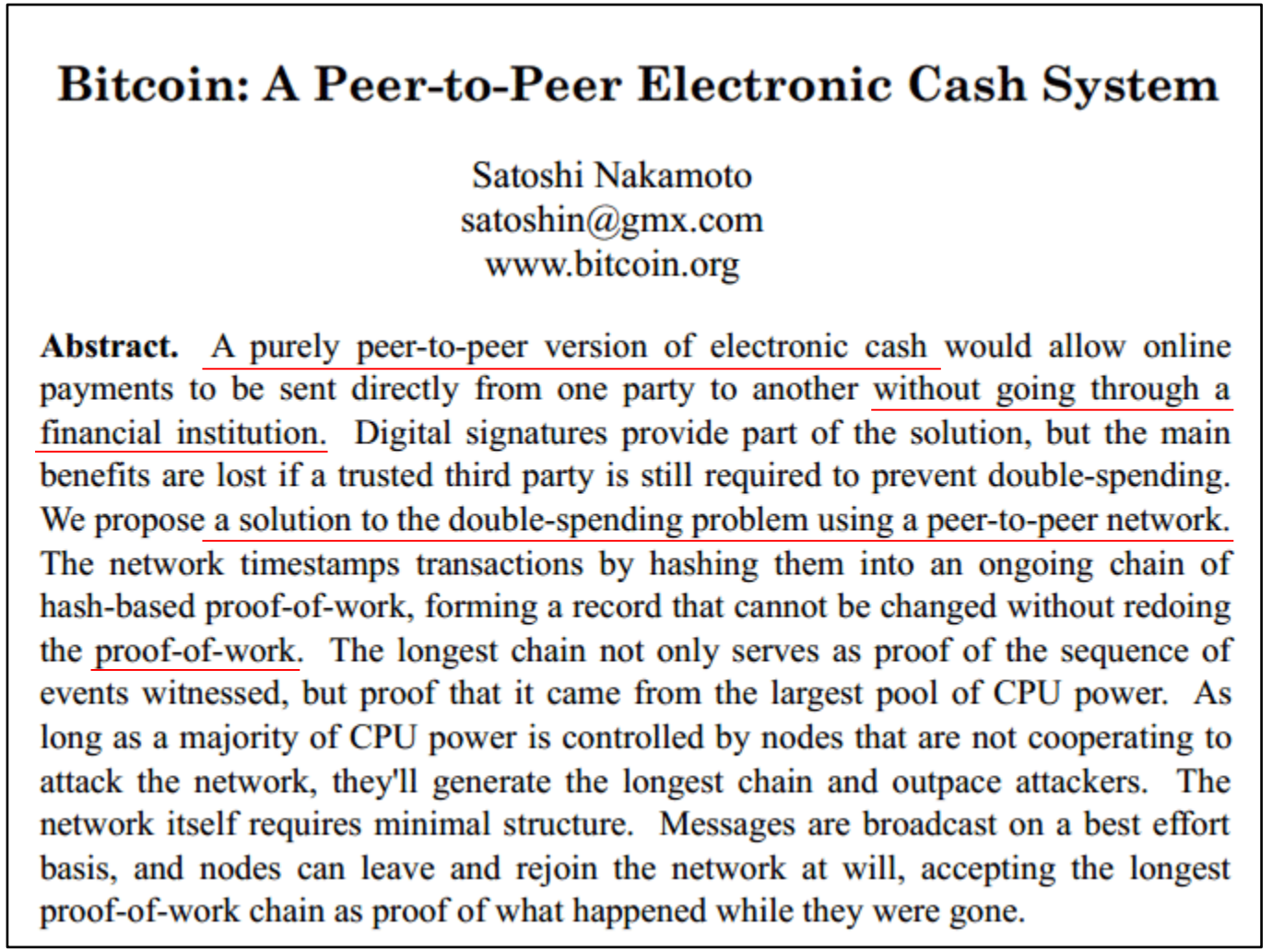
1. 비트코인 네트워크 개요

1-1. 비트코인의 탄생
1-2. 가상화폐의 역사
1-3. 블록체인 기술의 의의
1-4. Peer-to-Peer (P2P) 네트워크
1-5. 비트코인 네트워크 개요
1-6. 노드 유형 및 기능 (Full Node, SPV, Miner, 3rd-Parth API client)
1-7. 참여 노드 현황 관찰
1-8. 블록체인의 구조
1-9. 블록체인 데이터 확인 (블록 헤더, 거래 내역)
1-10. 블록 Size 제한과 비트코인의 확장성 (Scalability)
1-11. 비트코인 지갑
1-12. 거래 (Transaction) 생성
1-13. Transaction 전송 및 전파
1-14. 노드 별 거래 승인 절차
1-15. 채굴 (Mining)



비트코인의 탄생

- 비트코인은 2008년 10월 사토시 나카모토라는 정체 불명의 인물 (혹은 그룹)이 만든 (오른쪽에 보이는) 한 편의 논문에서 시작되었음.
- 사토시는 이 논문을 Cypherpunk들의 메일링 리스트로 발송하였고, Cypherpunk들의 큰 관심을 받았음.
- 2009년, 이 논문을 바탕으로 여러 개발자들이 모여 비트코인 네트워크를 실현하였음. 오픈 소스 소프트웨어로 배포되어 운용되기 시작하였고 현재까지 운용, 업그레이드되고 있음.
- 이 논문은 기존의 가상화폐 시스템의 문제점들을 보완한 것임.
- 이 논문의 주요 내용은 아래와 같음.
  - 중앙 통제 시스템이 없는 peer-to-peer 네트워크임. (Decentralization)
  - 이중 지불 문제를 방지함. (Double spending problem)
  - 거래 원장은 네트워크에 공개되어 각 노드들이 공유함. (블록체인 : Blockchain)
  - 네트워크의 각 노드들이 거래 원장을 검증할 수 있는 규칙.
  - 블록 체인에 저장된 거래 원장을 신뢰할 수 있는 합의 시스템. (작업 증명 : Proof of Work)
- 비트코인은 기존 가상화폐 시스템의 문제점을 해결한 최초의 가상화폐 시스템임. 기존 가상화폐 시스템은 많은 문제점 때문에 실현되지 못하고 개념적인 수준이었음.



Bitcoin: A Peer-to-Peer Electronic Cash System

Satoshi Nakamoto
satoshin@gmx.com
www.bitcoin.org

Abstract. A purely peer-to-peer version of electronic cash would allow online payments to be sent directly from one party to another without going through a financial institution. Digital signatures provide part of the solution, but the main benefits are lost if a trusted third party is still required to prevent double-spending. We propose a solution to the double-spending problem using a peer-to-peer network. The network timestamps transactions by hashing them into an ongoing chain of hash-based proof-of-work, forming a record that cannot be changed without redoing the proof-of-work. The longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power. As long as a majority of CPU power is controlled by nodes that are not cooperating to attack the network, they'll generate the longest chain and outpace attackers. The network itself requires minimal structure. Messages are broadcast on a best effort basis, and nodes can leave and rejoin the network at will, accepting the longest proof-of-work chain as proof of what happened while they were gone.

가상화폐의 역사

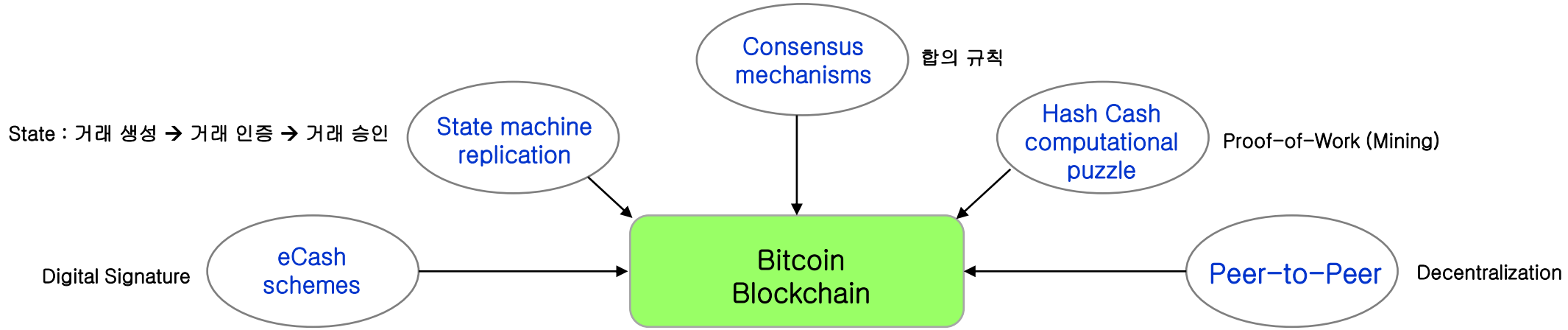
- 가상화폐는 기본적으로 암호학 (Cryptography)을 기반으로 만들어짐 (가상화폐 = 암호화폐). 비트코인도 암호화폐의 일종임.
- 가상화폐는 1980년대부터 Cypherpunk들을 중심으로 연구되었음. Cypherpunk는 암호 (Cipher)와 사이버펑크 (Cyberpunk)의 합성어로, 다국적 기업이나 정부가 정보를 독점하는 것에 저항하고, 통제 시스템으로부터 개인의 프라이버시를 보호하기 위해, 암호 기술을 바탕으로 개인간 네트워크를 개발하는 활동가를 의미함.
- 1984년 David Chaum은 디지털 화폐 (e-cash)를 소개하였고, 1990년 전자화폐를 발행, 관리하는 DigiCash를 설립하였음.
- e-cash는 암호학을 기반으로한 blind signatures와 이중 지불을 방지하기 위한 secret sharing 기능을 가지고 있었음.
- 1992년 Eric Hughes, Timothy C. May and John Gilmore 등에 의해 Cypherpunk 모임이 시작되었고, 이들에 의해 다양한 암호 거래 기술이 개발되었음.
- 1992년 Cynthia Dwork와 Moni Naor는 스팸 메일과 DoS 공격을 방지하기 위해 computational puzzle을 푸는 방식을 제안하였고, 1997년 Adam Back은 이 아이디어를 기반으로 HashCash를 제안하였음. 이 아이디어는 비트코인에서 Mining을 통한 Proof-of-Work (PoW)의 기초가 되었음.
- 1998년 Wei Dai는 B-Money 시스템을 제안하였음. B-Money는 HashCash의 computational puzzle을 푸는 과정에서 가상화폐가 생성되는 방식이며, Peer-to-Peer 네트워크를 통한 가상화폐 시스템의 기초가 되었음.
- 2005년 Nick Szabo는 BitGold를 제안하였음. BitGold도 B-Money와 유사하게 Mining을 통해 가상 화폐가 발행되도록 하였음.
- 2005년 Hal Finney는 HashCash, B-Money등의 아이디어를 모아 가상화폐 시스템을 제안하였음.
- 기존의 가상화폐 (혹은 암호화폐, 전자화폐, 디지털 화폐)들은 많은 문제점들 때문에 실용화되지 못하였음.
- 큰 문제점으로는 Peer-to-peer 네트워크 상의 각 노드들이 보유한 데이터의 일치성 (Consistency)을 보증하기 어려웠고, 신뢰할 3rd-party 서버가 필요하다는 점 등을 지적할 수 있음 (Centralized Network).

가상화폐의 역사

- 기존의 가상화폐의 주요한 문제점들은 아래와 같음.

  - 내가 받은 가상화폐를 신뢰할 수 있는가 ?. 위조된 것이 아닌 진본인가 ?
  - 내가 받은 가상화폐가 이중으로 지불된 것은 아닌가 ? (어떤 사람의 잔고가 100원인데, 나에게 100원을 보내고, 또 다른 사람에게도 100원을 송금하는 것을 이중지불 (Double spending) 문제라 함. 이중으로 지불되지 않았다는 사실을 보증할 방법이 필요함.)
  - 내가 받은 가상화폐의 소유권을 다름 사람들도 인정하는가 ? 등.

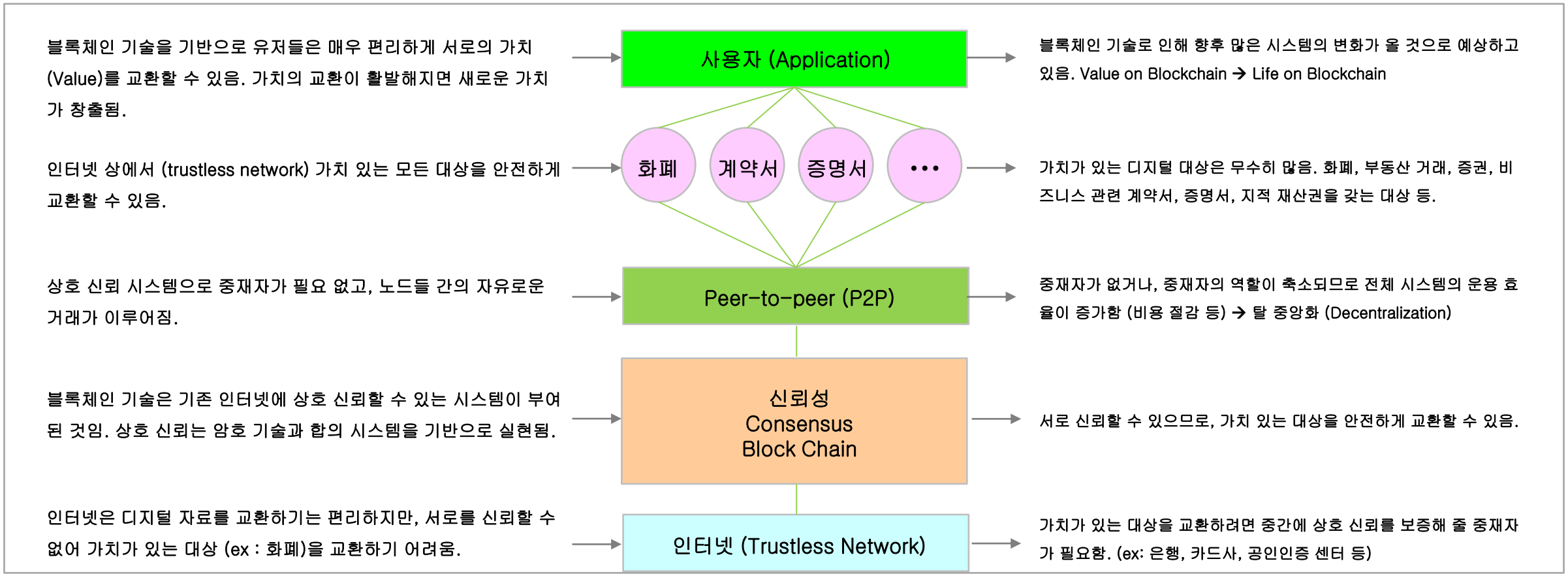
- 기존의 가상화폐는 위의 문제점들에 대한 보증 (검증) 문제로 중앙화 (Centralized) 개념을 사용하였고, 중앙 시스템은 공격에 대한 근본적인 취약점이 있음.
- 사토시 나카모토는 기존 가상화폐들을 참조하여 완전히 탈 중앙화 (Decentralized)된 개념을 사용하면서도 대부분의 문제점을 해결하였음.
- 사토시 나카모도의 비트코인은 최초로 실용화된 가상화폐 시스템이며, 2009년 이후 현재까지 큰 문제없이 운용되고 있음.



출처 : Imran Bashir, 2017, Mastering Blockchain (P. 16)

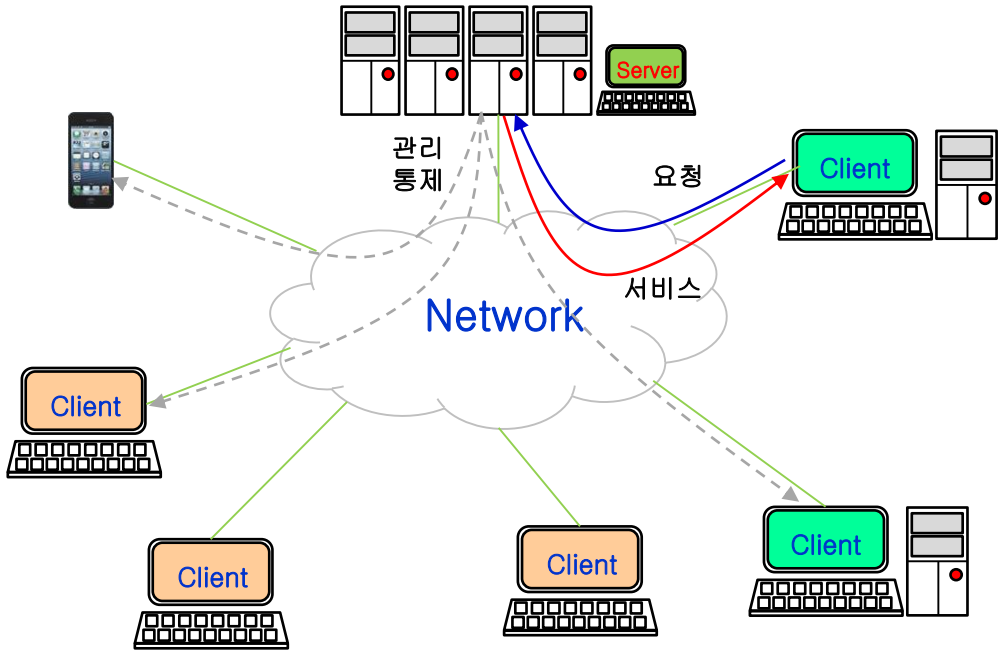
블록체인 기술의 의의

- 블록체인 기술은 기존의, 디지털 자료를 교환하기는 편리하나 서로를 신뢰할 수 없는 비 신뢰 기반의 (Trustless Network), 인터넷에 상호 신뢰 기능을 부여하여 사용자들이 화폐나 비즈니스 계약 등 가치 (Value)를 지닌 수 많은 디지털 대상을 안전하고 편리하게 교환할 수 있는 시스템으로 요약할 수 있음.
- 블록체인 기술로 인해 새로운 가치가 창출되고, 향후 우리 사회와 산업 시스템에 많은 변화가 올 것으로 전망하고 있음. → 4차 산업혁명의 요소 중 하나로 평가됨.
- 사토시 나카모토의 비트코인 시스템은 이러한 디지털 가치 교환 시스템을 가능하게 한 최초의 기술로 평가받고 있음.

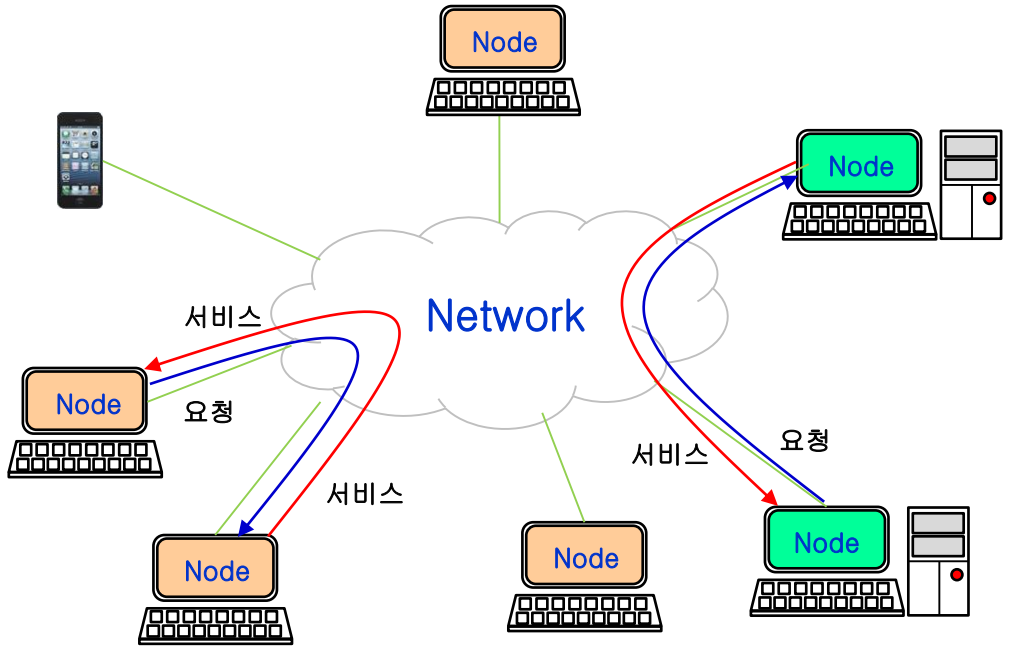


Peer-to-peer (P2P) 네트워크

- 중앙 집중 시스템은 중앙 시스템이 관리, 통제, 유지, 서비스를 제공하는 구조임. 많은 장점이 존재하지만 중앙 시스템 자체를 잘 관리하는 것이 관건임.
- 중앙 집중 시스템은 계좌 관리, 거래 승인, 취소, 거래 내역 원장 관리 등이 용이하고, 네트워크의 기능 보완 (업그레이드) 등에 유리하지만, 집중 시스템 자체를 안전하게 유지하는 것이 어려움. 예를 들어 집중 시스템이 손상되거나 공격 당하면 전체 네트워크의 기능이 마비될 수 있음.
- 중앙 집중 시스템은 네트워크의 영속성 (Business Continuity Plan : BCP, Disaster Recovery : DR)을 위해 유지 비용이 많이 들어감.
- P2P 네트워크는 중앙 시스템이 존재하지 않고, 각 노드들이 대등한 위치에서 서로의 요청에 대해 서로 서비스하면서 자율적으로 유지되는 네트워크임.
- P2P 네트워크는 중앙 시스템이 없기 때문에 관리, 통제, 유지가 어려우나, 중앙 시스템 자체에 대한 문제점 (BCP, DR 등)이 없음.
- P2P 네트워크는 거래 원장 관리 (분산 원장), 거래 승인 (채굴 : Mining) 등의 업무를 각 노드들이 분담하면서 전체 노드의 합의 (Consensus) 시스템으로 네트워크가 안전하게 유지됨. → 합의 규칙이 필요함. → 비트코인 네트워크의 핵심 사항.



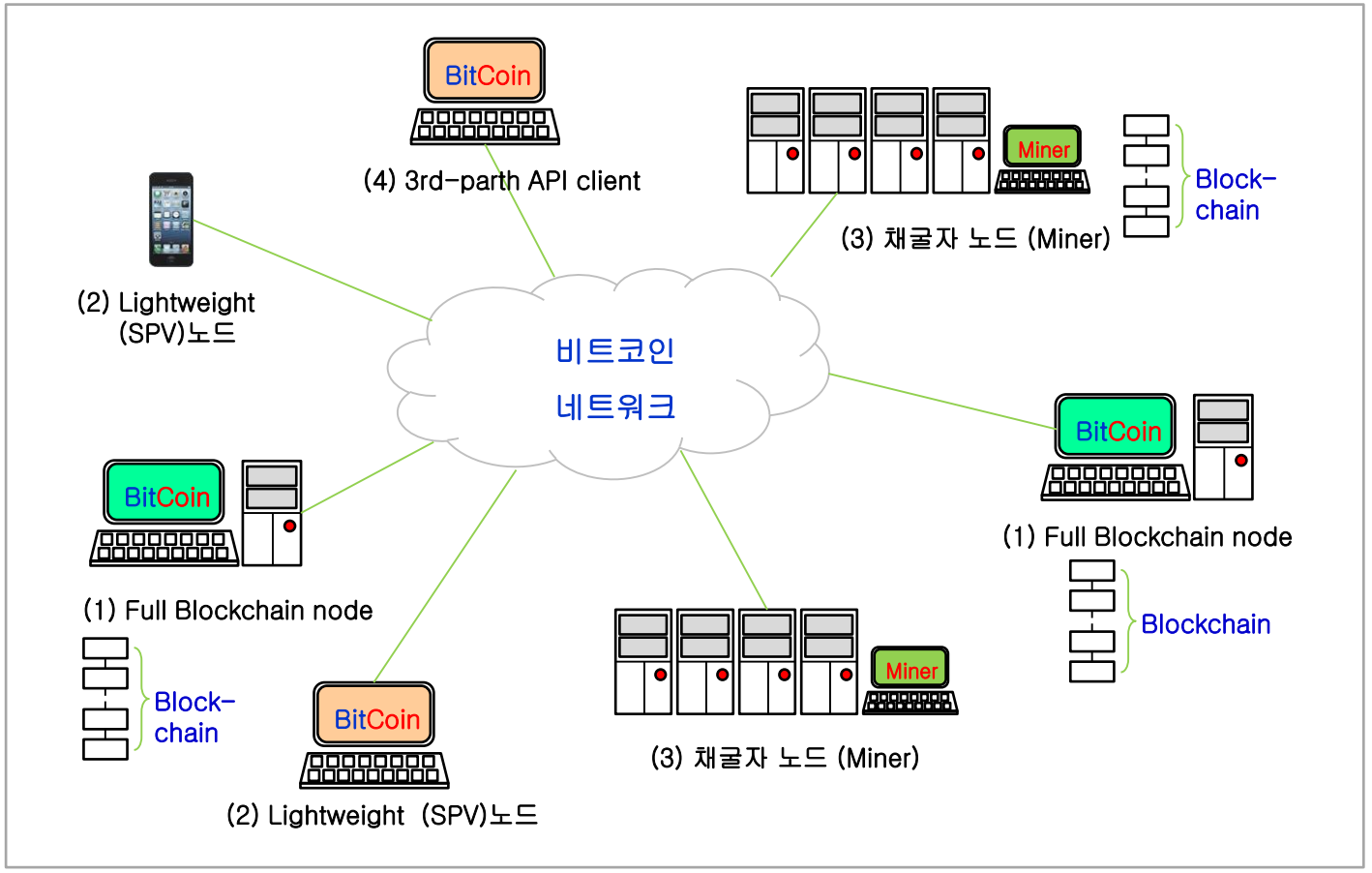
[중앙 집중 시스템 (서버-클라이언트)]



[P2P 네트워크]

비트코인 네트워크 개요

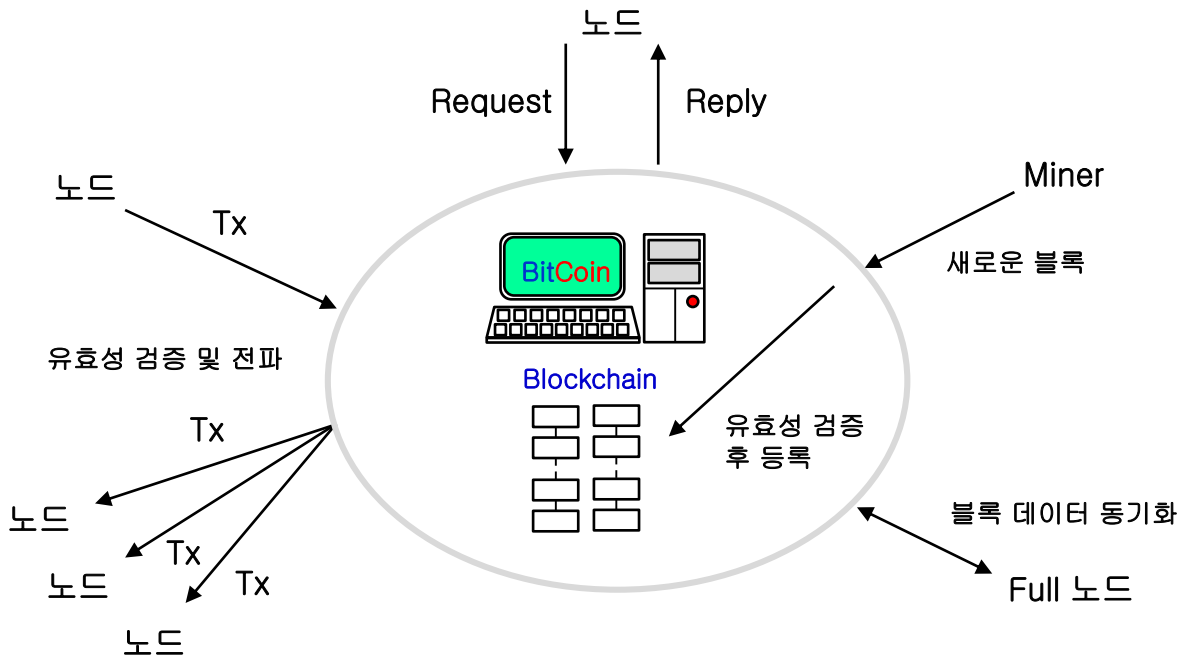
- 비트코인은 P2P (peer-to-peer) 네트워크로 중앙 통제와 같은 특별한 주체없이 동등한 노드들로 구성되고, 각 노드들의 합의 절차를 거쳐 자발적으로 운영됨.
- 비트코인은 소프트웨어이므로 스마트폰, 노트북, 데스크탑 등 일반 컴퓨터로 누구나 네트워크에 참여할 수 있음.
- 비트코인 네트워크는 (1) Full Blockchain 노드, (2) Lightweight (혹은 SPV : Simplified Payment Verification) 노드, (3) 채굴자 (Miner) 노드로 구성됨.
- 기타로는 (4) 3rd-party API client 형태가 있을 수 있음. 이 형태는 안전성 (Security) 문제 때문에 비트코인 네트워크의 기본 요소라고 할 수는 없음.
- Full Block chain 노드와 Miner 노드는 주로 데스크탑을 이용하고, Lightweight (SPV) 노드는 스마트 기기, 그리고 3rd-party API client는 Web 형태로 사용되기도 함.
- 거래 생성, 거래 원장 확인 요청은 어느 노드나 서로에게 할 수 있으며, 거래 확인, 거래 승인, 거래 원장 (블록체인) 저장 관리는 각 기능별 노드가 수행함.
- Full node가 관리하는 블록체인에는 모든 거래 내역이 저장되어 있음. → 공공 거래 장부 (Public ledger)
- 공공 거래 장부는 많은 Full node들이 가지고 있으므로 분산 저장됨. 일부가 잘못되어도 자체 복원 능력을 가지고 있음. → Distributed public ledger
- 모든 노드들은 거래를 생성 할 수 있으며 (비트코인을 주고 받는 행위), Miner 노드는 거래 들을 승인할 수 있음. Miner는 거래를 승인하는 대가로 보상을 얻을 수 있음.



노드의 유형 및 기능

1. Full Blockchain 노드 (Full 노드)

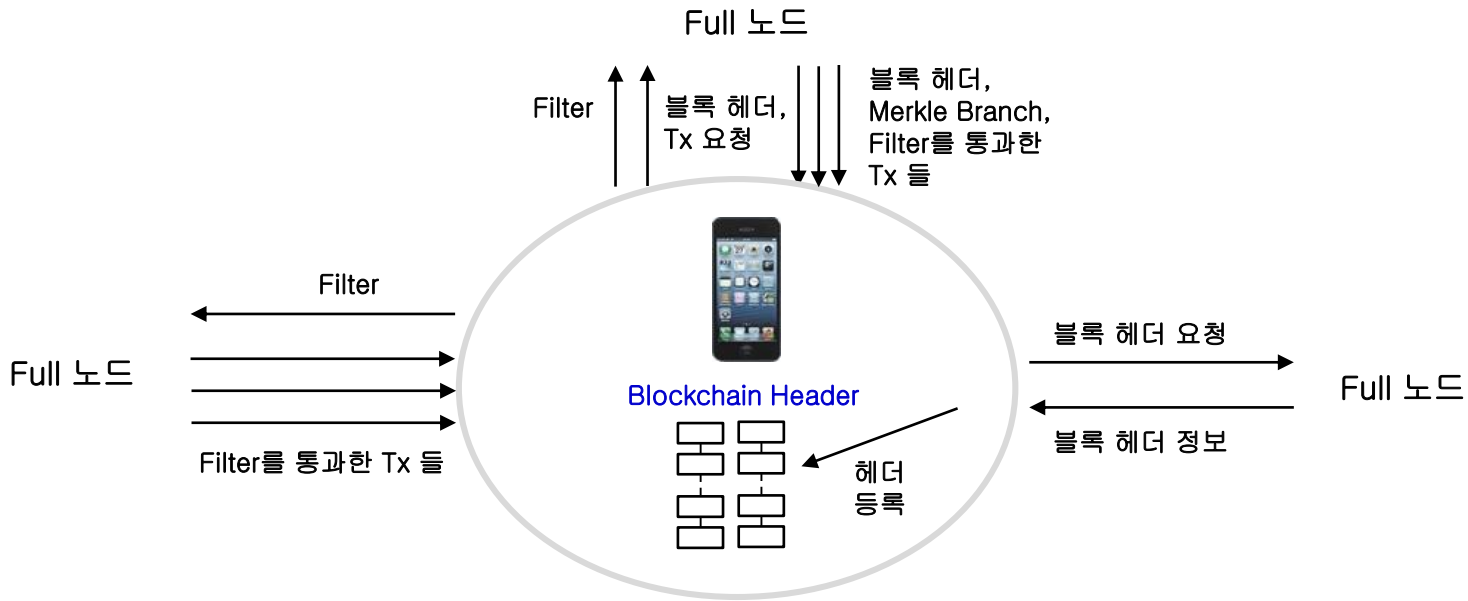
- 블록체인 데이터 전체를 가지고 있으며 Miner로부터 거래 목록이 담긴 새로운 블록을 받으면, 해당 블록의 유효성을 검증하고 기존 블록체인에 연결 시킴.
- 블록체인에는 거래 내역들이 저장되어 있고 (거래 원장), 지갑이 사용 가능한 잔고를 파악하기 위한 DB를 가지고 있음 (UTXO set in LevelDB)
- 아직 블록에 등록되지 않은 거래들의 유효성을 검증하고 (UTXO, Signature), Memory pool에 저장한 후 다른 노드로 전파 (Relay)함
- 다른 노드에서 블록체인 데이터의 내용을 요청할 때 데이터를 제공함 (블록 데이터, 블록 헤더, Merkle branch, 개별 거래 내역 (Tx) 등).
- 다른 Full 노드들과 블록체인 데이터들 동기화 시켜 데이터의 신뢰성 (Consistency)을 확보하는 기능을 수행함 (블록체인의 연결 상태, 블록 높이 등).



노드의 유형 및 기능

2. Lightweight (SPV) 노드

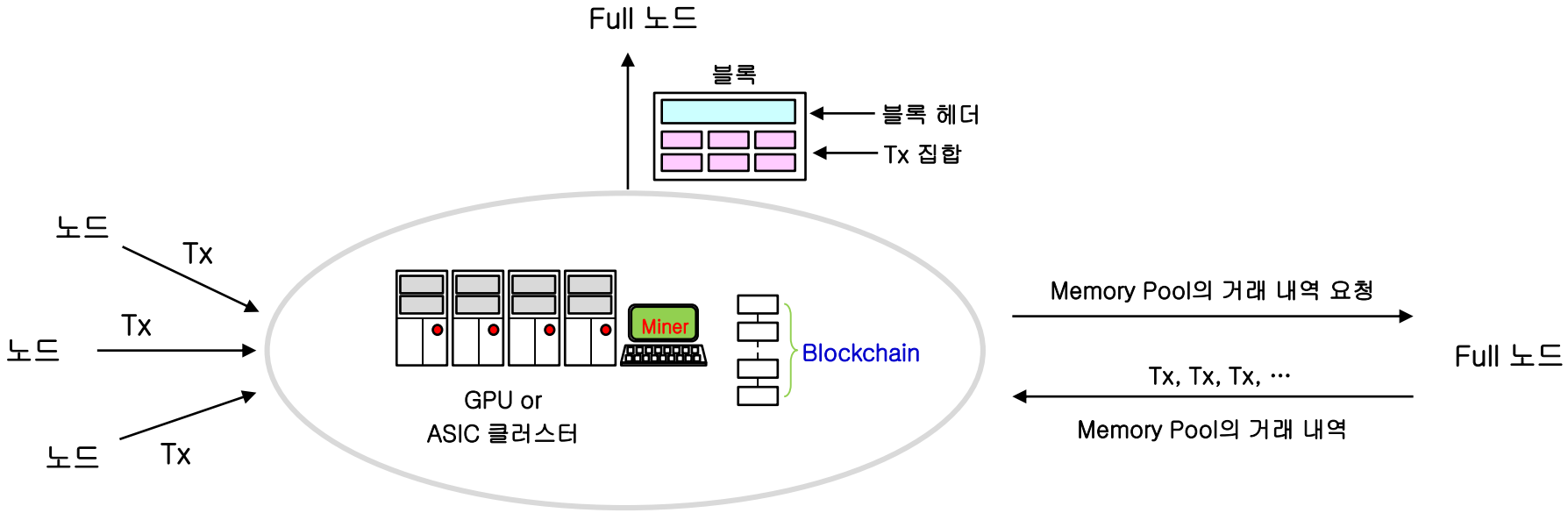
- 블록체인 데이터 전체를 가지고 있지 않고, 각 블록의 헤더 정보만 가지고 있음. 블록의 헤더는 크기가 작으므로 (헤더 당 80 byte) 저장 공간을 절약할 수 있음.
- 지갑 Application들은 주로 SPV 노드이며, 자신의 지갑과 관련된 거래 (입금, 출금)의 유효성만 검증함 (Merkle branch 인증). 다른 거래 내역은 검증할 수 없음.
- 자신의 거래에 대한 검증은 Full 노드의 도움을 받아야함. Full 노드로부터 자신과 관련된 거래 정보를 (Bloom Filter) 받을 수 있고, 자신의 거래 내역이 특정 블록에 속해 있는지 (Block height) 검증할 수 있음. 또한, 자신의 거래가 포함된 블록 이후로 몇 개의 블록이 쌓여 있는지 (Block depth) 등을 검증할 수 있음.
- 이 노드는 블록체인 데이터 전체를 관리하기 곤란한 소형 기기 (스마트폰 등)에 적합함.
- Full 노드에 의존해야하기 때문에 특정 Full 노드가 아닌 여러 개의 Full 노드를 랜덤하게 선택할 필요가 있음. Full 노드 방식에 비해 안전성 (Security)은 떨어짐.



노드의 유형 및 기능

3. 채굴자 (Miner) 노드

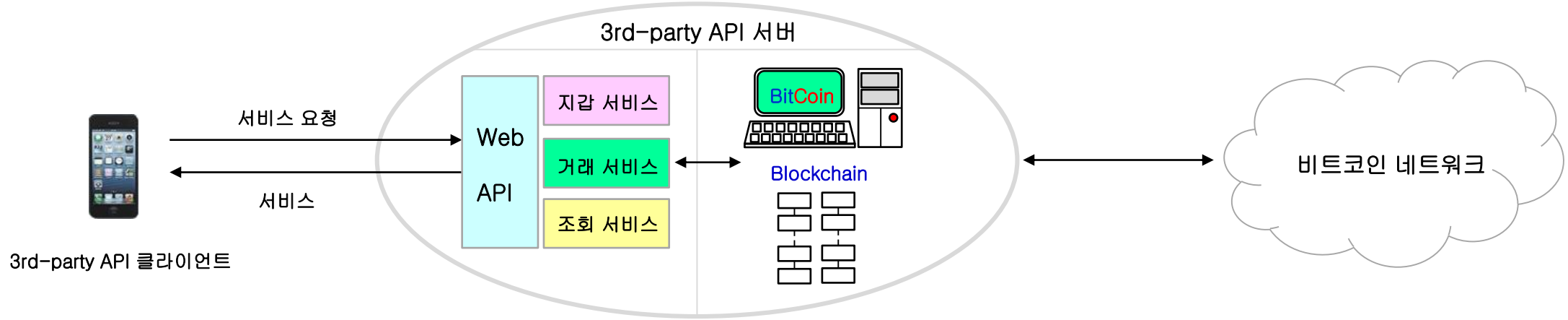
- 각 노드들이 보내는 거래 내역 (Tx)들의 유효성을 검증하고 유효한 Tx들을 모아서 새로운 블록을 생성함. (거래 승인 과정, Proof of Work : PoW).
- 생성한 블록을 Full 노드로 전송함. Full 노드는 이 블록의 유효성을 검증하고 기존 블록체인에 추가함.
- Miner들은 서로 경쟁적으로 블록을 생성하고 경쟁에서 이긴 Miner는 (유효한 블록을 빨리 만들면) 그 대가로 보상을 받음 (Coinbase Transaction).
- Miner는 경쟁에서 이기기 위해 GPU, ASIC 같은 장비를 이용하여 최대한 빨리 유효한 블록을 만들기 위해 노력함. Miner들의 경쟁을 통해 비트코인 네트워크의 신뢰성이 높아진다고 할 수 있음. 아무나 블록을 쉽게 생성할 수 없으므로 블록에 있는 거래 내역을 위,변조할 수 없고, 신뢰할 수 있음.
- Miner 노드는 직접 거래 내역을 수집하기도 하고, Full 노드에게 아직 블록에 등록되지 않고 Memory pool에 남아 있는 거래 내역을 요청할 수도 있음 (mempool).
- 유효한 거래 내역이라도 수수료 (Fee)가 포함되어 있지 않으면 Miner들이 승인하지 않을 수도 있음. 거래 수수료 (Fee)는 Miner에게 지급되는 몫이기 때문임.



노드의 유형 및 기능

4. 3rd-party API client

- 노드 자신이 지갑을 소유한 경우 : 이 노드는 블록체인 데이터나 헤더, 어느 정보도 가지고 있지 않고 오직 자신의 거래를 위한 지갑의 키 정보만 가지고 있음. 자신의 거래가 인증되어 블록에 등록되었는지, 자신의 잔고 (Balance)가 얼마인지, 자신이 사용할 수 있는 거래의 포인터 (UTXO)는 무엇인지 등은 3rd-party에서 운영하는 서버의 API를 이용함 (ex : Blockchain.info).
- 지갑 자체를 3rd-party 서버에 위탁하는 경우 : 3rd-party 서버에 가입하면 지갑을 발급해 주고, 서버를 통해 거래가 이루어짐. 자신의 지갑이 서버에 저장되어 있고, 자신은 Web을 통해 잔고를 확인하거나, 거래를 할 수 있음.
- 두 경우 모두 비트코인 네트워크의 구성 요소로 볼 수 없고, 3rd-party 서버를 통해 간접적으로 접속된 형태임.
- 지갑을 소유한 형태는 3rd-party 서버에 전적으로 의존하므로 서버를 신뢰해야 하는 문제와, 자신의 IP가 노출되어 공격 대상이 될 수 있는 위험이 존재함.
- 지갑을 서버에 위탁하는 형태는 자신의 개인키가 무방비로 노출되어 있으므로 더욱 위험함. 서버가 지갑의 잔고를 모두 사용할 수도 있음.
- 두 경우 모두 3rd-party가 개입되어 있는 면에서 비트코인 네트워크가 지향하는 바는 아니나 (탈 중앙화 개념에 어긋남), 편리성 때문에 사용되기도 함 (ex : 자신이 직접 지갑, 거래용 프로그램을 간단히 제작해서 사용하는 경우 3rd-party API 기능을 이용하면 편리함).



참여 노드 현황 관찰

- 2018년 4월 10일 현재 비트코인 네트워크에 상시 접속되어 있는 노드들의 수는 11,031개 임. 상시 접속되어 있는 노드들은 Full node들임.
- 아래 결과는 미국이 24.24%로 가장 높고 독일이 18.26%, 그리고 중국이 9.61% 순으로 높음. 주로 미국, 유럽, 중국에 많이 분포해 있음. 우리 나라 경인 지역에도 약 12개가 운영되고 있음.
- 아래 "11031 NODES"를 클릭하면 각 노드들의 버전, 보유하고 있는 블록 개수, 지역 등의 상세 정보를 조회할 수 있음.
- 아래 사이트는 Full Node를 운영하면서 비트코인 네트워크 상의 노드들이 보내오는 프로토콜 (Version, Tx, Inv 등)을 수집하여 분석하였을 것으로 추정됨. 참고로 Version 메시지에는 해당 노드가 사용중인 프로토콜 버전, 보유한 블록의 개수의 정보가 들어 있고, 해당 노드의 IP 주소를 추적하면 지역 정보 등을 알 수 있음. 또한, Ping-Pong 메시지를 주고 받으면서 해당 노드가 살아있는지 실시간으로 확인할 수 있음.

GLOBAL BITCOIN NODES DISTRIBUTION

Reachable nodes as of Tue Apr 10 2018 11:13:49 GMT+0900 (대한민국 표준시).

11031 NODES

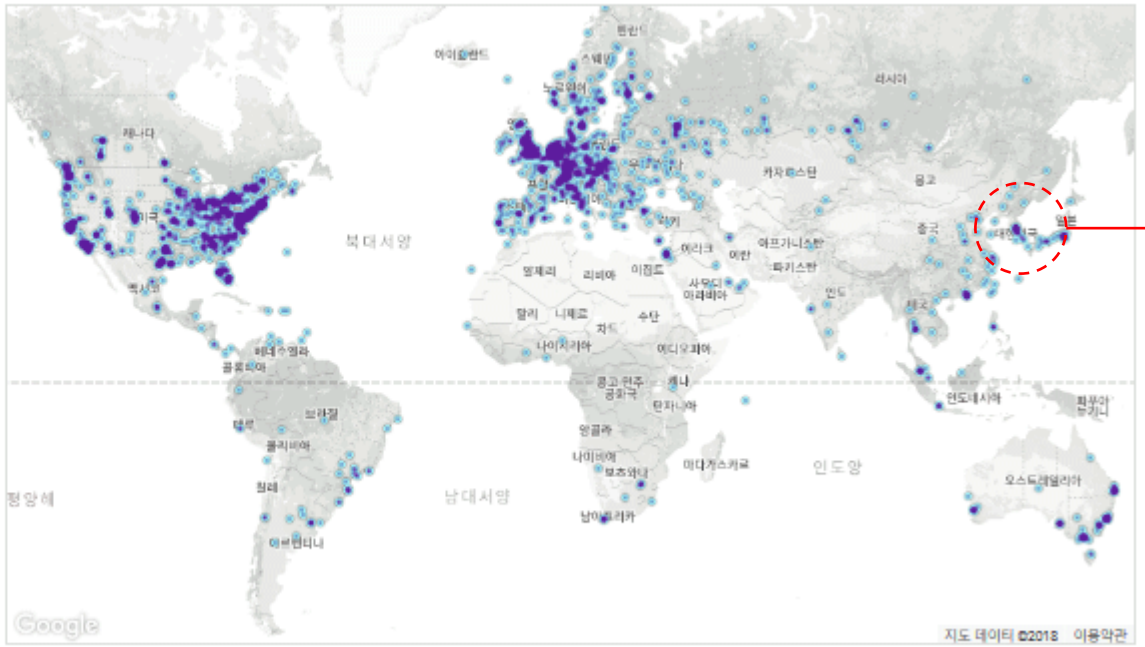
24-hour charts »

Top 10 countries with their respective number of reachable nodes are as follow.

RANK	COUNTRY	NODES
1	United States	2674 (24.24%)
2	Germany	2016 (18.28%)
3	China	1060 (9.61%)
4	France	662 (6.00%)
5	Netherlands	491 (4.45%)
6	Canada	411 (3.73%)
7	United Kingdom	391 (3.54%)
8	Russian Federation	362 (3.28%)
9	n/a	312 (2.83%)
10	Singapore	240 (2.18%)

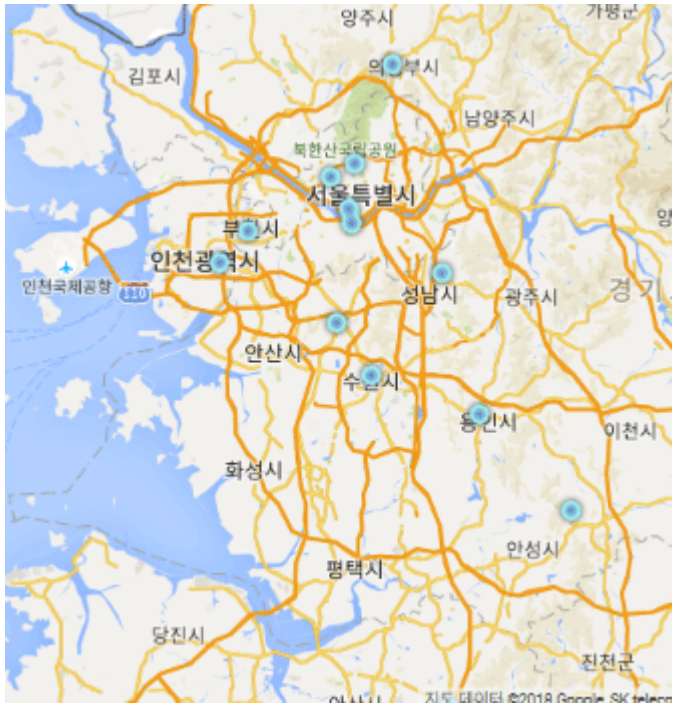
More (107) »

* 자료 출처 : https://bitnodes.earn.com/



Map shows concentration of reachable Bitcoin nodes found in countries around the world.

LIVE MAP



참여 노드 현황 관찰

* 자료 출처 : https://bitnodes.earn.com/nodes/

NETWORK SNAPSHOT

Snapshot of reachable nodes as of Tue Apr 10 2018 10:55:09 GMT+0900 (대한민국 표준시).

11031 Nodes
9283 IPv4
1483 IPv6
303 .onion
+59/-43 Churn
517464 Height

USER AGENTS | COUNTRIES | NETWORKS

Top 6 user agents with their respective number of reachable nodes.

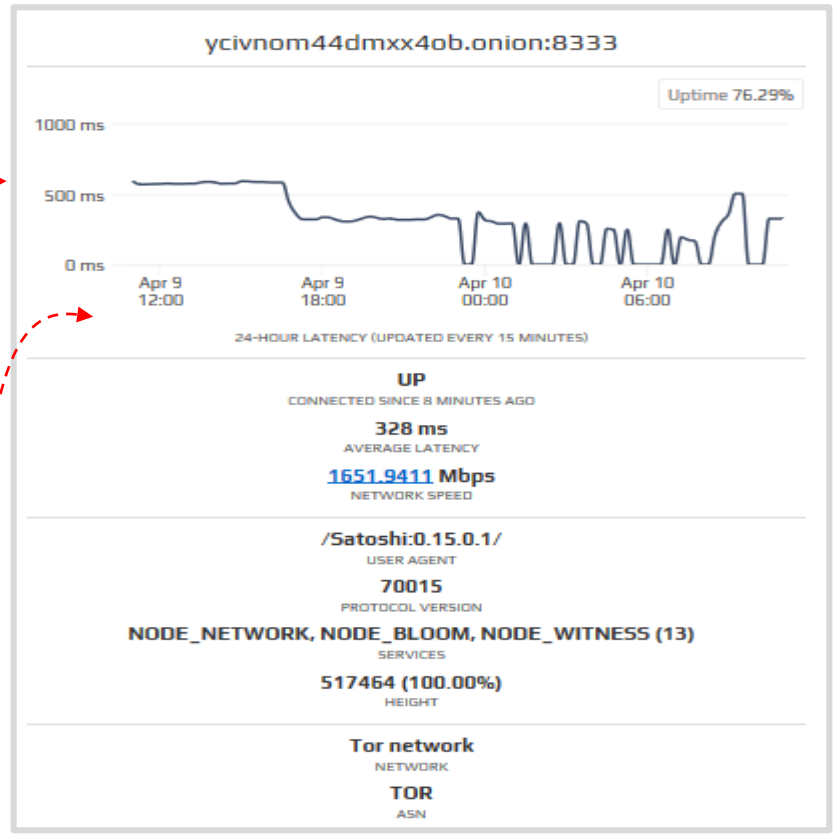
RANK	USER AGENT	NODES
1	Satoshi:0.16.0	3729 (33.69%)
2	Satoshi:0.15.1	3103 (28.03%)
3	Bitcoin ABC:0.16.1	766 (6.92%)
4	Satoshi:0.14.2	560 (5.06%)
5	Satoshi:0.15.0.1	548 (4.95%)
6	Satoshi:0.15.0	264 (2.39%)

← 지역 별 참여 노드 요약 정보

특정 노드의 상세 정보 →

- 지역 별 참여 노드 상세 정보
- 프로토콜 버전으로 해당 노드가 최근에 추가된 기능을 지원하는지 여부를 알 수 있음.
- Satoshi 0.16.0 : Bitcoin Core 버전
- 70015 : 프로토콜 버전

ADDRESS	USER AGENT	HEIGHT	LOCATION	NETWORK
78.56.66.249:8333 78-56-66-249.static.zebra.lt Since 2 minutes ago	/Satoshi:0.15.1/ (70015) NODE_NETWORK, NODE_BLOOM, NODE_WITNESS (13)	515153	Vilnius, Lithuania Europe/Vilnius	Telia Lietuva, AB AS8764
81.182.23.34:8333 51B61722.dsl.pool.telekom.hu Since 2 minutes ago	/Satoshi:0.13.0/ (70014) NODE_NETWORK, NODE_BLOOM (5)	516704	Budapest, Hungary Europe/Budapest	Magyar Telekom plc. AS5483
209.33.232.230:8333 Since 2 minutes ago	/Satoshi:0.8.6/ (70001) NODE_NETWORK (1)	357143	Saint George, United States America/Denver	InfoWest, Inc. AS11071
123.2.217.46:8333 46.217.2.123.dv.iprimus.net.au Since 2 minutes ago	/Satoshi:0.16.0/ (70015) NODE_NETWORK, NODE_BLOOM, NODE_WITNESS (1037)	502096	Melbourne, Australia Australia/Victoria	M2 Telecommunicatio AS38285
[2001:0:5ef5:79fb:24f3:849:bc0d:91f9]:8333 Since 2 minutes ago	/Satoshi:0.15.1/ (70015) NODE_NETWORK, NODE_BLOOM, NODE_WITNESS (13)	517464	-	-
ycivnom44dmxx4ob.onion:8333 Since 2 minutes ago	/Satoshi:0.15.0.1/ (70015) NODE_NETWORK, NODE_BLOOM, NODE_WITNESS (13)	517464	-	Tor network TOR



- 네트워크에서 오래 활동하는 노드들은 신뢰할 수 있음.
- 네트워크에 자주 들락거리는 노드들은 운영이 불안정하거나, 기타 다른 목적 (예를 들면 자료 수집이나, 악의적인 목적)을 위한 것으로 신뢰하기 어려움.
- 실제 블록 체인을 운영하지 않고도 다른 노드들과 프로토콜만 교환해도 이 목록에 나타날 수 있을 것임.

참여 노드 수의 변화 관찰 – Python 실습

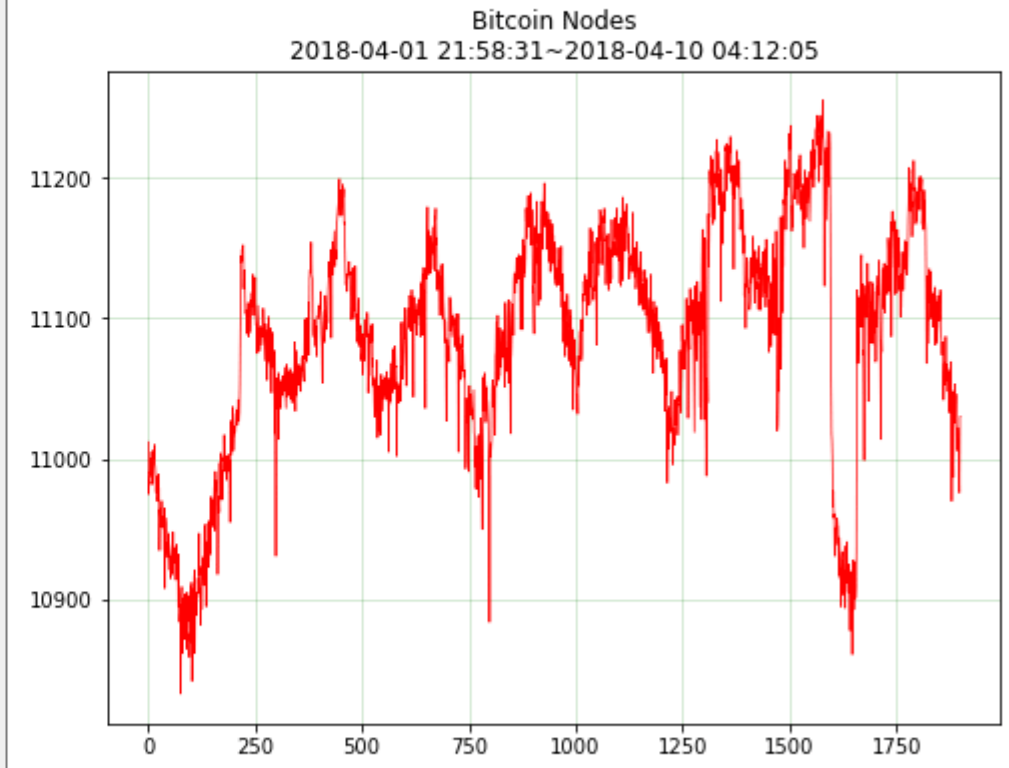
(실습 파일 : 1-1.BitcoinNodes.py)

- 참조 사이트의 API 서비스를 이용하여 최근 비트코인 네트워크에서 활동중인 노드 수의 변화를 관찰함. 아래 결과는 최근 약 10일간의 변화임.

```
# 비트코인 네트워크의 노드 수의 변화를 관찰한다.
# 참조 : https://bitnodes.earn.com/api/
#
# 2018.4.10
# 아마추어 퀀트 (조성현)
# ----------------------------------------
import requests
import time
import matplotlib.pyplot as plt

# 100 page까지만 조회한다. 이 사이트는 최근 60일 까지 데이터를 제공함.
nPage = 20
if nPage > 100:
    print("요청 페이지가 너무 많아 시간이 오래 걸립니다.")
else:

    t = []
    n = []

    for page in range(1, nPage):
        # 페이지 당 100개 씩 요청한다. (Max = 100)
        url = 'https://bitnodes.earn.com/api/v1/snapshots/?limit=100&page=' + str(pa
        resp = requests.get(url=url)
        data = resp.json()
        print("page %d loaded." % page)

        for i in range(len(data['results'])):
            ts = time.gmtime(data['results'][i]['timestamp'])
            t.append(time.strftime("%Y-%m-%d %H:%M:%S", ts))
            n.append(data['results'][i]['total_nodes'])

    t = t[::-1]
    n = n[::-1]
```

```
page 16 loaded.
page 17 loaded.
page 18 loaded.
page 19 loaded.
```

* 자료 출처 : https://bitnodes.earn.com/api/

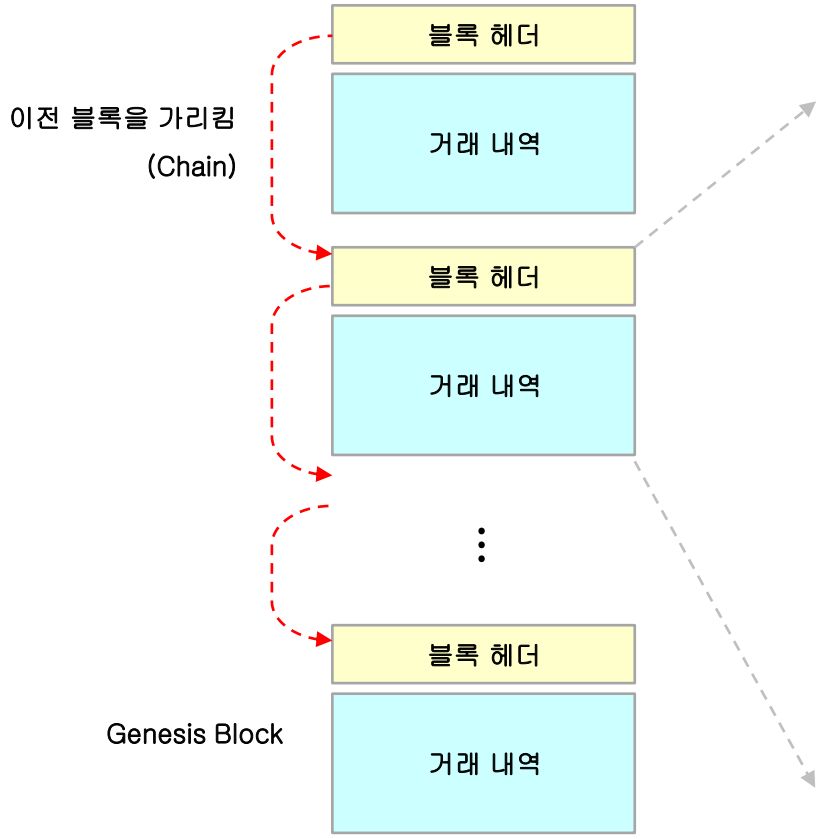



```
In [5]:
```

History log | IPython console

블록체인의 구조

- 블록체인 데이터는 아래와 같이 블록 헤더와 거래 내역의 집합으로 구성됨. 아래 예시는 2018.4.11 13:41:02에 실제 생성된 블록체인의 517,707번째 블록임.
- 최초 블록 (블록 #0)은 2009.1.3 18:15:05에 Satoshi가 생성한 것임. → Genesis Block
- 블록 헤더는 버전, 이전 블록 헤더의 해시 값 등 총 80 bytes로 구성되어 있고, 거래 내역은 총 거래 수, Coinbase 거래, Transaction의 집합으로 구성되어 있음.
- 블록 데이터는 모두 해시 값으로 요약되어 헤더에 기록되어 있고, 블록들은 체인으로 연결되어 있으므로, 블록 데이터의 내용은 변경될 수 없음. 위조, 변조가 현실적으로 불가함.



블록 #517,707
블록 해시 : 00000000000000000003de39f5ad668699c0cbc2236585d585f0d99e….

Version	4 bytes	20000000
Previous Block Header Hash	32 bytes	00000000000000000002098ab5e2ac4d5a6f9cc3cc8c764dcaecd5a9c6290523
Merkle Root	32 bytes	19c6d65057762a53c153b681b3ad6d46edaf59aaf723ecbbd9eeb9b365ed6db6
Timestamp	4 bytes	5acd91de = 2018-04-11 13:41:02
Bits	4 bytes	17502ab7
Nonce	4 bytes	5c81558

블록 헤더

Transaction count	4 bytes	05e8
Coinbase transaction		
Transaction #1		
Transaction #2		
⋮		
Transaction #1512		

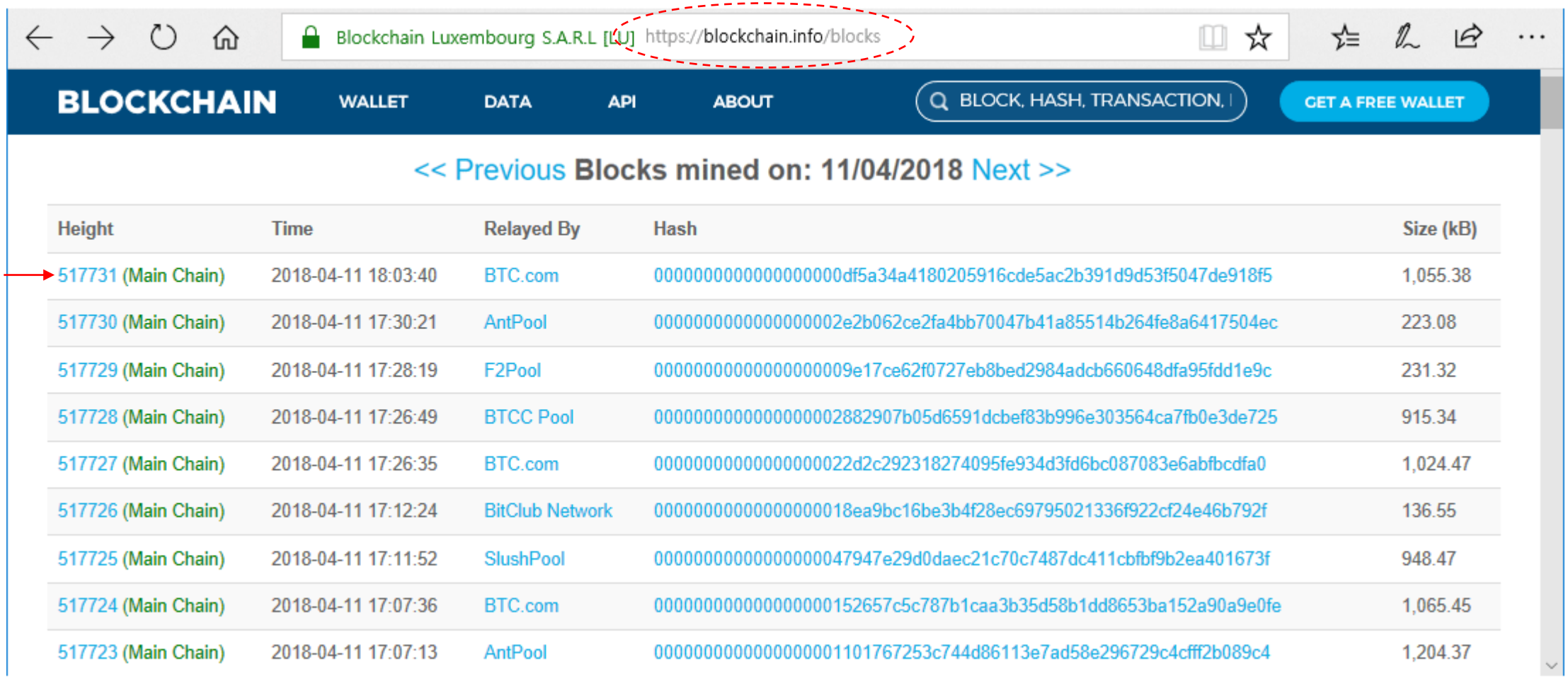
블록 데이터
거래 내역

블록체인의 구조

- 블록의 버전 (Version)은 비트코인 S/W의 버전 정보임. S/W가 업그레이드되면서 추가된 기능을 Version으로 표시함. Genesis block의 version = 1 이었고, version = 2,3,4 까지 진행되다가, 현재는 BIP-9 표준으로 Soft Fork가 진행될 때 Miner들이 어떤 기능을 지원하는지 표시함.
- Previous block header hash는 이전 블록 헤더의 해시 값임. 이 값은 이전 블록을 가리키는 해시 포인터 역할을 하고, 블록들을 체인으로 연결함. 이 블록 안에 있는 임의의 거래 내역 (Transaction)이 변조되면 블록 헤더의 해시 값이 변하므로 다음 블록으로의 연결이 끊어짐. 블록 안에 있는 거래들은 변경될 수 없음.
- Merkle Root는 거래 내역들을 요약한 정보임 (Merkle Tree의 Root). 일부 거래 내역이 변조되면 Merkle Root가 변경되므로 블록 헤더가 변하고, 블록 헤더의 해시 값이 변하므로 블록의 체인이 끊어짐. Merkle Root는 특정 거래가 이 블록에 존재하는지 확인할 때 사용될 수 있음 (SPV가 자신의 거래가 이 블록에 실재하는지 확인할 때도 사용됨).
- Timestamp는 이 블록이 생성된 날짜, 시간임. 시간은 UTC (세계 협정시, 영국 그리니치 천문대 기준) 기준임.
- Bits (or Target) 와 Nonce는 Miner가 블록을 생성할 때 블록 해시 값의 최솟값과 관련이 있음 (난이도 변수). 계산한 블록 해시 값이 Bits에서 제시하는 값 이하 (블록 해시 앞 부분의 0의 개수)가 되도록 Nonce 값을 설정해야 함. 즉 Miner는 이 조건을 만족하는 Nonce 값을 찾는 것이 관건임. (상세 내역은 Mining 편에서 다룸)
- 이 블록 헤더 자체의 해시 값은 저장되어 있지 않음. 필요시 헤더로부터 계산해서 사용함 (double-SHA256 해시).
- Transaction Count는 거래 내역의 총 개수임.
- Coinbase Transaction은 Miner가 이 블록을 생성한 대가로 받은 보상을 (현재는 12.5 BTC + Fee) 자기 지갑으로 송금하는 거래임. 대가를 누가 지급하는 것이 아니라 Miner 스스로 챙기는 것임. 비트코인 총 발행량은 블록이 생성될 때마다 (약 10분 마다) 12.5 BTC (현재 기준으로)씩 증가함.
- Transaction 들은 비트코인 네트워크에서 발생한 (약 10분 동안) 거래 들임. 이 블록이 블록체인에 등록되면 (정확히는 이후 6개 이상의 블록들이 쌓이면), 거래들이 최종 승인된 것임. Miner가 이 거래들을 승인한 것이고, Miner는 그 대가로 보상을 받는 시스템임. → 비트코인 네트워크의 핵심 부분이라 할 수 있음.

블록체인 데이터 확인

- 아래 사이트 (https://blockchain.info)에서 실제 블록체인 데이터를 확인할 수 있음. 이 사이트는 3rd-party API 서버 사이트로 많은 서비스를 제공하고 있음.
- 블록체인에는 총 517,731개의 블록이 체인으로 연결되어 있음. 블록들은 평균적으로 약 10분 마다 생성됨. (평균 10분 마다 생성되도록 난이도가 조절됨.)



블록체인 데이터 확인 – 블록 헤더 (Header)

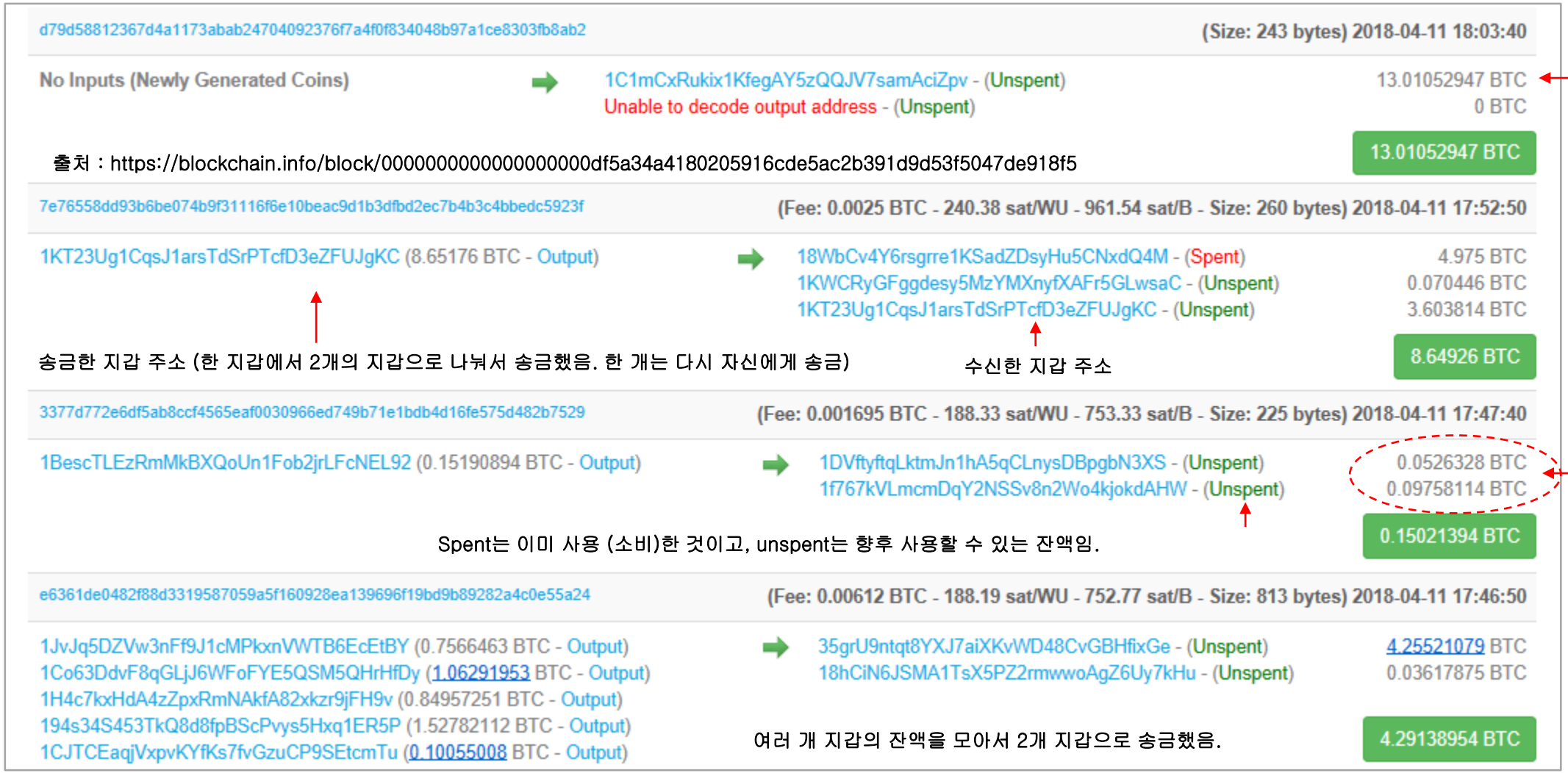
- 아래 내용은 517,731 번 블록의 헤더 정보임. 이 블록은 2018.4.11.18:03:40에 생성되었고, 총 1,323 개의 거래 내역을 가지고 있음.

Height	517731 (Main chain)
Hash	00000000000000000000df5a34a4180205916cde5ac2b391d9d53f5047de918f5
Previous Block	00000000000000000002e2b062ce2fa4bb70047b41a85514b264fe8a6417504ec ← 517,730 번 블록 헤더의 해시 값
Time	2018-04-11 18:03:40
Difficulty	3,511,060,552,899.72
Bits	391129783
Number Of Transactions	1323 ← 승인된 총 거래 내역
Output Total	6,498.90301862 BTC
Estimated Transaction Volume	885.15704864 BTC
Size	1055.375 KB
Version	0x20000000
Merkle Root	5cb40d5eb117638710b31fde9771491e7c6e1c6ed73439732650885df20d6e8f
Nonce	1513409972
Block Reward	12.5 BTC
Transaction Fees	0.51052947 BTC ← 1,323 거래가 지불한 수수료 합계. 거래 당 약 2,800원 정도를 지불하고 있음. (현재 시세 7,39,1000 적용 시)

출처 : https://blockchain.info/block-height/517731
(일부 필드는 제외했음.)

블록체인 데이터 확인 – 거래 내역 (Transaction)

- 아래 내용은 517,731 번 블록의 거래 내역임 (총 1,323 개). 첫 번째 거래 내역은 Coinbase Transaction임. 잔액 (Balance)은 블록에 기록된 것이 아님.



Coinbase Transaction. 이 블록을 생성한 대가로 Miner가 이만큼 가져감. 12.5 BTC는 신규로 발행된 것이고, 나머지는 132개 거래가 지불한 수수료 (Fee) 임.

수신자가 사용할 수 있는 금액 (UTXO)

블록 헤더 관찰 – Python 연습

(실습 파일 : 1-2.BitcoinBlock.py)

- https://blockchain.info/blocks API 서비스를 이용하여 블록이 생성되는 시간 간격의 분포를 관찰함. 최근 10일 동안 생성된 블록 헤더를 관찰함.

```
# Miner에 의해 블록이 생성되는 시간 간격을 관찰한다.
# 참조 : https://blockchain.info/blocks
#
# 블록이 생성되는 시간 간격은 지수분포를 따르므로 내 거래가
# 몇 분 이내에 Mining될 확률이 얼마인지 계산해 볼 수 있다.
# Satoshi 논문에는 포아송 분포로 설명하고 있음.
#
# 2018.4.12 아마추어 퀀트 (조성현)
# ------------------------------------------------------
import requests
import time
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

print("금일 생성된 블록일 읽어옵니다")
url = 'https://blockchain.info/blocks?format=json'
resp = requests.get(url=url)
data = resp.json()

header = []
block = data['blocks']
for n in range(len(block)):
    height = block[n]['height']
    btime = block[n]['time']
    bhash = block[n]['hash']
    header.append([height, btime, bhash])

# 어제 생성된 블록을 읽어온다.
stime = btime - 24 * 60 * 60

# 이전 10일 동안 생성된 블록 정보를 읽어온다
for nDay in range(0, 10):
```

```
In [143]: sdf[0:20]
Out[143]:
    index  Height        Time                                                Hash
0    1445  516202  1522628182  00000000000000000000b6efbb3461adf1e6afac0c84f19...
1    1446  516203  1522628852  00000000000000000000347da3f8ff2720a21dda47fadd74...
2    1447  516204  1522629473  00000000000000000000110ecc3cc4f603aa01f55dae7fe6...
3    1448  516205  1522629850  00000000000000000000045947a16e7eda3f561a1876ce0fa...
4    1449  516206  1522629961  00000000000000000001dc7256cea157ae23f1788e662c0...
5    1450  516207  1522630178  00000000000000000000037ac86ba79d83ebda92edef8ab13...
6    1451  516208  1522631224  00000000000000000000497abfd2d649f64f4d3ef4a8b308...
7    1452  516209  1522632008  00000000000000000001da73c8fba2485d8d2b9d59ccb8e...
8    1453  516210  1522632382  00000000000000000003bb1cd2556b009df9b710bb773ea...
9    1454  516211  1522633070  00000000000000000000140112037f53276ccbd24a5c5c9d...
10   1455  516212  1522633660  00000000000000000000414badec9f686288f81329e00f46...
11   1456  516213  1522633755  00000000000000000000eb8770d61eac5e51148edd10d9b...
12   1457  516214  1522634084  00000000000000000003a3b756ef9ca20c0650e266f6246...
13   1458  516215  1522635016  00000000000000000000743a1abddf37e9a1a7cf82b700a...
14   1459  516216  1522635295  00000000000000000001f42e867df19d9e9fde79c9f8218...
15   1460  516217  1522635327  00000000000000000000026efdc725e74e17ab5b661d136c4...
16   1461  516218  1522635871  00000000000000000000b09ee6c28fc5bdd1d6db3d8264...
17   1462  516219  1522635947  00000000000000000001ef82609a307e528ac8c6f598ab7...
18   1463  516220  1522636468  00000000000000000003dbf93c81132fa73c1d7eff23d0a...
19   1464  516221  1522636514  00000000000000000029afe4e93ee00f52aaa82f1d65f9...
```

최근 10일 동안 생성된 블록의 헤더 정보 (요약본)

History log | IPython console

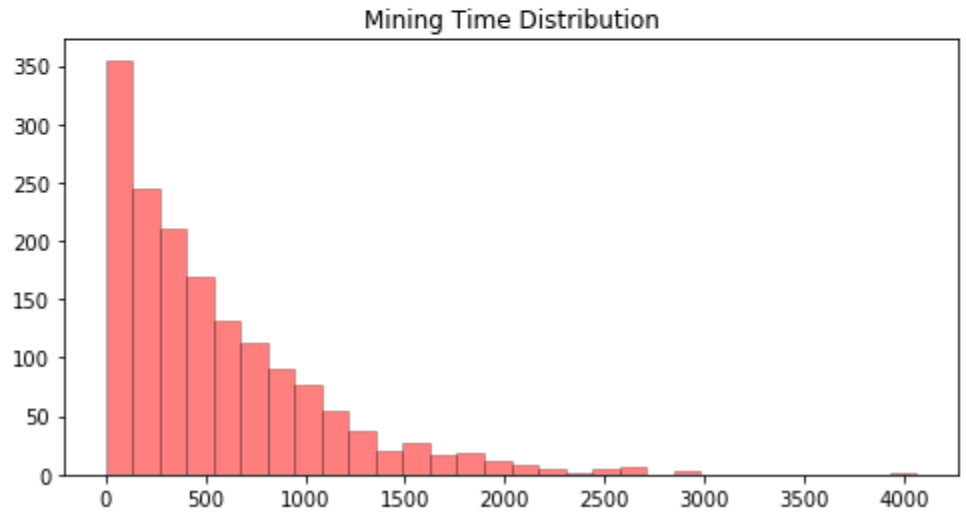
블록 헤더 관찰 – Python 연습

(실습 파일 : 1-2.BitcoinBlock.py)

- Miner에 의해 블록이 생성된 시간 간격을 관찰함. 최근 10일 평균 = 548 (초), 표준편차 = 514 (초)로 관찰됨.

```
    data = resp.json()

    block = data['blocks']
    for n in range(len(block)):
        height = block[n]['height']
        btime = block[n]['time']
        bhash = block[n]['hash']
        header.append([height, btime, bhash])

    stime = block[0]['time'] - 24 * 60 * 60

df = pd.DataFrame(header, columns=['Height', 'Time', 'Hash'])
sdf = df.sort_values('Time')
sdf = sdf.reset_index()
print('총 %d 개 블록 헤더를 읽어왔습니다.' % len(df))

# 블록 생성 소요 시간 분포 관찰
mtime = sdf['Time'].diff().values
mtime = mtime[np.logical_not(np.isnan(mtime))]
print("평균 Mining 시간 = %d (초)" % np.mean(mtime))
print("표준편차 = %d (초)" % np.std(mtime))

plt.figure(figsize=(8,4))
n, bins, patches = plt.hist(mtime, 30, facecolor='red', edgecolor='black', linewi
plt.title("Mining Time Distribution")
plt.show()

# 5분 이내에 내 거래가 Mining될 확률
s = 60 * 5
p = 1 - np.exp(-s / np.mean(mtime))
print("5분 이내에 내 거래가 Mining될 확률 = %.2f (%s)" % (p * 100, '%'))
```

```
2018-04-10 00:16:24 생성된 블록을 읽어옵니다
2018-04-09 00:02:02 생성된 블록을 읽어옵니다
2018-04-08 00:06:00 생성된 블록을 읽어옵니다
2018-04-07 00:11:17 생성된 블록을 읽어옵니다
2018-04-06 00:18:25 생성된 블록을 읽어옵니다
2018-04-05 00:00:35 생성된 블록을 읽어옵니다
2018-04-04 00:00:46 생성된 블록을 읽어옵니다
2018-04-03 00:01:28 생성된 블록을 읽어옵니다
2018-04-02 00:02:58 생성된 블록을 읽어옵니다
총 1605 개 블록 헤더를 읽어왔습니다.
평균 Mining 시간 = 547 (초)
표준편차 = 513 (초)
```




```
5분 이내에 내 거래가 Mining될 확률 = 42.17 (%)
```

History log | IPython console

(실습 파일 : 1-2.BitcoinBlock.py)

블록 헤더 관찰 – Python 연습

- 거래 생성 후 이 거래가 Mining되어 블록에 등록되는 시간의 확률을 추정함. 단위 시간 당 블록이 생성되는 횟수는 포아송 분포를 따르고, 블록이 생성되는 시간 간격에 대한 분포는 지수분포를 따름. 이 예제에서는 블록이 생성되는 시간 간격을 측정하였으므로 지수분포를 이용함.
- 지수분포를 이용하면 이 확률을 계산해 볼 수 있음. 평균대기시간 = 최근 10일 간 블록이 생성되는 시간 간격의 관측 값.
- Satoshi Nakamoto의 2008년 원문 (Bitcoin: A Peer-to-Peer Electronic Cash System – 우측 박스 참조)에는 포아송 분포로 설명하고 있음.

$$p(t) = \lambda e^{-\lambda t}$$

$$p(t < T) = \int_0^T \lambda e^{-\lambda t}\, df = \left[-e^{-\lambda t}\right]_0^T$$

$$\lambda = \frac{1}{548}$$

$$p(t < 300) = \left[-e^{-\lambda t}\right]_0^{300} = -e^{-\frac{300}{548}} + 1 = 0.4216$$

- 거래 생성 후 이 거래가 5분 이내에 Mining되어 블록에 등록될 확률은 약 42.16% 임.

(참고 : 2008년 Satoshi 논문의 마지막 부분)

The receiver generates a new key pair and gives the public key to the sender shortly before signing. This prevents the sender from preparing a chain of blocks ahead of time by working on it continuously until he is lucky enough to get far enough ahead, then executing the transaction at that moment. Once the transaction is sent, the dishonest sender starts working in secret on a parallel chain containing an alternate version of his transaction.

The recipient waits until the transaction has been added to a block and z blocks have been linked after it. He doesn't know the exact amount of progress the attacker has made, but assuming the honest blocks took the average expected time per block, the attacker's potential progress will be a Poisson distribution with expected value:

$$\lambda = z\frac{q}{p}$$

To get the probability the attacker could still catch up now, we multiply the Poisson density for each amount of progress he could have made by the probability he could catch up from that point:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{cases} (q/p)^{(z-k)} & \text{if } k \le z \\ 1 & \text{if } k > z \end{cases}$$

← 포아송 분포

Rearranging to avoid summing the infinite tail of the distribution...

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!}\left(1 - (q/p)^{(z-k)}\right)$$

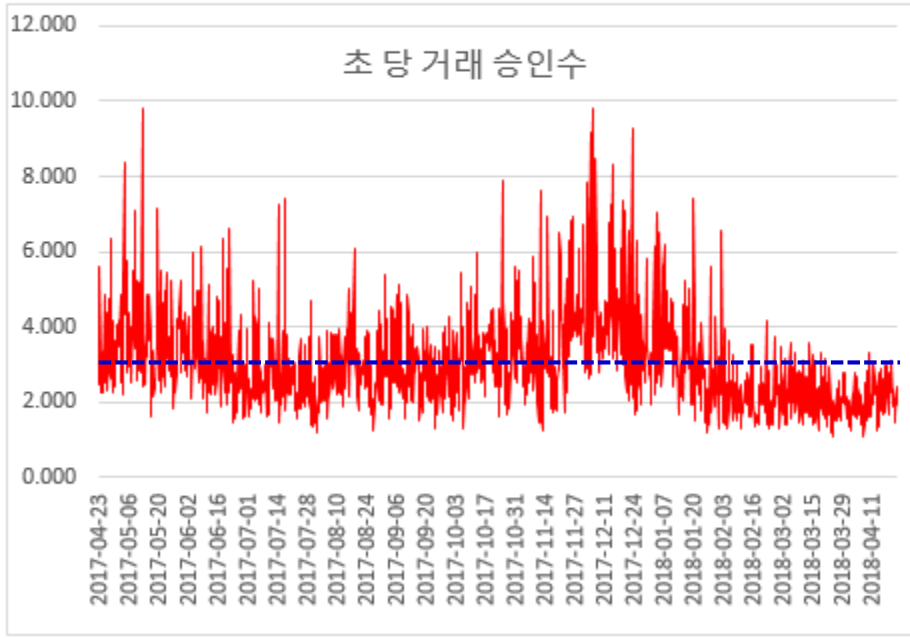
(실습 파일 : 1-3.transactions-per-second.xlsx)

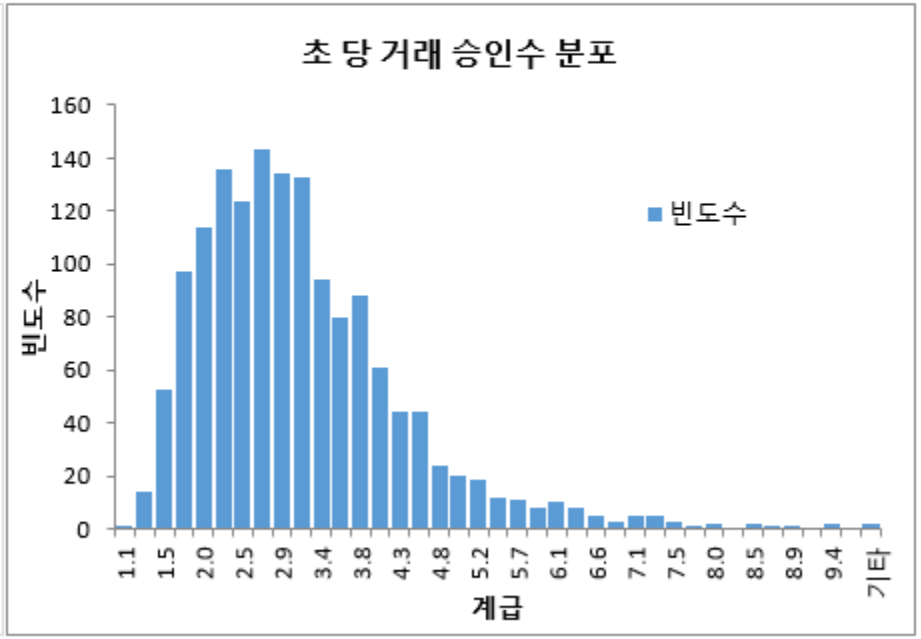
초 당 거래 승인 개수

- 현재 비트코인 네트워크는 초 당 약 3개의 거래 (Transaction)를 처리 (승인)하고 있음.
- 블록이 생성되는 시간과, 블록에 포함된 총 Transaction수를 측정하면 초 당 처리하는 Transaction 개수를 측정할 수 있음.
- 아래 예시는 최근 1년 간 측정된 Transaction 개수와 분포임. (평균 = 3.0 개, 표준편차 = 1.22 개)

	A	B
1	날짜-시간	초당 거래 승인수
2	2017-04-23 9:11	2.450
3	2017-04-23 14:58	5.633
4	2017-04-23 20:45	3.000
5	2017-04-24 2:32	2.683
6	2017-04-24 8:19	2.250
7	2017-04-24 14:06	2.283
8	2017-04-24 19:53	3.350
9	2017-04-25 1:40	2.483
10	2017-04-25 7:27	2.550
11	2017-04-25 13:14	4.850
12	2017-04-25 19:01	3.717
13	2017-04-26 0:48	2.533
14	2017-04-26 6:35	2.500
15	2017-04-26 12:22	3.467
16	2017-04-26 18:09	4.400
17	2017-04-26 23:56	2.817
18	2017-04-27 5:43	2.317
19	2017-04-27 11:30	3.400
20	2017-04-27 17:17	4.017
21	2017-04-27 23:04	4.750
22	2017-04-28 4:51	2.850

평균	3.038	표준편차	1.220





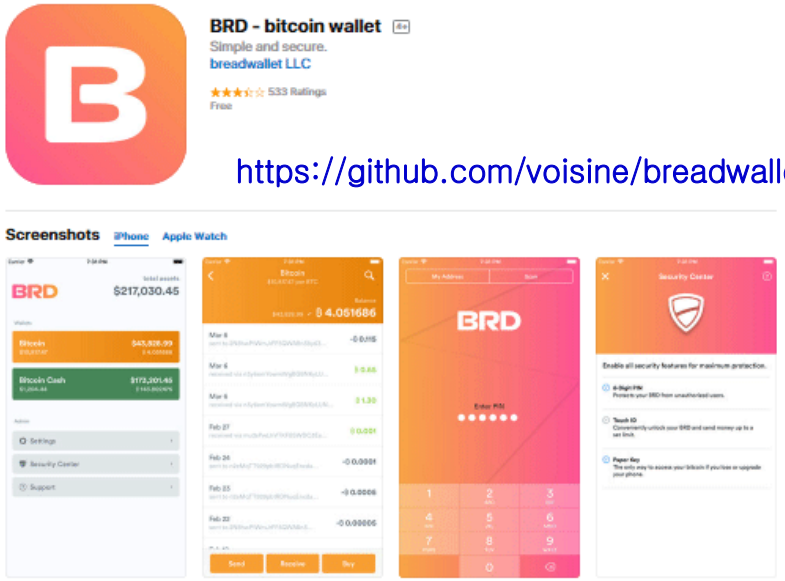
출처 : https://blockchain.info/ko/charts/transactions-per-second?timespan=1year

블록 Size 제한과 비트코인의 확장성 (Scalability)

- 블록체인의 각 블록의 크기는 1MB를 초과할 수 없음. 초기에는 크기의 제한이 없었지만, DoS 공격 등의 가능성 때문에 Satoshi가 1MB로 변경했다고 함.
- 현재는 초 당 약 3개의 Transaction을 처리하고 있어 블록 크기 제한에 큰 영향은 없으나, 향후 Transaction이 증가하면 문제가 될 수 있음.
- 만약, 비트코인이 가상화폐로 자리 잡아 실 생활에 널리 사용된다면 Transaction이 엄청나게 증가할 것이므로 현재의 처리 용량으로는 큰 문제가 발생할 수 있음.
- Transaction은 증가하는데, 처리 용량이 제한적이라면, 승인을 필요로 하는 수요가 공급보다 많아지게 되고, Miner들은 높은 수수료를 지불하는 Transaction 들만 골라서 승인하려는 경향이 발생할 것임. 따라서 거래 수수료가 점차 상승하는 문제도 발생할 수 있음.
- 이러한 문제를 비트코인의 Scalability 문제라고 함. 1MB라는 블록 크기 제한 때문에 향후 확장성 (Scalability)에 큰 문제가 발생할 수 있음.
- 이 문제를 해결하려면 한 블록에 들어가는 Transaction 개수가 많아 지도록 블록의 크기를 늘릴 필요가 있음.
- 블록의 크기를 늘리려면 전체 노드들의 S/W를 업그레이드해야 할 필요가 있고, 그러려면 참여 노드들의 전체적인 합의가 이루어져야 함. 합의가 이루어지지 않은 상태에서 업그레이드를 진행하면 블록체인이 두 개로 분리되는 (네트워크가 분리됨) 현상이 발생하기도 함 (Hard fork).
- Scalability 문제와 (향후 논의될) Transaction Malleability 문제를 해결하기 위해 2017년 8월 Segregated Witness (SegWit) 기능이 추가되었음 (Soft fork 방식) .
- 참고로, 2017년 Soft fork 방식으로 SegWit 기능이 추가되었을 때 일부 반대하는 주체들이 있었음 (일부 Major급 Miner). 이들은 블록의 크기를 8MB로 늘리는 Hard fork를 진행하여 비트코인 캐시를 탄생시켰음. 비트코인 캐시는 거래 용량을 확장하여 처리 속도를 증가 시켰음 (화폐 유통을 강조하여 캐시라는 이름을 붙임).
- SegWit에 대해서는 거래 (Transaction 편에서, Fork에 대해서는 Mining 편에서 자세히 다룰 것임).

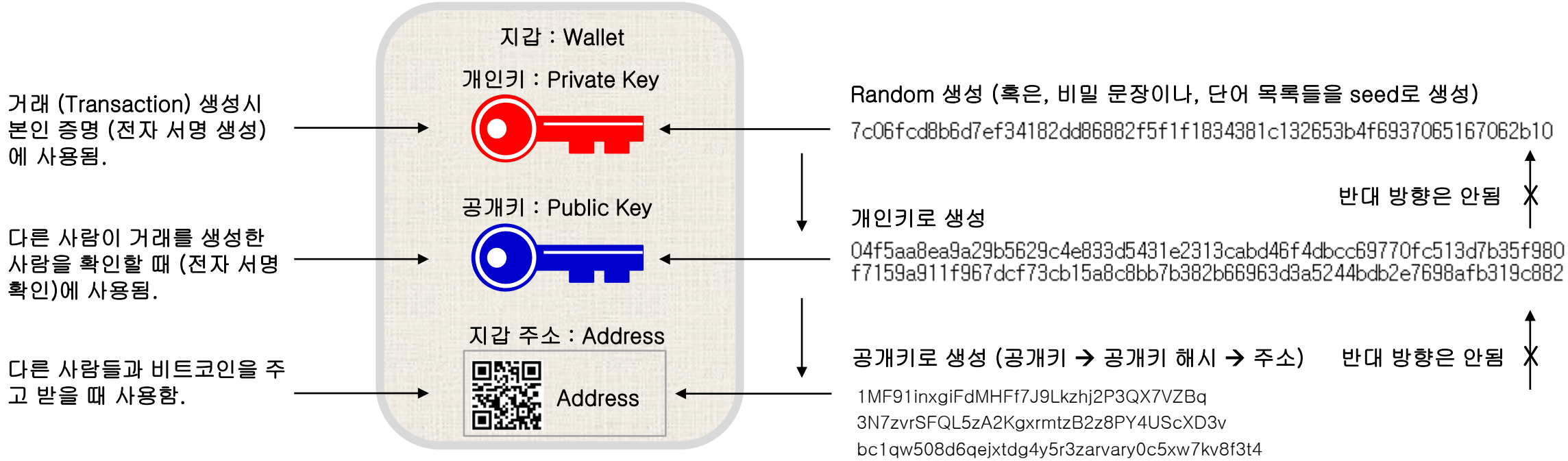
비트코인 지갑 : 일반 사용자 측면의 지갑

- 비트코인 지갑은 Key 관리, 잔고 관리, 거래 생성, 거래 인증 , 거래 내역 관리 등의 기능을 수행함.
- Full 노드에서의 지갑은 블록체인 데이터를 모두 관리하고 있으므로 위의 기능을 모두 쉽고 안전하게 수행할 수 있으며 보안 위험에 대한 안정성 측면에서는 가장 바람직한 형태의 지갑임. 그러나 블록체인 데이터를 유지/관리하는 비용이 크다고 할 수 있음.
- Lightweight (SPV) 지갑은 블록체인 데이터가 없으므로 주변 Full 노드에 의존해야 함. 이 지갑은 블록 헤더만 가지고 주변 Full 노드의 도움을 받아 잔고를 직접 관리해야 하고, 거래 생성 후 거래 인증도 직접 관리해야 함. 블록체인 데이터를 직접 관리하지 않아도 된다는 이득에 대한 대가로 약간의 보안 위험에 노출될 수 있으며, 블록 헤더만으로 인증해야 하기 때문에 구현이 복잡해짐 (Bloom Filter, Merkle branch 인증 등).
- 아래의 지갑 App. (Bread wallet, Android wallet)들은 주로 Lightweight (SPV) 형태이며, 일부 App. 들은 간단히 사용하기 위해 3rd-party API 기능을 사용하는 것들도 있음 (편리성). 3rd-party API를 이용한 App. 들은 보안에 취약할 수밖에 없으므로 가장 바람직하지 못한 형태의 지갑임.



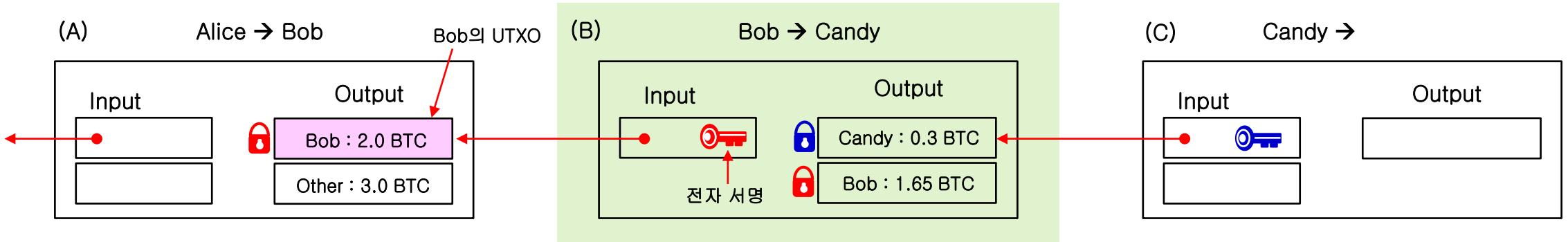
비트코인 지갑의 : 개발자, 전문가 측면의 지갑

- 비트코인 지갑은 개인키 (혹은 비밀키 : Private key), 공개키 (Public Key), 지갑 주소 (Address)로 구성되어 있음 (Key의 집합체).
- 비트코인의 잔액은 지갑에 보관되는 것이 아니라 네트워크 상의 블록체인에 기록되어 있음. 거래 내역에 누구로부터 얼마를 받았고, 이 중 얼마를 사용하지 않았다는 기록 (UTXO : Unspent transaction output)이 해당 지갑의 잔고 (Balance) 임.
- 개인키는 블록체인에 기록된 내 잔고를 사용할 때 본인 인증 (전자 서명 : Digital signature) 용으로 사용되고, 공개키는 내가 서명했다는 사실을 다른 사람들이 확인할 때 사용함 (서명 확인). 개인키는 내가 보관하고, 공개키는 블록체인 데이터 상에 공개되어 있음.
- 지갑 주소는 비트코인을 주고 받을 때 사용하고, 한 지갑에서 개인키, 공개키, 지갑 주소는 하나가 아니라 여러 개를 사용하는 것이 보통임 (안전성).
- 개인키로 공개키를 만들고 공개키로 지갑 주소를 만듦 (반대 방향으로는 안됨). 개인키만 알고 있으면 공개키와 지갑 주소는 언제든지 생성할 수 있음.
- 만약 개인키를 잃어버리면 블록체인에 기록된 잔고를 사용할 수 없음. → 해당 지갑으로 송금된 비트코인은 영원히 아무도 사용할 수 없음.



거래 (Transaction : Tx) 생성

- 이전에 Alice가 Bob에게 2.0 BTC를 보낸 상태임 (A). Bob이 Candy에게 0.3 BTC를 보내는 거래 (B)는 Input과 Output으로 구성 함.
- Input은 Bob이 사용 가능한 잔고 (UTXO : Unspent Transaction Output)가 기록된 거래 원장 (A)의 정보임 (포인터 역할). 이 정보는 블록체인에 기록되어 있음.
- (A)의 UTXO는 오직 Bob 만이 사용할 수 있도록 Bob의 공개키 (해시)로 잠가 놓았음. Bob은 (B) input에 이 잠금 장치를 풀 수 있는 키를 제시해야 함.
- 잠금 장치 해제는 개인키로 전자 서명 (Digital Signature)을 하고, 서명과 함께 공개키를 input에 제시함. 전자 서명은 개인키를 가지고 있는 Bob만 할 수 있음.
- Output은 Candy에게 0.3 BTC를 보낸다는 내용과 나머지는 다시 자신에게 보낸다는 내용을 기록함.
- Input의 총액은 2.0이고, Output의 총액은 1.95 BTC임. Input 총액과 Output 총액의 차이인 0.05는 수수료 (Fee)로 Miner가 거래 승인의 대가로 가져감.
- 만약 Output에 다시 자신에게 보내는 항목이 없다면 수수료는 1.7 BTC가 되어 Miner가 모두 가져감. → 지갑 프로그램을 만들 때 주의 사항임.
- 수수료가 없다면 (Fee = 0) Miner는 이 거래를 블록에 포함시키지 않을 것이므로, 이 거래는 블록체인에 추가되지 못함. → 거래가 승인되지 않음.
- 향후 Candy가 0.3 BTC를 사용하려면 Bob이 Candy에게 보낸 거래 (B)의 Output을 참조해야 함.
- Bob이 Candy에게 0.3 BTC를 보낼 때 (B) 부분만 비트코인 네트워크로 전송됨.
- (B)의 Input 부분에는 Bob의 실제 잔고가 기록되어 있지 않고, Output에 수수료도 기록되어 있지 않음. 따라서 (B)만 가지고는 수수료를 계산할 수 없음.
- Miner가 (B)를 수신하면 Input이 가리키는 거래 (A)를 참조하여 Bob의 잔고를 알아 내서 수수료를 계산함.
- Alice가 Bob에게 보낸 2.0 BTC 이외에 다른 사람이 Bob에게 보낸 다른 거래 내역의 Output에도 Bob이 사용할 수 있는 잔고가 있을 수 있음. → 총 잔고



Alice가 Bob에게 2.0 BTC를 보낸 거래 내역.
이 정보는 블록체인에 기록되어 있음.

Bob이 Candy에게 0.3 BTC를 보내는 거래 원장.
수수료 (Fee) = 2.0 – 0.3 – 1.65 = 0.05 BTC

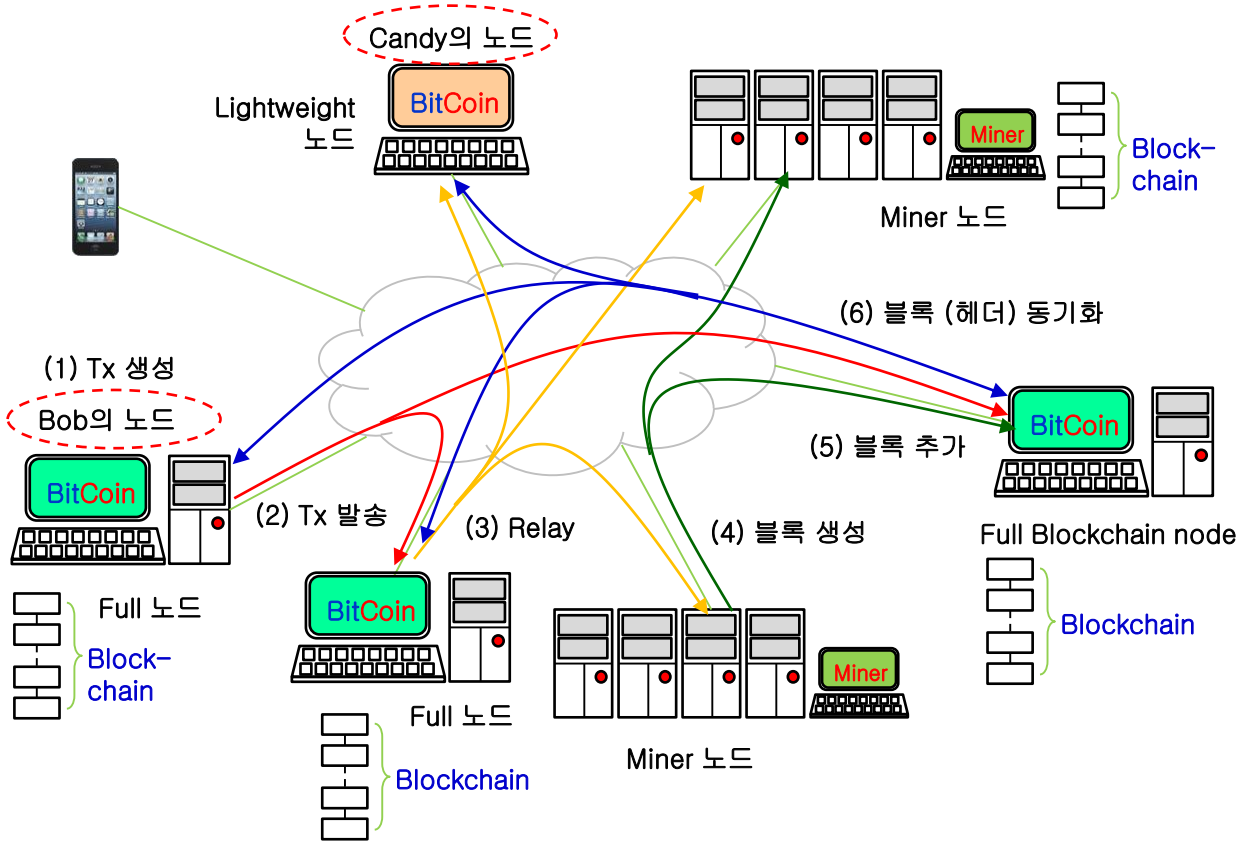
향후 Candy가 누군가에게 보낼 때

Transaction (Tx) 전송 및 전파 (Relay)

- Bob의 지갑은 합의 규칙에 맞게 Transaction Tx (B)를 생성한 후 자신이 알고 있는 인근 노드들에게 이 Transaction을 전송함 (Tx 메시지).
- 인근 노드들이 Tx를 받으면 규칙에 맞게 생성되었는지 검증하고 전파 (Relay) 함. Full 노드가 Tx (B)를 받으면 자신의 블록체인 데이터에서 Tx (B)의 Input이 가리키는 Tx (A)를 찾아서 Output이 Bob의 UTXO인지 확인하고 (Bob이 사용할 수 있는 잔고가 맞는지), Tx (B)의 Input에 기록된 열쇠로 UTXO의 자물쇠가 풀리는지 (Tx (B)가 Bob이 생성한 것이 맞는지, Bob이 사용할 권한이 있는지) 확인함. (Tx (A) Output의 자물쇠는 Locking Script이고, Tx (B) Input의 열쇠는 Unlocking Script 임. 나중에 Transaction 편에서 자세히 다룸).
- Full 노드가 Tx (B)의 유효성을 검증한 후에는 자신이 알고 있는 인근 노드들에게 Tx (B)를 전파함. → Tx (B)는 비트코인 네트워크로 퍼져나감.
- Miner 노드가 Tx (B)를 받으면 Full 노드와 동일하게 유효성을 확인하고, 다른 Tx 들과 합쳐서 새로운 블록을 생성한 후 Full 노드들에게 보냄.
- Tx (B)를 포함한 블록이 생성되어 기존 블록체인에 연결되면 Bob이 Candy에게 보내는 거래가 완료된 것임. 실제는 이후 블록이 6개 정도 더 추가되어야 이 거래가 완전히 승인된 것으로 판단함 (나중에 Mining 편에서 자세히 다룸).
- Full 노드가 Tx (B)의 Input이 가리키는 곳의 Output이 사용 가능한 잔고인지 (unspent), 아니면 이미 사용한 잔고인지 (spent) 확인하는 방법은 UTXO set을 이용함. Full 노드는 UTXO 확인을 위한 별도의 DB (UTXO set : Google LevelDB 사용)를 관리하고 있어서 고속으로 Tx (A)의 Output을 찾을 수 있음.
- Full 노드는 이 거래가 최종 승인된 후에는 이중 지불 (Double spending)을 방지하기 위해 UTXO set에 기록된 Tx (A)의 Output 상태를 “spent” 상태로 표시함. 블록체인에 기록된 Tx (A) Output의 내용이 바뀌는 것이 아님 (블록에 기록된 내용은 바꿀 수 없음. 모든 해시 값이 달라지기 때문임).

거래 생성 및 승인 절차 : Bob이 Full 노드인 경우

- Bob은 Alice에게 받은 2.0 BTC를 사용할 수 있고, 이 중 0.3 BTC를 Candy에게 보내려고 함. Bob은 Full 노드에서 Candy에게 0.3 BTC를 보내려고 함.
- Bob이 사용할 수 있는 2.0 BTC의 UTXO는 Bob의 지갑에 들어있는 것이 아니라, 자신이 관리하는 블록체인 (Blockchain)에 기록되어 있음.



(1) Bob은 자신의 UTXO set과 블록체인 데이터에서 사용 가능한 UTXO를 찾아서 생성할 Tx의 Input에 기록하고 Candy에게 송금할 부분인 Output을 작성하여 Tx를 생성함.

(2) 생성한 Tx를 인근 노드들에게 전송함.

(3) 인근 노드가 블록체인을 관리하는 노드라면 Bob이 보낸 Tx의 유효성을 검증하고, 다른 인근 노드로 이 Tx를 Relay 함. 이 Tx가 Candy와 관련된 거래이므로 Candy의 Lightweight (SPV)노드에도 전달됨. (Candy는 Bloom Filter로 자신과 관련된 거래를 Relay해 달라고 Full 노드에게 요청해 둔 상태임.)

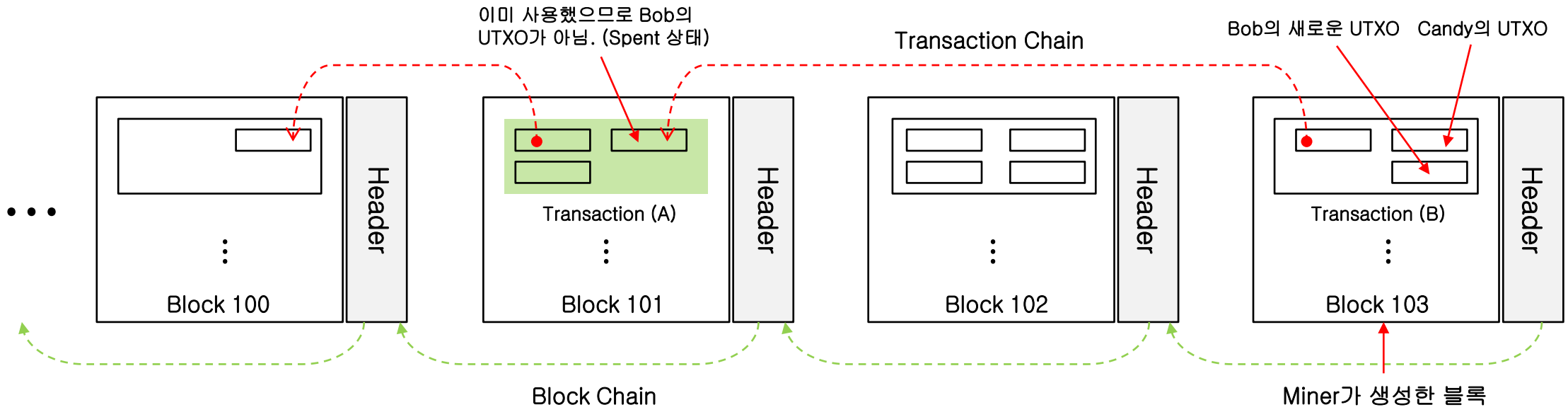
(4) Miner 노드가 Bob의 Tx를 받으면 유효성과 수수료 (Fee)의 적정성을 확인한 후, 다른 Tx들과 합쳐서 새로운 블록을 생성함. Miner 노드들은 서로 경쟁적으로 블록을 생성하여 다른 노드들에게 전송함 (거래 승인 과정 : PoW). Miner가 생성한 블록에는 자신의 지갑으로 일정량의 비트코인과 Tx들에 포함된 수수료를 보낼 수 있음 (거래 승인의 대가).

(5) Full 노드가 새로운 블록을 받으면, 블록의 유효성을 검증한 후 자신의 블록체인에 이 블록을 연결함 (블록 승인 과정). Bob이 Candy에게 보낸 거래 내역이 블록체인에 기록되고 거래가 완료됨.

(6) 새로운 블록이 블록체인에 연결되면 다른 Full 노드들과 동기화 함.

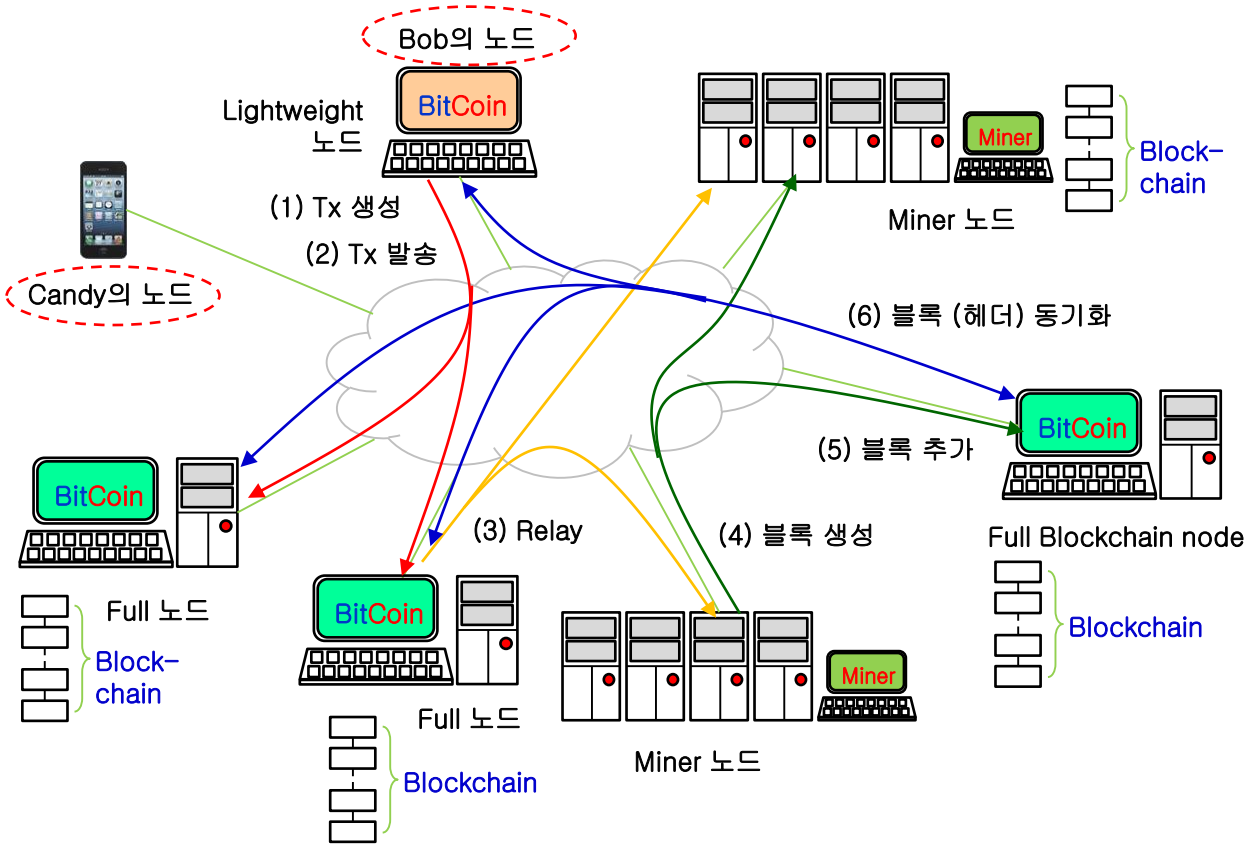
거래 승인

- Miner가 Tx (B)를 새로운 블록 (Block 103 번)에 기록하면 Bob이 Candy에게 보내는 거래가 (1차) 승인된 것임. (이후 6개 정도의 블록이 추가되어야 완전히 승인된 것으로 판단함).
- Tx (B)가 승인된 후의 블록체인 데이터는 아래와 같음. Bob은 Block 101번에 기록된 Tx (A)의 잔고를 사용해서 Candy에게 송금한 것임.
- Tx (A)의 잔고는 이미 Candy에게 지불 되었으므로 더 이상 사용할 수 없음 (Spent 상태). → 이중 지불 (Double Spending) 문제가 해결됨.
- 이 상태에서 Bob이 Tx (A)의 0.3 BTC를 다른 사람에게 보내려고 하면 네트워크의 노드들은 Tx (A)에 Bob이 사용할 수 있는 잔고가 없으므로 거절함.
- Candy는 Block 103에 기록된 Tx (B)의 첫 번째 Output에 기록된 비트코인을 사용할 수 있음. Bob이 했던 것과 동일한 절차를 거침.
- Bob은 Tx (A)의 Output은 더 이상 사용할 수 없고, Tx (B)의 두번 째 Output에 기록된 비트코인을 사용할 수 있음.
- 아래 그림과 같이 블록들은 서로 체인으로 연결되어 있고 (Block chain), Tx 들도 서로 Output과 Input으로 연결되어 있음 (Transaction chain).



거래 생성 및 승인 절차 : Bob이 Lightweight (SPV) 노드인 경우

- Bob은 Alice에게 받은 2.0 BTC를 소유하고 있고, 이 중 0.3 BTC를 Candy에게 보내려고 함. Bob은 Lightweight (SPV) 노드에서 Candy에게 0.3 BTC를 보냄.
- Bob의 지갑은 블록체인 데이터는 가지고 있지 않고, 블록 헤더만 가지고 있음. Full 노드와 블록 헤더를 동기화 함. 자신의 UTXO는 자체 관리함.



(1) Bob은 Full 노드들과 블록 헤더 정보를 동기화하고 있고, 자신의 지갑이 사용할 수 있는 잔고 (UTXO)를 관리하고 있는 상태임. Bob의 지갑은 Full 노드와 동일하게 Tx의 Input, Output을 생성함.

(2) 생성한 Tx를 인근 노드들에게 전송함.

(3) 인근 노드는 이 Tx를 네트워크에 Relay 함.

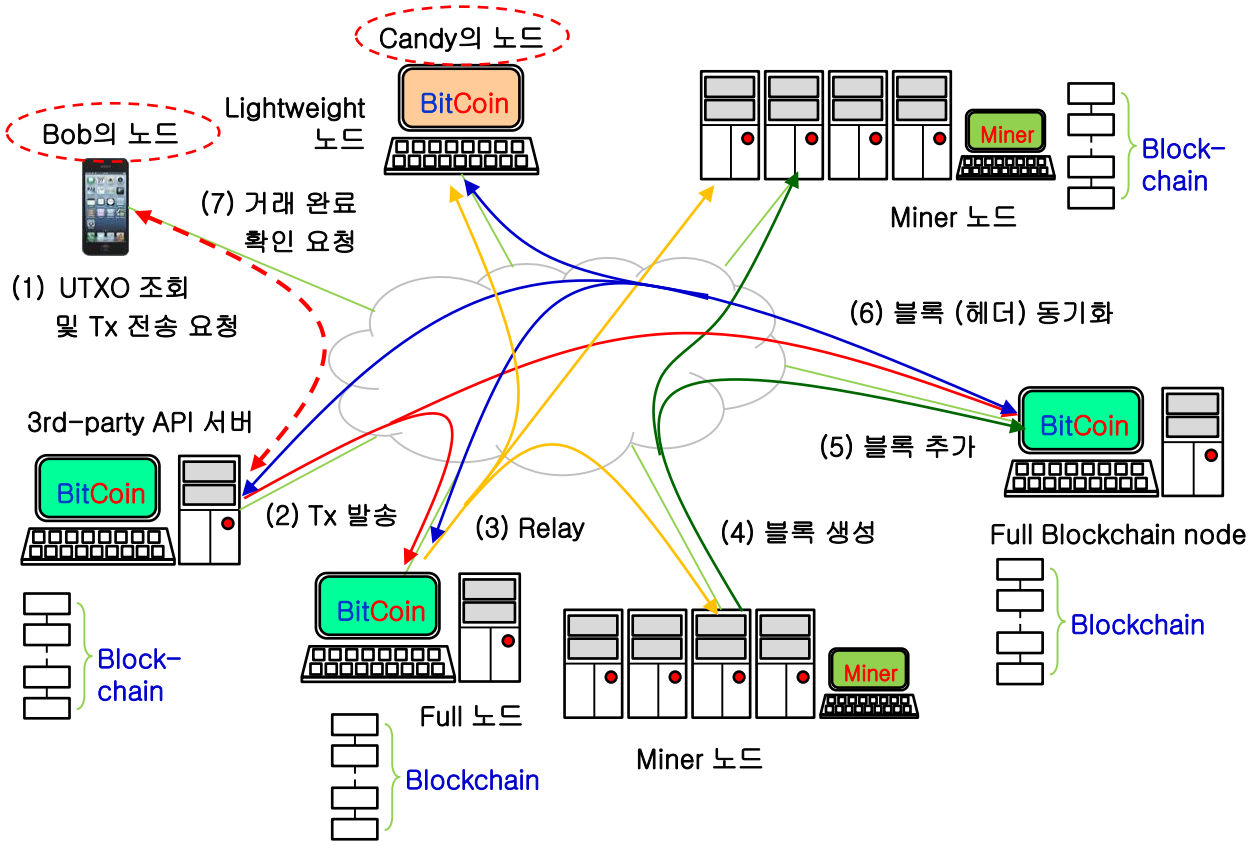
(4) Miner 노드가 Bob의 Tx를 포함한 새로운 블록을 생성함..

(5) Full 노드는 새로운 블록을 자신의 블록체인에 연결함.

(6) Full 노드는 새로운 블록이 생겼다고 다른 Full 노드와 SPV 노드에게 알려줌. SPV 노드는 Full 노드에게 블록 데이터를 요청함. Full 노드는 SPV에게 블록 헤더와, SPV가 자신의 거래가 이 블록에 포함되었는지 확인할 수 있는 재료 (Merkle branch와 Bloom Filter를 통과한 Tx 들)를 보냄. SPV는 자신의 Tx가 이 블록에 기록되었는지 검증할 수 있고 (블록 Height), 자신의 UTXO set을 업데이트할 수 있음. 이후 6개의 블록이 추가되면 이 Tx가 최종 승인되었다고 판단함 (블록 Depth 로 확인).

거래 생성 및 승인 개요 : 3rd-party API client 지갑의 경우 (비트코인의 표준 절차는 아님)

- Bob은 Alice에게 받은 2.0 BTC를 소유하고 있고, 이 중 0.3 BTC를 Candy에게 보내려고 함. Bob은 3rd-party API client 노드를 사용하고 있음.
- Bob의 지갑은 오직 거래를 위한 키와 지갑 주소만 가지고 있음. UTXO 조회, Tx 전송, Tx 완료 확인은 3rd-party API 서버에 의존함. 비트코인의 구성 요소는 아님.



(1) Bob은 자신이 사용할 수 있는 UTXO 정보를 모르므로 자신의 지갑 주소를 API 서버에 알려 주고 UTXO를 찾아달라고 요청함. Bob은 이 UTXO를 사용하여 Tx의 Input과 Output을 생성하고, 생성한 Tx를 API로 보내 비트코인 네트워크로 전송해 달라고 요청함. Bob은 UTXO 조회와 거래 확인만 API 서버에 의존하고 Tx 는 직접 전송할 수도 있음.

(2) API 서버는 Bob의 Tx를 인근 노드들에게 전송함.

(3) 인근 노드는 이 Tx를 네트워크에 Relay 함.

(4) Miner 노드가 Bob의 Tx를 포함한 새로운 블록을 생성함..

(5) Full 노드는 새로운 블록을 자신의 블록체인에 연결함.

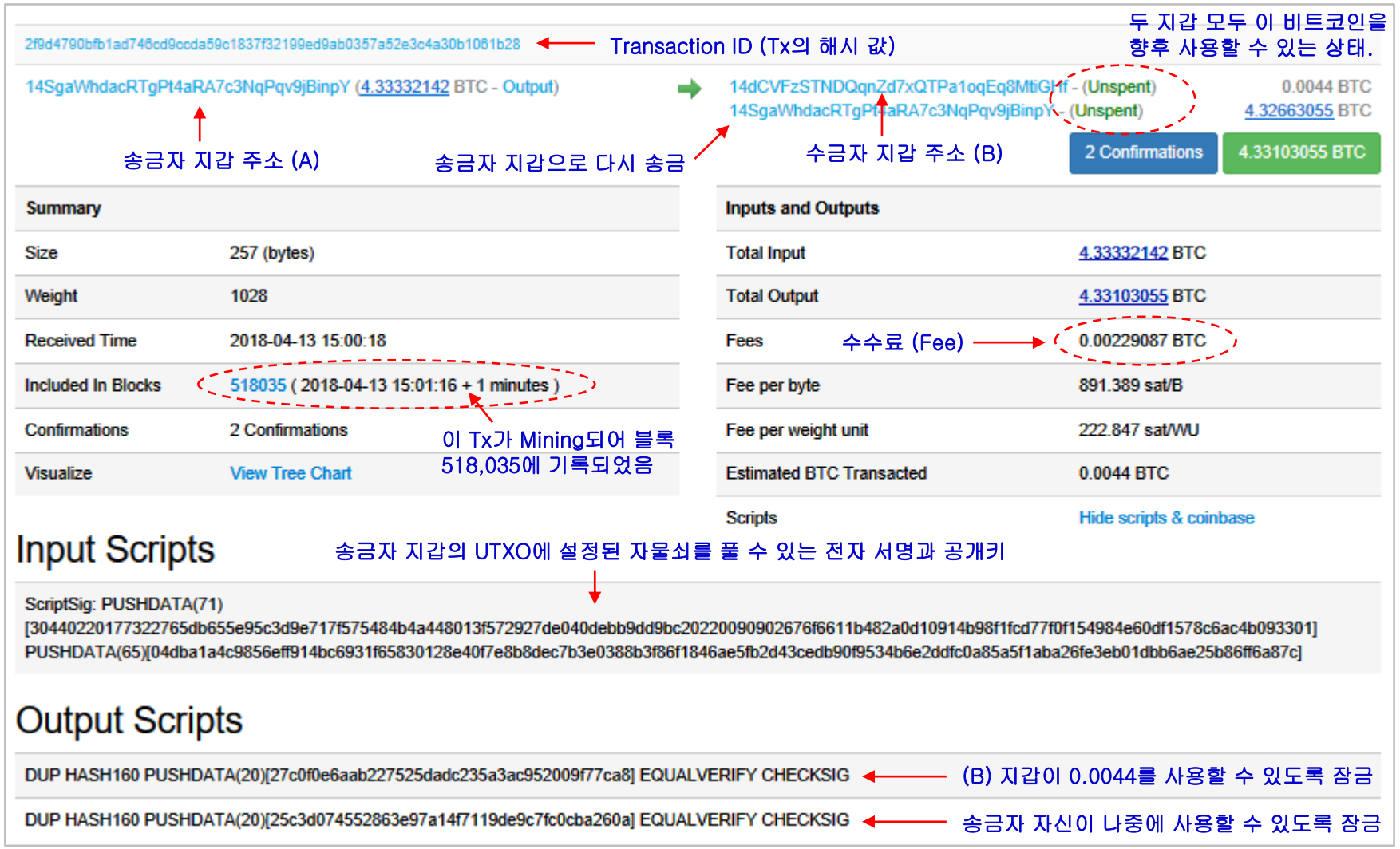
(6) API 서버도 Full 노드로 동작하므로 새로운 블록을 동기화함.

(7) Bob의 지갑은 자신의 거래가 완료되었는지 API 서버에 요청함.

* 이 경우는 보안상 취약점이 많기 때문에 비트코인의 표준은 아님. 단지 편리하기 때문에 간단히 구현한 일부 Wallet 들이 사용하는 경우도 있고, Bob이 스스로 간단한 지갑 S/W를 직접 만들어 사용하는 경우도 있음.

Transaction 실제 사례 확인

출처 : https://blockchain.info/tx/2f9d4790bfb1ad746cd9ccda59c1837f32199ed9ab0357a52e3c4a30b1061b28?show_adv=true



- 왼쪽 거래 원장은 14Sg… 지갑 (A)가 14dC… 지갑 (B)로 0.0044 BTC를 보내는 거래임.
- (A)가 사용할 수 있는 UTXO는 4.33332142 BTC이고, 이 중 0.0044는 (B)에게 보내고 수수료 0.00229087을 제외한 나머지 4.32663055는 다시 자신에게 송금한 거래임.
- (A)와 (B)는 모두 나중에 우측 상단의 금액만큼 사용할 수 있음 (Unspent 상태).
- 이 거래는 Miner에 의해 인증되어 블록 번호 518,035에 기록되었음.
- (A)는 자신의 UTXO에 기록되었던 4.3332142 를 사용하기 위해 Input Scripts로 잠금 장치를 풀었음. 이 UTXO는 Spent 상태로 더 이상 사용할 수 없음.
- (A)는 나중에 (B)가 0.0044를 사용할 수 있도록 Output Scripts로 잠가 놓았음. 또한 (A) 자신이 향후 4.32663055를 사용할 수 있도록 잠가 놓았음. (Script는 나중에 Transaction 편에서 자세히 다룸)

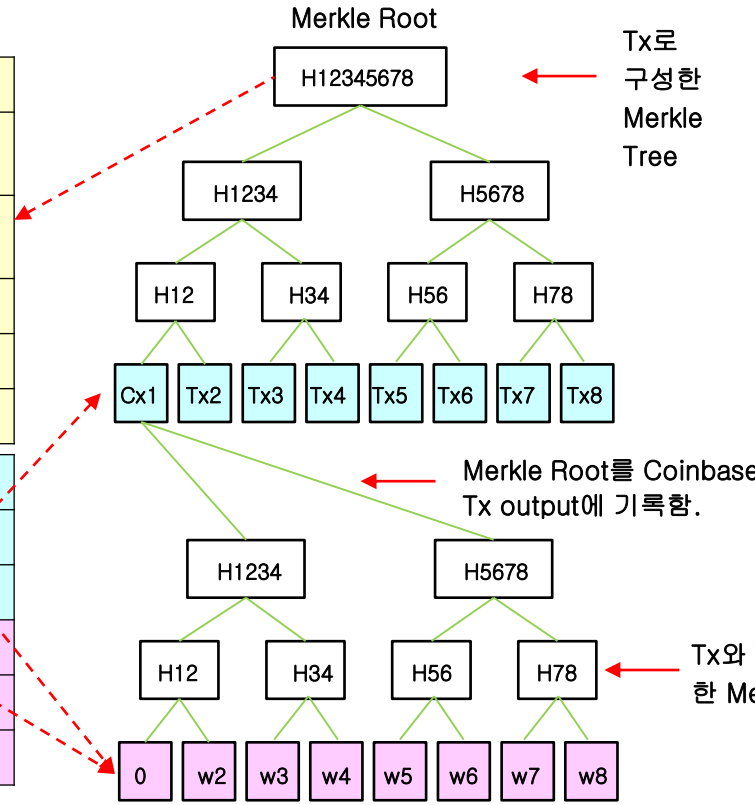
Mining (채굴) : Mining 방법

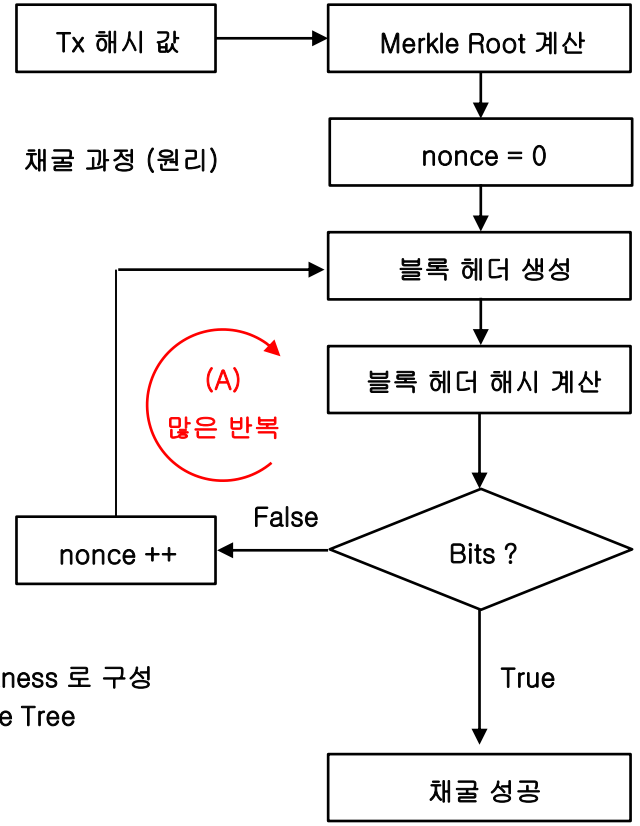
- Miner는 Tx 들을 모아서 아래와 같이 새로운 블록을 생성함. Tx 집합의 첫 번째는 Mining의 보상을 자신에게 송금하는 Tx 임. → Coinbase Transaction.
- Tx 들로 Merkle Root를 계산하고, 이전 블록의 해시 값 등을 이용하여 블록 헤더를 생성하고 (nonce는 초깃값 = 0), 블록 헤더의 해시 값을 계산함.
- 블록 헤더의 해시 값이 Bits 에서 제시하는 난이도에 만족하면 Mining에 성공한 것이고, 만족하지 못하면 nonce 값을 증가시켜 가면서 난이도 조건에 만족할 때까지 반복함. 헤더의 해시 값의 앞 부분에 '0'이 많이 붙어야 난이도 조건을 만족함. (난이도 조건은 나중에 Mining 편에서 자세히 다룸)
- 2017년 8월 SegWit이 도입되어 현재는 Coinbase Tx에 Witness용 Merkle Root가 추가로 붙어 있음 (SegWit에 대해서는 Transaction 부분에서 자세히 다룸)

블록 #517,707
블록 해시 : 00000000000000000003de39f5ad668699c0cbc2236585d585f0d99e….

Version	4 bytes	20000000
Previous Block Header Hash	32 bytes	00000000000000000002098ab5e2ac4d5a6f9cc3cc8c764dcaecd5a9c6290523
Merkle Root	32 bytes	19c6d65057762a53c153b681b3ad6d46edaf59aaf723ecbbd9eeb9b365ed6db6
Timestamp	4 bytes	5acd91de = 2018-04-11 13:41:02
Bits	4 bytes	17502ab7
Nonce	4 bytes	5c81558

Transaction count	4 bytes	05e8
Coinbase transaction		
Transactions		
Witness		
Witness		
Witness		





Mining (채굴)

- 아래 예시는 블록 518,035번의 블록 헤더와 Transaction임. 아래 결과는 3rd-party API 서버에서 제공한 것으로 실제 블록 데이터 이외에 정보가 추가로 표시되어 있음. (ex : 블록 헤더의 해시 값은 실제 기록되어 있지 않고 필요할 때 계산해서 사용함)
- 첫 번째 Transaction은 Mining의 보상을 Miner 자신에게 송금한 거래임 (Coinbase Tx). 현재 Block Reward는 12.5 BTC이고 약 4년 주기로 절반씩 줄어듦.

Block #518035

Summary	
Number Of Transactions	1421
Output Total	6,327.70781064 BTC
Estimated Transaction Volume	619.29817409 BTC
Transaction Fees	0.13837282 BTC
Height	518035 (Main Chain)
Timestamp	2018-04-13 15:01:16
Received Time	2018-04-13 15:01:16
Relayed By	ViaBTC
Difficulty	3,511,060,552,899.72
Bits	391129783
Size	1032.597 kB
Weight	3992.829 kWU
Version	0x20000000
Nonce	2900625785
Block Reward	12.5 BTC

Hashes	
Hash	0000000000000000004c9fefb4091b7f9065b9a3c5f959b7b6c609e180c3d407
Previous Block	0000000000000000004f3cb99e2c6e2815c2c14e358d42b31043aae1d8e18a9d
Next Block(s)	0000000000000000001e16c8cd32d93f2167f181204589d08471e185386ff83e
Merkle Root	89c8eabdbd223f4595fd04052123039624278877 32dcac7c7294b5bbd432156a

← 블록 헤더 (총 80 byte)

Transaction (총 1,42 개)

Transactions

48f47331b48b21595295281359083 4a03b097fe21399d5316bc773bfcf116af3 — 2018-04-13 15:01:16

No Inputs (Newly Generated Coins) → 18cBEMRxXHqz... (ViaBTC Bitcoin Mining Pool) — 12.63837282 BTC
Unable to decode output address — 0 BTC
12.63837282 BTC

Miner 자신에게 송금한 거래 : Coinbase Transaction Input이 없음. 비트코인 12.5 BTC가 신규 발행된 것임.

참고 : Witness Merkle Root 임.

a433861b1f31528fc8f18a7a897d1f33f775f04b7148e0c53559e8eeed2cbd75 — 2018-04-13 15:00:17

13MiXXPYin6ZjBm8wQP1AywDM7MKiM2ebB → 16gSwcGK7i6MADbwsTUBGhXojZpPZA95qw — 0.088773 BTC
13MiXXPYin6ZjBm8wQP1AywDM7MKiM2ebB — 3.77174923 BTC
3.86052223 BTC

2f9d4790bfb1ad746cd9ccda59c1837f32199ed9ab0357a52e3c4a30b1061b28 — 2018-04-13 15:00:18

출처 : https://blockchain.info/block/

(실습 파일 : 1-3.채굴자지갑.py)

Mining (채굴)

- 최근 생성된 100개 블록의 Coinbase Transaction 지갑 주소를 조회해 봄. 아래 지갑 주소들은 모두 Miner 소유일 것임.

```
# 3rd-party API 서버로부터 Coinbase Transaction을 수집하여
# Miner의 지갑을 관찰한다. 최근 몇 개의 블록만 조회해 본다.
#
# 3d-party API 서버 (예시) : https://blockchain.info
#
# 2018.4.14 아마추어 퀀트 (조성현)
# ------------------------------------------------------
import requests
import time
import pandas as pd
import matplotlib.pyplot as plt

# 마지막 block 번호를 조회한다
resp = requests.get(url='https://blockchain.info/latestblock')
data = resp.json()
nHeight = data['height']

# 마지막부터 몇 개 블록을 읽어서 coinbase transaction의 지갑 주소를 수집한다
mining = []

for n in range(nHeight, nHeight-100, -1):
    url = 'https://blockchain.info/block-height/' + str(n) + '?format=json
    resp = requests.get(url=url)
    data = resp.json()
    block = data['blocks'][0]

    stime = block['time']
    addr = block['tx'][0]['out'][0]['addr']
    value = block['tx'][0]['out'][0]['value']

    ts = time.gmtime(stime)
    date = time.strftime("%Y-%m-%d %H:%M:%S", ts)

```

```
#518272 : 2018-04-15 05:13:10   1C1mCxRukix1KfegAY5zQQJV7samAciZpv      12.641763
#518271 : 2018-04-15 05:00:37   1C1mCxRukix1KfegAY5zQQJV7samAciZpv      12.592176
#518270 : 2018-04-15 04:54:48   1CK6KHY6MHgYvmRQ4PAafKYDrg1ejbH1cE      12.523103
#518269 : 2018-04-15 04:54:18   1KFHE7w8BhaENAswwryaoccDb6qcT6DbYY      12.568512
#518268 : 2018-04-15 04:48:51   1Nh7uHdvY6fNwtQtM1G5EZAFPLC33B59rB      12.502625
#518267 : 2018-04-15 04:48:36   1C1mCxRukix1KfegAY5zQQJV7samAciZpv      12.632157
#518266 : 2018-04-15 04:40:47   1Hz96kJKF2HLPGY15JWLB5m9qGNxvt8tHJ      12.714512
#518265 : 2018-04-15 04:25:20   1C1mCxRukix1KfegAY5zQQJV7samAciZpv      12.500000
#518264 : 2018-04-15 04:23:34   1Nh7uHdvY6fNwtQtM1G5EZAFPLC33B59rB      12.531354
#518263 : 2018-04-15 04:22:44   1C1mCxRukix1KfegAY5zQQJV7samAciZpv      12.531992
#518262 : 2018-04-15 04:20:41   1C1mCxRukix1KfegAY5zQQJV7samAciZpv      12.624102
#518261 : 2018-04-15 04:09:36   1Nh7uHdvY6fNwtQtM1G5EZAFPLC33B59rB      12.575664
#518260 : 2018-04-15 04:03:19   13TEThZNnKPk34HYAuo1QqYMwDdjF3qeHx      12.624571
#518259 : 2018-04-15 03:56:31   1Nh7uHdvY6fNwtQtM1G5EZAFPLC33B59rB      12.590583
#518258 : 2018-04-15 03:50:49   1CK6KHY6MHgYvmRQ4PAafKYDrg1ejbH1cE      12.580678
#518257 : 2018-04-15 03:46:38   1CK6KHY6MHgYvmRQ4PAafKYDrg1ejbH1cE      12.643370
#518256 : 2018-04-15 03:34:04   1KFHE7w8BhaENAswwryaoccDb6qcT6DbYY      12.520926
#518255 : 2018-04-15 03:34:09   1C1mCxRukix1KfegAY5zQQJV7samAciZpv      12.535777
#518254 : 2018-04-15 03:31:40   1C1mCxRukix1KfegAY5zQQJV7samAciZpv      12.622676
#518253 : 2018-04-15 03:26:02   1Nh7uHdvY6fNwtQtM1G5EZAFPLC33B59rB      12.546474
#518252 : 2018-04-15 03:23:16   1C1mCxRukix1KfegAY5zQQJV7samAciZpv      12.629876
#518251 : 2018-04-15 03:12:30   1ACAgPuFFidYzPMXbiKptSrwT74Dg8hq2v      12.521845
#518250 : 2018-04-15 03:12:29   1CK6KHY6MHgYvmRQ4PAafKYDrg1ejbH1cE      12.644420
#518249 : 2018-04-15 03:01:37   1Hz96kJKF2HLPGY15JWLB5m9qGNxvt8tHJ      12.706411
#518248 : 2018-04-15 02:45:36   147SwRQdpCfj5p8PnfsXV2SsVVpVcz3aPq      12.745851
#518247 : 2018-04-15 02:27:15   18cBEMRxXHqzWWCxZNtU91F5sbUNKhL5PX      12.773150
#518246 : 2018-04-15 02:10:18   1J7FCFaafPRxqu4X9VsaiMZr1XMemx69GR      0.125477
#518245 : 2018-04-15 02:07:32   1CK6KHY6MHgYvmRQ4PAafKYDrg1ejbH1cE      12.566368
#518244 : 2018-04-15 02:03:54   13TEThZNnKPk34HYAuo1QqYMwDdjF3qeHx      12.543937
#518243 : 2018-04-15 02:01:51   147SwRQdpCfj5p8PnfsXV2SsVVpVcz3aPq      12.751417
#518242 : 2018-04-15 01:41:18   1ACAgPuFFidYzPMXbiKptSrwT74Dg8hq2v      12.579658
#518241 : 2018-04-15 01:37:12   18cBEMRxXHqzWWCxZNtU91F5sbUNKhL5PX      12.525600
#518240 : 2018-04-15 01:36:24   18cBEMRxXHqzWWCxZNtU91F5sbUNKhL5PX      12.655198
History log   IPython console
```

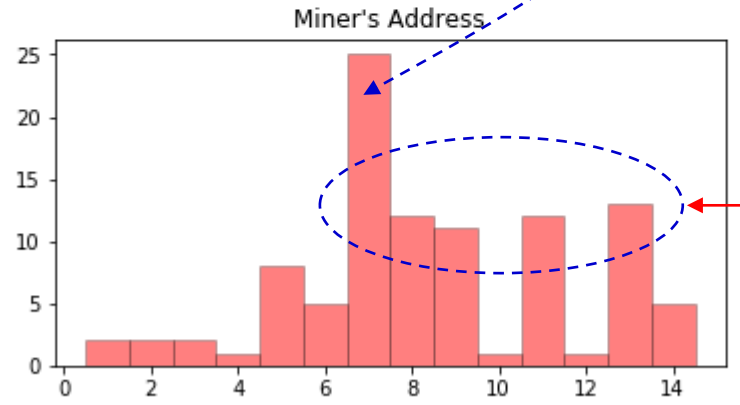
(실습 파일 : 1-4.채굴자지갑.py)

Mining (채굴)

- 최근 100개 블록을 생성한 지갑의 분포를 관찰함. 1C1mCxRukix1KfegAY5zQQJV7samAciZpv 지갑이 100개 블록 중 25개를 Mining 하였고 (25%), 5개 지갑이 75%를 차지하고 있음. → Mining의 독점 현상은 탈중앙화 목적에 반하는 현상임. (ASIC과 Pool Mining의 등장 때문임)

```
22      url = 'https://blockchain.info/block-height/' + str(n) + '?format=json'
23      resp = requests.get(url=url)
24      data = resp.json()
25      block = data['blocks'][0]
26
27      stime = block['time']
28      addr = block['tx'][0]['out'][0]['addr']
29      value = block['tx'][0]['out'][0]['value']
30
31      ts = time.gmtime(stime)
32      date = time.strftime("%Y-%m-%d %H:%M:%S", ts)
33
34      # 결과를 list에 저장한다
35      mining.append([date, addr, value])
36
37      # 중간 결과를 표시한다
38      print("#%d : %s\t%s\t%f" % (n, date, addr, value*1e-8))
39
40  # 결과를 dataframe에 저장한다
41  df = pd.DataFrame(mining, columns=['Date', 'Address', 'Reward'])
42
43  # 같은 지갑끼리 묶어본다
44  grp = df.groupby('Address').Address.count()
45  print()
46  print(grp)
47
48  # Histogram
49  plt.figure(figsize=(6,3))
50  plt.title("Miner's Address")
51  x = list(range(1, len(grp.values)+1))
52  plt.bar(x, grp.values, width=1, color="red", edgecolor='black', linewidth=0.5, alpha=0.5)
53  plt.show()
```

```
Address
13TEThZNnKPk34HYAuo1QqYMwDdjF3qeHx      2
13hQVEstgo4iPQZv9C7VELnLWF7UWtF4Q3      2
147SwRQdpCfj5p8PnfsXV2SsVVpVcz3aPq      2
14yd17779fySwxwPJVxLtdacpd1QMJhabo      1
18cBEMRxXHqzWWCxZNtU91F5sbUNKhL5PX      8
1ACAgPuFFidYzPMXbiKptSrwT74Dg8hq2v      5
1C1mCxRukix1KfegAY5zQQJV7samAciZpv     25
1CK6KHY6MHgYvmRQ4PAafKYDrg1ejbH1cE     12
1Hz96kJKF2HLPGY15JWLB5m9qGNxvt8tHJ     11
1J7FCFaafPRxqu4X9VsaiMZr1XMemx69GR      1
1KFHE7w8BhaENAswwryaoccDb6qcT6DbYY     12
1LayuyG7T2BptABEAJdkCkGRaidRNPDFNt      1
1Nh7uHdvY6fNwtQtM1G5EZAFPLC33B59rB     13
1Q7Jmho4FixWBiTVcZ5aKXv4rTMMp6CjiD      5
Name: Address, dtype: int64
```



이 지갑이 100개 블록 중 25개를 Mining 했음 (25%)

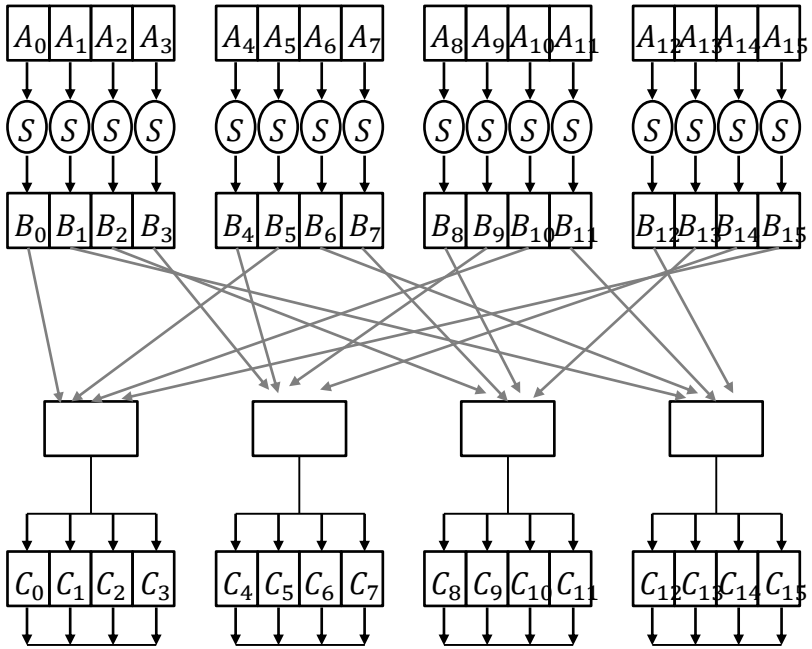
5개 지갑이 73%를 차지함

```
In [15]:
```

History log | IPython console

2. 암호학 (Cryptography)

2-1. 암호학의 역사
2-2. 대칭키 암호의 특징
2-3. 암호문의 요건
2-4. 대칭키 암호 (Block Cipher & Stream Cipher)
2-5. 대칭키 암호 동작 모드
2-6. DES 암호 알고리즘
2-7. AES 암호 알고리즘
2-8. 공개키 기반 암호 시스템
2-9. RSA 암호 알고리즘
2-10. Diffie-Hellman Key 교환 알고리즘
2-11. Elgamal 암호 알고리즘
2-12. Square-and-Multiply 알고리즘
2-13. 타원곡선 (ECC) 암호 알고리즘
2-14. 타원곡선 알고리즘에 의한 개인키와 공개키 생성
2-15. Hash 알고리즘
2-16. Digital Signature (전자서명)
2-17. 타원곡선 전자서명 (ECDSA) 알고리즘



암호학의 역사 : 고대 ~ 근대 암호학

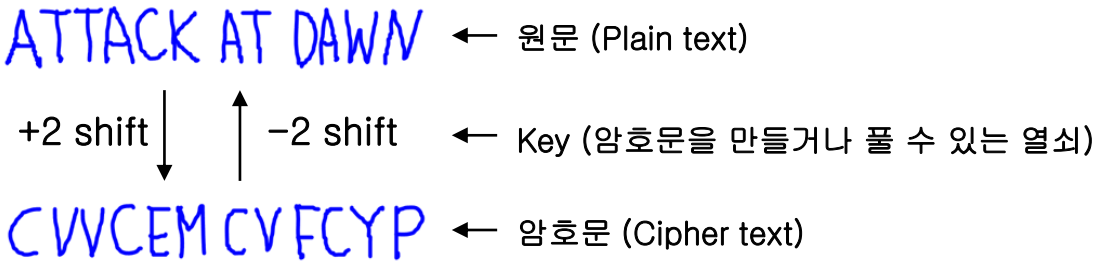
- 고대 암호 : 암호는 B.C. 3000년 전부터 사용되었다고 알려져 있음. 고대의 암호는 주로 다른 문자로 바꾸는 치환암호 (Substitution Cipher), 문자의 위치를 바꾸는 전치암호 (Transposition Cipher, Shift Cipher) 등이 사용되었음. (ex : Spartan Scytale, Julius Caesar)

Spartan scytale (Transposition Cipher)

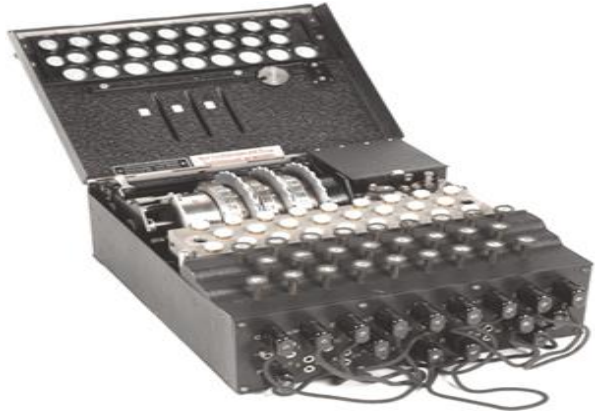
암호문
(Cipher text)

Key

Julius Caesar (Substitution Cipher)



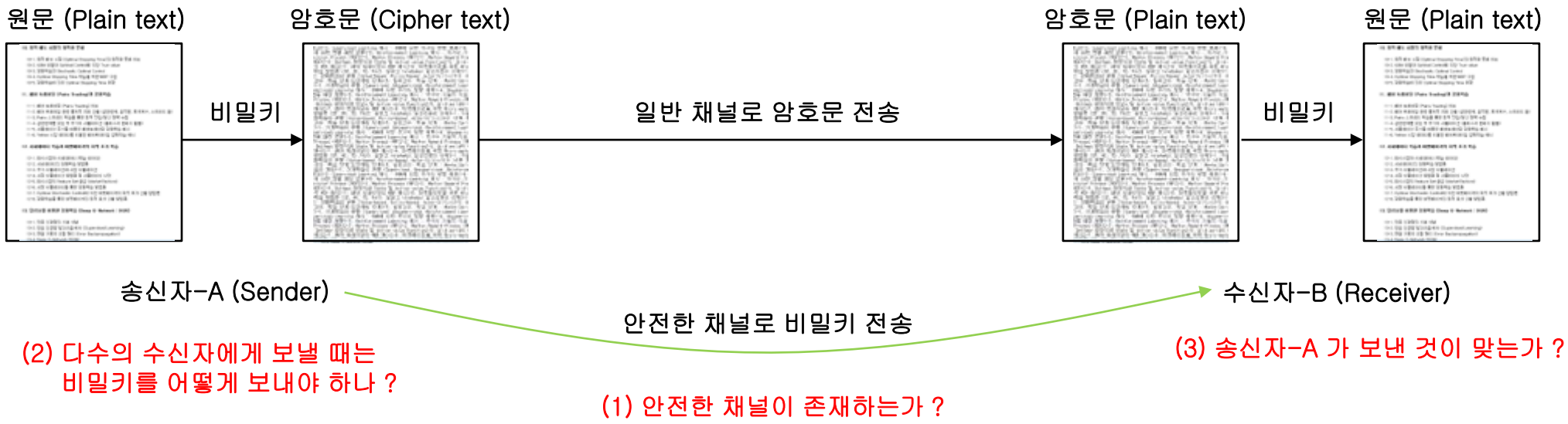
- 근대 암호 : 근대 암호는 세계대전과 컴퓨터의 발달로 더욱 정교하게 이루어 짐. 수학적 암호.
- 사례 : Jefferson disks (1790), The Hagelin Machine (1920), Enigma Machine (1919), Herbert Yardley's Black Chamber (1929) 등.
- 고대와 근대의 암호는 Key를 이용하여 원문 (평문)을 암호문으로 변환하고, 동일한 Key를 이용하여 암호문을 원문으로 변환함. Key는 비밀키 (Secret key)로 송신자와 수신자가 사전에 약속하여 가지고 있어야 함.
- Key가 복잡해 질수록 암호문을 해독하기 어려워 짐.
- 암호문과 Key를 동시에 수신자에게 보낼 경우, 암호문은 일반적인 루트로 보내고, Key는 특별히 안전한 루트로 보내야함 (Key가 노출되면 안됨). → 비밀키 방식의 암호 체계의 문제점.



(Enigma Machine 출처 : Wikimedia Commons)

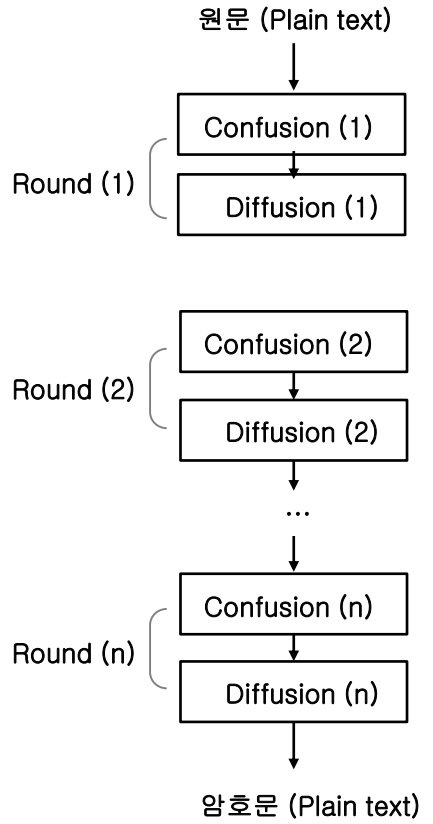
대칭키 암호의 특징 (Secret Key, Symmetric Cryptography : 비밀키)

- 송신자는 비밀키를 이용하여 원문을 암호문으로 변환한 후 수신자에게 암호문과 비밀키를 전송하고, 수신자는 비밀키로 암호문을 해독함.
- 비밀키는 한 번 노출되면 이후의 모든 암호문도 노출되는 것이므로, 송신자는 비밀키를 자주 바꿔야함. 비밀키는 매 번 바꾸는 것이 가장 안전함. (한 번만 사용)
- 비밀키는 충분히 복잡해야 함. 비밀키가 128 bit 인 경우보다 256 bit 이상인 경우가 훨씬 안전함.
- 암호문은 일반적인 채널 (네트워크)로 보내도 되지만, 비밀키는 전송 중간에 노출 (Sniffing)되지 않도록 안전한 채널로 보내야함.
- 현실적으로 완전히 안전한 채널은 존재하기 어려움. 비밀키를 수신자에게 보내야 한다는 것이 가장 큰 문제점 임.
- 송신자가 다수의 수신자에게 동일한 암호문을 발송하는 경우 동일한 비밀키를 전송하면 노출될 위험이 증가함. (타임 스탬프를 이용한 세션키를 사용하기도 함)
- 비밀키 방식은 정상적인 송신자가 보낸 것인지 확인할 수 있는 인증 (Authentication) 기능이 없음. 반대로 부인 방지 (Non-repudiation) 기능도 없음.
- 현대 암호학은 개인키 (Private Key) – 공개키 (Public Key) 페어 (Pair) 방식으로 위의 문제점들을 해결하였음.



암호문의 요건

- 정보이론 학자인 Claude Shannon은 강력한 암호문을 위해서는 Confusion과 Diffusion의 동작이 필요하다고 하였음.
- Caesar의 Shift cipher나, 2차 세계대전 독일의 Enigma는 Confusion 동작만 수행하였으므로 안전하지 못하였음.
- 현대의 암호문은 Confusion과 Diffusion 동작을 번갈아 가면서 반복적으로 수행함. → Product Cipher

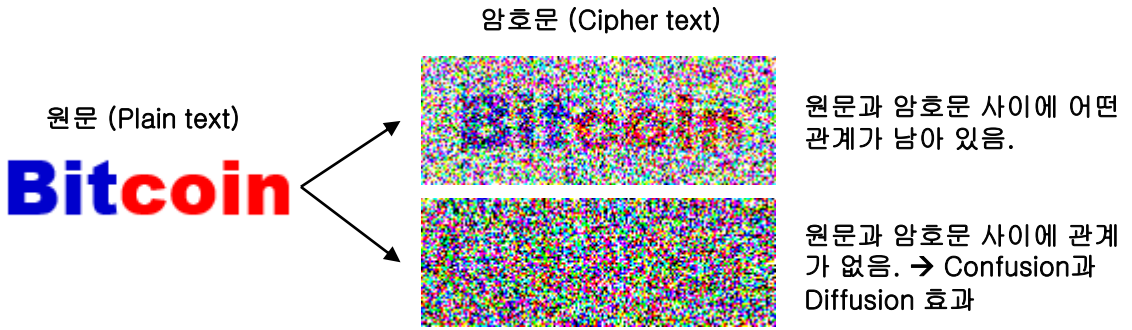


❖ Confusion

- 비밀키와 암호문 혹은 원문과 암호문의 관계가 모호해야 함. 비밀키 (혹은 원문)와 암호문에 어떤 관계가 존재하면 (ex: 상관관계) 암호문으로 비밀키나 원문을 추정할 소지가 있음.

❖ Diffusion

- 원문의 특정 위치의 내용은 암호문의 여러 위치에 영향을 미쳐야 함. 원문의 특정 위치의 내용이 암호문의 특정 위치에만 영향을 미치면 원문과 암호문 사이의 관계가 드러날 수 있음.
- 아래 예시에서 원문 (x)은 1 bit만 바뀌어도, 암호문 (y)은 여러 bit가 바뀌었음 (5 bit가 바뀜).



대칭키 암호 : Block Cipher와 Stream Cipher

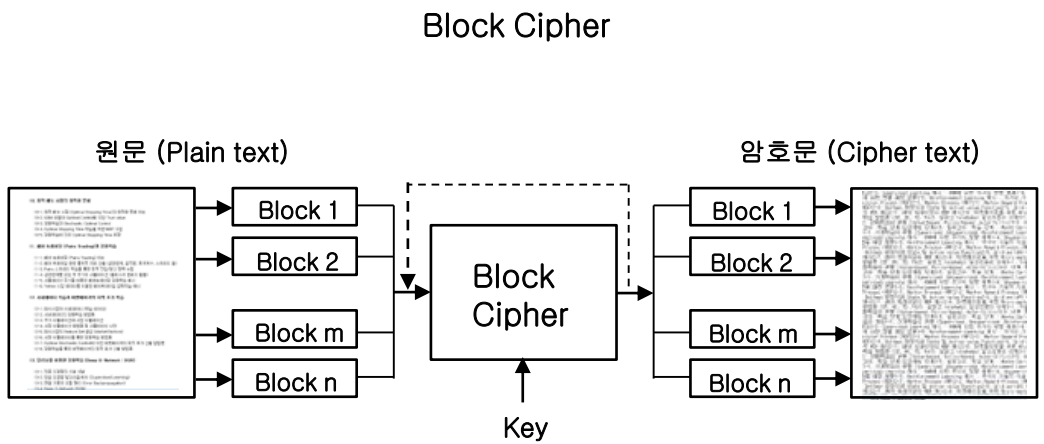
- 대칭키 암호 방식은 블록 암호 방식 (Block Cipher)과 스트림 암호 방식 (Stream Cipher)으로 나눌 수 있음.

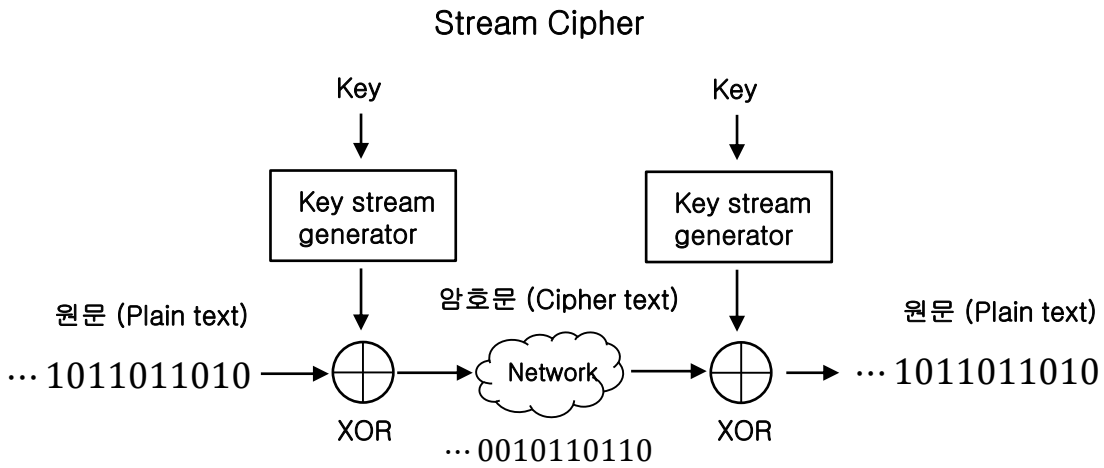
❖ 블록 암호 방식 (Block Cipher)

- 원문 (Plain text)을 여러 블록으로 나누어서 블록 별로 암호화 함. 일반적인 암호 방식임.
- 원문의 블록과 암호문의 블록이 1:1로 대응되게 할 수도 있고 (ECB 모드), 1:1로 대응되지 않고 섞이게 할 수도 있음 (CBC 모드).

❖ 스트림 암호 방식 (Stream Cipher)

- 원문의 내용을 비트 단위로 암호화함. 원문이 문서 형태가 아닌 실시간 데이터 스트림인 경우에 적합함.
- 예를 들어 음성 통화 (VoIP)내용을 암호화하거나 (비화기), 휴대폰과 기지국 사이의 신호 전달 등에 이용될 수 있음.



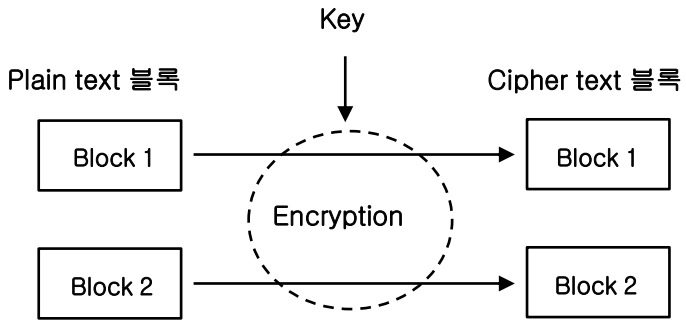


대칭키 암호 : 동작 모드

- 대칭키 암호 방식은 크게 결정적 (Deterministic) 방식과 확률적 (Probabilistic) 방식으로 나눌 수 있음.
- 결정적 방식은 동일한 키를 사용하는 경우 항상 동일한 암호문을 만드는 방식이고, 확률적 방식은 동일한 키를 사용해도 암호문이 바뀌는 방식임.
- 결정적 방식에는 ECB 방식이 있고, 확률적 방식에는 CBC, OFB, CFB 방식이 있음.

❖ Electronic Code Block (ECB)

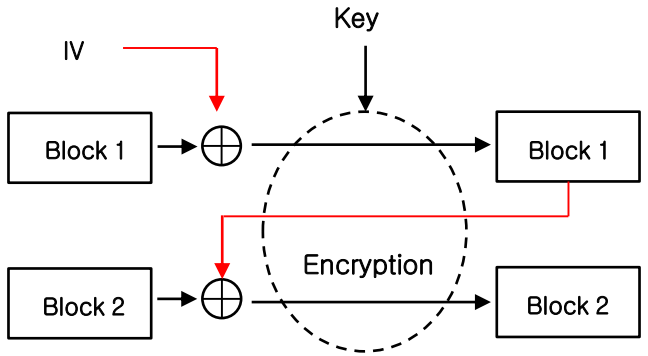
- 결정적 방식의 Block Cipher로 원문의 블록과 암호문의 블록이 1:1로 대응됨.
- 블록들이 서로 독립적이므로 병렬 처리가 가능하고 전송 중 일부 블록이 누락되어도 정상적으로 수신한 블록들은 복호화 (Decryption)가 가능함.
- 블록 단위의 Diffusion이 없으므로 취약할 수 있음.



[ECB]

❖ Cipher Block Chaining (CBC)

- 확률적 방식의 Block Cipher로 원문의 블록과 암호문의 블록이 1:1로 대응되지 않음.
- 원문의 첫 번째 블록은 Initialization Vector (IV)와 섞어서 암호화되고, 두 번째 블록부터는 이전의 암호화된 블록과 섞어서 암호화됨. 블록들이 체인 (Chain)으로 연결됨.
- IV는 랜덤하게 생성할 수도 있고 (주로 사용됨), 카운터가 사용될 수도 있음.
- 복호화할 때도 IV가 필요하므로 암호문과 함께 IV도 수신자에게 보내야함.
- 병렬 처리는 곤란하지만 블록 간 Diffusion이 존재하므로 안전함.



[CBC]

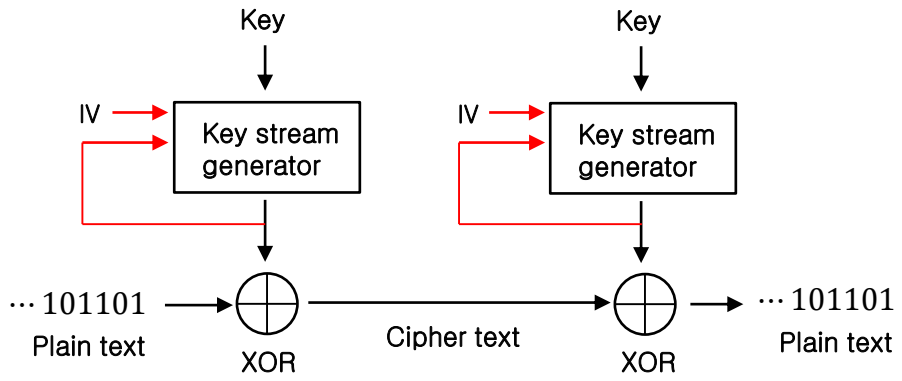
대칭키 암호 : 동작 모드

❖ Output Feedback (OFB)

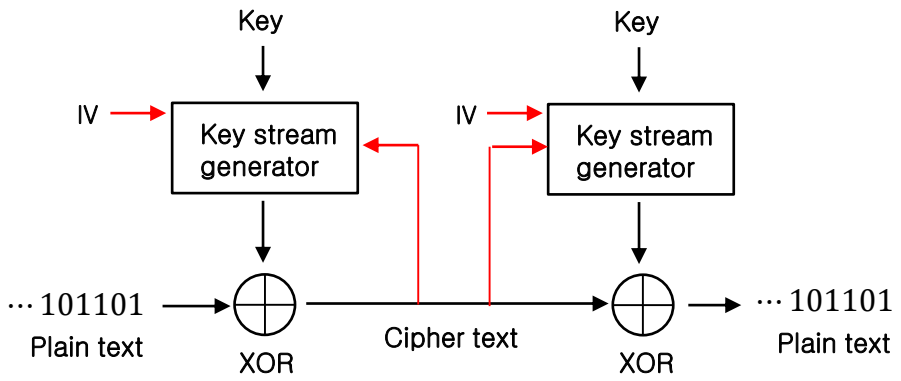
- 확률적 방식의 Stream Cipher임.
- Key는 블록 단위로 생성되고, 최초 Key 블록에는 IV가 사용되고, 두 번째 Key 블록부터는 이전의 Key 블록이 Feedback되어 사용됨.
- 블록 단위로 구성된 Key의 각 비트는 Plain text의 각 비트와 XOR로 연산됨.
- 복호화할 때도 동일한 과정을 수행하고, Key와 IV는 송신자와 수신자가 공유해야 함.

❖ Cipher Block Chaining (CFB)

- OFB 방식과 유사한 확률적 방식의 Stream Cipher임.
- Key는 블록 단위로 생성되고, 최초 Key 블록에는 IV가 사용되고, 두 번째 Key 블록부터는 이전의 Cipher 블록이 Feedback되어 사용됨.
- 복호화할 때도 동일한 과정을 수행하고, Key와 IV는 송신자와 수신자가 공유해야 함.



[OFB]



[CFB]

비밀키 (대칭키) 기반 암호 알고리즘 : Data Encryption Standard (DES)

❖ DES의 역사

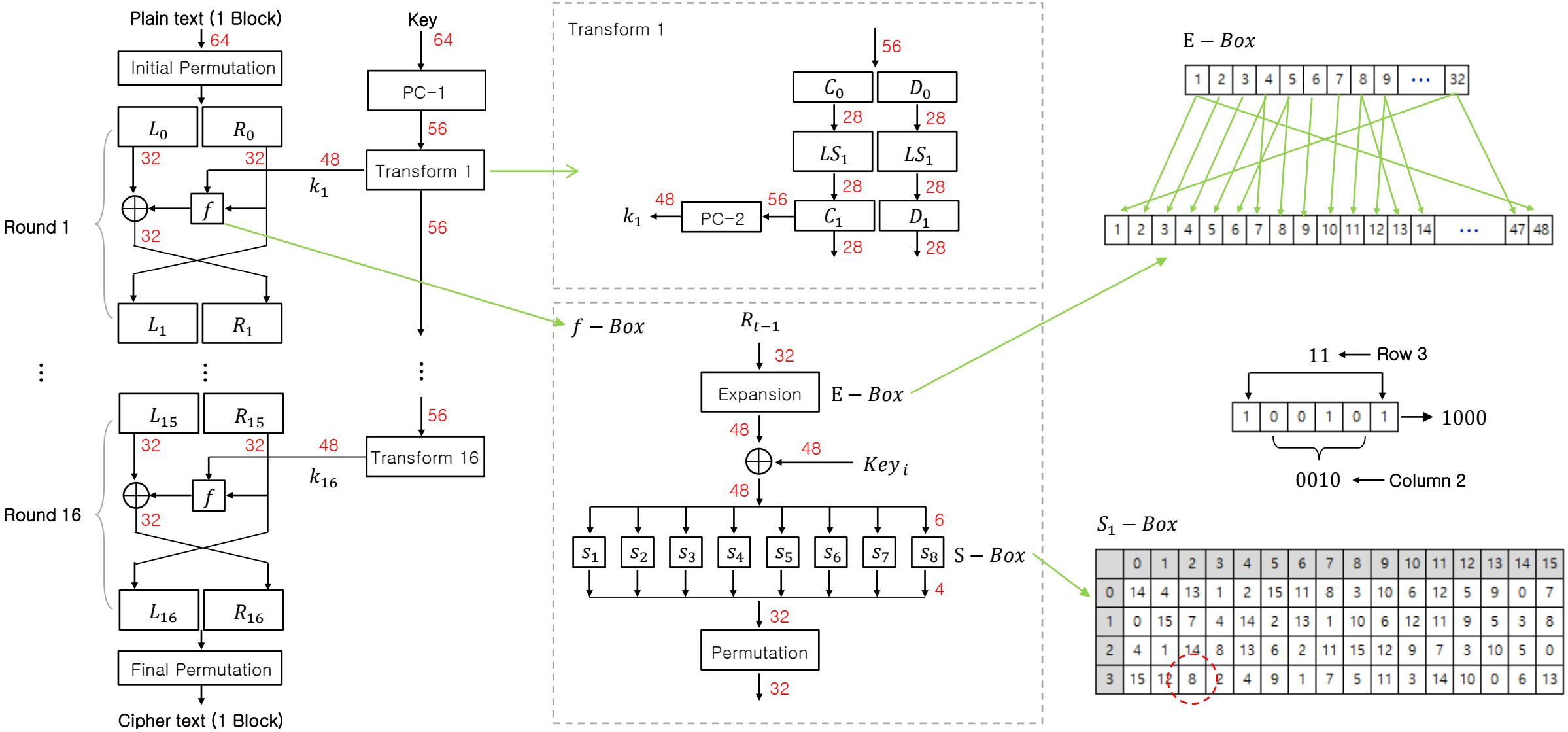
- 1972년 미국 NIST에서 표준 암호 제정을 위해 제안을 요청하였고, IBM이 루시퍼라는 알고리즘을 제안함.
- 1974 ~ 1977년 IBM에서 제안한 루시퍼 알고리즘을 수정하여 미국 암호 표준으로 채택하였고 (Data Encryption Standard), 1998년 까지 널리 사용됨.
- 1999년 이후 DES 암호의 취약성이 대두되면서 새로운 암호 표준이 요구되었음. → Advanced Encryption Standard (AES)
- DES는 56 bit의 비밀키를 사용하였고 키의 크기가 작은 취약점 때문에 3DES가 사용되었음. 3DES는 3개의 키로 DES를 세 번 수행하므로 매우 안전함.

❖ DES의 동작 원리 (다음 페이지 그림 참조)

- DES는 블록 암호 방식 (Block Cipher)으로 원문을 64 비트의 블록으로 나누고, 각 블록에 56 비트 키를 사용하였음 (56 bit + 8 parity bit = 64 bit).
- DES는 Feistel Network 구조임. 64 비트의 원문을 32 비트씩 나누고 (Left, Right), Left 측 32 비트를 서브키로 암호화한 후 Left와 Right를 교차하여 다음 Round의 Feistel로 보냄. 이 과정을 16번 반복 (Round 1~16)함.
- Feistel 구조는 암호화 과정과 복호화 과정이 동일하다는 장점이 있음.
- 비밀키는 64 비트를 생성하여 8번째 비트 (패리티 비트)를 빼고 56 비트만 사용함. 56 비트의 키를 48 비트의 서브 키로 변환하여 각 Round에서 사용함.
- 각 Round의 서브키와 원문 블록의 Right 측은 $f-Box$에서 섞이고 $f-Box$의 결과로 원문 블록의 Left 측을 암호화함.
- $f-Box$로 인해 Confusion과 Diffusion이 발생함.
- $f-Box$는 $E-Box$와 8개의 $S-Box$로 구성되어 있음. $E-Box$는 32 비트를 48 비트로 확장하고, $S-Box$는 6비트를 4비트로 변환함.

참고 사항 : DES 내부 구조

(출처 : Christof Paar, 2010, Understanding Cryptography, Chapter 3)



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
1	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

(실습 파일 : 2-1.DES(ECB).py)

비밀키 (대칭키) 기반 암호 알고리즘 : DES 연습 (ECB 모드 Block Cipher)

- DES 알고리즘, ECB 모드로 원문을 암호문으로 변환한 후, 다시 암호문을 원문으로 변환함. (대칭키 = 56 bit, 블록 크기 = 64 bit)

```
1  # Data Encryption Standard (DES) 알고리즘 연습
2  # ECB (Electronic Code Book) 모드로 암호화한다. (Plaintext 블록과 Cipher
3  #
4  # 2018.2.23
5  # 아마추어 퀀트 (조성현)
6  # ------------------------------------------------------------------
7  from Crypto.Cipher import DES
8
9  # 64 bit (8-byte)의 대칭키를 만든다. 처음 8-bit는 parity 용이고, 다음 56-
10 secretKey = b'01234567'
11
12 # 암호화할 원문
13 plainText = 'This is Plain text. It will be encrypted using DES with
14 print("\n\n")
15 print("원문 : ", plainText)
16
17 # ECB 모드에서는 plain text가 64-bit (8 byte)의 배수가 되어야 하므로, pa
18 # padding으로 NULL 문자를 삽입함. 수신자는 별도로 padding을 제거할 필요없음
19 n = len(plainText)
20 if (n % 8) != 0:
21     n = n + 8 - (n % 8)
22     plainText = plainText.ljust(n, '\0')
23
24 # 송신자는 secretKey로 plainText를 암호문으로 변환한다.
25 des = DES.new(secretKey, DES.MODE_ECB)
```

```
원문 :  This is Plain text. It will be encrypted using
DES with ECB mode.

                    Key ──→ Encryption

암호문 :  b'\x84g\xe8\x0ey\xf0\xfd\xfa\xb0\xe7k
\x98\x16\xa7=O\xca\x90?\x94\xcf\x9dyB)\xcfm\xa6\xf4I
\xd1\xd4\xd2IlL\xd4\xc1\xb4\xd1\x1a\xa9P%\x86\xc1\xcdW
\x13?j|\xeb\xd3\x9d\x00\xfd\x0f\xad7j\x04\x03\x7f\xbb
\xe6\x9a\xf7g\x04\xda\x97'

                    Key ──→ Decryption

해독문 :  This is Plain text. It will be encrypted
using DES with ECB mode.

In [82]:
```

(실습 파일 : 2-2.DES(CBC).py)

비밀키 (대칭키) 기반 암호 알고리즘 : DES 연습 (CBC 모드 Block Cipher)

- DES 알고리즘, CBC 모드로 원문을 암호문으로 변환한 후, 다시 암호문을 원문으로 변환함. (대칭키 = 56 bit, IV = 64 bit, 블록 크기 = 64 bit)

```
1  # Data Encryption Standard (DES) 알고리즘 연습
2  # CBC (Cipher Block Chain) 모드로 암호화한다. (Plaintext 블록과 Cipherte
3  #
4  # 2018.2.23
5  # 아마추어 퀀트 (조성현)
6  # ------------------------------------------------------------------------
7  from Crypto.Cipher import DES
8  from Crypto import Random
9  import numpy as np
10
11 # 64 bit (8-byte)의 대칭키를 만든다. 처음 8-bit는 parity 용이고, 다음 56-
12 secretKey = b'01234567'
13
14 # 암호화할 원문
15 plainText = 'This is Plain text. It will be encrypted using DES with
16 print("\n\n")
17 print("원문 : ", plainText)
18
19 # CBC 모드에서는 plain text가 64-bit (8 byte)의 배수가 되어야 하므로, pa
20 # padding으로 NULL 문자를 삽입함. 수신자는 별도로 padding을 제거할 필요없음
21 n = len(plainText)
22 if (n % 8) != 0:
23     n = n + 8 - (n % 8)
24     plainText = plainText.ljust(n, '\0')
25
```

```
원문 :  This is Plain text. It will be encrypted using
DES with CBC mode.

                    Key ──→ Encryption

암호문 :  b'\xb6F\xdd\xad\xcd\x83Roo-?\xb9><\xa8n\xbc
\xa5\xac\x9f&\x93\xde\x04\xd2JX\x1a\x8fv/\x88\xfb"\x0b
\xe0\x07V\tl\x91{\xbb\x99\xf1K\xee\x01\xd0\xeaw
\x87\xc2\xf4\xf4&\xdd\xa8e\xf0!\x07\x8b\xa0\xc2\xfd
\x0cTR\xd1\xd3)'

                    Key ──→ Decryption

해독문 :  This is Plain text. It will be encrypted
using DES with CBC mode.

In [83]:
```

비밀키 (대칭키) 기반 암호 알고리즘 : Advanced Encryption Standard (AES)

❖ AES의 역사

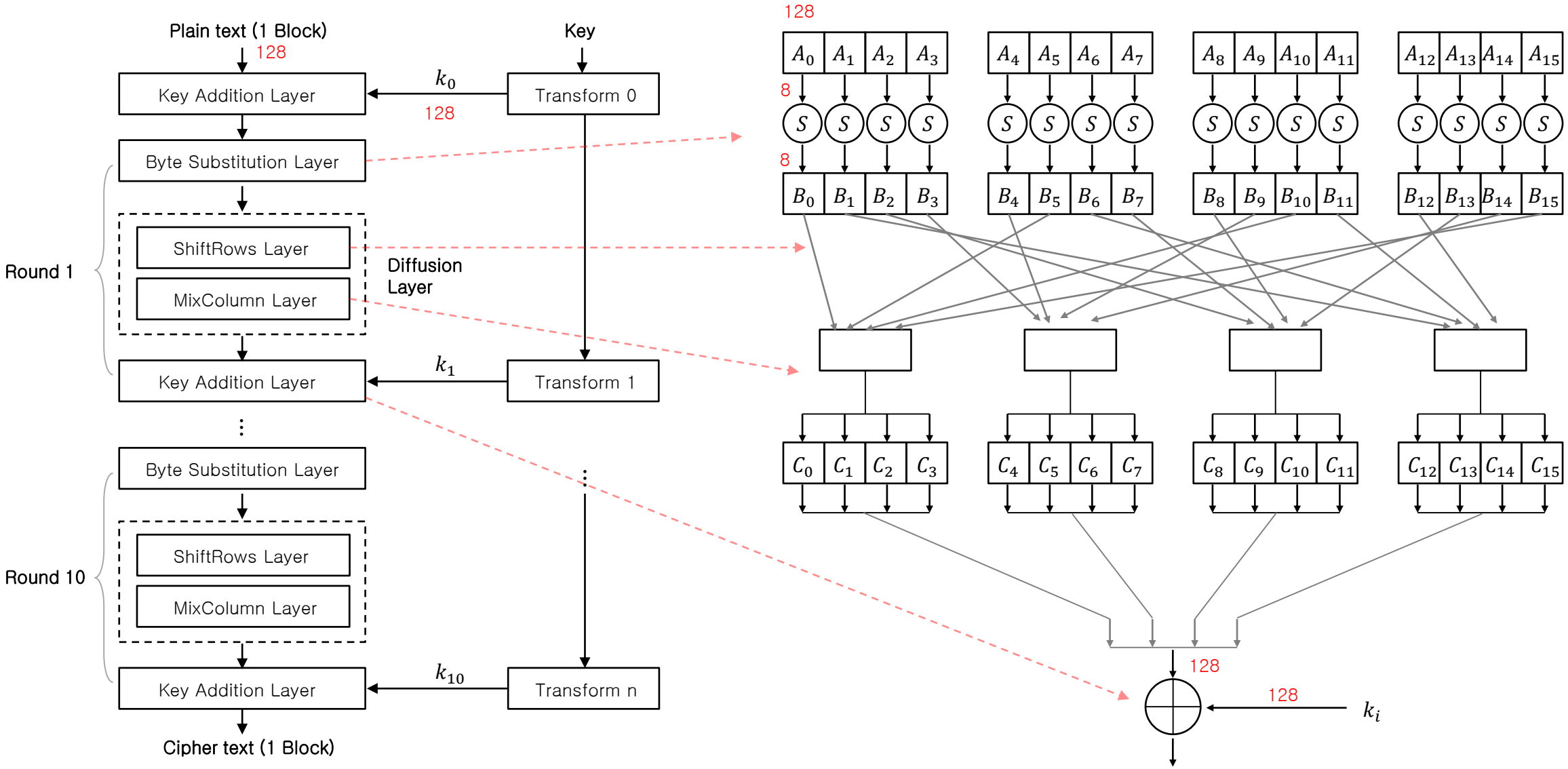
- 1997년 미국 NIST에서 DES를 대체하기 위해 새로운 Advanced Encryption Standard의 제안을 요청하였음.
- 1998년 15개의 AES 후보 알고리즘이 제출되었고, 1999년 이 중 5개 알고리즘을 선정하였음. (MARS, RC6, Rijndael, Serpent, Twofish)
- 2000년 NIST는 두 명의 벨기에 암호학자인 존 대먼과 빈센트 라이먼에 의해 개발된 Rijndael 알고리즘을 최종 AES로 선정하였음.
- 2001년 미국 정부는 공식적으로 AES 암호 표준 (Rijndael)을 제정하였음.

❖ AES의 동작 원리 (다음 페이지 그림 참조)

- DES는 Feistel Network 구조인 반면, AES는 Substitution-Permutation Network (SPN) 구조임.
- AES는 블록 암호 방식 (Block Cipher)으로 원문을 128 비트의 블록으로 나누고, 각 블록에는 128, 192, 256 비트의 키가 사용될 수 있음.
- AES의 키는 세 가지가 사용될 수 있으며, 이 중 128 비트 키가 표준으로 채택되었음. 56 비트 키인 DES 방식보다 안전함.
- AES 알고리즘은 10개의 Round로 구성되어 있으며, 각 Round는 Key Addition layer, Byte Substitution layer, Diffusion layer로 구성되어 있음.
- Key Addition layer는 128 비트 키와 128 비트의 데이터 블록이 XOR로 섞이는 과정임.
- Byte Substitution layer는 16개의 S-Box 테이블를 이용하여 Confusion 과정을 수행함.
- Diffusion layer는 ShiftRows layer와 MixColumn layer로 Diffusion 과정을 수행함.

참고 사항 : AES 내부 구조

(출처 : Christof Paar, 2010, Understanding Cryptography, Chapter 4)



(실습 파일 : 2-3.AES(CBC).py)

비밀키 (대칭키) 기반 암호 알고리즘 : AES 연습 (CBC 모드 Block Cipher)

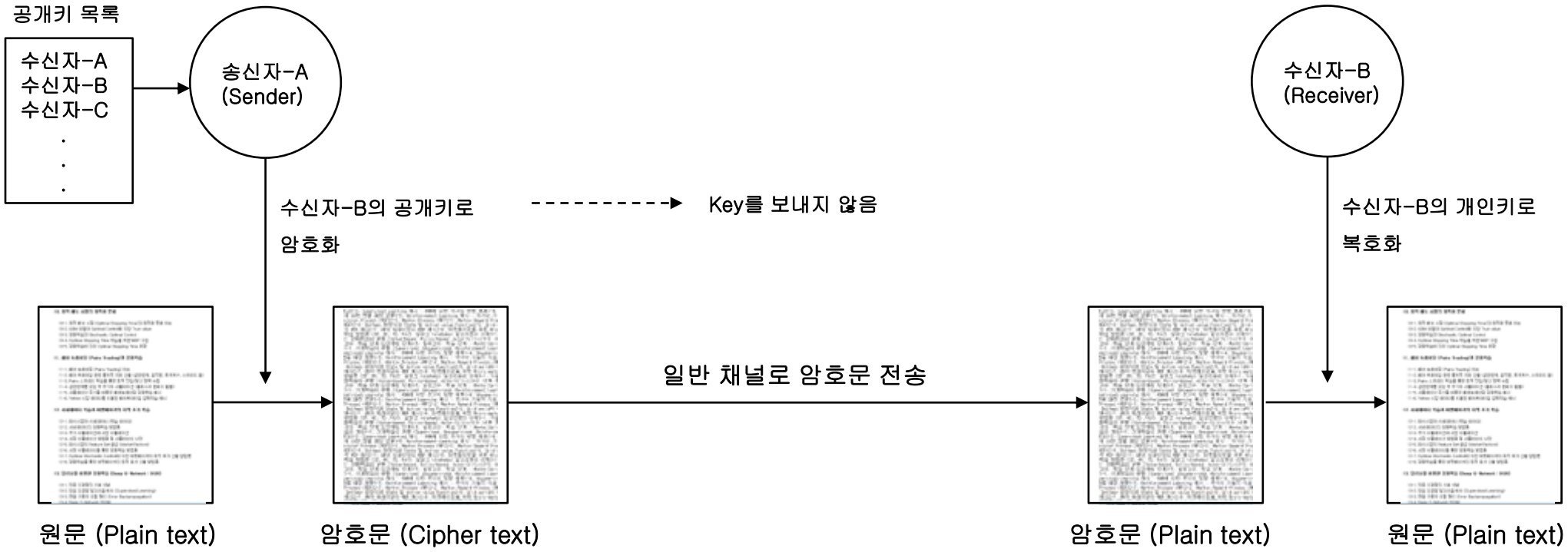
- AES 알고리즘으로 원문을 암호문으로 변환한 후, 다시 암호문을 원문으로 변환함. (대칭키 = 256 bit, 블록 크기 = 128 bit 사용 예시)

```
1  # Advanced Encryption Standard (AES) 알고리즘 연습
2  # CBC (Cipher Block Chain) 모드로 암호화한다. (Plaintext 블록과
3  #
4  # 2018.2.23
5  # 아마추어 퀀트 (조성현)
6  # ------------------------------------------------------------
7  from Crypto.Cipher import AES
8  from Crypto import Random
9  import numpy as np
10
11 # 대칭키를 만든다. 대칭키는 128-bit, 192-bit, 256-bit를 사용할 수
12 secretKey128 = b'0123456701234567'
13 secretKey192 = b'012345670123456701234567'
14 secretKey256 = b'0123456701234567012345670123456701234567'
15
16 # 256-bit key를 사용한다.
17 secretKey = secretKey256
18 plainText = 'This is Plain text. It will be encrypted using A
19 print("\n\n")
20 print("원문 : ", plainText)
21
22 # CBC 모드에서는 plain text가 128-bit (16 byte)의 배수가 되어야
23 # padding으로 NULL 문자를 삽입함. 수신자는 별도로 padding을 제거할
24 n = len(plainText)
25 if (n % 16) != 0:
```

```
원문 :  This is Plain text. It will be encrypted using AES
with CBC mode.

                                Key ──→ Encryption

암호문 :  b'EsD\xaa\x85>\xf6.%FH0Zxm\xc88\xe3\x8bZ\x08\x862BF
\xd4?\xa7\xb2\x91\xb7\x07B\x88\xe1\x04\xfa\xee\x85\x8c\xff
\xb0\xa9\x08\xb8 |i"\x87\xaes\x9a@A\xa2\x04\x93\xae\x07\xaeP*
\xcd\x13M63\x86\x8c0m\xbc\xe4\x8c\xb2\xc7y\x13``

                                Key ──→ Decryption

해독문 :  This is Plain text. It will be encrypted using AES
with CBC mode.

In [8]:
```

공개키 (비대칭키) 기반 암호 시스템 – Public (Asymmetric) Key Cryptosystems

- 공개키 방식은 개인키 (Private Key)와 공개키 (Public Key) 쌍을 이용함. 송신자-A가 수신자-B에게 암호문을 보내고자 할 때, 수신자-B의 공개키로 암호문을 만들어 수신자-B에게 보냄. 수신자-B는 자신의 개인키로 암호문을 해독 (복호화)할 수 있음. Key를 보내지 않기 때문에 대칭키 방식보다 안전함.
- 수신자-B의 공개키로 암호화한 문서는 해당 공개키와 쌍을 이루는 수신자-B의 개인키로만 해독할 수 있음. (수신자-C는 해독할 수 없음)
- 공개키 암호 방식은 암호학 역사에서 가장 큰 혁명적 업적으로 평가되고 있음.
- 개인키로 암호화한 문서는 공개키로 해독할 수 있고 (전자 서명에 사용), 공개키로 암호화한 것은 개인키로 해독할 수 있음.
- 현재, 공인인증서, 전자상거래 시스템, 웹보안, 이메일 보안등에 널리 사용되고 있으며, 블록체인 (Block Chain)에도 사용되고 있음.

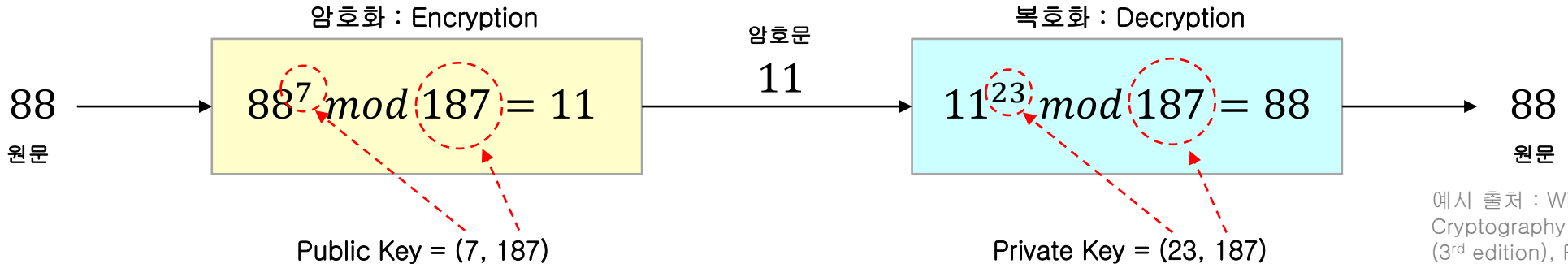


공개키 (비대칭키) 암호 알고리즘 : RSA

- 개인키와 공개키 쌍은 수학적 근거로 생성함. (정수론, 유한체, 타원곡선군에 대한 이산대수 등의 복잡한 수학적 이론에 근거함.)
- 한 개의 키로 다른 키를 추정할 수 없음. 개인키는 소유자가 보관하고, 공개키는 여러 사람에게 공개함. 비트코인의 경우 지갑의 공개키는 블록체인에 공개됨.
- 개인키/공개키를 이용하면 대칭키 방식의 3가지 문제점을 모두 해결할 수 있음.

❖ RSA 알고리즘

- Rivest, Shamir, and Addleman에 의해 제안된 것으로 어떤 수를 두 개의 큰 소수 (Prime number)로 분해하는 것이 어렵다는 원리를 이용한 것임.
- RSA 알고리즘 예시 : 이 알고리즘을 제대로 이해하려면 정수론의 페르마, 오일러의 정리 등을 이해해야 함.
  - RSA 알고리즘 예시 :
  - 두 개의 (큰) 소수를 선정함 (p, q) : $p = 17, q = 11$
  - $n = pq = 17 * 11 = 187$
  - $\varphi(n) = (p-1)(q-1) = 16 * 10 = 160$
  - $\varphi(n) = 160$ 과 서로소 관계에 있으며 $\varphi(n)$ 이하인 정수 한 개 (e)를 선택함. $e = 7$
  - $de \equiv 1 \, mod \, 160, and \, d < 160$ 인 d 를 선택함. $d = 23$. $\leftarrow 23 * 7 = 161 = 10 * 160 + 1$ 이므로 $d = 23$은 올바른 선택임.
  - $Public\ Key = (e, n) = (7, 187),\ Private\ Key = (d, n) = (23, 187)$



예시 출처 : William Stallings, 2003, Cryptography and Network Security (3rd edition), P.270

(실습 파일 : 2-4.RSA.py)

공개키 (비대칭키) 암호 알고리즘 : RSA – Python 실습

- RSA 알고리즘으로 개인키와 공개키를 만들고 문서를 암호화하는 과정을 Python 프로그램으로 확인함.

```python
# Public Key (RSA) 알고리즘 연습
#
# 2018.2.23
# 아마추어 퀀트 (조성현)
# ------------------------------------------------------
from Crypto.PublicKey import RSA

# Private key와 Public key 쌍을 생성한다.
# Private key는 소유자가 보관하고, Public key는 공개한다.
keyPair = RSA.generate(2048)
privKey = keyPair.exportKey()   # 키 소유자 보관용
pubKey = keyPair.publickey()    # 외부 공개용

# keyPair의 p,q,e,d를 확인해 본다
keyObj = RSA.importKey(privKey)
print("p = ", keyObj.p)
print("q = ", keyObj.q)
print("e = ", keyObj.e)
print("d = ", keyObj.d)

# 암호화할 원문
plainText = 'This is Plain text. It will be encrypted us
print()
print("원문 : ", plainText)

# 공개키로 원문을 암호화한다.
cipherText = pubKey.encrypt(plainText.encode(), 10)
print("\n")
print("암호문 : ", cipherText)

# Private key를 소유한 수신자는 자신의 Private key로 암호
# pubKey와 쌍을 이루는 privKey 만이 이 암호문을 해독할 수
```

```
p =
136074375738021708199693599551883592714528779714081039723827867750172017283701993053658534116313834480069
134991647598875702320659444888142842109673137788523038756860715827504622348167411115117841143305367855865
717963214921496597647999304954111882210808561513299452127112624103909055274286263170882237976338789
q =
144141369425193203458000248063646444502723141194162140180039416210635091445323021314673953254200786252196
906761041330850583417305093413138111757954324347757617608395319826788165180514815040721436372293789727028
659841110662410875461817945778308011603434483488494225952512162758839511777609498713440746499308837
e =  65537
d =
142487451382627540246188492363604881901525401195955996075614740812858573324108797903763331286483239410064
780686251630652757822771637584798682745672963090331778001777607281144228884352157714155242563008972367161
314826277767066888545019153625584837567532632157591686981262539040543986763691115725887180427660350415306
276453127122305554679714259695516314408204477234248210512679136846250112434014248912750053935555125589910
803930276447350858822791213251584028288126387574500612947067814895131436865071117764399941721880231894422
686442301737053887273052379492712178903812768145315135040506266143600304466328822100716737 7

원문 :  This is Plain text. It will be encrypted using RSA.

암호문 :  (b"VN?V\xfc2\xa9\x11\xa6\x87\x14B.\x07\x1dO);\xfcG\xf3[\xe7iq\xbc?\xc8`\xa9\xeb\x04\x132U\x19J
\xfc\x03\xa8\xd7\xb9Q\xbc%\xd8v\xdd\xfd\xf7\x86\xb5l\xeb\xfb\xc8\xb2\xb7B\xa2\xa88\x1d\xeb/\x14\xfa\x9e
\x02p\xc8#1e\xc0\x05(\xf2\x1d\xec\xc9\xc1\x90\xac\xab\x02!c\xe9\xdf%\xec_\x99FF\xe8y%p\xe4\xf0G\x197c
\x93\xd1\xedv\x85\xb0\xf1\xcd?\xdfOU\x1e\xbf\xa9\xe5fnB3\xe2\xd9\xb8W`\xd4\x16\xa1(\xf6\xaa\xd4\x8c0mM
\xf9\xbap\x03\xd5Op\xff\xf1\xd8\xe0\xfa\xe4T\x8b\x86`\xa6\x1c\xc3F\xcd\x0f\x8dM<\xe5av\x9b,
\xf6\xe6\x10\xb8\x94\xe0\x01<\xc4#N\xc5\xf19\xcd\xe24`\x82\xff\xea\xce\xcf\xb8\xc7}Q\xad\x15\x7f
\x94\x87\xef\x99\x99\xe5\x0b'!\xe2^DU)\xbd\xd1\xa7c\x90\x13|\xc4\xcc\x13r7\xdf\xc8\xfff\xbc~\rc\xd8m
\x86\xb4\xd7((\xdb\x0cS\x02\xb4\xe3/\xc5\x07\xe8\xdd\xb6",)

해독문 :  This is Plain text. It will be encrypted using RSA.
```

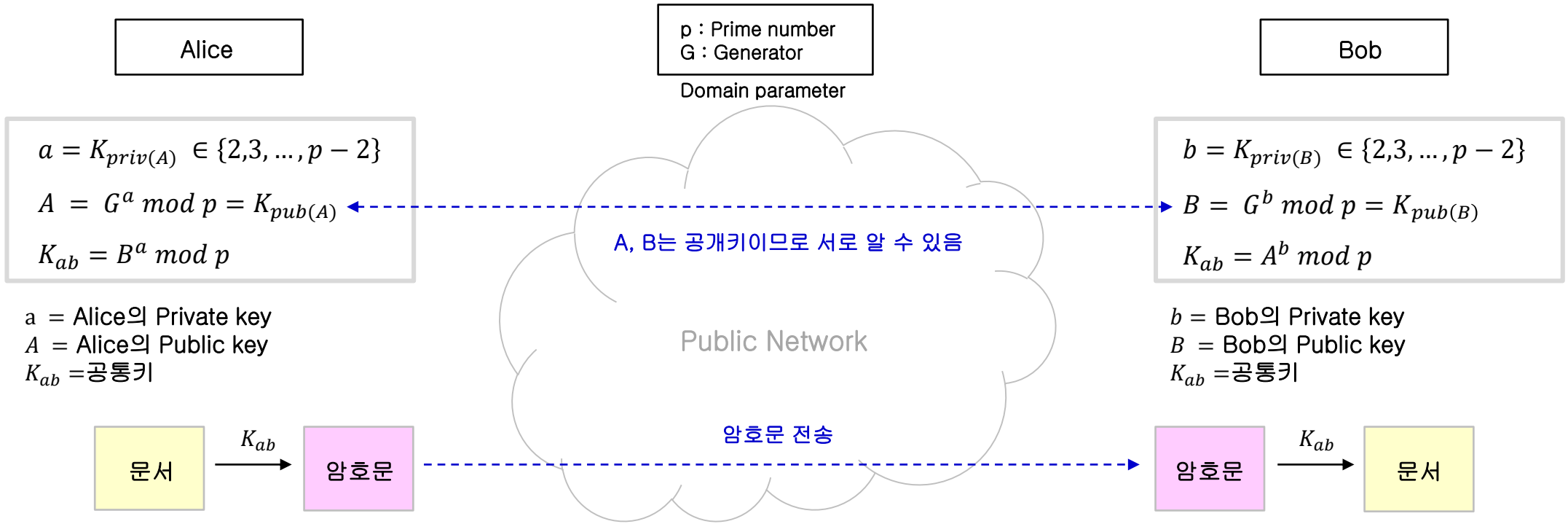
Encryption

Decryption

History log | IPython console

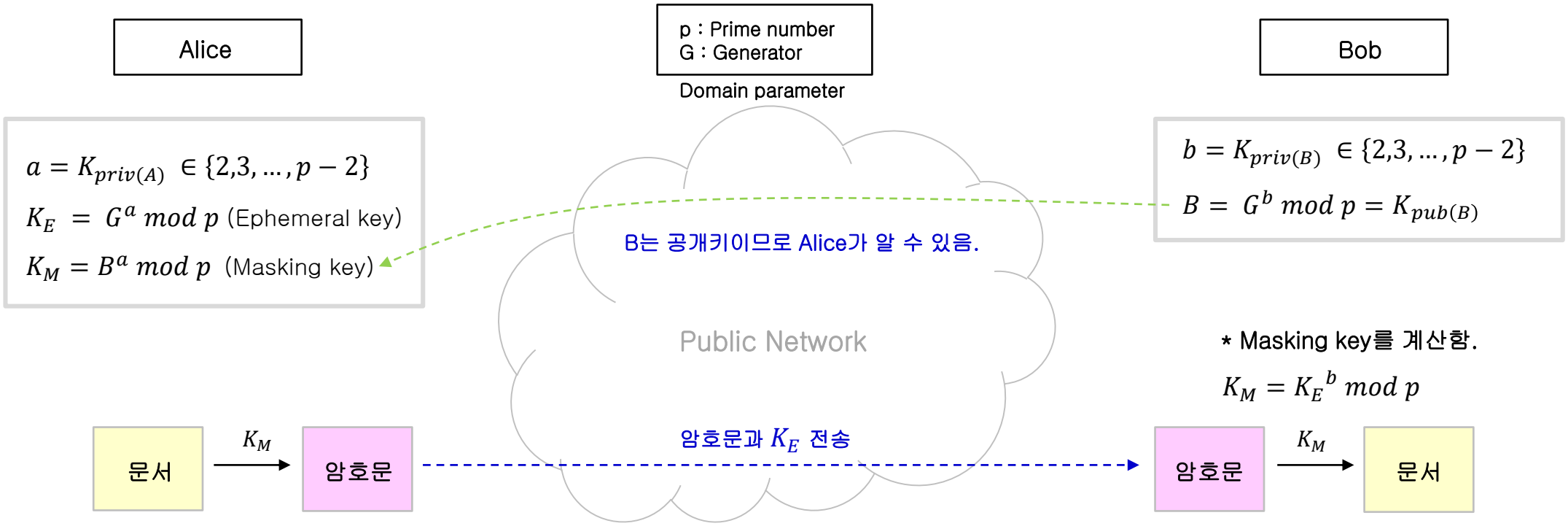
공개키 (비대칭키) 암호 알고리즘 : Diffie-Hellman Key Exchange (DHKE)

- 1976년 W.Diffie와 M.E. Hellman에 의해 개발된 키 교환 알고리즘임. 개인키와 공개키를 이용하면 각자의 키를 이용하여 공통키를 만들 수 있음.
- 송신자와 수신자는 사전에 정의된 정보를 (Public parameter : p = prime number, G = generator) 이용하여 각자의 비밀키와 공개키를 만들 수 있음.
- Public parameter와 각자의 공개키는 모두에게 알려진 키이고, 개인키는 자신만 알고 있는 키임.
- 상대방의 공개키와 자신의 개인키를 이용하면 공통키를 만들 수 있고, 공통키로 문서를 암호화하면, 공통키를 보내지 않고도 암호문을 해독할 수 있음.
- 공통키는 서로 만들 수 있으므로, 공중망으로 전송할 필요가 없음.



공개키 (비대칭키) 암호 알고리즘 : Elgamal

- 1985년 Taher Elgamal이 제안한 알고리즘임. DHKE와 유사하지만 암호문을 보낼 때마다 키를 바꿀 수 있어 더욱 안전한 방식임.
- Domain parameter를 이용하여 자신의 개인키를 만들고 (DHKE와 동일함), 암호문을 보낼 때 사용할 임시키 (Ephemeral key)와 상대방의 공개키를 이용하여 (Bob 의 공개키) 문서를 암호화할 키 (Masking key)를 만듦.
- Masking key로 문서를 암호화한 후 암호문과 임시키를 상대방에게 보냄. 상대방은 암호문과 함께 받은 임시키와 자신의 개인키를 이용하여 암호문을 복호화할 Masking key를 만듦. 송신자와 수신자의 Masking 키는 동일하므로 수신자는 암호문을 풀 수 있음.
- 암호문을 보낼 때마다 임시키가 달라지므로 동일한 문서를 보내도 보낼 때마다 암호문을 달라짐. (Probabilistic encryption scheme)



공개키 (비대칭키) 암호 알고리즘 : 공개키 $(G^a \ mod \ p)$ 계산 – Square-and-Multiply 알고리즘 (Modular Exponentiation)

- 큰 수에 대해 공개키를 계산하는 Modular 연산은 Square-and-Multiply (or Modular Exponentiation) 알고리즘을 사용함. 이 알고리즘의 세부 원리는 정수론 (수학)을 참조해야 함. Square-and-Multiply 알고리즘을 이용하면 아래 예시와 같이 몇 번의 스텝 만으로도 쉽게 결과를 얻을 수 있음.
- 아래 예시에서 승수인 234는 Private Key를 의미하고, 계산 결과는 Public Key를 의미함. Private Key로 Public Key를 쉽게 계산할 수 있음.
- 그러나 Public Key로 Private Key를 계산하는 것은 매우 어려움. Brute Force 방식으로 알아내는 것은 현실적으로 불가함.

$$3^{234} \ mod \ 15 = ?$$

$$234_{(10)} = 11101010_{(2)}$$

- exponent 234를 2진수로 변환함.
- 좌측 비트부터 우측으로 가면서 Square & Multiply 알고리즘 진행
- 첫 번째 비트는 base인 3으로 초기화
- 두 번째 비트부터 진행.
- 비트가 '1' 이면 Square (이전 결과의 제곱 mod 15) 계산 후 Multiply (이전 결과 * base mod 15)
- 비트가 '0' 이면 Square만 계산.
- 마지막 비트까지 진행하면 끝.
- 정답은 9 임.

$bit \ 1 \ (1) : 3 \ \text{mod} \ 15$	Initialize
$\text{bit} \ 2 \ (1) : \ 3^2 \ mod \ 15 = 9 \ mod \ 15$	Square
$9 \ * 3 \ mod \ 15 = 12 \ mod \ 15$	Multiply
$bit \ 3 \ (1) : \ 12^2 \ mod \ 15 = 9 \ mod \ 15$	Square
$9 \ * 3 \ mod \ 15 = 12 \ mod \ 15$	Multiply
$bit \ 4 \ (0) : 12^2 \ mod \ 15 = 9 \ mod \ 15$	Square
$bit \ 5 \ (1) : \ 9^2 \ mod \ 15 = 6 \ mod \ 15$	Square
$6 \ * 3 \ mod \ 15 = 3 \ mod \ 15$	Multiply
$bit \ 6 \ (0) : 3^2 \ mod \ 15 = 9 \ mod \ 15$	Square
$bit \ 7 \ (1) : 9^2 \ mod \ 15 = 6 \ mod \ 15$	Square
$6 \ * 3 \ mod \ 15 = 3 \ mod \ 15$	Multiply
$bit \ 8 \ (0) : 3^2 \ mod \ 15 = 9 \ mod \ 15$	Square

$$x = Private \ Key$$

$$3^x \ mod \ 15 = 9 \ = Public \ Key$$

- x=234 라면 Square-and-Multiply 알고리즘으로 결과=9 를 쉽게 계산할 수 있음.
- 그러나 결과=9를 아는 상태에서 큰 수인 x 를 구하는 것은 대단히 어려운 일임.
- x 는 Private Key이고, 결과=9는 Public Key 임.
- Private Key를 알고 있으면 Public Key를 쉽게 계산할 수 있지만, Public Key를 알고 있어도 Private Key는 쉽게 계산할 수 없음.

(실습 파일 : 2-5.modularExp.py)

공개키 (비대칭키) 암호 알고리즘 : Square & Multiply 알고리즘 (Modular Exponentiation) – Python 실습

- 큰 수에 대해 공개키를 계산하는 과정을 Python 프로그램으로 확인함. (Left-to-Right 방식과 Right-to-Left 방식)

```python
# 1) Modular exponentiation (Square & Multipy) 알고리즘 연습
#    built-in 함수인 pow(base, exp, p)와 비교해 본다
#
# 2) 이 알고리즘으로 Private Key와 Public Key를 생성한다
#    Private Key로 Public Key를 생성하는 것은 쉬우나,
#    Public Key로 Private Key를 알아내는 것은 매우 어렵다. why ?
#
# 2018.4.6
# 아마추어 퀀트 (조성현)
# ----------------------------------------------------------------
import random

# Left-to-Right 방식
def modExp(base, exp, p):
    bits = bin(exp)
    bits = bits[2:len(bits)]

    # initialize. bits[0] = 1 (always)
    r = base

    # 두 번째 비트부터 square & multiply
    bits = bits[1:len(bits)]
    for bit in bits:
        # Square
        r = (r * r) % p

        # Multiply
        if bit == '1':
            r = (r * base) % p
    return r

# Right-to-Left 방식
# 참조 : https://en.wikipedia.org/wiki/Modular_exponentiation 의 psuedo code
```

Name	Size	Type	Date Modified
2-4.RSA.py	1 KB	py File	2018-03-07 오후 11:16
2-5.modularExp.py	2 KB	py File	2018-04-06 오전 4:07

Variable explorer | File explorer

IPython console

Console 1/A

$$1234567890^{78948454537162537 45} \; mod \; 12345678 = ?$$

```
# Square & Multiply algorithm test :
1234567890 ** 78948354537162537 45 mod 12345678 = ?
1) Modular Exp (Left to Right) =  10168668
2) Modular Exp (Right to Left) =  10168668
3)              Built-in pow() =  10168668

# Private Key, Public Key generation test :
Private Key =  0x1f10622a69ab20442e1ebf8f0e3b31bfa25c744e639b9c0fe424e89f5d726f52
 Public Key =  0xedd27bd4089a50a53f4b44472cd9dacc35a5a22a9832e4dd7e894b1872607fdf

In [13]:
```

계산 결과

256 bit의 큰 수를 랜덤하게 생성하여 Private Key로 사용하고, 위의 방식으로 Public Key를 생성한 예시. 실제, 비트코인의 Public Key는 타원곡선 암호(ECC) 방식으로 생성함.

History log | IPython console

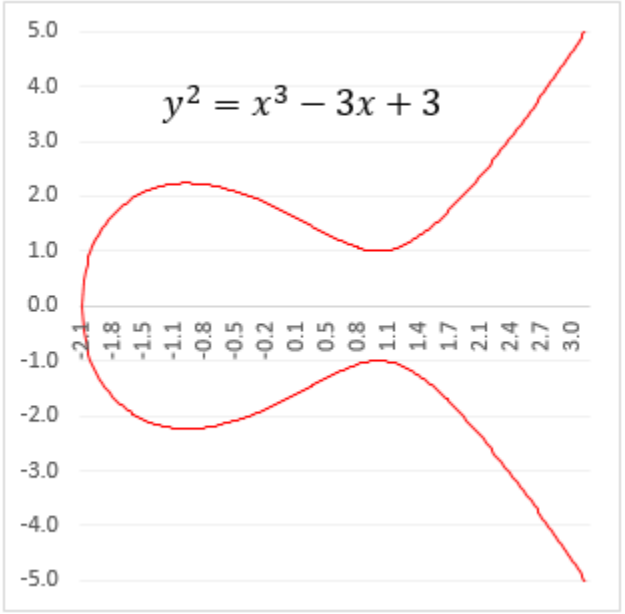
(실습 파일 : 2-6.타원곡선그래프.xlsx)

공개키 (비대칭키) 암호 알고리즘 : 타원곡선 암호 (Elliptic Curve Cryptography : ECC)

- 타원곡선 암호 (ECC)는 1985년 Neal Koblitz 와 Victor S. Miller에 의해 제안된 방식으로, 작은 수로 키를 만들어도 RSA 만큼의 성능을 지닌 우수한 암호 방식으로 알려져 있음. 160~256 비트의 ECC로도 1024~3072 비트의 RSA 정도의 성능을 보임. 계산 속도도 빠르고 네트워크의 전송 성능도 좋아짐.
- ECC는 $y^2 = x^3 + ax + b$ 형태의 함수식을 사용하고 (실제 타원 함수는 아님), 아래와 같은 모습임 ($a = -3, b = 3$ 인 경우). 우측 그림은 스케일을 크게 그린 것임. 단, 이 그림은 실수 영역 (x, y, a, b 모두 실수)에서 그린 것이며, 실제 ECC는 $y^2 = x^3 + ax + b \pmod{p}$ 로 유한체 ($Finite\ Field, x, y, a, b$ 모두 자연수) 상의 점들로 구성함. 여기서 유한체 상의 점들을 간단히 그릴 수는 없고, 뒤에서 mod p 에 대한 (Cyclic) Group의 점들을 모두 찾아 그려볼 것임 (Python 예제).
- ECC는 아래 곡선상의 점들을 연산하는 규칙을 정의하여 (덧셈 규칙) 특정 위치 (Generator G)의 점을 여러 번 더해서 Private Key와 Public Key를 만듦.
- 다른 알고리즘과 마찬가지로 Private Key로 Public Key를 쉽게 만들 수 있지만, 거꾸로 Public Key로는 Private Key를 만들 수는 없음.

Elliptic Curve in real number

x	x^3 - 3x +3	y	-y
-2.1038	0.0000	0.0001	-0.0001
-2.0838	0.2030	0.4506	-0.4506
-2.0638	0.4011	0.6333	-0.6333
-2.0438	0.5942	0.7708	-0.7708
-2.0238	0.7824	0.8845	-0.8845
-2.0038	0.9657	0.9827	-0.9827
-1.9838	1.1442	1.0697	-1.0697
-1.9638	1.3180	1.1480	-1.1480
-1.9438	1.4870	1.2194	-1.2194
-1.9238	1.6514	1.2851	-1.2851
-1.9038	1.8111	1.3458	-1.3458
-1.8838	1.9663	1.4023	-1.4023
-1.8638	2.1170	1.4550	-1.4550
-1.8438	2.2632	1.5044	-1.5044
-1.8238	2.4050	1.5508	-1.5508

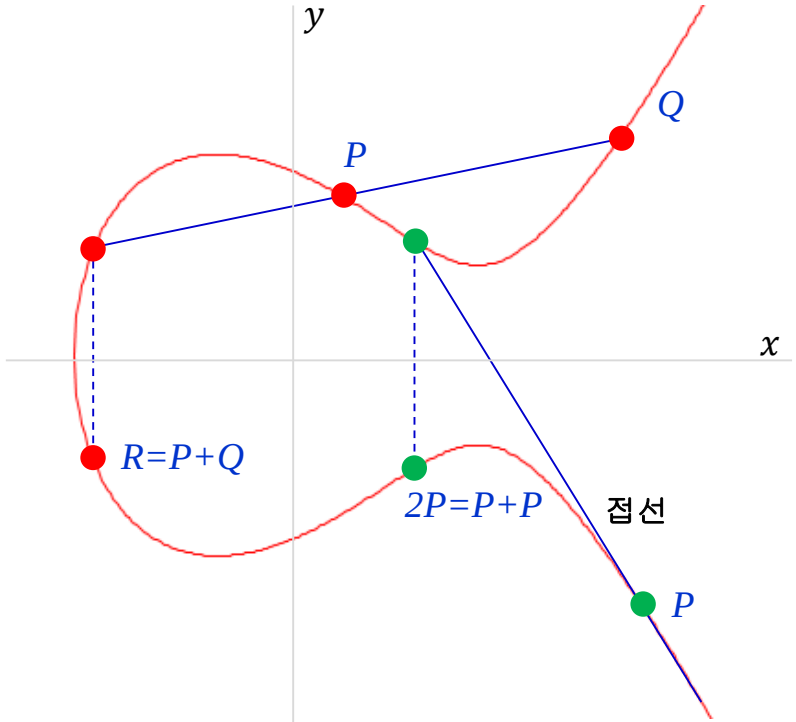


Elliptic Curve in real number

x	x^3 - 3x +3	y	-y
-2.1	0.0	0.0	0.0
-1.7	3.2	1.8	-1.8
-1.3	4.7	2.2	-2.2
-0.9	5.0	2.2	-2.2
-0.5	4.4	2.1	-2.1
-0.1	3.3	1.8	-1.8
0.3	2.1	1.5	-1.5
0.7	1.2	1.1	-1.1
1.1	1.0	1.0	-1.0
1.5	1.9	1.4	-1.4
1.9	4.1	2.0	-2.0
2.3	8.2	2.9	-2.9
2.7	14.5	3.8	-3.8
3.1	23.4	4.8	-4.8
3.5	35.2	5.9	-5.9

공개키 (비대칭키) 암호 알고리즘 : 타원곡선 암호 – 덧셈 연산자

- ECC에서 덧셈 연산자는 아래와 같이 정의함. EC (Elliptic Curve) 상의 서로 다른 두 점 P와 Q의 덧셈은, P와 Q를 잇는 선을 연장하여 EC와 만나는 점을 구하고, 그 점이 x 축과 대칭인 점 R로 정의함 $(P + Q = R)$. → Addition 연산자
- 한 점 P에 대해 P를 두 배 하려면 (2P), P + P로 표시하고, 아래 그림과 같이, 점 P에 접하는 접선이 EC와 만나는 점을 구하고, 그 점이 x 축과 대칭인 점으로 정의함. → Doubling 연산자
- 기하학적으로 점 R과 2P를 구하기 위해서는 각 선의 기울기를 구해야 하고, 그 직선과 EC의 교점을 구하면 됨.
- 단, 아래의 기하학적 해석은 실수 세계에서 직관적으로 이해하기 위한 그림이고, 실제 EC 에서는 유한체 (Finite Field) 상에서 계산에 의해 구함.



- 두 점을 알고 있는 상태에서 (P와 Q 혹은 P와 P) 세 번째 점은 아래와 같이 구할 수 있음.

$$P = (x_1, y_1), \qquad Q = (x_2, y_2), \qquad R \; or \; 2P = (x_3, y_3)$$

$$x_3 = s^2 - x_1 - x_2 \; mod \; p$$

$$y_3 = s(x_1 - x_3) - y_1 \; mod \; p$$

$$s = \begin{cases} \dfrac{y_2 - y_1}{x_2 - x_1} \; mod \; p & ; if \; P \neq Q \; (point \; addition) \\ \dfrac{3{x_1}^2 + a}{2y_1} \; mod \; p & ; if \; P = Q \; (point \; doubling) \end{cases}$$

(예제 출처 : Christof Paar, 2010, Understanding Cryptography, Chapter 9)

공개키 (비대칭키) 암호 알고리즘 : 타원곡선 암호 – 덧셈 연산자 (연습)

- EC가 아래와 같을 때 EC 위에 있는 점 P=(5,1)에 대해 2P 를 계산함.
- 기울기 s를 구할 때 분모는 나눈다는 의미가 아님. Modular p 세계에서는 나눗셈을 할 수 없으므로 (실숫값이 나오기 때문), 역원을 구해서 곱함. 역원을 구할 때는 Fermat's Little Theorem을 사용함.

$EC: y^2 = x^3 + 2x + 2 \ mod\ 17$

$P = (5,1) \rightarrow P + P = 2P = ?$

$2P = P + P = (5,1) + (5,1) = (x_3, y_3)$

$$s = \frac{3{x_1}^2 + a}{2y_1} \ mod\ 17 = (2 \cdot 1)^{-1}(3 \cdot 5^2 + 2) \ mod\ 17 = 2^{-1} \cdot 9 \ mod\ 17$$

$2^{-1} \ mod\ 17 = 2^{17-2} \ mod\ 17 = 2^{15} \ mod\ 17 = 9 \ mod\ 17$

$s = 9 \cdot 9 \ mod\ 17 = 13 \ mod\ 17$

$x_3 = 13^2 - 5 - 5 \ mod\ 17 = 6 \ mod\ 17$

$y_3 = 13(5 - 6) - 1 \ mod\ 17 = 3 \ mod\ 17$

$2P = (6,3)$

- Fermat's Little Theorem (p가 소수일 때)

$$a^p \equiv a \ mod\ p, \quad a^{p-1} \equiv 1 \ mod\ p, \quad a^{-1} \equiv a^{p-2} \ mod\ p$$

- 좌측 방법으로 EC 위의 모든 점을 찾아 보면 ?
- 아래 결과와 같이 총 18개의 점이 존재함. 19P는 infinity를 의미하고, 20P 부터는 다시 P로 돌아감 (Cyclic Group).
- Infinity 개념과 확인 방법은 뒷 페이지의 Python 예시에서 프로그램 코드와 함께 자세히 설명하기로 함.

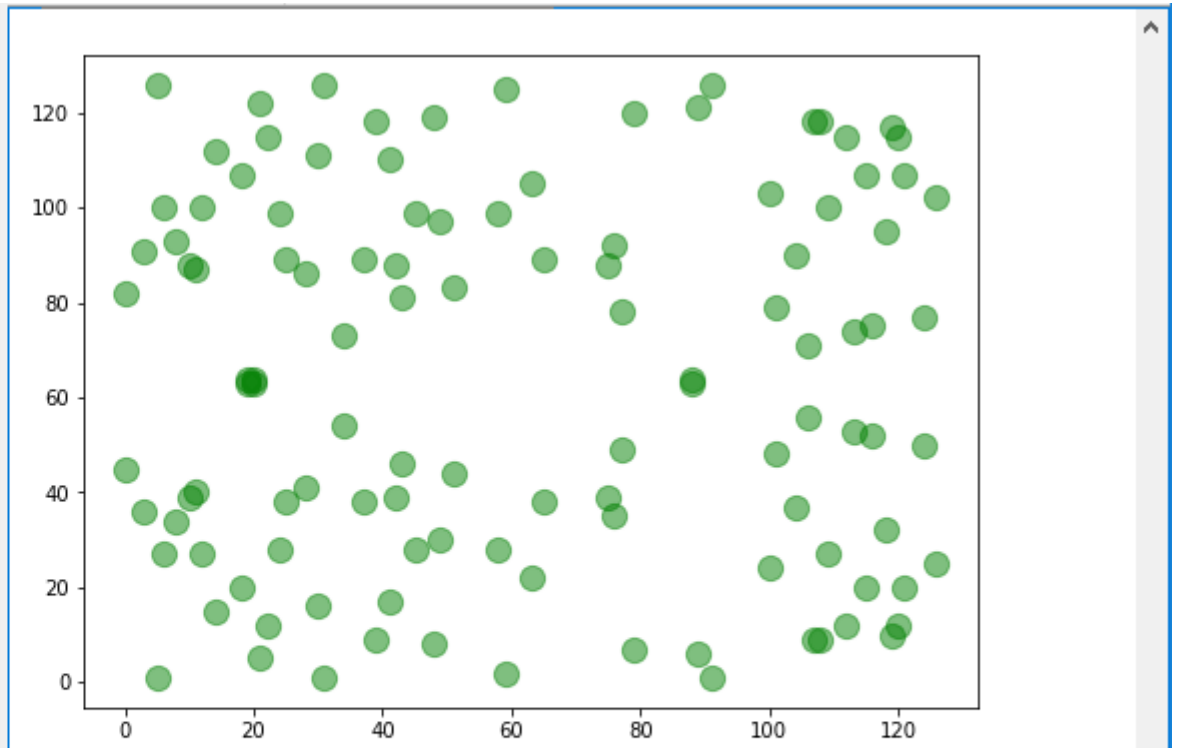
$P = (5,1)$
$2P = P + P = (6,3)$
$3P = 2P + P = (10,6)$
$4P = 3P + P = (3,1)$
$5P = 4P + P = (9,16)$
$6P = 5P + P = (16,13)$
$7P = 6P + P = (0,6)$
$8P = 7P + P = (13,7)$
$9P = 8P + P = (7,6)$
$10P = 9P + P = (7,11)$
$11P = 10P + P = (13,10)$
$12P = 11P + P = (0,11)$
$13P = 12P + P = (16,4)$
$14P = 13P + P = (9,1)$
$15P = 14P + P = (3,16)$
$16P = 15P + P = (10,11)$
$17P = 16P + P = (6,14)$
$18P = 17P + P = (5,16)$
$19P = 18P + P = inf$
$20P = 19P + P = P = (5,1)$
$21P = 20P + P = 2P = (6,3)$
$22P = 21P + P = 3P = (10,6)$
$23P = 22P + P = 4P = (3,1)$
$24P = S3P + P = 5P = (9,16)$

(실습 파일 : 2-7.ECC(Group).py)

공개키 (비대칭키) 암호 알고리즘 : 타원곡선 암호 – Python 연습 (Cyclic Group의 모든 점을 찾음)

- 이전 페이지 예제에서 mod 127인 경우 Cyclic Group에 속한 모든 점을 찾음. 유한체에서 실제 ECC 모습은 아래와 같음. (타원곡선은 실수 체계에서의 그림임)
- ECC에서는 이 점 중에 하나를 Public Key로 사용하는 것임. → (실습 파일 2-8.BitcoinPubKey.py 참조)

```python
# 타원곡선 암호 연습 : Finite field 타원곡선의 모든 점을 찾는다.
#
# 관련된 이론 : Additive operation, Fermat's little theorem, cyclic group
# 참조 : Christof paar, Understanding cryptography, Chap. 9
#
# 2018.4.7 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
import math
import numpy as np
import matplotlib.pyplot as plt

# Additive Operation
def addOperation(a, b, p, q, m):
    if q == (math.inf, math.inf):
        return p

    x1 = p[0]
    y1 = p[1]
    x2 = q[0]
    y2 = q[1]

    if p == q:
        # Doubling
        # slope (s) = (3 * x1 ^ 2 + a) / (2 * y1) mod m
        # 분모의 역원부터 계산한다 (by Fermat's Little Theorem)
        r = 2 * y1
        rInv = pow(r, m-2, m)   # Fermat's Little Theorem
        s = (rInv * (3 * (x1 ** 2) + a)) % m
    else:
        r = x2 - x1
        rInv = pow(r, m-2, m)   # Fermat's Little Theorem
```




```
[(5, 1), (107, 9), (34, 54), (48, 8), (89, 6), (121, 107), (118, 95), (75, 88),
(24, 28), (106, 56), (65, 38), (91, 1), (31, 126), (88, 64), (76, 92), (3, 91),
(112, 115), (59, 2), (12, 100), (126, 102), (43, 46), (14, 112), (119, 117),
(21, 5), (109, 100), (8, 34), (108, 9), (0, 82), (39, 118), (120, 115), (124,
50), (101, 79), (42, 88), (41, 110), (113, 53), (77, 49), (116, 52), (100, 24),
(104, 90), (49, 30), (10, 39), (58, 28), (11, 87), (6, 27), (30, 111), (25,
```

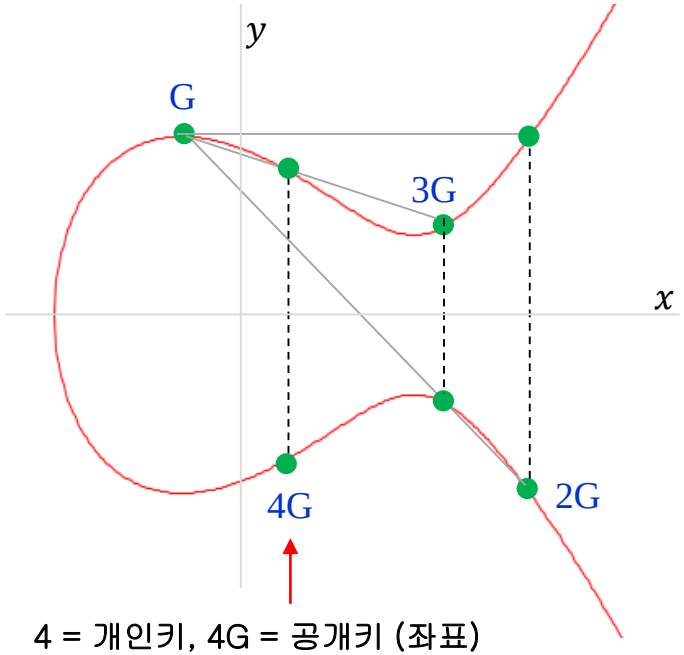
History log | IPython console

공개키 (비대칭키) 암호 알고리즘 : 타원곡선 암호 – 개인키와 공개키 생성

- ECC 상의 점 G를 d번 더하면 점 K가 될 때, d는 Private Key가 되고, K는 Public Key가 됨. 점 G는 주어진 것이고 (secp256k1), 사용자가 d를 선택하여 (랜덤 혹은 기타 방법) 자신의 Private Key로 사용하고, 점 G를 d번 더해서 Public Key로 사용함.
- 점 G를 d번 더하는 방법은 "Double-and-Add" 알고리즘으로 계산함. d를 알면 K를 쉽게 계산할 수 있으나, K를 알고 d를 구하는 것은 매우 어려움.

$$d \cdot G = K$$

$G = Generator$
$d = Private\ Key$
$K = Public\ Key$



4 = 개인키, 4G = 공개키 (좌표)

- Double-and-Add 알고리즘 (Square-and-Multiply 알고리즘과 동일한 원리임)
- Private Key = 26이면 Public Key는 아래와 같이 6 step 만에 구할 수 있음.
- Private Key를 모르고 Public Key만 알고 있으면 26 step의 계산이 필요함.
- Private Key가 매우 큰 수 (256 bit) 라면 알아내기 어려움.

$26G = K$ $\quad 26_{(10)} = 11010_{(2)}$

$bit\ 1\ (1) : G$	←	Initialize
$bit\ 2\ (1) : G + G = 2G$	←	Double
$2G + G = 3G$	←	Add
$bit\ 3\ (0) : 3G + 3G = 6G$	←	Double
$bit\ 4\ (1) : 6G + 6G = 12G$	←	Double
$12G + G = 13G$	←	Add
$bit\ 5\ (0) : 13G + 13G = 26G$	←	Double

Private Key를 알고 있으면 Public Key를 쉽게 계산할 수 있지만, Public Key를 알고 있어도 Private Key는 쉽게 계산할 수 없음.

(실습 파일 : 2-8.PublicKey.py)

공개키 (비대칭키) 암호 알고리즘 : 공개키 생성 – Python 연습

- 개인키와 Double-and-Add 알고리즘을 이용하여 공개키를 생성함.

```python
# 타원곡선 암호 연습 : 개인키, 공개키 생성
# Double-and-Add 알고리즘으로 공개키를 생성한다
#
# 2018.4.8 : 아마추어 퀀트 (조성현)
# ----------------------------------------
import math

# Additive Operation
def addOperation(a, b, p, q, m):
    if q == (math.inf, math.inf):
        return p

    x1 = p[0]
    y1 = p[1]
    x2 = q[0]
    y2 = q[1]

    if p == q:
        # Doubling
        # slope (s) = (3 * x1 ^ 2 + a) / (2 * y1) mod m
        # 분모의 역원부터 계산한다 (by Fermat's Little Theorem)
        # pow() 함수가 내부적으로 Square-and-Multiply 알고리즘을 수행한다.
        r = 2 * y1
        rInv = pow(r, m-2, m)   # Fermat's Little Theorem
        s = (rInv * (3 * (x1 ** 2) + a)) % m
    else:
        r = x2 - x1
        rInv = pow(r, m-2, m)   # Fermat's Little Theorem
        s = (rInv * (y2 - y1)) % m
    x3 = (s ** 2 - x1 - x2) % m
    y3 = (s * (x1 - x3) - y1) % m
    return x3, y3
```

Name	Size	Type	Date Modified
2-7.ECC(Group).py	1 KB	py File	2018-04-08 오후 8:31
2-8.PublicKey.py	1 KB	py File	2018-04-08 오후 8:31
2-9.hash.py	1,005 bytes	py File	2018-04-09 오후 2:09

IPython console

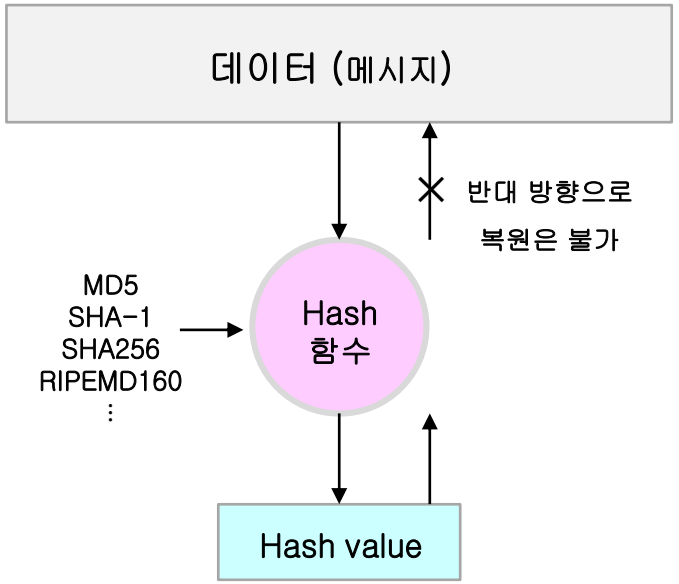
Console 1/A

```
Private Key =  0x499602d2

Public Key = 1234567890 * (5, 1) = (0x380db, 0x7b63)

In [25]:
```

History log | IPython console

Hash 알고리즘

- Hash는 임의 데이터 (메시지)의 내용을 고정된 길이의 데이터로 변환 (매핑)하는 것을 의미함. 데이터를 변환하는 함수를 Hash 함수라 함.
- Hash는 데이터의 지문 같은 역할로 Digest 혹은 Hash value라 하고, 데이터의 일부가 바뀌면 Hash 값은 완전히 바뀌게 되어 데이터가 위, 변조 되었는지 확인할 때 유용하게 사용할 수 있음.
- Hash는 암호적 목적으로도 활용될 수 있고 (데이터의 위, 변조, 무결성 검출), 비암호적 목적으로 사용될 수도 있음 (Filter, 파일 다운로드 오류 검출 등).
- Hash 함수는 단 방향성 특징을 갖음. 데이터로부터 Hash 값을 생성할 수는 있지만, Hash 값으로부터 데이터를 생성할 수는 없음.
- Hash 함수의 동작 원리는 블록 암호 (Block Cipher)와 유사한 방식으로 데이터를 블록으로 나눠서 각 블록의 데이터를 뒤섞거나 XOR 등의 연산을 수행함.
- Hash 함수는 서로 다른 데이터의 Hash 값이 우연히 같은 경우가 발생할 수도 있으나 (Collision), 그 확률은 대단히 작음.
- Collision 확률이 높다면, 동일한 Hash 값이 나오는 방향으로 데이터를 변조할 위험이 존재함.
  (ex : Birthday attack). 강력한 Hash 함수일수록 Collision 확률이 매우 낮음 (Collision Resistance).



암호 목적의 Hash 알고리즘

- MD5 : 1991년 Rivest가 만든 128 bit 알고리즘.
- SHA-1 : 1995년 미국 NIST에서 표준으로 채택한 160 bit 알고리즘
- RIPEMD160 : 1996년 Hans Dobbertin 등이 만든 160 bit 알고리즘
- SHA256, 384, 512 : 2002년 미국 NIST에서 표준으로 채택한 256, 384, 512 bit 알고리즘.
- MD5, SHA-1은 Collision Resistance가 낮아 현재는 SHA256 이상이 널리 사용되고 있음.
- SHA256 (이상)은 미국 주도하의 표준이지만, RIPEMD160은 학술적 오픈 소스로 만들어 짐.
- 비트코인 네트워크에서는 SHA256과 RIPEMD160이 사용됨.

(실습 파일 : 2-9.hash.py)

Hash 알고리즘

- Secp256k1 규격과 Double-and-Add 알고리즘을 이용하여 비트코인 지갑의 개인키와 공개키를 생성하고, Vitalik Buterin의 pybitcointools 결과와 비교함.

```
# Hash 알고리즘 연습
#
# 2018.4.9 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------
from Crypto.Hash import MD5, RIPEMD, SHA, SHA256

msg = '이 문서의 Hash value를 계산한다'
print("\nMessage : ", msg)

msg = msg.encode()

# MD5
h = MD5.new()
h.update(msg)
hv = h.hexdigest()
print("\n          MD5 (%d bit) : %s" % (len(hv) * 4, hv))

# RIPEMD
h = RIPEMD.new()
h.update(msg)
hv = h.hexdigest()
print("       RIPEMD (%d bit) : %s" % (len(hv) * 4, hv))

# SHA
h = SHA.new()
h.update(msg)
hv = h.hexdigest()
print("          SHA (%d bit) : %s" % (len(hv) * 4, hv))

# SHA256
h = SHA256.new()
h.update(msg)
hv = h.hexdigest()
print("       SHA256 (%d bit) : %s" % (len(hv) * 4, hv))
```

Name	Size	Type	Date Modified
2-8.BitcoinPubKey.py	2 KB	py File	2018-04-09 오후 1:17
2-9.hash.py	2 KB	py File	2018-04-09 오후 1:17

IPython console

Console 1/A

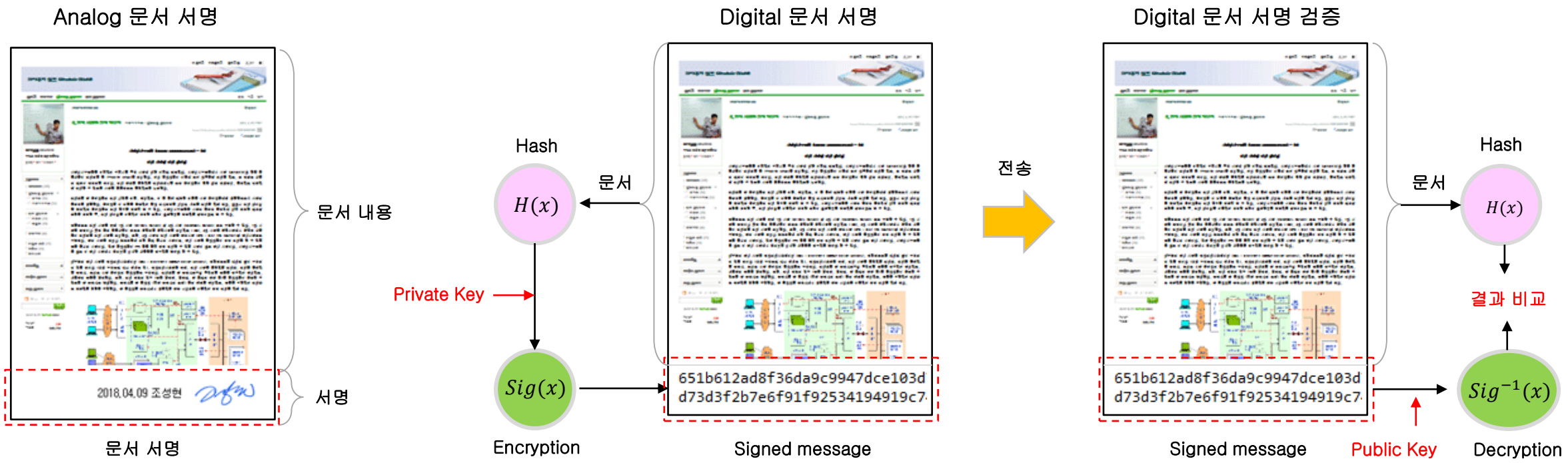
```
Message :  이 문서의 Hash value를 계산한다

          MD5 (128 bit) : cc3d8654c13642da1b96a75b6ee07dbe
       RIPEMD (160 bit) : 0abad3222bc687dcd34a2fa24b146df60155e54c
          SHA (160 bit) : c1a09dbb5f16a297e080be0757cf92711d7c7137
       SHA256 (256 bit) : ba5a1dd7d78c2a69dcaaa99a5766b28248a6bab573a30061cd20b89727ed4412
Double-SHA256 (256 bit) : 0486b0c4d3be761afc1c4457eab08d0b7730726d11ed02de78f99e9db30390fe



Message :  이 문서의 Hash value를 계산한다

          MD5 (128 bit) : 072a3e2a659d68c26eb4934bb46414bb
       RIPEMD (160 bit) : 5bd9c366e4fb4a92066be15bfd0fe8f5e2982ae9
          SHA (160 bit) : f70d7c107614e8bee22af3156299e56d8e65c6d1
       SHA256 (256 bit) : 651b612ad8f36da9c9947dce103d7bdb2d5c355e86a05615ac5d2f3bd9b9ccb2
Double-SHA256 (256 bit) : d73d3f2b7e6f91f92534194919c74b5d6eda2fe3825588068b19f6fdccc385a9

In [66]:
```

Message의 마침표만 달라졌는데,
Hash 값은 완전히 달라 졌음.

History log | IPython console

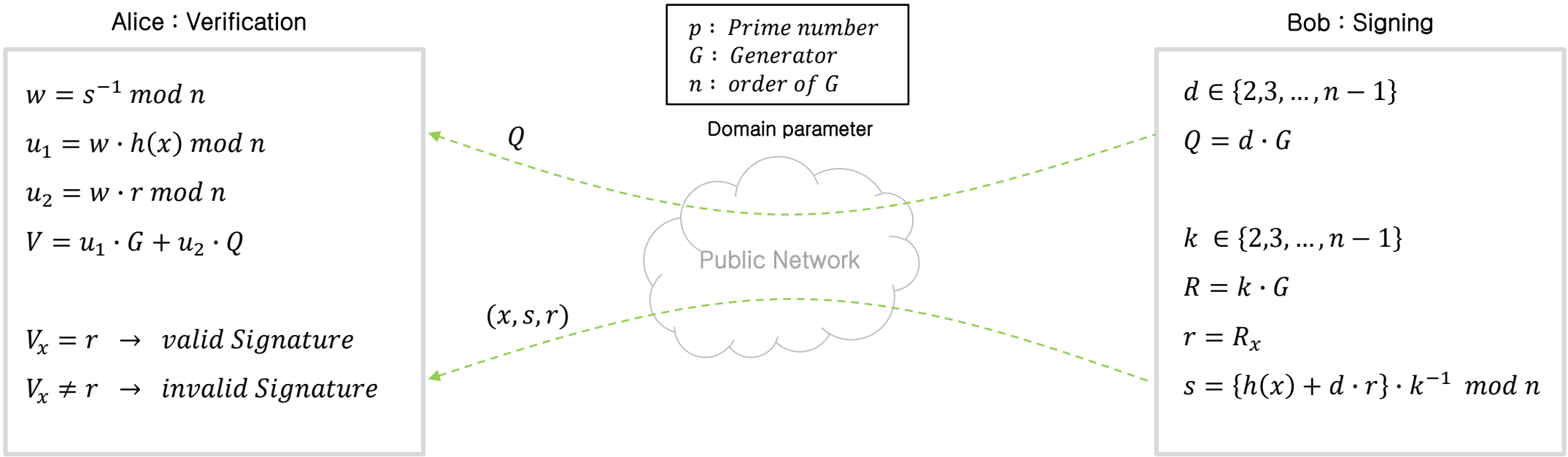
Digital Signature (전자 서명)

- Digital Signature는 공개키 기반의 암호 기술을 이용하여 Digital 문서에 대한 인증 (Authentication)과 부인방지 (Nonrepudiation) 목적으로 사용됨.
- 본인의 Private key로 문서에 대한 Hash 값을 암호화하면 (전자 서명), 다른 사람들은 서명한 사람의 공개키로 암호문 (전자 서명)을 풀 수 있고, 문서의 Hash 값을 계산한 후 전자 서명과 비교하면 해당 문서는 서명자가 작성한 것이라는 게 증명됨.
- Private key를 가지고 있는 사람만이 전자 서명을 할 수 있으므로 본인 인증 (Authentication)이 가능하고, 또한, 서명한 사람이 자신이 생성한 문서가 아니라고 부인할 수 없음 (Nonrepudiation)
- 공개키 기반의 암호 방식 (RSA, Elgamal, ECC)은 모두 Digital Signature에 사용될 수 있음.
- 비트코인 네트워크에서는 지갑 소유자가 다른 사람에게 송금할 때 자신의 Private key로 서명하는 절차가 필요함 (Transaction input의 Unlock 스크립트).
- 비트코인 네트워크의 다른 노드들은 이 서명을 보고 지갑의 소유자가 맞는지 검증할 수 있음.



Digital Signature (전자 서명) : Elliptic Curve Digital Signature Algorithm (ECDSA)

- Bob은 문서 서명자이고 Alice는 문서 서명의 검증자임. Bob은 자신의 Private key (d)와 Public key (Q)를 생성함. Alice는 Q를 알 수 있음.
- Bob은 문서 (x)를 서명하기 위해 임시 Private key (k)와 Public key (R) (k, R : Ephemeral key)를 생성함 (Stochastic cryptography).
- Bob은 문서 (x)의 Hash (h(x))를 생성하고, d, k, R을 이용하여 전자 서명 s를 생성한 후 Alice에게 x, s, r을 보냄.
- Alice는 Q, x, s, r을 이용하여 V를 생성한 후 서명을 확인함.
- 이 방식은 Stochastic cryptography 방식으로 동일 문서라도 서명할 때마다 전자 서명 값이 달라짐. Deterministic 방식은 임시키를 만들지 않고 자신의 개인키로 전자 서명을 생성하는 방식이며 동일한 문서에 대해 동일한 서명 값이 생성됨. Stochastic 방식이 더 안전함.



(실습 파일 : 2-10.ECDSA(1).py)

Digital Signature (전자 서명) : ECDSA – Python 연습 (1)

- 비트코인의 secp256k1 Domain parameter를 이용하여 ECDSA 알고리즘의 Signature & Verification 과정을 확인함.

```
# Digital Signature (ECDSA) 연습
#
# 2018.4.10 : 아마추어 퀀트 (조성현)
# ---------------------------------
import math
import random
from Crypto.Hash import SHA256

# secp256k1의 Domain parameters
# y^2 = x^3 + 7 mod m
a = 0
b = 7
m = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
G = (0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798,
     0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8)
n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

# Additive Operation
def addOperation(a, b, p, q, m):
    if q == (math.inf, math.inf):
        return p

    x1 = p[0]
    y1 = p[1]
    x2 = q[0]
    y2 = q[1]

    if p == q:
        # Doubling
        # slope (s) = (3 * x1 ^ 2 + a) / (2 * y1) mod m
        # 분모의 역원부터 계산한다 (by Fermat's Little Theorem)
        # pow() 함수가 내부적으로 Square-and-Multiply 알고리즘을 수행한다.
        r = 2 * y1
        rInv = pow(r, m-2, m)   # Fermat's Little Theorem
```

```
Message = 이 문서를 서명합니다.

전자서명 생성 :
h(x) = 0x6ec7029ee2c6b1397301cd793f1f6bd96865e26d0b858fa687746410d722fb95
   r = 0x28f9194d1806d0ad344d8547427f193324fa9e5cc43b5614fb4d4423b9e6fe8c
   s = 0x2f9c7ea9b0485e2e7416c2d9d3d810e527d0e5625240a4620c8c1255245eedd8

전자서명 확인 :
h(x) = 0x6ec7029ee2c6b1397301cd793f1f6bd96865e26d0b858fa687746410d722fb95
   V = 0x28f9194d1806d0ad344d8547427f193324fa9e5cc43b5614fb4d4423b9e6fe8c
   r = 0x28f9194d1806d0ad344d8547427f193324fa9e5cc43b5614fb4d4423b9e6fe8c

* Valid Signature



Message = 이 문서를 서명합니다.

전자서명 생성 :
h(x) = 0x6ec7029ee2c6b1397301cd793f1f6bd96865e26d0b858fa687746410d722fb95
   r = 0x2b533be73883bd4ca6a48ecad67eb18407c7bafe0309e38c5e2295883fd6f8f1
   s = 0xcc8df8448da7f69635c7e331844b29d7ae28839b8fafa556a55f497e356d8886

전자서명 확인 :
h(x) = 0x6ec7029ee2c6b1397301cd793f1f6bd96865e26d0b858fa687746410d722fb95
   V = 0x2b533be73883bd4ca6a48ecad67eb18407c7bafe0309e38c5e2295883fd6f8f1
   r = 0x2b533be73883bd4ca6a48ecad67eb18407c7bafe0309e38c5e2295883fd6f8f1

* Valid Signature

In [6]:
```

History log | IPython console

(실습 파일 : 2-11.ECDSA(2).py)

Digital Signature (전자 서명) : ECDSA - Python 연습 (2)

- Vitalik Buterin의 pybitcointools 패키지를 이용하여 ECDSA 알고리즘의 Signature & Verification 결과를 확인함.

```
# Digital Signature (ECDSA) 연습
# 패키지 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
#          written by Vitalik Buterin
#
# 2018.4.10 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------
import bitcoin.main as btc

# 개인키와 공개키를 생성한다.
d = btc.random_key()
Q = btc.privkey_to_pubkey(d)

# 서명할 문서
x = '이 문서를 서명합니다.'
x = x.encode()

# Signature
sig = btc.ecdsa_sign(x, d)

# Verification
v = btc.ecdsa_verify(x, sig, Q)

print("\nMessage = ", x.decode())
if v:
    print("\n* Valid Signature")
else:
    print("\n* Invalid Signature")
```

Name	Size	Type	Date Modified
2-10.ECDSA(1).py	2 KB	py File	2018-04-09 오후 1:17
2-11.ECDSA(2).py	1 KB	py File	2018-02-20 오전 2:07

IPython console

Console 1/A

```
Message = 이 문서를 서명합니다.
     h(x) = 0x6ec7029ee2c6b1397301cd793f1f6bd96865e26d0b858fa687746410d722fb95
        V = 0x459cc74242103da28e8667eed17897dffc7005f910a8f691dd1352630400e822
        r = 0x459cc74242103da28e8667eed17897dffc7005f910a8f691dd1352630400e822

* Valid Signature
```

* Vitalik Buterin의 pybitcointools 패키지 실행 결과

```
Message =  이 문서를 서명합니다.

* Valid Signature

In [136]:
```

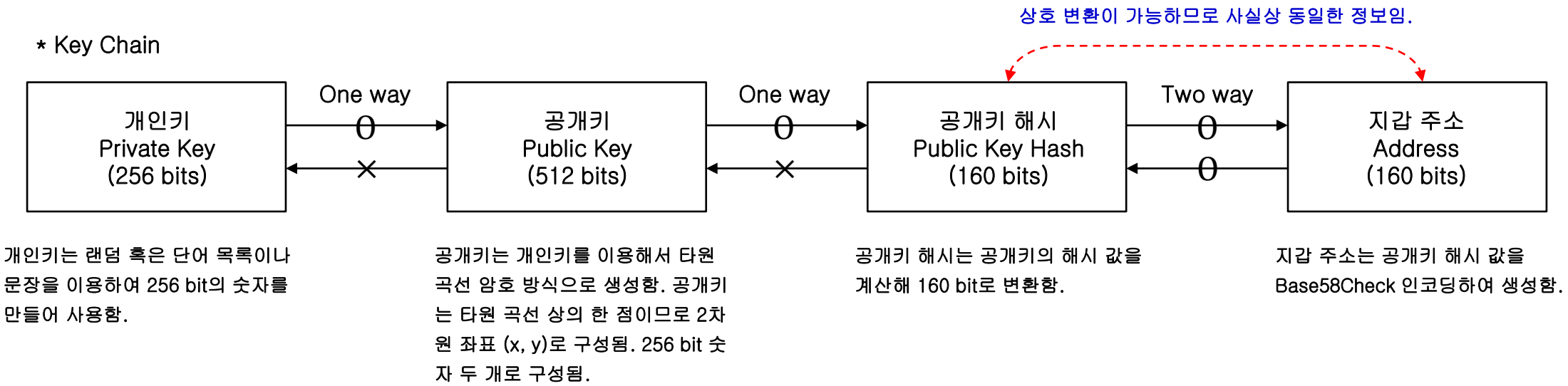
History log | IPython console

3. 지갑 (Wallet)

3-1. 비트코인 지갑 구조
3-2. 개인키 (Private Key)
3-3. 공개키 (Public Key)
3-4. 지갑 주소 (Address)
3-5. 지갑 관리 및 백업
3-6. 키 (key) 관리
3-7. Paper Wallet
3-8. Brain Wallet 과 Vanity Address
3-9. Nondeterministic과 Deterministic Wallet
3-10. HD (Hierarchical Deterministic) Wallet
3-11. Mnemonic Code (BIP-39)

비트코인 지갑 구조

- 비트코인 지갑은 비트코인의 잔고를 보관하는 것아 아니라, 개인키, 공개키, 지갑 주소를 관리하는 것이 목적임.
- 개인키를 생성하고, ECC 알고리즘으로 공개키를 생성함. 공개키로 공개키 해시를 계산하고, 공개키 해시로 지갑 주소를 생성함.
- 지갑 어플리케이션은 개인키, 공개키, 공개키 해시, 지갑 주소를 생성하고, 다양한 형태 (Key Formats)로 변환하거나, 상호 변환 등의 기능을 수행함.
- 만약 개인키를 잃어버리면 블록체인에 기록된 잔고를 사용할 수 없음. → 해당 지갑으로 송금된 비트코인은 영원히 아무도 사용할 수 없음.
- 공개키 → 공개키 해시는 역변환이 불가하고 (One way), 공개키 해시 ←→ 지갑 주소는 상호 변환이 가능함.
- 사용자는 지갑 주소를 이용하여 잔고를 관리하거나 입.출금하지만, 블록체인 데이터에는 지갑 주소가 기록되지 않고, 지갑 주소의 공개키 해시 값이 기록됨.
- 거래를 생성할 때 지갑 주소를 공개키 해시 값으로 변환하여 Transaction에 기록함 (output의 Locking Script 부분에 기록함).
- 3rd-party API 서버에서 거래 내역을 보여줄 때 지갑 주소로 보여주는 것은 블록체인 데이터의 공개키 해시 값을 읽어 지갑 주소로 변환하여 보여주는 것임.



개인키 : Private Key

- 개인키는 256 bit의 임의의 숫자임 $(1 \sim 2^{256})$. 정확히는 secp256k1의 Domain parameter인 N 값보다는 작으므로 2^{256} 보다는 약간 작음.
- $2^{256} \approx 10^{77}$ 이고 (눈에 보이는) 우주에 존재하는 원자의 총 개수는 10^{80} 정도로 알려져 있음. 개인키는 이 정도로 큰 숫자임.
- 프로그램 언어에서 기본적으로 제공하는 rand(), random.random() 함수 등은 보안상 취약한 것으로 알려져 있음. → 완전 랜덤이 아님. 패턴이 존재할 수 있음.
- Python의 경우 os.urandom(), Linux의 경우 /dev/random, /dev/urandom이 더 안전한 것으로 알려져 있음.
- 단순히 랜덤 값이 아닌 단어 목록이나 문장 등을 이용한 해시 값을 이용할 수 있음. → Mnemonic words, Brainwallet.

Random Number Generator (RNG)

- TRNG (True Random Number Generators) : 실제로 동전, 주사위 같은 걸 던져서 랜덤 비트를 생성하는 방식. (반도체 노이즈, Clock jitter, 마우스 움직임 등).
- PRNG (Pseudorandom Number Generators) : seed 값을 이용하여 시계열을 생성함. seed 값은 주로 clock을 이용함. (ex : random.random() in Python).
  $X_{n+1} = (aX_n + seed)\ mod\ m$ (ex : ANSI C의 rand() → $S_0 = 12345,\ S_{i+1} = 1103515245 \cdot S_i + 12345\ mod\ 2^{31}$, 출처 : Christof paar, Understanding Cryptography)
  PRNG 방식은 시계열 간의 연관성이 있을 수 있으며, 이후 시계열의 추측 가능성이 있으므로 보안상 안전하지 못함.
- CSPRNG (Cryptographically Secure Pseudorandom Number Generators) : 추측 불가능한 PRNG 방식. Python의 경우 os.urandom (Linux의 경우 /dev/random, /dev/urandom)은 OS 디바이스 등 H/W 장치에서 발생하는 입력 노이즈 들을 모아서 랜덤 seed로 사용하는 방식으로 CSPRNG에 가까움.
  Python의 경우 random.random() 혹은 numpy.random() 보다 os.urandom() 방식이 보안상 안전함.

(실습 파일 : 3-1.개인키(생성).py)

개인키 : Private Key 생성 – Python 연습

- 1 ~ N 사이의 Random (CSPRNG) 숫자를 생성하고 Hash (SHA256) 값을 계산해서 개인키로 사용한 예시.

```
# Vitalik Buterin의 pybitcointools에서 개인키를 생성한 방법
# 참고 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
#
# CSPRNG : Cryptogrphically Secure Pseudorandom Number Generatc
# PRNG : Pseudorandom Number Generator
#
# PRNG는 seed 값을 이용해 시계열을 계산하는 (X = (a*X + seed) moc
# 시계열간 종속성이 있을 수 있음. 이후 시계열 추측 가능성. - ex : r
# CSPRNG는 이후 시계열 추측이 불가한 방식
#
# 2018.4.16 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------
import os
import random
import time
import hashlib

# secp256k1의 Domain parameter (order of G)
N = 115792089237316195423570985008687907852837564279074904382660

# CSPRNG 방식, os.urandom()와 random()를 적당히 섞어서 hash256으
def random_key():
    r = str(os.urandom(32)) \
        + str(random.randrange(2**256)) \
        + str(int(time.time() * 1000000))
    r = bytes(r, 'utf-8')
    h = hashlib.sha256(r).digest()
    key = ''.join('{:02x}'.format(y) for y in h)
    return key

while (1):
```

Name	Size	Type	Date Modified
2-11.ECDSA(2).py	677 bytes	py File	2018-04-10 오전 3:04
3-1.개인키(생성).py	1 KB	py File	2018-04-16 오전 3:18

IPython console

Console 1/A

```
PrivKey (Hex) :  c44dcd9e52966107ac3fbfc398b4cd5a0352dd6c96efe38742b030df777390be
PrivKey (Dec) :  887907846726782834488017344698513796536648269751697011577093375771562467370 86

PrivKey (Hex) :  f3f9f92cdd2d6f787647f21ad862f085362581b9557882acdd6d3a373fb7afae
PrivKey (Dec) :  110353686869234261789091599658296454363616633702242897874079598990787498979246

PrivKey (Hex) :  4d2325dd938c0e618df00e8ec896b38253de6fff6a085c8c61bd3a1f17cc62e8
PrivKey (Dec) :  348901903264803226917088448889956092794233689162640414049828896218529364958 48

PrivKey (Hex) :  b55f3c868243639741da69a042eaedcf06e88156f14c94dc4f900d951238997f
PrivKey (Dec) :  820368937958571437934660881167188492739928482241290225394664118344406516760 31

In [6]:
```

History log | IPython console

개인키 유형 : Key Formats 및 상호 변환

- 개인키는 여러 형태 (Format)로 변환할 수 있음. 거래에 사용하기 위한 유형과 관리를 위한 유형으로 나눌 수 있음.
- 거래에는 Raw와 Hex Format이 사용되고, 관리를 위해서는 WIF (Wallet Import Format), WIF-Compress Format이 있음.
- 지갑 S/W는 거래 생성을 위해 내부적으로 Raw, Hex Format을 사용하고, 보안상 이유로 외부로 표시하지는 않음 (일반적으로). 사용자가 요청하면 WIF Format을 표시함.
- WIF Format은 Raw, Hex Format을 Base58 encode 형태로 변환한 것으로 외부 입.출력 용으로 사용함. 개인키 관리 목적으로 QR 코드에 사용되기도 함.
- WIF-Compressed Format은 Compressed Public Key를 생성할 때 사용됨.
- WIF Format에는 Check Sum (4 bytes) 값이 들어 있어서 키의 유효성을 확인할 수 있음.
- 아래 4가지 Format은 상호 변환 가능함. 모두 동일한 정보이므로, 외부로 유출되면 안됨.

Private key representations (encoding formats)

Type	Prefix	Description
Raw	None	32 bytes, 256 bits
Hex	None	64 hexadecimal digits
WIF	5	Base58Check encoding : Base 58 with version prefix of 128-and 32-bit checksum
WIF-Compressed	K or L	As above, with added suffix 0x01 before encoding

출처 : Mastering Bitcoin 2nd Edition (p. 70, Table 4-2)

Key Format (예시)

Hex : 69236c9bd75194a301a21a5671a2b5b5671c840a86b39caa94353fe32ef81f3c
Dec : 47555438338583104046036392420609779792096882768935954893573390972430384111420
WIF : 5JcbCZmfJrT7hjLq2CosvyLLbyNLXpkht3PLtKh2nuB45ZwWCSz
WIF-Compressed : Kzk5x3e85oGGamtWxK7uoDeMXMoBD3muSHXcmXVgubHNb3bdKYMv

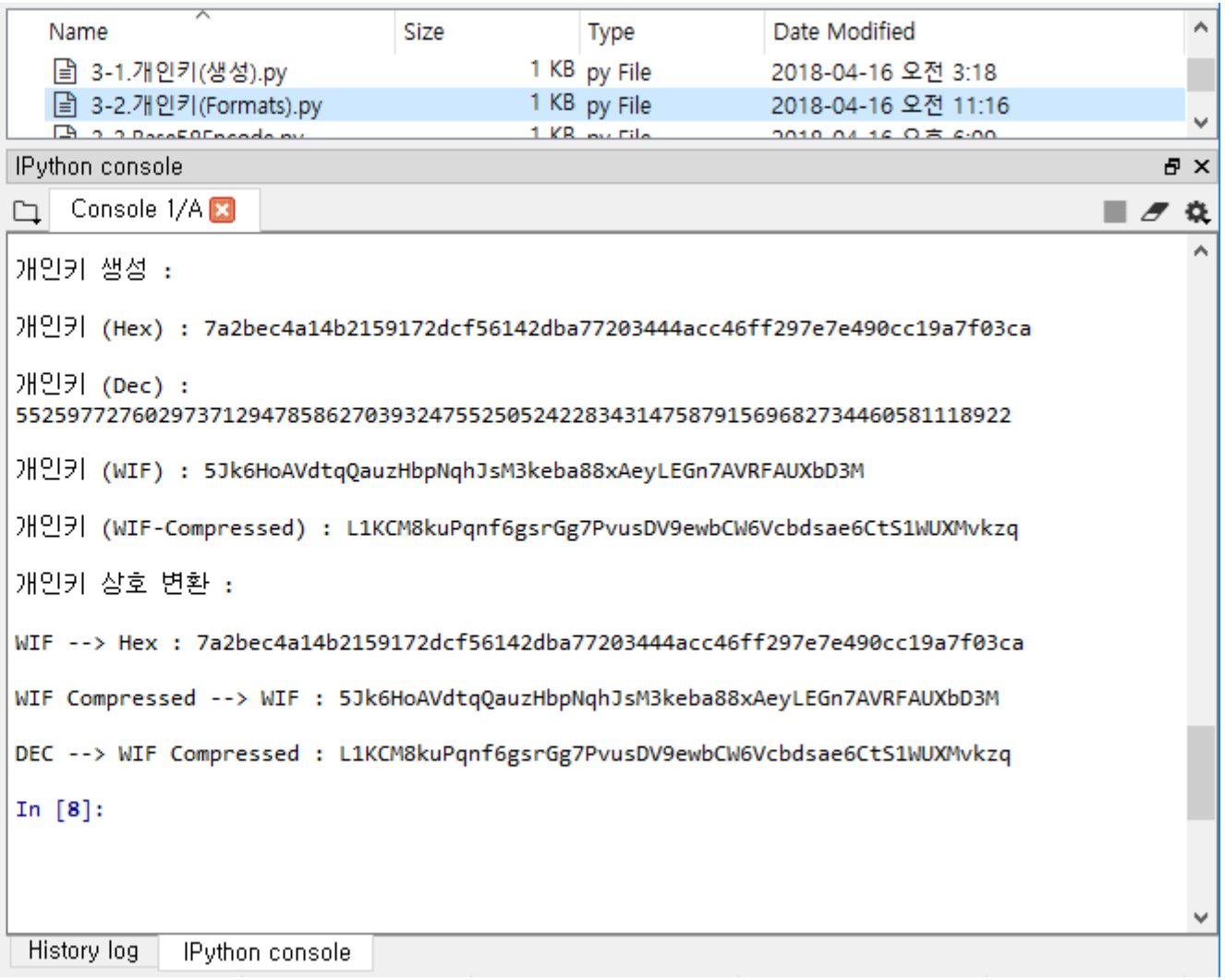
개인키 상호 변환 :
WIF --> Hex : 69236c9bd75194a301a21a5671a2b5b5671c840a86b39caa94353fe32ef81f3c
WIF Compressed --> WIF : 5JcbCZmfJrT7hjLq2CosvyLLbyNLXpkht3PLtKh2nuB45ZwWCSz
DEC --> WIF Compressed : Kzk5x3e85oGGamtWxK7uoDeMXMoBD3muSHXcmXVgubHNb3bdKYMv

(실습 파일 : 3-2.개인키(Formats).py)

Key Formats 생성 및 상호 변환 – Python 연습

- 개인키를 다양한 형태로 표시함. 모든 형태는 상호 변환됨을 확인함.

```python
# 개인키를 생성하고 다양한 형태로 변환한다.
# 참고 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
#
# Hex : 16진수
# Decimal : 10진수
# WIF  : Wallet Import Format
# WIF compressed
#
# 2018.4.16 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------
import bitcoin as btc  # Vitalik Buterin의 pybitcointools

# 개인키 (Hex format)를 생성한다
while (1):
    privKey = btc.random_key()                      # 256 bit Random n
    dPrivKey = btc.decode_privkey(privKey, 'hex')   # 16진수 문자열을 1
    if dPrivKey < btc.N:                            # secp256k1 의 N 노
        break

print("\n개인키 생성 :")
print("\n개인키 (Hex) :", privKey)
print("\n개인키 (Dec) :", dPrivKey)

# 개인키를 WIF (Wallet Import Format) 형태로 변환한다
wifPrivKey = btc.encode_privkey(dPrivKey, 'wif')
print("\n개인키 (WIF) :", wifPrivKey)

# 개인키를 wif_compressed 형태로 변환한다
cWifPrivKey = btc.encode_privkey(dPrivKey, 'wif_compressed')
print("\n개인키 (WIF-Compressed) :", cWifPrivKey)

# Key format은 상호 변환이 가능하다
```



Base58Check 인코딩과 WIF Format

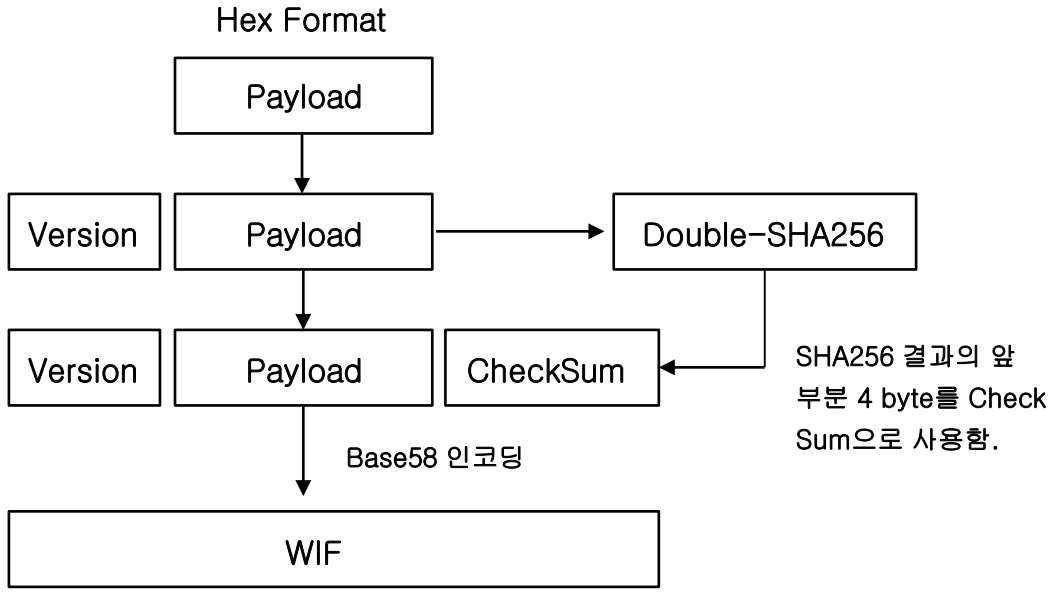
- Base58Check 인코딩은 숫자 (혹은 문자열)를 58개의 문자와 Check Sum 값으로 변환하여 사람이 읽기 쉬운 형태로 변환하고 에러를 검출하기 위해 사용함.
- Base58Check 인코딩은 개인키 뿐만 아니라, 공개키, 지갑 주소에도 사용됨. 구별하기 쉽게 Version prefix를 붙임.
- 개인키의 경우 Hex Format (Payload) 앞 부분에 0x80을 붙이고, SHA256 해시를 두 번 수행한 후 앞 부분 4 byte를 Check Sum 값으로 사용함.
- Version + Payload + Check sum 값을 Base58 인코딩으로 변환함.

- Base58Check 인코딩 예시
- 58개 문자 (0, I, O, l, +, - : 혼돈하기 쉬운 문자는 사용하지 않음)
- "123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz"
- 숫자 12548 / 58 → 몫 = 216, 나머지 = 20 → 나머지에 해당하는 문자 : 'M'
  216 / 58 → 몫 = 3, 나머지 = 42 → 'j'
  3 / 58 → 몫 = 0, 나머지 = 3 → '4'
- 숫자 12548 → '4jM'

Base58Check version prefix and encoded result examples

Type	Version prefix	Base58 result prefix
Bitcoin Address	0x00	1
Pay-to-Script-Hash Address	0x05	3
Bitcoin Testnet Address	0x6F	m or n
Private Key WIF	0x80	5, K, or L
BIP-38 Encrypted Private Key	0x0142	6P
BIP-32 Extended Public Key	0x0488B21E	xpub

출처 : Mastering Bitcoin 2nd Edition (p. 68, Table 4-1)

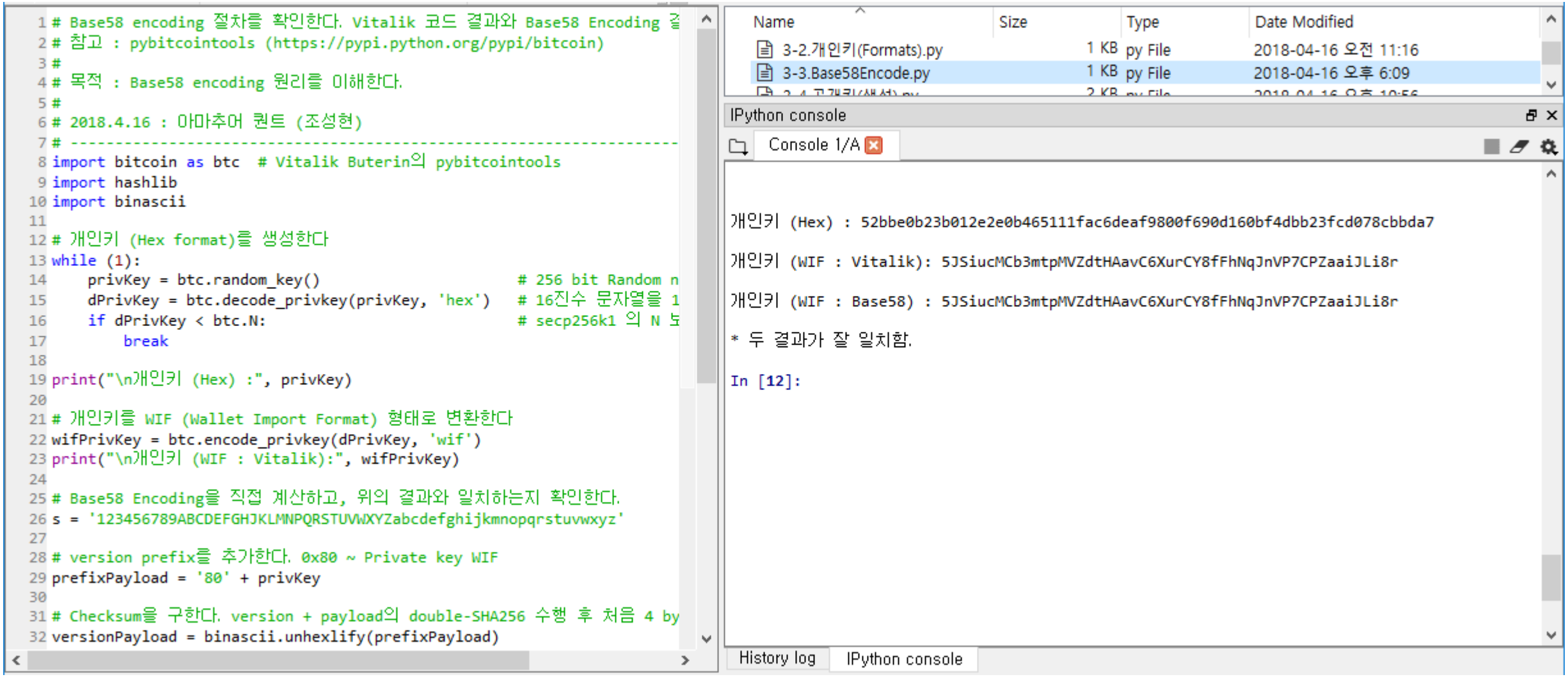


(실습 파일 : 3-3.Base58Encode.py)

Base58Check 인코딩 – Python 연습

- 개인키를 Base58Check 인코딩으로 변환함. 이전 페이지 절차를 직접 계산하여 원리를 이해하고. pybitcointools 결과와 비교함.

```
# Base58 encoding 절차를 확인한다. Vitalik 코드 결과와 Base58 Encoding 결
# 참고 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
#
# 목적 : Base58 encoding 원리를 이해한다.
#
# 2018.4.16 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
import bitcoin as btc  # Vitalik Buterin의 pybitcointools
import hashlib
import binascii

# 개인키 (Hex format)를 생성한다
while (1):
    privKey = btc.random_key()                      # 256 bit Random n
    dPrivKey = btc.decode_privkey(privKey, 'hex')   # 16진수 문자열을 1
    if dPrivKey < btc.N:                            # secp256k1 의 N 노
        break

print("\n개인키 (Hex) :", privKey)

# 개인키를 WIF (Wallet Import Format) 형태로 변환한다
wifPrivKey = btc.encode_privkey(dPrivKey, 'wif')
print("\n개인키 (WIF : Vitalik):", wifPrivKey)

# Base58 Encoding을 직접 계산하고, 위의 결과와 일치하는지 확인한다.
s = '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'

# version prefix를 추가한다. 0x80 ~ Private key WIF
prefixPayload = '80' + privKey

# Checksum을 구한다. version + payload의 double-SHA256 수행 후 처음 4 by
versionPayload = binascii.unhexlify(prefixPayload)
```



공개키 : Public Key

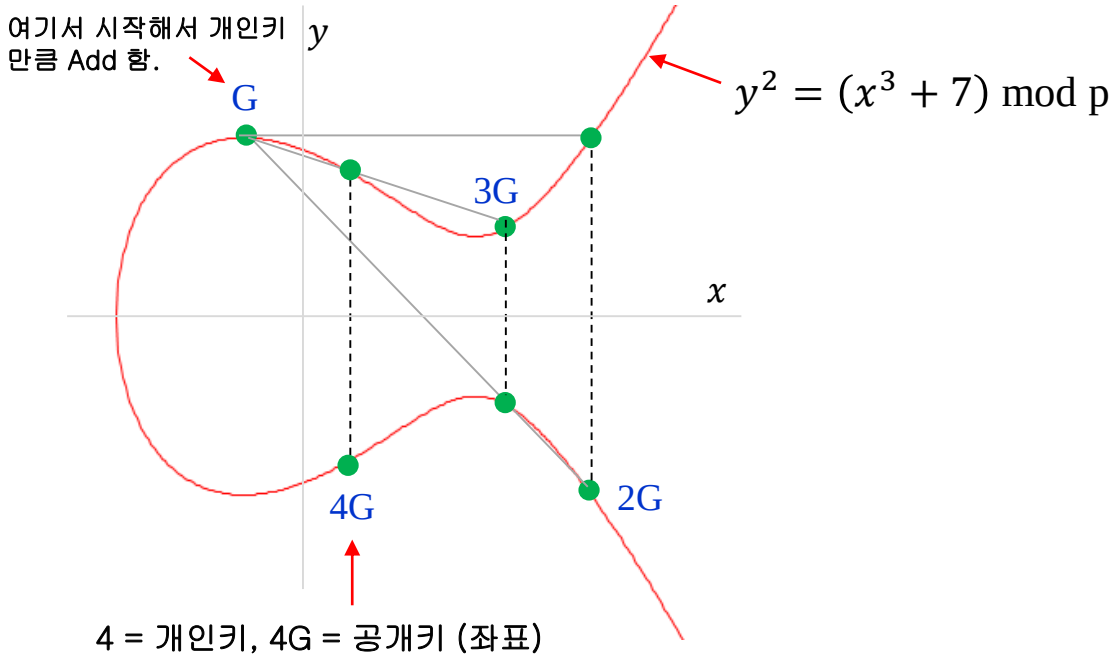
- 공개키는 개인키를 이용하여 타원곡선 암호 (ECC) 방식으로 계산함. 비트코인 공개키는 secp256k1에 정의된 ECC Domain Parameter (a, b, p, G, n)를 적용함.
- 공개키는 타원곡선 상 (실제 타원은 아님)의 점 G를 개인키 만큼 곱한 것으로 정의함. 공개키 (K) = 개인키 * G. → 세부 내용은 2. 암호학의 타원곡선 암호 참조.
- 공개키 (K) = G + G + G + … + G (개인키 만큼 덧셈)로 계산할 수 있고, 아래 그림의 (실수 공간에서의) 덧셈으로 생각할 수 있음. (실제는 이산체 공간).
- 개인키를 알고 있으면 위의 계산을 쉽게 할 수 있으나 (Double-and-Add 알고리즘), 공개키만 알고 개인키를 찾으려면 위의 덧셈을 일일이 해 봐야 함 (Brute Force). 개인키가 큰 수이기 때문에 (2^{256}) 일일이 계산하는 것은 현실적으로 가능하지 않음.
- 예를 들어 개인키 (d) = 4 라면 공개키는 아래 우측 그림의 4G인 점의 좌표임. 공개키는 (x, y) 형태의 2차원 좌표임.

공개키 (K) 생성 방법

d 만큼 Add 함 (Double-and-Add 알고리즘)

$$K = d \cdot G = G + G + G + \cdots + G$$

$$G = Generator, d = Private\ Key, K = Public\ Key$$



공개키 생성 : secp256k1 규격

- 비트코인 개인키, 공개키 규격은 secp256k1을 채택함. (참고 자료 : http://www.secg.org/sec2-v2.pdf)
- EC 는 $y^2 = x^3 + 7 \ mod\, p$ 를 사용하고 Domain parameter 인 p와 G는 아래와 같음. Compress 모드의 G는 G의 x-좌표만 표시한 것이고, Uncompress 모드의 G는 (x, y) 좌표 모두를 표시한 것임. Domain parameter는 모두 256 bits 임.

- Secp256k1 규격

The elliptic curve domain parameters over F_p associated with a Koblitz curve secp256k1 are specified by the sextuple $T = (p, a, b, G, n, h)$ where the finite field F_p is defined by:

$p = FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFE\ FFFFFC2F$
$= 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$

The curve E: $y^2 = x^3 + ax + b\ over\ F_p$ is defined by:
$a = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000$
$b = 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000007$

The base point G in compressed form is:
$G = 02\ 79BE667E\ F9DCBBAC\ 55A06295\ CE870B07\ 029BFCDB\ 2DCE28D9\ 59F2815B\ 16F81798$

and in uncompressed form is:
$G = 04\ 79BE667E\ F9DCBBAC\ 55A06295\ CE870B07\ 029BFCDB\ 2DCE28D9\ 59F2815B\ 16F81798$
$483ADA77\ 26A3C465\ 5DA4FBFC\ 0E1108A8\ FD17B448\ A6855419\ 9C47D08F\ FB10D4B8$

Finally the order n of G and the cofactor are:
$n = FFFFFFFF\ FFFFFFFF\ FFFFFFFF\ FFFFFFFE\ BAAEDCE6\ AF48A03B\ BFD25E8C\ D0364141$
$h = 01$

- 비트코인 지갑의 개인키 공개키 생성 절차

1. 왼쪽의 Domain Parameter를 참조하여
2. EC 함수는 $y^2 = x^3 + 7 \ mod\, p$ 를 사용함.
   $(a = 0, b = 7)$
3. 개인키 privKey (d) = 1 ~ n-1 사이의 랜덤값을 사용함.
4. 공개키 pubKey (K) = d G를 계산하여 사용함.

- 참고 사항 :
- 미국 NIST는 ECC 암호 표준으로 secp256r1 규격을 채택하였음 (NIST curve).
- ECC 암호 방식으로 secp256r1 이 널리 사용됨.
- Secp156k1 은 secp256r1 보다 (아주) 약간 약하다고 알려져 있음.
- Satoshi 는 왜 비트코인에 secp256k1을 채택했을까?
  (참고 : https://koclab.cs.ucsb.edu/teaching/ecc/project/2015Projects/Bjoernsen.pdf)

(실습 파일 : 3-4.공개키(생성).py)

공개키 생성 Python 연습

- Secp256k1 규격과 Double-and-Add 알고리즘을 이용하여 비트코인 지갑의 개인키와 공개키를 생성하고, Vitalik Buterin의 pybitcointools 결과와 비교함.

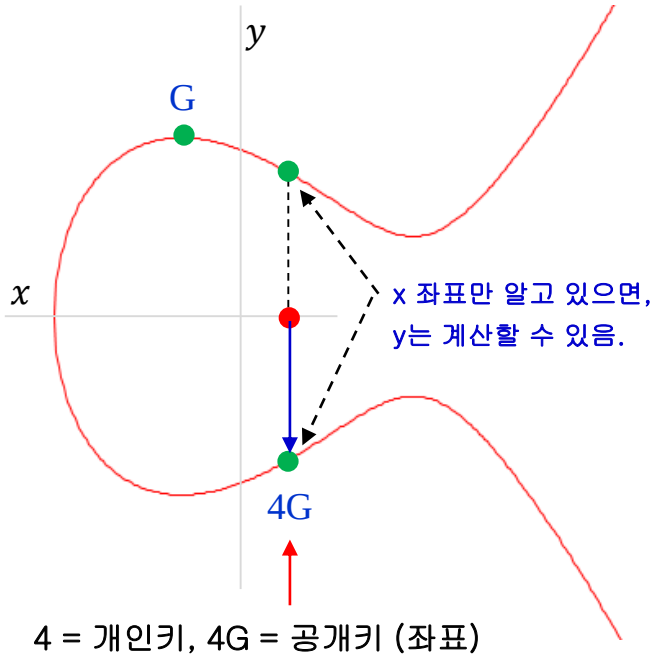
```
# secp256k1의 Domain parameters
# y^2 = x^3 + 7 mod m
a = 0
b = 7
m = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
G = (0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798,
     0x483ADA7726A3C4655DA4FBFC0E1108A8FD17B448A68554199C47D08FFB10D4B8)
n = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

# 개인키를 생성한다.
while(1):
    d = random.getrandbits(256)
    if d > 0 & d < n:
        break

# Double-and-Add 알고리즘으로 공개키를 생성한다
bits = bin(d)
bits = bits[2:len(bits)]

# initialize. bits[0] = 1 (always)
K = G

# 두 번째 비트부터 Double-and-Add
bits = bits[1:len(bits)]
for bit in bits:
    # Double
    K = addOperation(a, b, K, K, m)

    # Multiply
    if bit == '1':
        K = addOperation(a, b, K, G, m)

privKey = d
```

```
Name                  Size    Type     Date Modified
3-2.개인키(2).py       1 KB    py File  2018-04-16 오전 11:16
3-3.Base58Encode.py   1 KB    py File  2018-04-16 오후 6:09
3-4.공개키.py          2 KB    py File  2018-04-16 오후 10:56

IPython console
Console 1/A

Private Key :
0xd138126b799ca796ebb969d4e861c789b2ab31202acf9e73a01edec3b53afb66

Public Key :
(0x365d2a5c40056fb7264998b01242c2953d82a6a8cfc0e4c76d471e86ef92d62c,
0x9e017447063c3e9867dff7dc4a8cd87985654024b7078d2eae9bcf60915c8fe5)

# Vitalik Buterin의 pybitcointools 결과 :

Public Key :
(0x365d2a5c40056fb7264998b01242c2953d82a6a8cfc0e4c76d471e86ef92d62c,
0x9e017447063c3e9867dff7dc4a8cd87985654024b7078d2eae9bcf60915c8fe5)

* 두 결과가 잘 일치함

In [2]:

History log  IPython console
```

두 결과가 잘 일치함

공개키 유형 : Key Formats 및 상호 변환

- 공개키도 여러 형태 (Format)로 변환할 수 있음. 공개키는 (x, y) 좌표이고, x, y는 각각 256 bit 숫자임.
- Uncompressed Format은 0x04 + x 좌표 + y 좌표 형태임. 길이는 8 bit + 256 bit + 256 bit = 520 bit임.
- Compressed Format은 y 좌표가 짝수인 경우 0x02 + x 좌표로 표시하고, y 좌표가 홀수인 경우는 0x03 + x 좌표로 표시함.
- 함수식을 알고 있으므로, x 값만 저장해 두면 y는 계산할 수 있음 (정수론 : 법 p에 대한 제곱수 찾기 문제). 단, EC는 x 축에 대칭이므로 y는 2개가 대응됨.
- Compressed Format으로 (x, y) 형태나 Uncompressed Format으로의 변환은 아래 방법을 사용함.



4 = 개인키, 4G = 공개키 (좌표)

* 예제 : x-좌표로 y-좌표 계산

$y^2 \ mod\ 127 = \ x^3 + 7 \ mod\ 127$

이 값에 따라 계산 방법이 달라짐.

$x = 3, y = ?$

$127 \equiv 3 \ mod\ 4$ ← 이 조건을 만족하면 아래 식을 이용할 수 있음.

$y = \pm\ (x^3 + 7)^{\frac{127+1}{4}} \ mod\ 127$

$y = \pm\ (3^3 + 7)^{\frac{127+1}{4}} \ mod\ 127$

$y = \pm\ 34^{32} \ mod\ 127$

$y = \pm\ 62$

* 결과 확인

$y^2 \ mod\ 127 = \ x^3 + 7 \ mod\ 127$

$62^2 \ mod\ 127 = 34$

$(x^3 + 7) = (3^3 + 7) = 34$

$y^2 = a \ mod\ p \ \ (a \neq 0, p = 3 \ mod\ 4)$

$y = \pm\ a^{\frac{p+1}{4}} \ mod\ p$

비트코인은 이 조건을 만족하므로,
이 공식을 적용할 수 있음

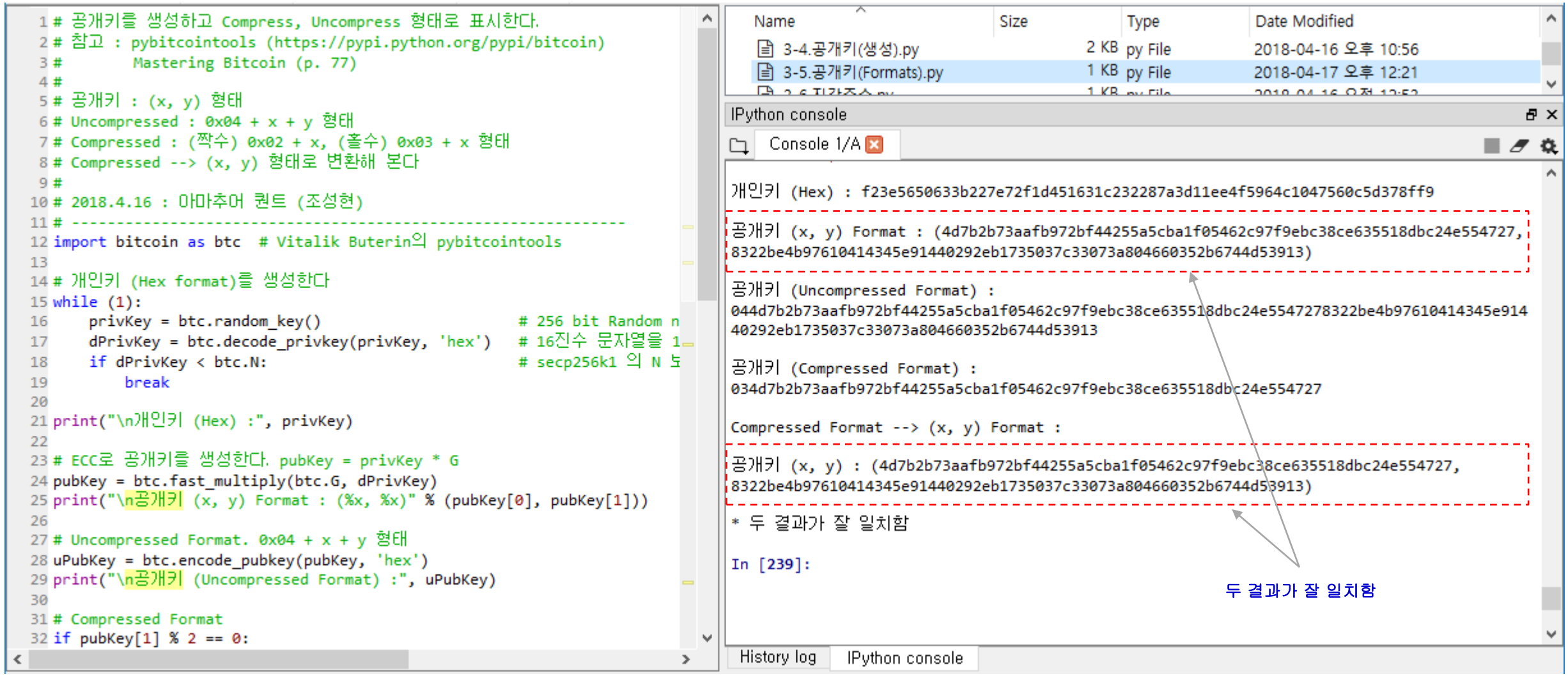
출처 : http://mersennewiki.org/index.php/Modular_Square_Root

(실습 파일 : 3-5.공개키(Formats).py)

공개키 : Key Formats 생성 및 상호 변환 – Python 연습

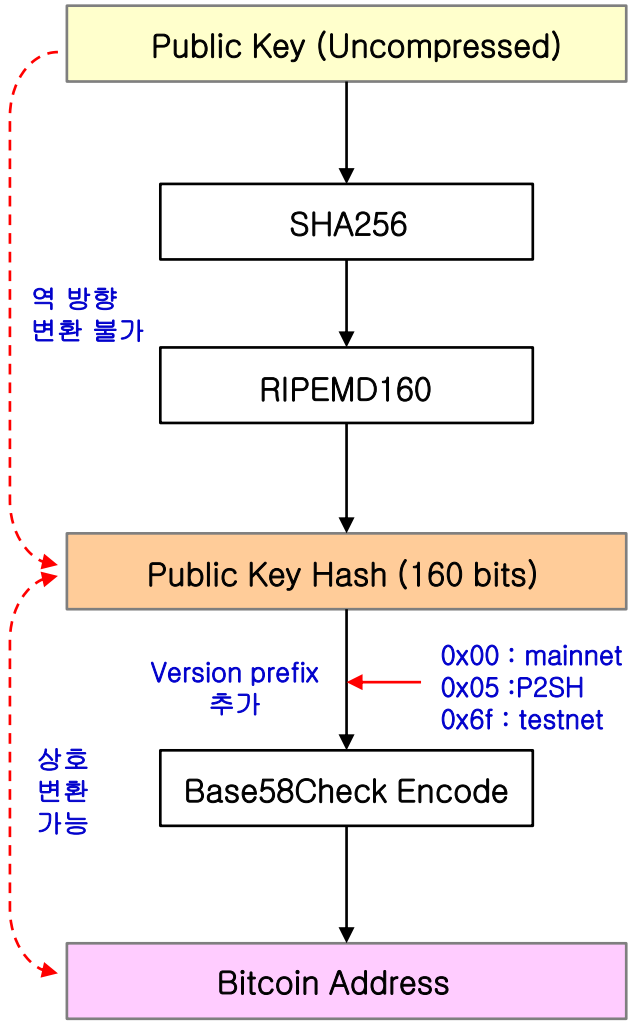
- 개인키로 공개키를 계산하고 (x,y) 형태, Uncompressed 형태, Compressed 형태로 표시함. 그리고 상호 변환을 확인함.

```python
# 공개키를 생성하고 Compress, Uncompress 형태로 표시한다.
# 참고 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
#        Mastering Bitcoin (p. 77)
#
# 공개키 : (x, y) 형태
# Uncompressed : 0x04 + x + y 형태
# Compressed : (짝수) 0x02 + x, (홀수) 0x03 + x 형태
# Compressed --> (x, y) 형태로 변환해 본다
#
# 2018.4.16 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------
import bitcoin as btc  # Vitalik Buterin의 pybitcointools

# 개인키 (Hex format)를 생성한다
while (1):
    privKey = btc.random_key()                      # 256 bit Random n
    dPrivKey = btc.decode_privkey(privKey, 'hex')   # 16진수 문자열을 1
    if dPrivKey < btc.N:                            # secp256k1 의 N 노
        break

print("\n개인키 (Hex) :", privKey)

# ECC로 공개키를 생성한다. pubKey = privKey * G
pubKey = btc.fast_multiply(btc.G, dPrivKey)
print("\n공개키 (x, y) Format : (%x, %x)" % (pubKey[0], pubKey[1]))

# Uncompressed Format. 0x04 + x + y 형태
uPubKey = btc.encode_pubkey(pubKey, 'hex')
print("\n공개키 (Uncompressed Format) :", uPubKey)

# Compressed Format
if pubKey[1] % 2 == 0:
```



지갑 주소 : Address

- 지갑 주소 (Address)는 다른 사람들에게 알려 주고 비트코인을 받을 때 사용하거나, 지갑 어플리케이션에서 잔고나 거래 내역을 관리할 때 주로 사용함. 주소는 공개 정보임.
- 지갑 주소는 개인키 → 공개키 (Uncompressed Format) → 공개키 해시 → 지갑 주소의 과정으로 생성됨.
- 개인키로 Uncompressed Format의 공개키를 생성하고 여기에 해시 값을 (SHA256 & RIPEMD160) 취해 160 bit의 공개키 해시 값을 계산함.
- 공개키 해시 값 앞에 Version prefix를 추가한 후 Base58Check 인코딩으로 지갑 주소를 생성함.
  Version prefix : 0x00 ~ 일반 지갑 주소 (mainnet), 0x05 ~ P2SH용 주소 (나중에 Transaction 편에서 다룸)
  0x6f ~ 개발자 테스트용 비트코인 네트워크 (testnet)용 지갑 주소
- mainnet의 주소는 '1'로 시작하고, P2SH용 주소는 '3'으로 시작하고, testnet용 주소는 'm or n'으로 시작함.
- 지갑 주소는 Uncompressed 형태의 공개키로 만드는 것이 일반적이나, Compressed 형태의 공개키로 만들 수도 있음. 그러면 공개키 해시 값도 달라지고 지갑 주소도 달라짐. → 비트코인 어플리케이션의 호환성 문제임.
- 지갑 주소는 외우기 어려운 형태이므로 아래 예시와 같이 QR 코스 형태로 다른 사람에게 공개하기도 함.



* 지갑 주소 예시

- Mainnet : 182mocjYj7PjCnJxUTSrVCK9dNkbjNqarA
- P2SH : 3H1FfAYW5sDYRqJCxZ3G7Jpf4jgdtigbk1
- Testnet : mnYj6fpXY8pyytnaC2REK7XUVNMJhs8AKL

QR code →

(실습 파일 : 3-6.지갑주소.py)

지갑 주소 생성 – Python 연습

- 개인키, 공개키, 공개키 해시 값을 생성하고, 지갑 주소를 생성함 (mainnet용, testnet용 지갑 주소 생성). 공개키 해시 ←→ 지갑 주소 변환도 확인함.

```
# 개인키와 공개키를 생성하고 공개키로부터 지갑 주소를 생성한다.
# 참고 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
#
# 2018.4.17 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------
import bitcoin.main as btc

# 개인키를 생성한다
while (1):
    privKey = btc.random_key()                      # 256 bit Random n
    dPrivKey = btc.decode_privkey(privKey, 'hex')   # 16진수 문자열을 1
    if dPrivKey < btc.N:                            # secp256k1 의 N 노
        break

# 개인키로 공개키를 생성한다.
pubKey = btc.privkey_to_pubkey(privKey)

# 공개키로 지갑 주소를 생성한다. (mainnet 용)
address1 = btc.pubkey_to_address(pubKey, 0)

# 공개키로 160-bit public key hash를 생성한다
pubHash160 = btc.hash160(btc.encode_pubkey(pubKey, 'bin'))

# 160-bit public key hash로 지갑 주소를 생성한다. (위의 addrerss와 동일하
address2 = btc.hex_to_b58check(pubHash160, 0)

# 지갑 주소를 160-bit public key hash로 변환한다. (위의 pubHash160과 동일
pubHash1601 = btc.b58check_to_hex(address2)

# 공개키로 testnet용 지갑 주소를 생성한다
address3 = btc.pubkey_to_address(pubKey, 0x6f)
```

Name	Size	Type	Date Modified
3-5.공개키(Formats).py	2 KB	py File	2018-04-17 오후 2:21
3-6.지갑주소.py	1 KB	py File	2018-04-18 오전 2:20
3-7.BrainWallet.py	1 KB	py File	2018-04-18 오후 12:21

IPython console

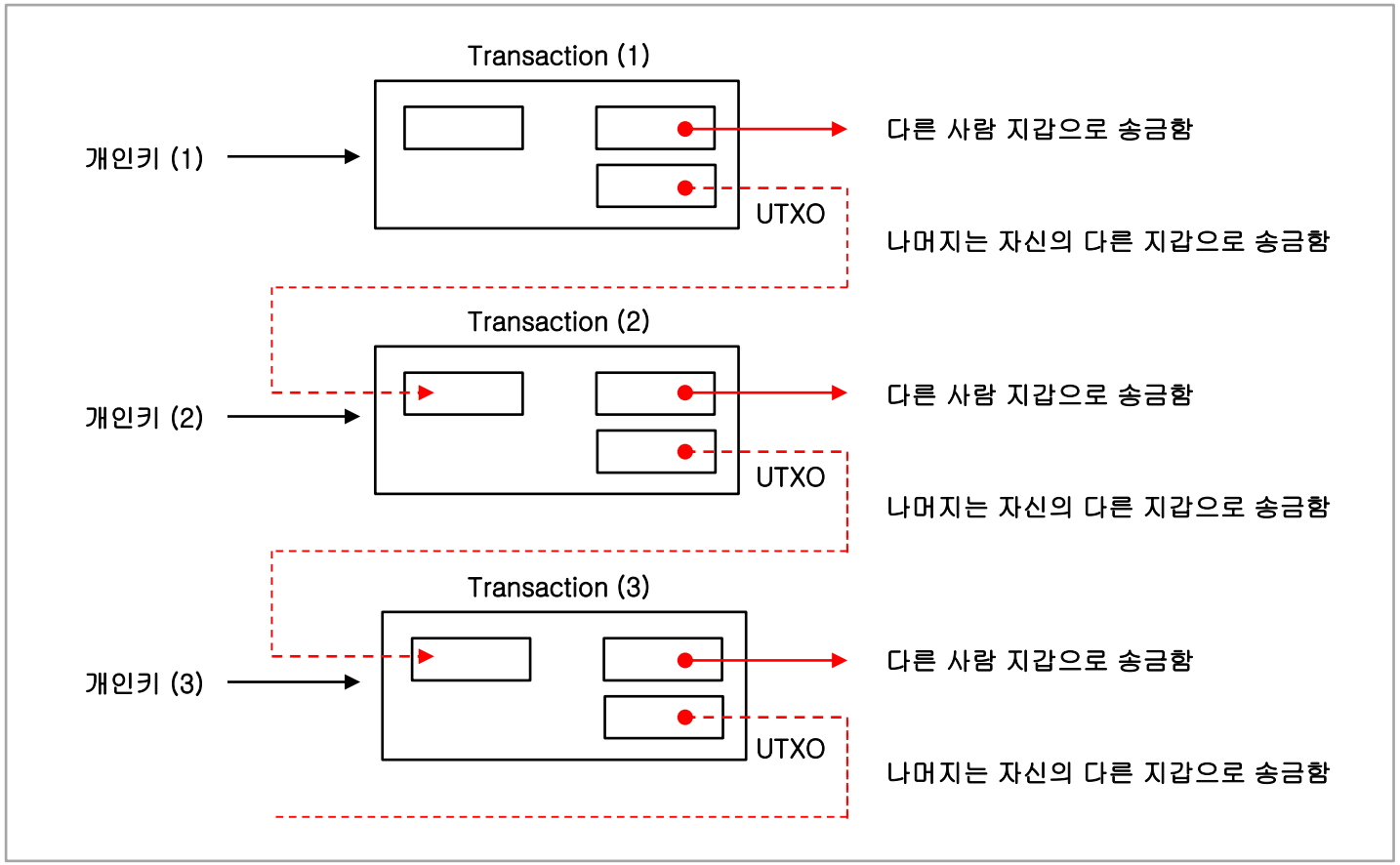
Console 1/A

```
개인키 :  970092e03adff47b2a9d9a40023c6557f1827b81a6fa62e1f53602d1d0b526bd
개인키 --> 공개키 :
04252f059f96ee3d9415bcd7d1576718a3805f0958942b3bcd5d2c4fdb96ef16fe84745e95770946d3021d8b6
2b80206210445d3c814960ab699362dd826caa049

공개키 --> 지갑주소 (1. mainet 용) :  1J9hADkjGQVc6MJaWifgTEvtKGa4fr116o
공개키 --> 공개키 해시 :  bc1efd314840e24202fdf9d7faafa712a93be1d3

공개키 해시 --> 지갑주소 (2. mainet 용) :  1J9hADkjGQVc6MJaWifgTEvtKGa4fr116o
지갑주소 --> 공개키 해시 :  bc1efd314840e24202fdf9d7faafa712a93be1d3
지갑주소 (Testnet 용) : mxfeTGqi5RvrsTnCEHe4HA9DBGAmbaR8xA

In [40]:
```

History log | IPython console

지갑 주소 관리 : 다수의 주소 사용

- 최근 지갑 어플리케이션 들은 개인키, 지갑 주소를 하나만 운영하는 것이 아니라, 보안상 안전을 위해 여러 개의 개인키를 생성하여 운용함 (ex : 100개).
- 거래할 때마다 다른 개인키로 지갑 주소를 새로 만들어서 운용함. 일회용 암호 (One Time Password : OTP) 처럼 한 번 쓰고 버리는 개념임 (Recommended).
- 예를 들어 Transaction (1)에서 다른 사람에게 송금하고 남은 나머지는 다시 자신에게 송금해야 하는데, 이 때 기존 주소를 사용하는 것이 아니라, 새로운 개인키 (2) 로 생성한 지갑 주소로 송금함.
- Transaction (2)에서는 다시 새로운 지갑으로 송금함.
- 입금시에도 사람들에게 다른 주소를 알려 줄 수 있음.
- 통합 잔고 관리, 키 백업 등은 지갑 어플리케이션에서 자동으로 관리함.
- 개인키가 계속 바뀌기 때문에 해킹으로부터 안전할 수 있음.
- 반면, 개인키가 여러 개이고 계속 바뀌면 백업 관리에 문제가 발생할 수도 있음. 빈번한 백업 관리가 필요함.
- 대부분 지갑 어플리케이션들은 백업 기능을 제공하기는 하지만, 디스크, USB, 휴대용 메모리 등으로 백업이 가능하기 때문에 전자 장치에 의존해야 함.
- 만약 자주 거래하지 않고, 가끔씩 거래하면서 자산을 장기간 보유하는 사용자라면 (수 년 정도) 자신의 개인키를 페이퍼에 기록 보관하는 것이 안전할 수도 있음. 이런 경우 거래할 때마다 개인키가 바뀌면 지속적으로 기록 관리하기가 용이하지 않을 수도 있고, 오히려 관리 소홀로 분실 위험 등이 발생할 수도 있음. → Deterministic wallet의 필요성.



지갑 백업 관리 : Paper Wallet

- 지갑을 백업한다는 것은 개인키를 백업한다는 의미임. 개인키만 있으면 공개키, 지갑 주소 등은 얼마든지 생성할 수 있으므로 개인키만 안전하게 보관하면 됨.
- 개인키를 CD, USB, 외장 하드 디스크 등의 전자 장치에 백업하는 것은 일차적인 백업 개념임 (Warm Storage). 전자 장치에 백업하는 것은 편리하기는 하지만 고장, 실수, 해킹 등의 위험이 있으므로 장기간 보관용으로는 안전하지 않을 수도 있음.
- 최종적인 백업은 개인키를 물리적으로 어딘가에 기록해 놓는 것임. 아래 예시와 같이 종이에 기록해 놓는 것을 Paper Wallet이라 함 (Cold Storage)
- 아래 예시는 http://bitaddress.org 에서 JavaScript 형식으로 사용자 컴퓨터에서 실행되는 스크립트로 만든 Paper Wallet 임. 인터넷에 접속한 상태에서 Paper Wallet을 생성하면 개인키가 해당 사이트로 노출될 위험이 있으므로, 웹 브라우저에서 해당 사이트 페이지 (HTML)를 저장해둔 다음, 인터넷 접속을 차단하고, 웹 브라우저에서 저장된 HTML을 읽어서, 아래와 같은 Paper Wallet을 생성한 후 프린트 출력하면 안전하게 생성할 수 있음.
- Paper Wallet이 완벽히 안전한 방식은 아니더라도 가장 안전한 방식이라고는 할 수 있음. (도난, 분실, 화재 등의 위험은 존재함.)
- 도난을 방지하기 위해서는 암호화된 개인키 (BIP 38)를 사용할 수도 있음. 사용자가 설정한 Passphrase를 이용하여 개인키를 암호화해서 Paper Wallet으로 보관하면 Passphrase를 모르는 사람은 Paper Wallet 상의 개인키를 사용할 수 없음.
- 비트코인을 보유한 사람이 불의의 사고 등으로 개인키를 더 이상 사용할 수 없으면 블록체인에 기록된 해당 지갑의 UTXO는 영원히 아무도 사용할 수 없는 상태로 남아있게 됨.
- 은행이나 증권, 보험사 등 일반 금융권에 보관된 자산은 상속이 이루어질 수 있지만 비트코인은 관리 주체가 없기 때문에 개인키를 전달해 주지 않으면 상속이 이루어 질 수 없음. Paper Wallet으로 개인키를 보관하고 가족끼리 공유할 필요도 있음.

* Paper Wallet 예시

출처 : http://bitaddress.org

(실습 파일 : 3-7.BrainWallet.py)
(실습 파일 : 3-8.VanityAddress.py)

지갑 키 관리 : Brain Wallet과 Vanity Address

- Brain wallet은 개인키를 랜덤하게 생성하지 않고 문자형 암호 (Passphrase)를 사용하는 방식임. 문자열만 기억하고 있으면 언제든지 개인키를 생성할 수 있음.

```
# Brain Wallet 시험
# 임의의 문자열 (Passphrase)로 개인키와 공개키를 생성하고 공개키로부터 지갑
#
# 2018.4.17 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------
import bitcoin.main as btc

# 특정 문자열로 256-bit 개인키를 생성한다 (long brainwallet passphrase).
passphrase = 'Brain Wallet 시험용 개인키 입니다. 잊어버리지 마세요.'
privKey = btc.sha256(passphrase)
dPrivKey = btc.decode_privkey(privKey, 'hex')   # 16진수 문자열을 10진수
if dPrivKey < btc.N:                            # secp256k1 의 N 보다 작
```

```
Passphrase : Brain Wallet 시험용 개인키 입니다. 잊어버리지 마세요.

개인키 : 82909821248a8e60f44ebefd37ee762442cddf20f3303d03751e9874939c6a72
개인키 --> 공개키 :
044d044117080d94266a1b170e2987a298aa3e897cda0855fc75253da01961968cab54dcb99fc32f33bb7ed1
a3fb6e260cca8e2fe01e887c646af1ed565f602b00

공개키 --> 지갑주소 : 1Jv3z4v3TKizgGUvRGbGRkVCYTpm1vhDof

In [36]:
```

Passphrase만 기억하고 있으면 언제든지 이 개인키와 지갑 주소를 만들어 낼 수 있음.

History log | IPython console

- Vanity wallet은 지갑 주소의 특정 위치에 사용자가 원하는 문자열이 나오도록 한 것임. 사용자 자신이나 다른 사람들이 알아보기 쉽게 한 것임.
- 주소에 원하는 문자열이 나타날 때까지 랜덤 개인키를 반복 생성함. (Brute Force 방식). Base58 인코딩에 사용되는 문자열만 가능함 (0, O, I, l, +, - 는 제외함).

```
# Vanity Address 시험
# 지갑 주소 앞 부분에 원하는 문자열이 포함되는 개인키를 찾는다 (Brute Force
#
# 2018.4.17 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------
import bitcoin.main as btc

bFound = False
for i in range(10000):
    # 개인키를 생성한다
    while (1):
        privKey = btc.random_key()                  # 256 bit Rando
```

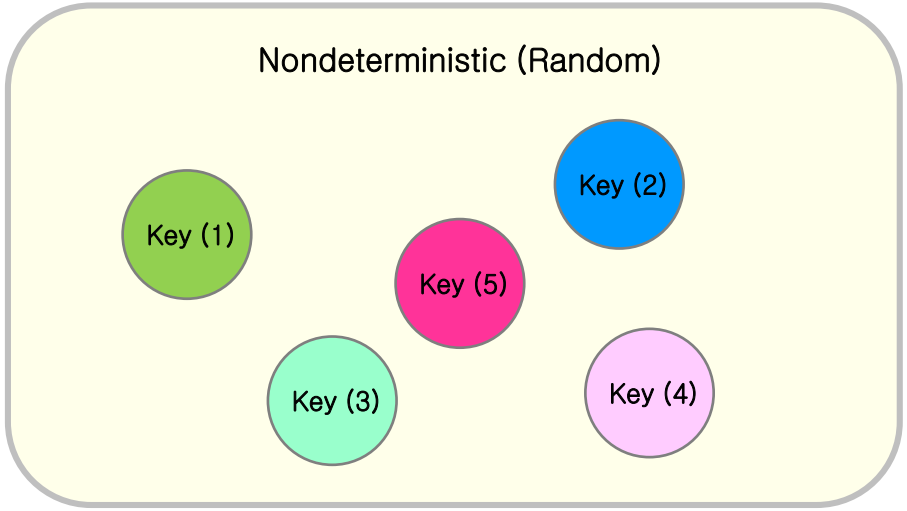
```
개인키 :  b76caed6e0957a5ecb6944d3958fd559cb4f3e9938dff10b51e9583772738069

개인키 --> 공개키 :
048f2c6af7621edce3de9fd3fed6b1868789cc2b2a3cffccca72c0e310096665f0d828c43dc09e852cc4f7b9
01737dd44e2b947da2205b09611157d75def2aefd8

공개키 --> 지갑주소 :  1Choy6HD1tAWtVLc4s32SRzLFjhgEevMms

In [32]:
```

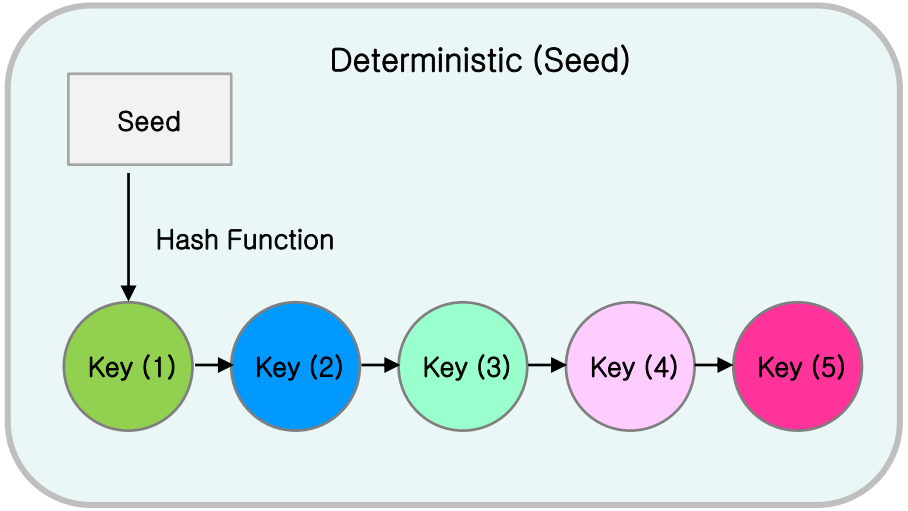
앞 부분 세 문자열이 'Cho' 가 나오도록 하였음.

History log | IPython console

지갑 키 관리 : Nondeterministic & Deterministic Wallets

- 다수의 지갑 주소를 Random 생성하는 방법은 Nondeterministic wallet에 해당함. 각각의 키를 독립적으로 생성하기 때문에 키 간의 연관성이 없음.
- Nondeterministic wallet은 키들 간의 연관성이 없기 때문에 안전한 방식일 수 있으나, 빈번히 바뀌는 많은 키를 자주 백업 관리하기 어려움.
- Deterministic wallet은 Seed 값을 생성 (Random 혹은 Passphrase, Mnemonic words) 한 후 해시 함수를 이용하여 다수의 키를 생성함.
- 예를 들어 Seed의 해시 값으로 마스터 키를 생성하고, 마스터 키의 해시 값으로 다음 키를 생성하고, 또 다음 키의 해시 값으로 그 다음 키를 생성할 수 있음.
- 다른 방법으로는, Seed 값에 index 번호를 붙여 해시 값을 계산해서 각 키를 생성할 수도 있음 (ex : Hash(seed + '5') = Key-5).
- Deterministic wallet은 키 간의 연관성이 존재할 수 있지만 Seed 값만 백업 보관하면 나머지 키들은 언제든지 복원할 수 있음.
- Nondeterministic wallet은 Type-0 wallet으로 "Just bunch of keys"의 의미로 JBOK wallet이라고도 하고, Deterministic wallet은 Type-1 wallet으로 Seeded wallet이라고도 함.
- Deterministic wallet은 Tree 형태로도 발전하고 있으며, BIP-32로 제안된 HD wallet (Type-2) 방식이 널리 사용되고 있음.

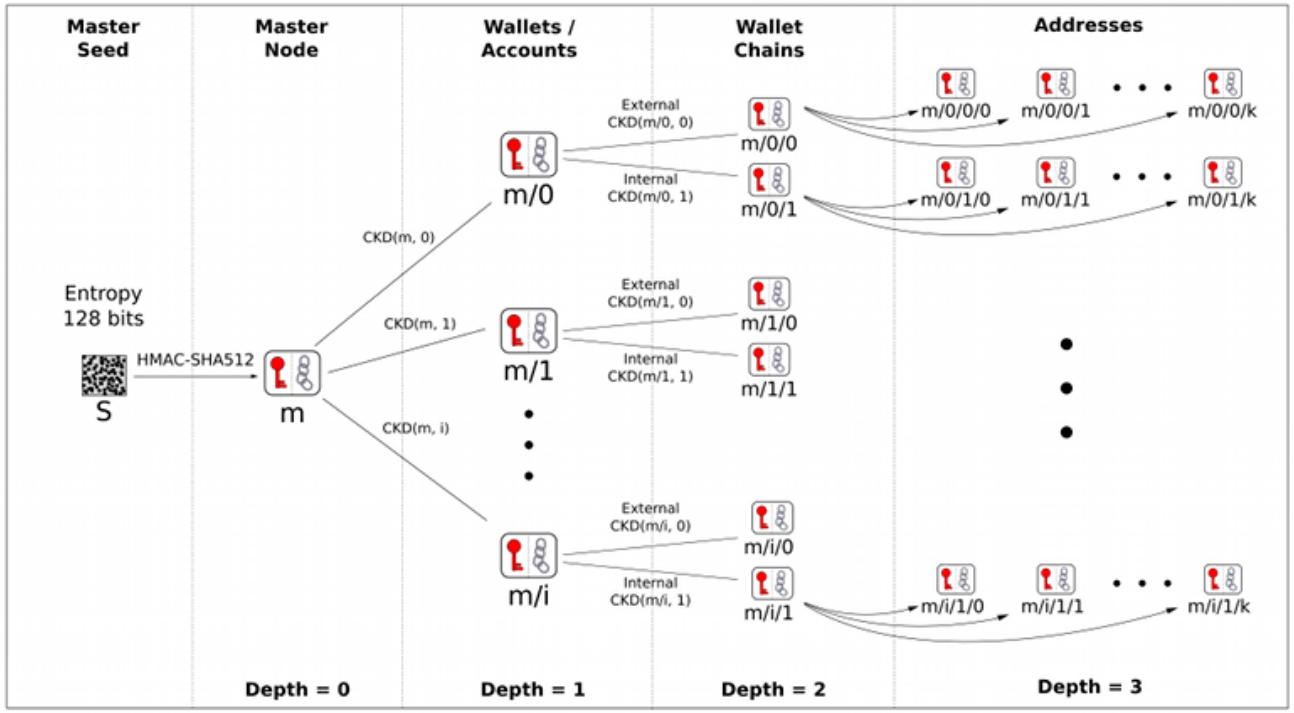




지갑 키 관리 : HD Wallets (Hierarchical Deterministic Wallets, BIP-32)

- HD wallet은 계층 구조로 키와 지갑 주소를 관리할 수 있음. Seed 값으로 마스터 개인키와 마스터 공개키를 생성하고 마스터 키로 하위 계층의 키와 주소를 생성함.
- Seed와 마스터 키를 사용하므로 Type-1 유형이고, 여기에 계층 구조를 적용하여 Type-2 유형으로 발전한 것임. (2012년 BIP-32로 제안된 이후 계속 발전하고 있음)
- Seed 값만 백업 보관하면 하위 계층의 키와 주소들은 모두 원래 상태로 복원할 수 있으므로, 백업 관리가 용이함.
- 계층 구조의 장점으로는, Tree의 Branch 별로 지갑의 용도를 달리할 수 있음. 예를 들어 Branch (A)에 속한 지갑은 수신 전용으로 사용할 수 있고, Branch (B)에 속한 지갑은 송금 후 잔돈을 수신할 목적으로 사용할 수 있음. 또한, 기업에서 부서 별로 지갑을 운영할 수도 있음.
- HD wallet의 또 다른 큰 장점은, 하위 계층의 지갑 주소는 개인키 없이 공개키 만으로도 만들 수 있어 개인키를 노출하지 않아도 된다는 것임.
- Seed 값으로 만든 마스터 개인키로 하위 계층의 개인키를 만들고, 마스터 공개키로 하위 계층의 공개키와 지갑 주소를 만들 수 있음. 즉, 하위 계층의 공개키는 개인키로 만드는 것이 아니라 상위 계층의 공개키로 만들 수 있음. 지갑 주소를 만들 때 개인키가 사용되지 않음. 그러면서도 하위 계층의 개인키 → 공개키 관계가 성립하기 때문에 그 지갑의 잔고를 사용할 수 있음 (개인키로 전자서명을 할 수 있음)
- 이 장점을 활용한 예시로는, 인터넷 쇼핑몰에서 외부 호스팅 서버에 상품 결제를 위한 지갑 주소를 올려 놓으면서, 지갑 주소를 자주 바꿀 경우에 유용하게 사용 할 수 있음. 호스팅 서버에는 공개키와 지갑 주소만 등록되고 개인키는 등록되지 않음.
- 하위 계층의 공개키는 Child Key Derivation (CKD) 함수가 사용되고, 하위 계층 별, 그룹 별로 지갑을 관리할 수 있음.

BIP 32 – Hierarchical Deterministic Wallets



Child Key Derivation Function ~ $CKD(x,n) = HMAC - SHA512(X_{chain}, X_{pubkey}||n)$

출처 : https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki

(실습 파일 : 3-9.HDwallet.py)

지갑 키 관리 : HD Wallets – Python 연습

- Vitalik의 pybitcointools 라이브러리를 활용하여 HD wallet 기능을 확인함. 계층-3의 첫 번째 Branch에 여러 지갑 주소를 생성함.

```
# HD wallet 기능을 시험한다. (BIP-32)
# Hierarchical Deterministic wallets
# 참고 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
#
# Master Private key와 Master Public key를 생성하고 하위 계층으로 키를 전파한다
# 하위 계층의 공개키는 개인키로 만들지 않고 상위 계층의 공개키로 만든다.
# 하위 계층의 지갑 주소는 개인키 없이 공개키 만으로 만들 수 있다.
# 그 계층의 개인키로 공개키를 생성한 결과와 잘 일치하는지 확인한다.
# Seed 값만 백업해 두면 하위 계층의 키와 주소를 모두 복원할 수 있다.
#
# 2018.4.19 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
import bitcoin.main as btc
import bitcoin.deterministic as det

# seed 값인 단어 목록 (Mnemonic words)를 임의로 설정하고
# Master private key와 master public key를 생성한다.
seed = b'cho previous andante apple hyun solee clk learn'
mPrv = det.bip32_master_key(seed)  # master private key
mPub = det.bip32_privtopub(mPrv)   # master public key

# 계층-1의 extended private key와 public key를 생성한다.
# Depth-1 (m=0, m/1)
xPrv01 = det.bip32_ckd(mPrv, 1)     # 마스터 개인키로 Depth-1의 개인키를 생성한
xPub01 = det.bip32_ckd(mPub, 1)     # 마스터 공개키로 Depth-1의 공개키를 생성한

# 계층-2의 extended private key와 public key를 생성한다.
# Depth-2 (m/1/0)
xPrv010 = det.bip32_ckd(xPrv01, 0)  # Depth-1의 개인키로 Depth-2의 개인키를 생
xPub010 = det.bip32_ckd(xPub01, 0)  # Depth-1의 공개키로 Depth-2의 공개키를 생
```

Name	Size	Type	Date Modified
3-8.VanityAddress.py	1 KB	py File	2018-04-18 오후 12:15
3-9.HDwallet.py	2 KB	py File	2018-04-19 오전 3:46

IPython console

Console 1/A

```
Depth-3 (m/1/0/0) :

xPrivate Key =
xprv9yekT3VbsD5UUZNL7Vg9DkfYzuCnbzPHZcKcMiAy9meXugcnDrf7znpLZekyDYctAxpiaZaYBno8v
PyptozPHm4knTjb7SpKFSjCn1e4BHq

xPublic Key =
xpub6Ce6rZ2Vhadmh3SoDXD9atcHYw3H1T78vqFDA6aai7BWnUwvmPyNYb8pQuaX4BrJuCGud9tPm7ggd
hJps9XoDwgEusZGqnApvfAdFKmrxkM

Private Key = ba22671f8898e5e8d317205317a3e6155f02da9ef205d40316adee1f9fadc83e01
Public Key = 029e64bd351a1c8330f22664ece4cd5ec3de6c07790db968c1bdbfe170fa5cda81
Address = 1G71vZSw3kj9ZnLvAzjhaGjqnrwTwwQsX5

Private Key --> Public Key 관계가 잘 성립함.

Depth-3의 지갑 주소는 개인키 없이 무수히 만들어 낼 수 있음.
m/1/0/1 : 1Loe7BnXch8G3iQPVC6fu8iEeHEF85urFX
m/1/0/2 : 18ghKGzP8VLo7mz1m2TEynJRvKUwZTZTiU
m/1/0/3 : 1CJYBpvFHyygTnTfZgvx4ix6NJEjg1yXAw
m/1/0/4 : 1CC9LAUaYfvqr3mV5wbmK7dovKBdFGnK3p
m/1/0/5 : 1BK8U7SqtnAVaVppuTqYcJiksuWuS7x2e6
m/1/0/6 : 1DuXvRDma4FrQAnKz7NsiAZwRXej39siks
```

HD wallet의 큰 장점 중 하나임

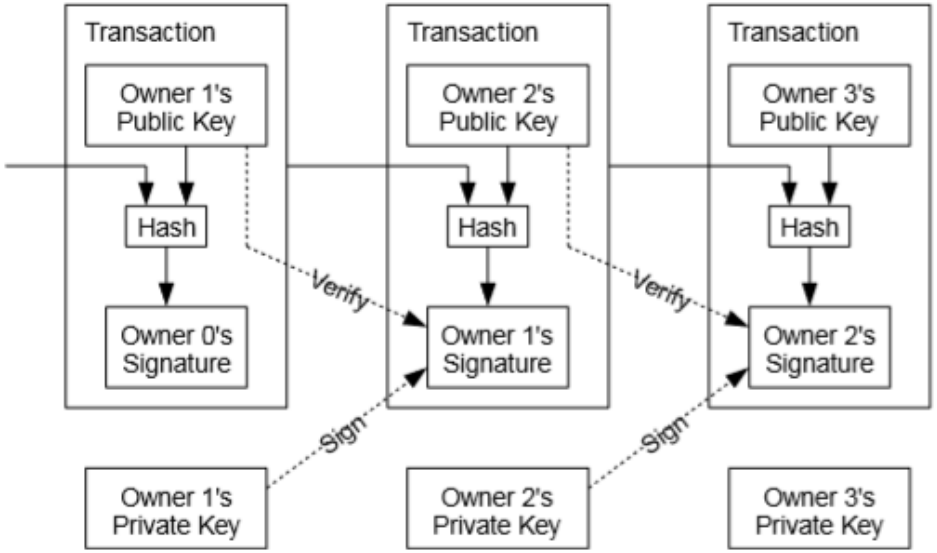
History log | IPython console

지갑 키 관리 : Mnemonic Code (BIP-39)

- Deterministic wallet의 Seed 값은 Random 생성, Passphrase (문장), Mnemonic Code (단어 목록)등을 이용할 수 있음.
- 단어 목록의 경우 사용자가 임의로 지정한 단어들을 사용할 수도 있고, 사전에서 랜덤하게 단어들을 선택할 수도 있음
- 사용자가 임의로 문장이나 단어 목록을 지정하면 Brain wallet 방식이 되고, 랜덤하게 선택하면 Mnemonic Code wallet 방식이 됨.
- 일반적으로 사용자가 지정하는 방식보다는 사전에서 랜덤하게 선택하는 방법이 훨씬 안전한 것으로 알려져 있음.
- BIP-39는 단어들을 랜덤하게, 안전하게 선택하는 방법을 제안하고 있음.
- HD wallet의 Seed 값을 Mnemonic Code로 생성하면 효과가 극대화됨 (BIP-39 + BIP-32).
- 최근 대부부의 지갑 어플리케이션들은 BIP-39와 BIP-32를 지원하고, Seed 값의 백업, 복원 기능을 지원하고 있음.

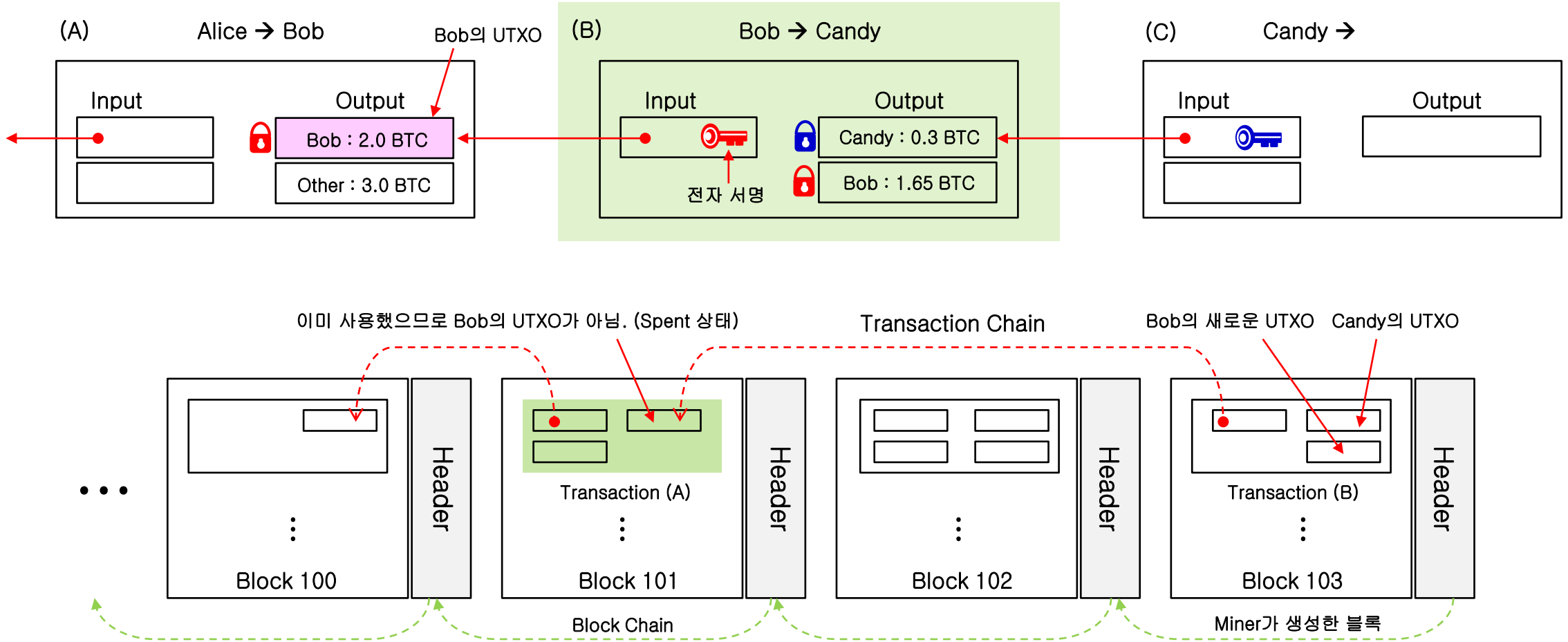
4. 거래 (Transaction)

4-1. Transaction 구조
4-2. ECDSA 전자서명과 Script
4-3. ECDSA 전자서명 검증
4-4. UTXO 조회
4-5. Transaction 생성
4-6. Transaction Malleability (거래 데이터 조작 가능성)
4-7. 다중 서명 (Multisig)
4-8. Pay-to-Script Hash (P2SH)
4-9. P2SH 주소 생성 (BIP-13)
4-10. P2SH 와 Multisig 거래 생성
4-11. Segregated Witness (SegWit : BIP-141)
4-12. SegWit과 Merkle Tree
4-13. SegWit Transaction 구조 (BIP-143, 144)
4-14. Bech32 주소 생성 (BIP-173)
4-15. SegWit의 Backward compatibility
4-16. Anyone-can-spend Transaction



Transaction 구조 리뷰

- 비트코인 Transaction의 구조는 아래와 같음. Bob → Candy에게 보내는 Transaction (B)는 Input과 Output으로 구성되어 있고, Input은 이전 Transaction과 연결되어 있음 (UTXO). 이러한 연결 구조로 인해 이중지불 문제, 사용자 인증 문제, 위,변조 방지 문제 등을 해결하고 있음.



참고 : https://bitcoin.org/en/developer-reference#raw-transaction-format

Transaction 구조 : Raw Data

- Transaction의 Raw Data는 아래와 같이 구성되어 있음.
- 크게 Input과 Output으로 구성되어 있음. Input은 이전 UTXO의 Transaction을 가리키는 정보이고, Output은 송금할 금액과 받을 지갑의 주소 정보임.
- Input에는 송금자의 전자 서명과 공개키가 들어있고, Output에는 수신자의 공개키 해시 값이 들어있음.
- 개수 (count)와 길이 (length)는 고정 byte가 아닌 Compact size uint 형태임. (우측)

* Compact Size uint Format

Value	Bytes	Format
>= 0 && <= 252	1	uint8_t
>= 253 && <= 0xffff	3	0xfd followed by the number as uint16_t
>= 0x10000 && <= 0xffffffff	5	0xfe followed by the number as uint32_t
>= 0x100000000 && <= 0xffffffffffffffff	9	0xff followed by the number as uint64_t

For example, the number 515 is encoded as 0xfd0302.

* Transaction Format

Name	Bytes	Description	예 시
Version	4	Transaction version 현재 버전은 '1'	01000000
input count	*C	Input 개수	01
previous output	32	Previous output의 Transaction ID (Hash pointer)	1e5080555fba889f404285d50bc398b98385eb314e6424f76a1a3691baf7ff31
output index	4	Previous output들 중 잔고를 사용할 output 번호	01000000
Script length	*C	Script 길이	8a
Script Sig	Var	Signature script. (Unlocking script), 전자서명	473044022041312c66889be78a8705e56cb336e395e5d8084a00e9bff80bbd0cece1cb1cd9022054484f4b8c69b300e8328e326564721a34099e5dca0bdecccb0fa01ad68915e9014104f5aa8ea9a29b5629c4e833d5431e2313cabd46f4dbcc69770fc513d7b35f980f7159a911f967dcf73cb15a8c8bb7b382b66963d3a5244bdb2e7698afb319c882
Sequence	4	Sequence 번호.	ffffffff
output count	*C	Output 개수	01
value	8	송금할 금액. satoshi 단위 (1e+8) 임.	404b4c0000000000
Script length	*C	Script 길이	19
Script PubKey	Var	Locking Script. 수신자의 공개키 해시 값	76a914063bb1590a928aeb9441f81f4e1f6ffd8488f4c988ac
Lock Time	4	500 million 이하이면 block height, 아니면 Unix timestamp	00000000

*C = CompactSize uint

Transaction 구조 : Script Sig (Signature Script, Unlocking Script)

- Transaction Input에 있는 Script Sig의 구조는 아래와 같음. Script Sig는 송신자인 Bob이 자신의 개인키로 이 Transaction 데이터에 서명한 값임.
- 전자 서명은 ECC (타원곡선 암호) 방식의 Digital Signature (ECDSA : Sig-r 과 Sig-s)이고, 다른 노드들이 서명을 검증 (Verification)할 수 있도록 서명자 (Bob)의 공개키를 첨부함. 공개키는 Uncompressed Format (04로 시작)이나 Compressed Format (02나 03으로 시작)을 사용할 수 있음. Compressed Format을 사용하면 Transaction Data의 크기를 줄일 수 있음. 수수료도 줄일 수 있음.
- Script Sig는 이전 UTXO의 output에 Script pubKey로 잠긴 잠금 장치를 풀 수 있는 열쇠의 역할을 함.
- Script Sig는 ASN.1의 DER-encoding (Distinguished Encoding Rules) 방식인, Sequence, Length, [Tag-Length-Value, Tag-Length-Value] 형태로 표시함 (2015.1, BIP-66에 의해 DER-encoding을 엄격히 따르도록 하였음. Transaction Malleability를 방지하기 위함.).

DER Encoding 구조

	Name	bytes	예 시
	signature length	1	47 : PUSHDATA ~ 0x47 (71) bytes를 stack에 push 하라는 의미
	sequence	1	30 : DER encoding의 sequnce를 의미함. Tag-Length-Value의 sequence를 의미함.
	script length	1	44 : Script length (68 bytes)
Tag-Length-Value	tag	1	02 : Tage - integer
Tag-Length-Value	sig-r length	1	20 : Signature-r length (32 bytes)
Tag-Length-Value	sig-r	Var	41312c66889be78a8705e56cb336e395e5d8084a00e9bff80bbd0cece1cb1cd9 : Signature-r
Tag-Length-Value	tag	1	02 : Tage - integer
Tag-Length-Value	sig-s length	1	20 : Signature-s length (32 bytes)
Tag-Length-Value	sig-s	Var	54484f4b8c69b300e8328e326564721a34099e5dca0bdecccb0fa01ad68915e9 : Signature-s
	hash type	1	01 - 01: SIGHASH_ALL, 02: SIGHASH_NONE, 03: SIGHASH_SINGLE
	public key length	*C	41 - Public key 길이
	public key	Var	04f5aa8ea9a29b5629c4e833d5431e2313cabd46f4dbcc69770fc513d7b35f980f7159a911f967dcf73cb15a8c8bb7b382b66963d3a5244bdb2e7698afb319c882 - Public Key (Uncompressed Form) : Compressed form이 사용될 수 있음.

Transaction 구조 : Serialization

- 비트코인 송금 거래 시 Transaction Data를 생성하고 Serialization 시킨 후 네트워크의 인근 노드들에게 전송함.
- Serialization 데이터는, Script (Script Sig, Script PubKey) 부분을 제외하고 , 모두 little endian 형태임. (ex : 송금 금액 = 0.14072919 BTC = 14,072,919 satoshi = 0x 0000000000d6bc57 (8byte padding, big-endian) → 0x 57bcd60000000000 (little-endian))

* Serialized Data

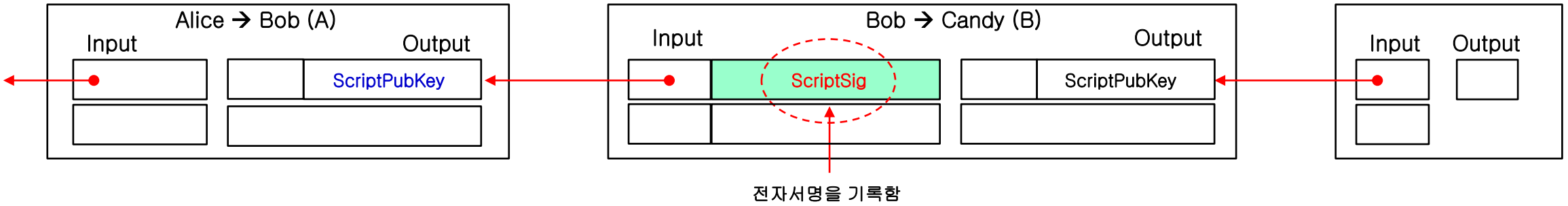
```
01000000011e5080555fba889f404285d50bc398
b98385eb314e6424f76a1a3691baf7ff310100000
08b483045022100d2dc378542d152d6f568b5c8
e491d809f3a634fc690092b29c44795e0fd69cf50
22032a86e22b7b86295b98000e01e7918c5b22a
70fcfa7c97ed05a0544f0478e58e014104f5aa8ea
9a29b5629c4e833d5431e2313cabd46f4dbcc69
770fc513d7b35f980f7159a911f967dcf73cb15a8
c8bb7b382b66963d3a5244bdb2e7698afb319c8
82ffffffff0240420f00000000001976a914063bb15
90a928aeb9441f81f4e1f6ffd8488f4c988ac57bcd
60000000000001976a914138a87f639de8074c46e6
200f7bbe57ea1db0d4e88ac00000000
```

* Deserialized Data (http://blockchain.info/decode-tx)


```
{
   "lock_time":0,
   "size":258,
   "inputs":[
      {
         "prev_out":{
         "index":1,
         "hash":"31fff7ba91361a6af724644e31eb8583b998c30bd58542409f88ba5f5580501e" },
"script":"483045022100d2dc378542d152d6f568b5c8e491d809f3a634fc690092b29c44795e0fd69cf5022032a86e22b7b86295b98000e01e7
918c5b22a70fcfa7c97ed05a0544f0478e58e014104f5aa8ea9a29b5629c4e833d5431e2313cabd46f4dbcc69770fc513d7b35f980f7159a911f9
67dcf73cb15a8c8bb7b382b66963d3a5244bdb2e7698afb319c882" } ],
         "version":1,
         "vin_sz":1,
         "hash":"9cdb2d01ea0c249d385fe9c0c7e49bc9e199ceaeb0a438f08084d11ad7d971e3",
         "vout_sz":2,
         "out":[ {
             "script_string":"OP_DUP OP_HASH160 063bb1590a928aeb9441f81f4e1f6ffd8488f4c9 OP_EQUALVERIFY OP_CHECKSIG",
             "address":"1ZxZgGRknj5ua6XZCyBkoVzTyay1LWy4w",
             "value":1000000,
             "script":"76a914063bb1590a928aeb9441f81f4e1f6ffd8488f4c988ac" },
           { "script_string":"OP_DUP OP_HASH160 138a87f639de8074c46e6200f7bbe57ea1db0d4e OP_EQUALVERIFY OP_CHECKSIG",
             "address":"12nKnxMdcm545aq9NT9fMgYqLrRQKGRAkE",
             "value":14072919,
             "script":"76a914138a87f639de8074c46e6200f7bbe57ea1db0d4e88ac" } ]
}
```

Digital Signature 생성 : ECDSA

- Bob이 Transaction (A)의 UTXO를 소비할 권한이 있는지를 증명하려면 자신의 개인키로 서명한 (ECDSA) 결과 (s, r)와 공개키를 ScriptSig에 기록해야 함.



서명 절차

- Input 마다 UTXO를 하나씩 가지고 있기 때문에 모든 Input에 대해 하나씩 서명해야 함. ScriptSig를 제외한 모든 데이터가 기록된 상태에서 서명함.
- 서명할 Input의 ScriptSig에는 UTXO의 ScriptPubKey 값을 복사하고 (not technical reason, but historical reason), 다른 Input의 ScriptSig는 blank 상태로 만듦.
- Transaction 데이터 전체를 Serialization 하고, 전체를 서명했다는 의미인 SIGHASH_ALL = 0x01 (Hash type)을 맨 뒤에 붙임 (전체가 아닌 일부만 서명할 수도 있음. 다음 페이지 참조).
- 위의 데이터가 서명할 메시지 (m)에 해당함 (ECDSA의 메시지 (m) 임).
- 메시지 (m)의 해시 값을 계산함. hash(m) = SHA256(SHA256(m)). ← double-SHA256 해시
- Bob의 개인키으로 메시지 해시 값에 서명하여 s, r을 계산함. (s, r) = sig(hash(m))
- 서명 결과인 s, r 과 Hash type (SIGHASH_ALL = 0x01), 그리고 Bob의 공개키를 합쳐서 DER-Encoding 구조로 ScriptSig에 기록함.
- 다른 Input도 동일한 절차로 서명함. 다른 Input을 서명할 때는 이미 서명이 완료된 Input의 ScriptSig는 blank로 지우고 서명함.

Digital Signature : SigHash Type

- 전자서명은 Transaction의 모든 Input과 Output을 대상으로 할 수도 있고, 일부 Input과 Output을 대상으로 할 수도 있음.
- ALL + ANYONECANPAY 유형은 해당 Input과 모든 Output에 대해 서명하는 것으로, Crowdfunding 같은 거래에 활용할 수도 있음 (Mastering Bitcoin 참조).
- Crowdfunding의 예시 :
  - 어떤 사람 (A)이 Input은 없고, 자신에게 10.0 BTC를 보내는, output만 있는, Tx를 생성하여 전파함. 이 Tx는 유효하지 않으므로 다른 노드들이 무시할 것임.
  - 만약 어떤 노드가 이 Tx에 관심이 있고 funding 하기 위해, 2.0 BTC를 사용하는 Input을 만들어 넣고 ALL + ANYONECANPAY 방식으로 서명하여 전파함.
  - 이 Tx도 Input = 2.0, Output = 10.0 이므로 유효하지 않아, 다른 노드들이 Reject 함.
  - 또, 관심있는 다른 노드들이 여기에 Input을 추가하여 총 Input이 10.0보다 커지면 이 Tx는 유효해 짐. 최초 Tx 생성자 (A)는 10.0 BTC를 모집할 수 있음.
  - 현실적으로 가능할 지는 의문이지만 (유효하지 않은 Tx는 널리 전파되지 못하고, 유효하지 않은 Tx를 전송한 노드는 Penalty를 받아 일정 기간 네트워크에서 격리됨), 향후 다양한 서비스 창출을 위한 아이디어가 될 수는 있음.
- NONE 유형은 모든 Input에 대해서만 서명하는 것으로 백지수표 (Blank check) 같은 역할을 할 수 있음. Tx 생성자가 Output을 서명에서 제외했기 때문에 다른 사람이 Output의 주소를 변경할 수 있음 (output의 금액은 서명에 포함되어 변경할 수 없다고 함. Mastering Bitcoin 참조)
- NONE + ANYONECANPAY 유형은 “dust collector”에 활용될 수 있음 (Mastering Bitcoin 참조). dust UTXO란 수수료도 지급하지 못할 정도의 작은 금액을 의미함. 누군가 자신의 dust UTXO를 아무나 사용하라는 의미로 Input에만 서명한 Tx를 전파함. 또 다른 사람들도 자신의 dust UTXO를 Input에 추가하면서, Input의 금액이 커지면 아무나 output을 임의로 만들어서 Input에 모여 있는 dust UTXO 들을 사용할 수 있음.

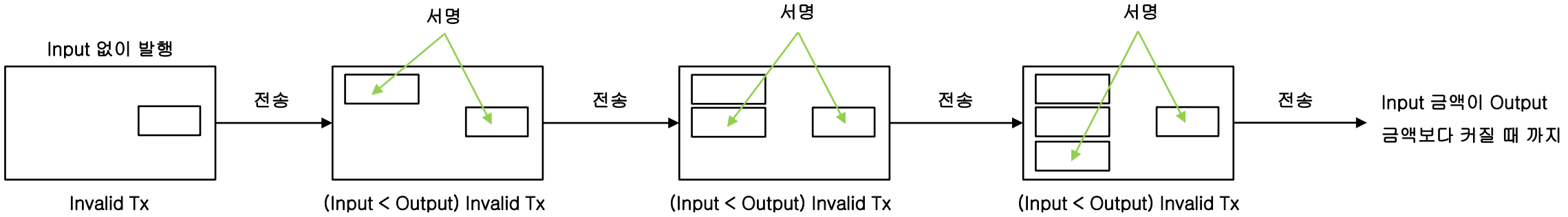
SigHashType	Value	Description
SIGHASH_ALL	0x01	모든 Input과 모든 Output에 대해 서명
SIGHASH_NONE	0x02	모든 Input에 대해 서명. 모든 Output은 제외
SIGHASH_SINGLE	0x03	모든 Input에 대해 서명. Input과 동일 index의 Output만 서명

SigHashType	Value	Description
ALL + ANYONECANPAY	0x81	해당 Input과 모든 Output에 대해 서명
NONE + ANYONECANPAY	0x82	해당 Input에 대해 서명. 모든 Output은 제외
SINGLE + ANYONECANPAY	0x83	해당 Input에 대해 서명. Input과 동일 index의 Output만 서명

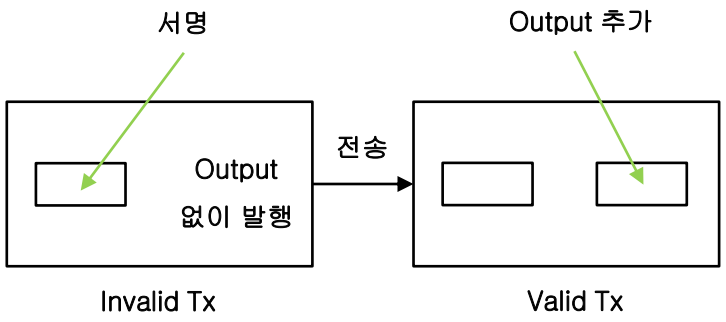
Digital Signature : SigHash Type을 응용한 예시

- Andreas Antonopoulos 가 Mastering Bitcoin에서 소개한 Crowdfunding, Blank check, Dust collector 시나리오.

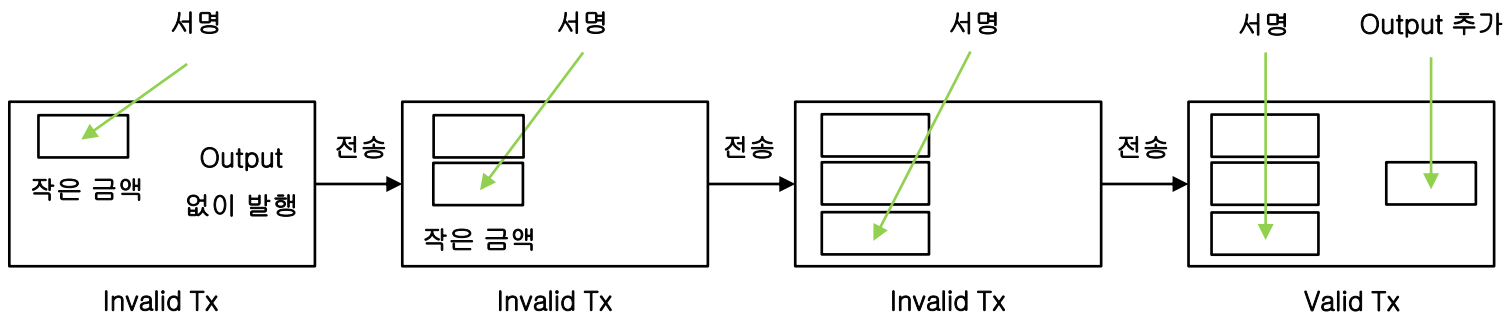
Crowdfunding



Blank Check

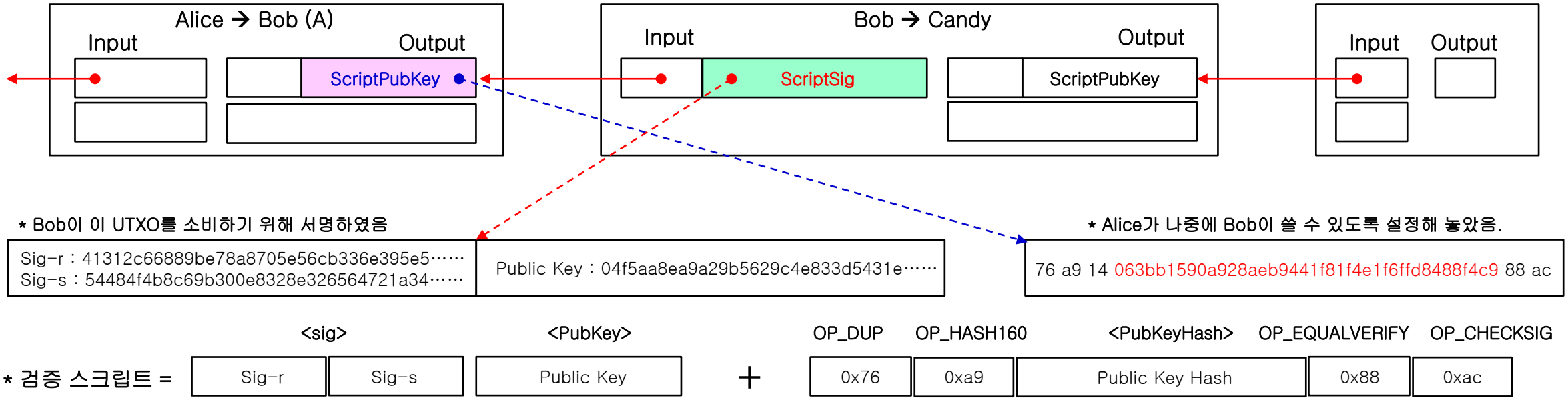


Dust Collector



Digital Signature 검증 (Verification) : ECDSA

- Bob이 Transaction (A)의 UTXO를 소비할 권한이 있는지 검증은 Script들을 통해 확인함. Script가 검증되면 모든 노드들이 인정함 (Consensus).



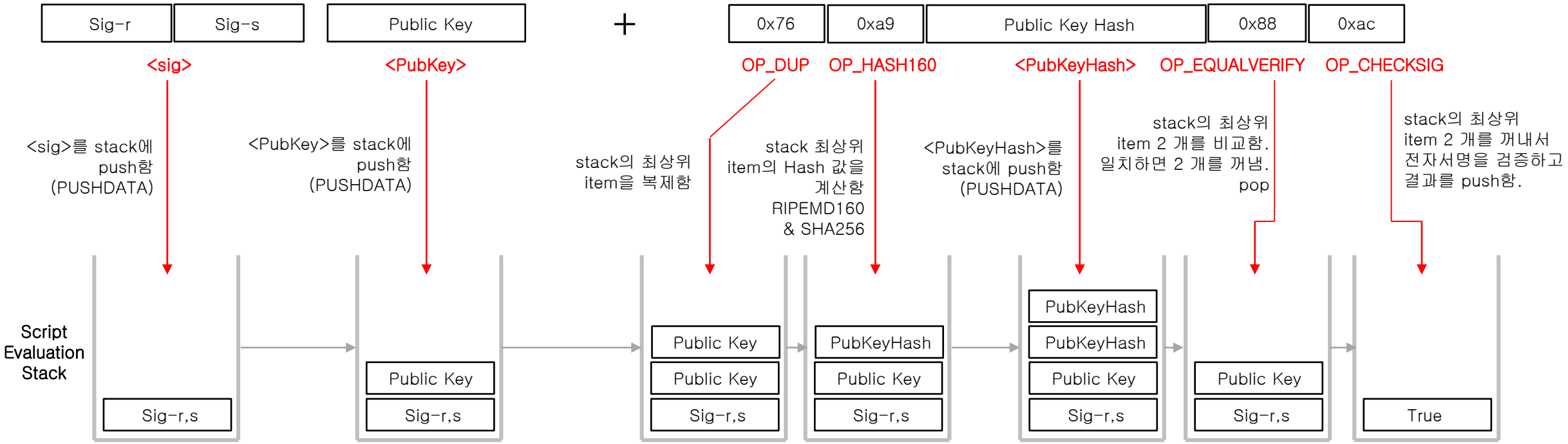
* 세부내역 참조 : https://en.bitcoin.it/wiki/Script

Word	Opcode	Hex	Input	Output	Description
N/A	1-75	0x01-0x4b	(special)	data	The next *opcode* bytes is data to be pushed onto the stack
OP_DUP	118	0x76	x	x x	Duplicates the top stack item.
OP_HASH160	169	0xa9	in	hash	The input is hashed twice: first with SHA-256 and then with RIPEMD-160.
OP_EQUALVERIFY	136	0x88	x1 x2	Nothing / *fail*	Same as OP_EQUAL, but runs OP_VERIFY afterward.
OP_CHECKSIG	172	0xac	sig pubkey	True / false	The entire transaction's outputs, inputs, and script (from the most recently-executed OP_CODESEPARATOR to the end) are hashed. The signature used by OP_CHECKSIG must be a valid signature for this hash and public key. If it is, 1 is returned, 0 otherwise.

Digital Signature 검증 절차 : Script Language (Script Evaluation Stack 이용)

- Script 검증은 아래와 같이 왼쪽에서 오른쪽으로 가면서 Item들을 Stack에 넣거나 (Push), 빼고 (Pop), 비교하는 절차로 진행됨.
- 아래 절차의 핵심은 Bob의 전자서명을 Bob의 공개키로 검증하는 것임. 먼저 Bob의 공개키가 맞는지 검증하고 (OP_EQUALVERIFY), 검증된 공개키로 전자서명을 검증하는 (OP_CHECKSIG) 것임.
- 아래 우측 스크립트는 Alice가 Bob을 위해 잠가 놓은 자물쇠에 해당하고 (Locking Script라고도 함), 좌측 스크립트는 Bob이 이 자물쇠를 풀 수 있는 열쇠에 해당함 (Unlocking Script라고도 함). 오직 Bob의 개인키로만 이 자물쇠를 풀 수 있음. → 개인키가 유출되면 안되는 이유임.

* 검증 스크립트



* OP-CODE 참조 : https://en.bitcoin.it/wiki/Script

Digital Signature 검증 절차 : Script Language (Stack 이용) – Raw Data

- 아래 예시는 Input (1개)와 Output (2개)로 구성된 Transaction의 Raw Data임. 아래 데이터에서 Script 부분은 Stack을 이용하여 검증함.
- Script를 1 byte씩 읽어가면서, 0x01 ~ 0x4b 범위에 있으면 길이를 의미하고 (비트코인에서 서명이나, Key 값들은 0x4b (75) byte를 넘는 게 없음), 그 이외의 범위는 OP_CODE를 의미함. 길이이면 이후 데이터를 그 길이 만큼 스택에 Push하고, OP_CODE이면 해당하는 Action을 취함.
- 0x4b (75 바이트) 보다 큰 데이터를 스택에 Push 하려면 OP_PUSHDATA1, OP_PUSHDATA2, OP_PUSHDATA4 를 이용함.

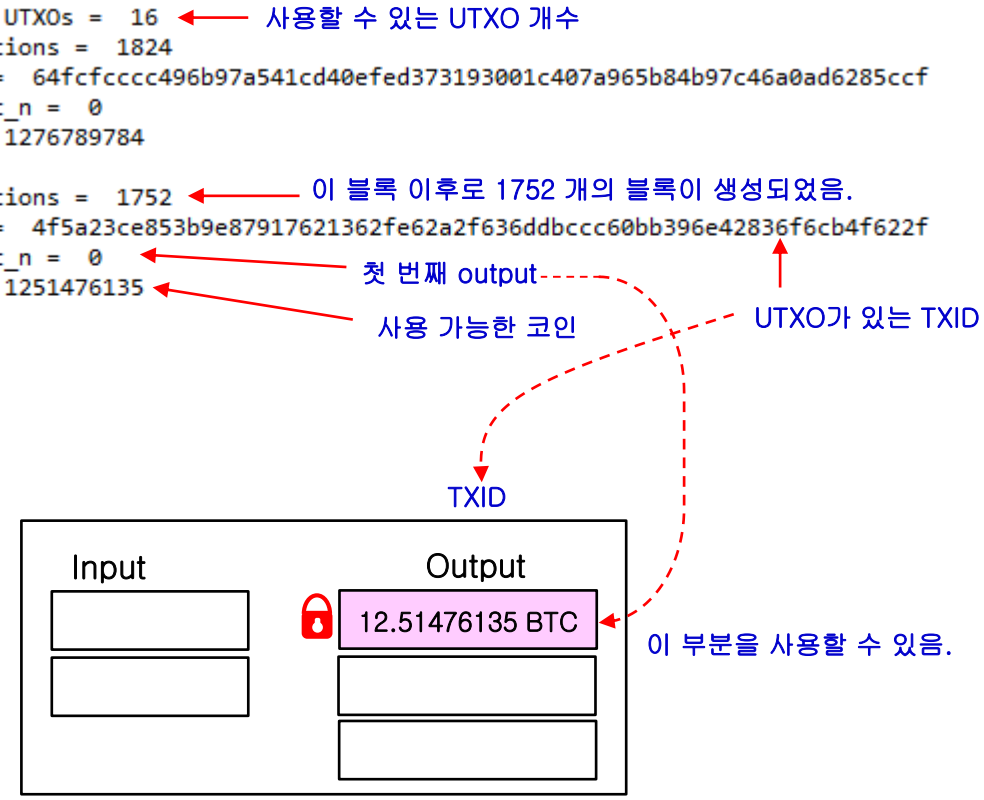
(실습 파일 : 4-1.UTXO조회.py)

UTXO 조회 : Python 연습

- 3rd-party API 서버 (ex : http://blockchain.info)로부터 특정 지갑 주소의 UTXO를 조회함. 거래를 생성할 때 아래 TXID의 첫 번째 Output을 사용할 수 있음.

```python
# 3rd-party API 서버로부터 특정 지갑 주소의 UTXO를 조회한다.
# 3d-party API 서버 (예시) : https://blockchain.info
#
# 2018.4.25 아마추어 퀀트 (조성현)
# ------------------------------------------------------
import requests

# 아래 주소의 UTXO를 조회한다
addr = '1AtQu3BesvK5L7cCE3v9QaY4jCTGVGAyWM'
url = 'https://blockchain.info/unspent?active=' + addr
resp = requests.get(url=url)
if resp.status_code != 200:
    print("\n", resp.text)
else:
    data = resp.json()
    utxo = data['unspent_outputs']

    # UTXO가 여러개인 경우 모두 출력한다
    totBalance = 0
    print("\nNumbr of UTXOs = ", len(utxo))
    for n in range(len(utxo)):
        print("Confirmations = ", utxo[n]['confirmations'])
        print("tx_hash = ", utxo[n]['tx_hash'])
        print("tx_output_n = ", utxo[n]['tx_output_n'])
        print("value = ", utxo[n]['value'])
        print()
        totBalance += utxo[n]['value']

    # UTXO의 value의 총 합이 이 지갑에서 사용할 수 있는 총 잔고임.
    print("\nTotal Balance = ", totBalance, '(Satoshi)')

```

```
Numbr of UTXOs =  16
Confirmations =  1824
tx_hash =  64fcfcccc496b97a541cd40efed373193001c407a965b84b97c46a0ad6285ccf
tx_output_n =  0
value =  1276789784

Confirmations =  1752
tx_hash =  4f5a23ce853b9e87917621362fe62a2f636ddbccc60bb396e42836f6cb4f622f
tx_output_n =  0
value =  1251476135

Total Balance =  2528265919 (Satoshi)
```

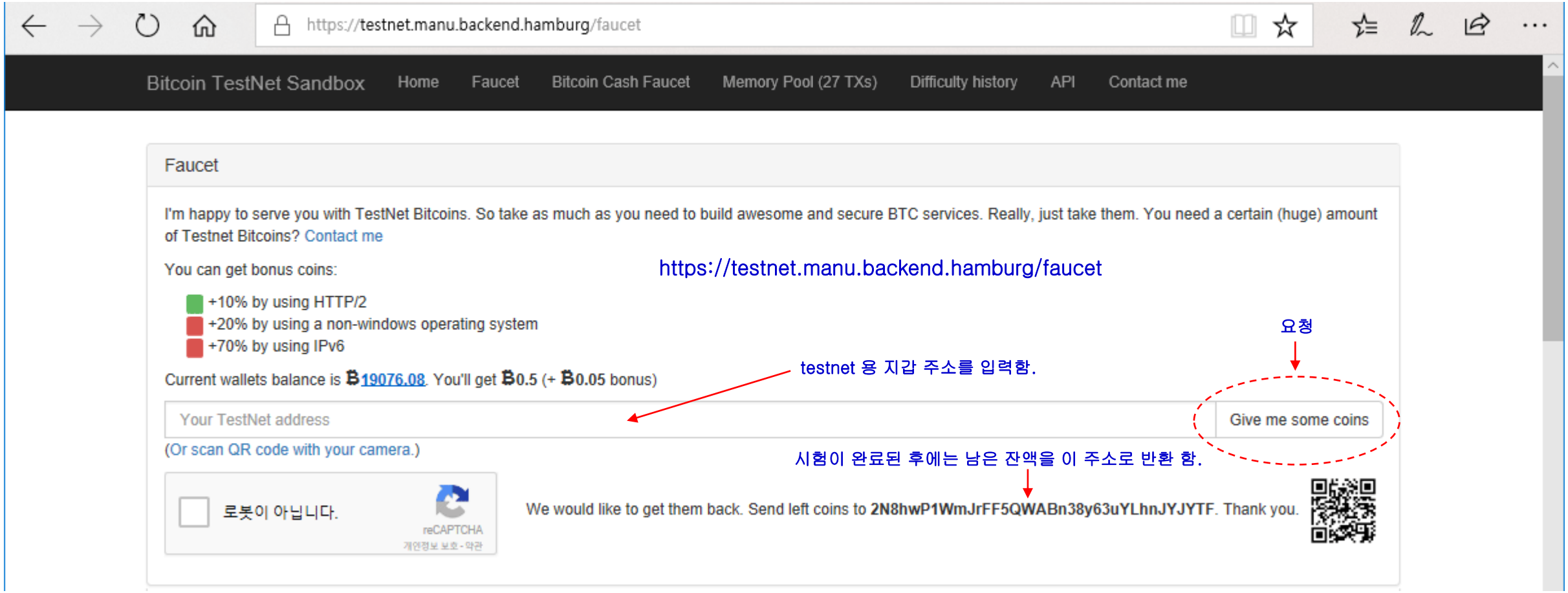


History log | IPython console

(실습 파일 : 4-2.Transaction(P2PKH).py)

Transaction 생성 : Python 연습 (Pay-to-Public-Key-Hash : P2PKH 방식)

- 많은 사이트에서 비트코인의 개발 시험용으로 testnet을 운용하고 있음 (ex : testnet.blockchain.info). mainnet과 동일하게 블록체인을 운영하고 있음.
- mainnet과 구별하기 위해, testnet에서는 주소의 앞 부분이 mainnet과 다름. m,n (P2PKH)으로 시작하거나, 2 (P2SH)로 시작함.
- Transaction 생성 시험은 testnet에서 수행하고, 초기 잔고는 아래 사이트에 요청하면 받을 수 있음.



(실습 파일 : 4-2.Transaction(P2PKH).py)

Transaction 생성 : Python 연습 (Pay-to-Public-Key-Hash : P2PKH 방식)

- 3rd-party API 서버 (ex : http://blockchain.info)를 이용하여 Transaction을 생성하고, 전송함. (A 지갑 → B 지갑으로 0.01 BTC를 송금함)

```
1  # 3rd-party API 서버를 이용하여 Transaction을 생성한다.
2  #
3  # Transaction 시험은 testnet에서 수행한다.
4  # 3d-party API 서버 (testnet) : https://blockchain.info
5  # usage : send(A, B, value, fee) ~ 주소 A 에서 주소 B로 value 금액을 송금한다.
6  #
7  # WARNING !!! 기능 시험용 코드이므로 mainnet에서 실제 비트코인 사용 금지.
8  #
9  # 2018.4.25 아마추어 퀀트 (조성현)
10 # ------------------------------------------------------------------------
11 from bitcoin.bci import history
12 from bitcoin.transaction import mktx, sign, deserialize
13 from urllib.request import urlopen
14 from urllib.parse import urlencode
15
16 url = "https://testnet.blockchain.info/"
17 A = 0   # Alice
18 B = 1   # Bob
19 address = ['mhJH61ScRnWJrhJm6283BbmACr27FjzT4Y', 'mg5urjMQZpALgga9GmwZaiiKKyB
20 privKey = ['7c06fcd8b6d7ef34182dd86882f5f1f1834381c132653b4f6937065167062b10'
21
22 # UTXO를 조회한다
23 def getUtxo(n=A):
24     if n == A or n == B:
25         h = history(address[n])
26         return list(filter(lambda txo: 'spend' not in txo, h))
27     else:
28         print("address error.")
29
30 # Transaction data packet을 생성한다
31 # input, output을 생성한다.
```

```
In [28]: utxo = getUtxo(A)

In [29]: utxo
Out[29]:
[{'address': 'mhJH61ScRnWJrhJm6283BbmACr27FjzT4Y',
  'block_height': 1294291,
  'output':
'44a7e169c111fdf489333faaedcc7669288790a0fd718719f850aa108037c5b5:0',
  'value': 2000000},
 {'address': 'mhJH61ScRnWJrhJm6283BbmACr27FjzT4Y',
  'block_height': 1293668,
  'output': '31fff7ba91361a6af724644e31eb8583b998c30bd58542409f88ba5f5580501e:
1',
  'value': 15082919}]

In [30]: tx, nInput = makeTx(utxo, A, B, 0.01, 0.0001)

In [31]: tx = signTx(utxo, tx, nInput, A)

In [32]: tx
Out[32]:
'0100000001b5c5378010aa50f8198771fda09087286976ccedaa3f3389f4fd11c169e1a74400000
0008a473044022025d425887b8a7cf56db3c96912956abe1e53e69ea0a29eb62aa5f947908b64490
220746b02b2a015d4610c515909d4014fab6ca5c8d23b5cfa0c7cdaf2eb534d9764014104f5aa8ea
9a29b5629c4e833d5431e2313cabd46f4dbcc69770fc513d7b35f980f7159a911f967dcf73cb15a8
c8bb7b382b66963d3a5244bdb2e7698afb319c882ffffffff0240420f00000000001976a914063bb
1590a928aeb9441f81f4e1f6ffd8488f4c988ac301b0f00000000001976a914138a87f639de8074c
46e6200f7bbe57ea1db0d4e88ac00000000'

In [33]:
```

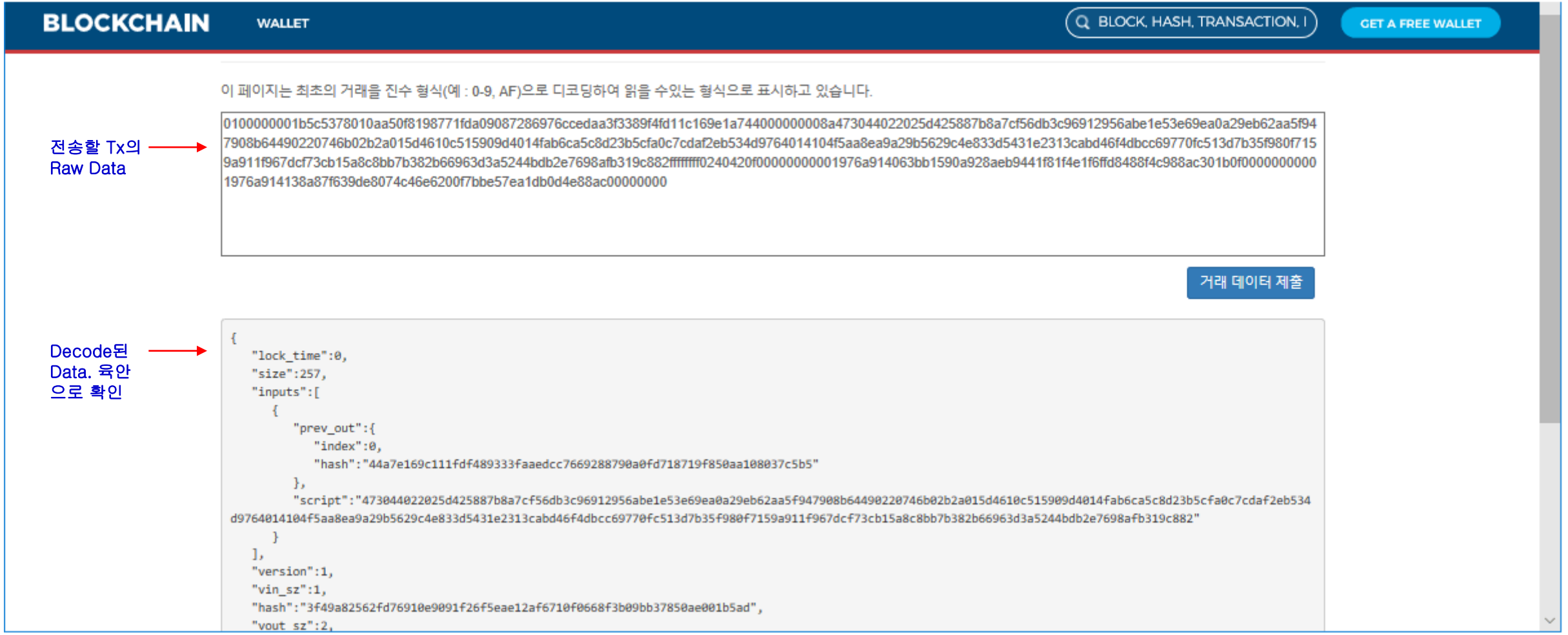
서명까지 완료된 최종 Tx임. 이 Tx를 비트코인 네트워크에 전송하면 됨.

History log | IPython console

(실습 파일 : 4-2.Transaction(P2PKH).py)

Transaction 생성 : Python 연습 (Pay-to-Public-Key-Hash : P2PKH 방식)

- 전자서명이 완료된 Transaction (Tx)을 네트워크에 전송하기 전에 decode하여 육안으로 확인함. (http://testnet.blockchain.info/decode-tx)



(실습 파일 : 4-2.Transaction(P2PKH).py)

Transaction 생성 : Python 연습 (Pay-to-Public-Key-Hash : P2PKH 방식)

- 전자서명이 완료된 Transaction (Tx)를 네트워크에 전송한 후 (3rd-party API 서버에게 전송해 달라고 요청), 정상적으로 전송되었음을 확인함.

```
# 3rd-party API 서버를 이용하여 Transaction을 생성한다.
#
# Transaction 시험은 testnet에서 수행한다.
# 3d-party API 서버 (testnet) : https://blockchain.info
# usage : send(A, B, value, fee) ~ 주소 A 에서 주소 B로 value 금액을 송금한다.
#
# WARNING !!! 기능 시험용 코드이므로 mainnet에서 실제 비트코인 사용 금지.
#
# 2018.4.25 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
from bitcoin.bci import history
from bitcoin.transaction import mktx, sign, deserialize
from urllib.request import urlopen
from urllib.parse import urlencode

url = "https://testnet.blockchain.info/"
A = 0   # Alice
B = 1   # Bob
address = ['mhJH61ScRnWJrhJm6283BbmACr27FjzT4Y', 'mg5urjMQZpALgga9GmwZaiiKKyB
privKey = ['7c06fcd8b6d7ef34182dd86882f5f1f1834381c132653b4f6937065167062b10'

# UTXO를 조회한다
def getUtxo(n=A):
    if n == A or n == B:
        h = history(address[n])
        return list(filter(lambda txo: 'spend' not in txo, h))
    else:
        print("address error.")

# Transaction data packet을 생성한다
# input, output을 생성한다.
def makeTx(utxo, n1=A, n2=B, value=0.01, fee=0.0001):
```

```
In [32]: tx
Out[32]:
'0100000001b5c5378010aa50f8198771fda09087286976ccedaa3f3389f4fd11c169e1a74400000
0008a473044022025d425887b8a7cf56db3c96912956abe1e53e69ea0a29eb62aa5f947908b64490
220746b02b2a015d4610c515909d4014fab6ca5c8d23b5cfa0c7cdaf2eb534d9764014104f5aa8ea
9a29b5629c4e833d5431e2313cabd46f4dbcc69770fc513d7b35f980f7159a911f967dcf73cb15a8
c8bb7b382b66963d3a5244bdb2e7698afb319c882ffffffff0240420f00000000001976a914063bb
1590a928aeb9441f81f4e1f6ffd8488f4c988ac301b0f00000000001976a914138a87f639de8074c
46e6200f7bbe57ea1db0d4e88ac00000000'

In [33]: sendTx(tx)
Transaction Submitted

In [36]: utxo = getUtxo(A)

In [37]: utxo
Out[37]:
[{'address': 'mhJH61ScRnWJrhJm6283BbmACr27FjzT4Y',
  'block_height': 1294307,
  'output': '3f49a82562fd76910e9091f26f5eae12af6710f0668f3b09bb37850ae001b5ad:
1',
  'value': 990000},
 {'address': 'mhJH61ScRnWJrhJm6283BbmACr27FjzT4Y',
  'block_height': 1293668,
  'output': '31fff7ba91361a6af724644e31eb8583b998c30bd58542409f88ba5f5580501e:
1',
  'value': 15082919}]
```

거래가 정상적으로 전송되었음.

- 잠시 기다리면 이 tx가 Mining 되어 블록체인에 기록될 것임.
- UTXO를 다시 확인해 보면 첫 번째 output이 변경 되었음.

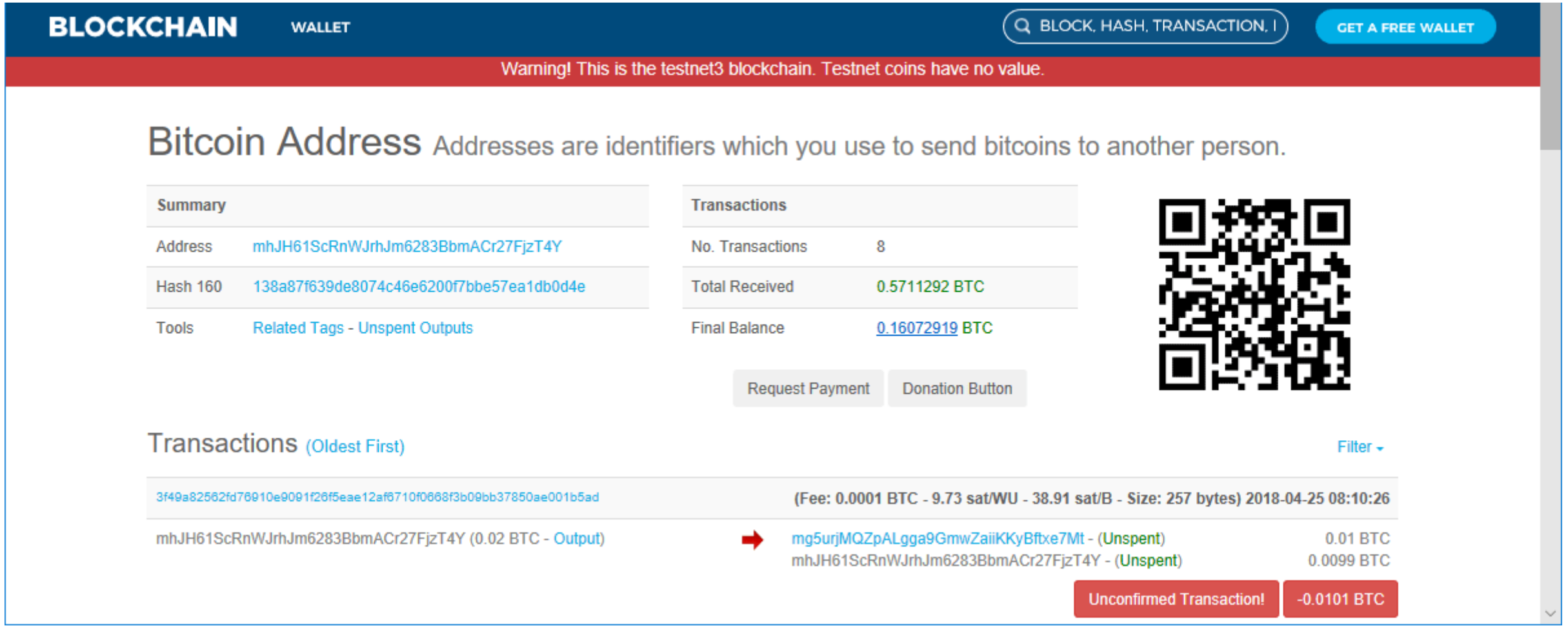
- 만약 이미 사용된 (Spent) 위의 tx를 다시 전송하면 (testnet.blockchain/pushtx 에서 재 전송), 오류가 발생함 (Transaction already exists). → 이중지불이 방지됨.

History log | IPython console

(실습 파일 : 4-2.Transaction(P2PKH).py)

Transaction 생성 : Python 연습 (Pay-to-Public-Key-Hash : P2PKH 방식)

- 네트워크에 전송된 Tx는 잠시 후 Mining되어 블록체인에 기록됨. 결과는 아래와 같이 확인할 수 있음. (아직은 Unconfirmed 상태임. → 무슨 의미인가 ?)



적정 거래 수수료

- 적정 거래 수수료는 최근 블록에 포함된 거래 내역을 참조하여 다른 사람들이 지불하는 평균 수수료 정도로 책정할 수 있음.
- 블록 521,829의 경우 2,011 거래가 지불한 총 수수료가 0.32517651 BTC 이므로 거래 당 0.0001617 BTC 임. Kilo-byte 당 평균 수수료 = 0.0002886 BTC 임.
- 2018년 들어서 거래 수수료는 급격히 낮아진 상태임. 비트코인 가격, Miner들의 Hash power 경쟁, 그리고 사용자의 수수료에 대한 인식 등이 주 원인 임.

Block #521829

Summary	
Number Of Transactions	2011
Output Total	9,587.11696863 BTC
Estimated Transaction Volume	607.43532291 BTC
Transaction Fees	0.32517651 BTC
Height	521829 (Main Chain)
Timestamp	2018-05-09 01:10:06
Received Time	2018-05-09 01:10:06
Relayed By	AntPool
Difficulty	4,022,059,196,164.95
Bits	390462291
Size	1126.78 kB

* 자료 출처 : https://blockchain.info

* 자료 출처 : https://coinmetrics.io/charts/

Transaction Malleability : 거래 데이터 조작 가능성

- 전자서명이 완료된 Transaction은 고유의 ID (txid)가 부여되고, 이 ID를 Key 값으로 블록체인에서 해당 Transaction을 검색하고, 거래 인증 등에 사용함.
- Txid는 서명이 완료된 Transaction의 해시 값 (SHA256)으로 부여함. Transaction의 일부분이 변경되면 Txid가 변경됨.
- Transaction의 내용이 변경되면 전자서명 값이 달라지므로 내용을 변경할 수 없음. 그러나 전자서명 값 자체가 변경되면 거래 내용은 동일하더라도 Txid가 달라짐.
- 전자서명을 유효하게 유지하면서 전자서명의 일부분을 변경하는 사례는 많이 있음. (BIP-62의 Several sources of malleability are known 부분 참조)
- 아래 사례의 경우 ScriptSig의 앞 부분이 0x4d4800 임. 이 부분을 decode 해 보면, 0x4d = OP_PUSHDATA2이고, 이 opcode는 다음 2 byte의 길이 만큼 stack에 push 하라는 의미임. 다음 2 바이트 (little endian)는 0x0048 이므로 다음 72 byte를 stack에 push 하게 됨.
- 이 부분의 원래 의미가 그냥 0x48 이라면 (0x4d4800 → 0x48) 다음 72 byte가 stack에 push 될 것임. 두 경우가 동일한 결과임.
- 전자 보다는 후자가 더 타당한 스크립트임. 따라서, 원래 0x48 부분을 누군가가 0x4d4800으로 변경하여 Txid가 변경되도록한 것이고, 변경된 거래가 Mining되어 정상 거래로 블록체인에 기록된 것임. 전송 금액이 크지 않은 것으로 보아 시험용이었을 것임.

* Malleability 사례

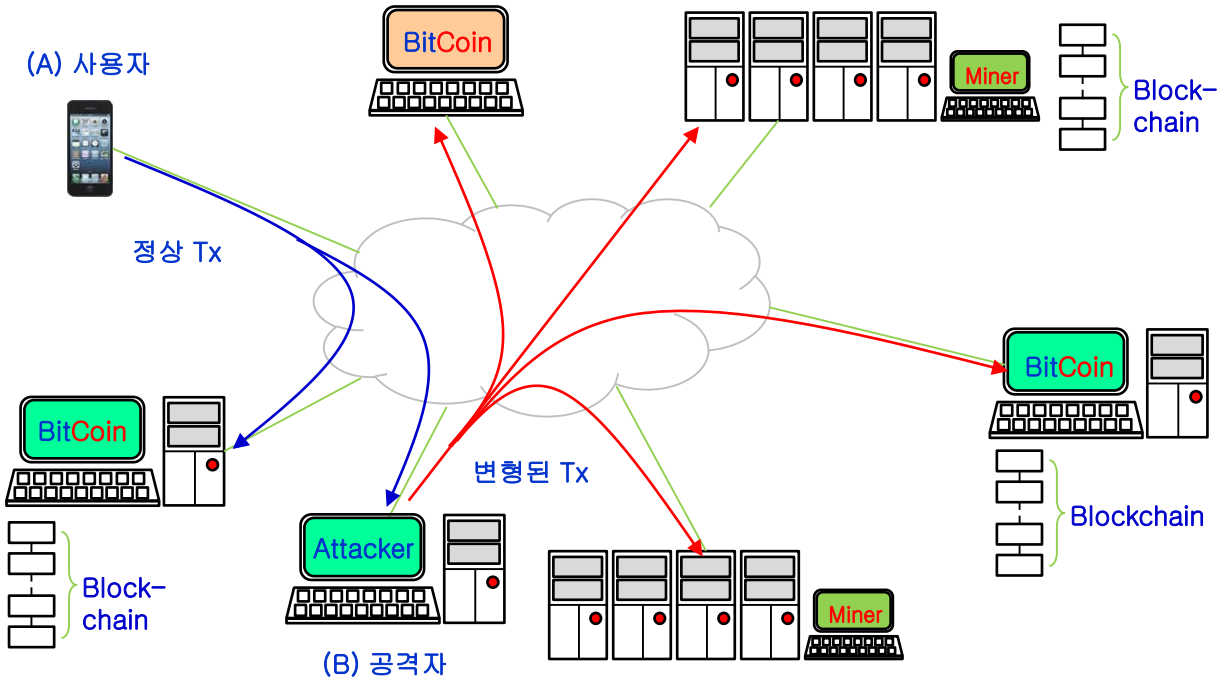
블록 285,201 (2014-02-11 03:42:45)의 txid = bba8c3d044828f099ae3bc5f3beaff2643e0202d6c121753b53536a49511c63f

0100000001d4884b2d7aaadc07c34a442098901ad772407dfaf0f0406c1a0964681dfbea3c010000008f4d480030450220539901ea7d6840eea8826c1f3d0d1fca7827e491deabcf17889e7a2e5a39f5a1022100fe745667e444978c51fdba6981505f0a68619f0289e5ff2352acbd31b3d23d87014d4100046c4ea0005563c20336d170e35ae2f168e890da34e63da7fff1cc8f2a54f60dc402b47574d6ce5c6c5d66db0845c7dabcb5d90d0d6ca9b703dc4d02f4501b6e44ffffffff01301b0f000000000001976a914b61c32ac39c63f919c4ce3a5df77590c5903d97588ac00000000

* 참조 : http://www.righto.com/2014/02/bitcoin-transaction-malleability.html

Transaction Malleability : 거래 데이터 조작 가능성

- Transaction Malleability 공격의 가능한 시나리오 예시.
- 정상 사용자 (A)가 (B)에게 송금하기로 하고 (인터넷 상품 구매 등) 정상 송금함. Transaction이 생성되어 네트워크로 전파됨.
- 공격자 (B)는 (A)가 보낸 Tx를 받아 ScriptSig의 일부분을 변경한 후 (금액, 지갑 주소 등, 다른 거래 조건은 변경이 불가함) 네트워크 상의 더 많은 노드들에게 전파함. 다른 노드들은 변형된 Tx를 받을 확률이 높을 수 있음.
- 다른 노드들이 변형된 Tx를 확인하면 정상적으로 검증되므로 정상적으로 다른 노드들로 Relay 할 것임.
- Miner도 변형된 Tx가 정상이므로 Mining할 것이고, Full 노드도 정상적으로 블록체인에 등록할 것임.
- 한편, 사용자 (A)의 지갑 S/W는 Tx를 송출한 후 Txid가 Mining될 것을 기대하고 기다리고 있는 상태임. 그러나 변형된 Txid가 이미 Mining 되었으므로, 정상 Txid는 폐기될 것임 (이중지불).
- 공격자 (B)는 (A)에게 입금 되지 않았다고 연락하면, 사용자 (A)는 이전 Tx 발송에 문제가 있었다고 생각하고, 다시 (B)에게 송금할 수 있음. 다른 UTXO를 이용하여 다시 송금할 수 있음.
- 공격자 (B)는 이중으로 받을 수 있음.
- 이러한 문제를 해결하기 위해 BIP-62으로 Tx 검증 규칙이 변경되었고, BIP-66으로 Script 형식 (DER)이 엄격해 졌음.
- 근본적 해결을 위해서는 Script를 Tx와 분리하는 Segregated Witness (SegWit) 방식이 개발 되었음.



(실습 파일 : 4-2.Transaction(P2PKH).py)

Transaction Malleability : 거래 데이터 조작 가능성 – Python 연습

- 정상적인 Tx를 생성한 후, 이전 사례와 같이 Tx를 조작함. Script length를 0x8b → 0x8d로 변경하고, Signature length 0x48 → 0x4d4800 으로 조작함.

```python
# 3rd-party API 서버를 이용하여 Transaction을 생성한다.
#
# Transaction 시험은 testnet에서 수행한다.
# 3d-party API 서버 (testnet) : https://blockchain.info
# usage : send(A, B, value, fee) ~ 주소 A 에서 주소 B로 value 금액을 송금한다.
#
# 참조 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
# WARNING !!! 기능 시험용 코드이므로 mainnet에서 실제 비트코인 사용 금지.
#
# 2018.4.25 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
from bitcoin.bci import history
from bitcoin.transaction import mktx, sign, deserialize
from urllib.request import urlopen
from urllib.parse import urlencode

url = "https://testnet.blockchain.info/"
A = 0   # Alice
B = 1   # Bob
address = ['mhJH61ScRnWJrhJm6283BbmACr27FjzT4Y', 'mg5urjMQZpALgga9GmwZaiiKKyB
privKey = ['7c06fcd8b6d7ef34182dd86882f5f1f1834381c132653b4f6937065167062b10'

# UTXO를 조회한다
def getUtxo(n=A):
    if n == A or n == B:
        h = history(address[n])
        return list(filter(lambda txo: 'spend' not in txo, h))
    else:
        print("address error.")

# Transaction data packet을 생성한다
```

```
In [50]: utxo = getUtxo(A)

In [51]: tx, n = makeTx(utxo)

In [52]: tx = signTx(utxo, tx)

In [53]: tx
Out[53]:
'0100000002adb501e00a8537bb093b8f66f01067af12ae5e6ff291900e9176fd6225a8493f01000
0008b48304502210095 70bd0e03a55cf2e0defe08ab34693594b97146e838d44b1cec0b973b29dac
402203fa1acde3626f06c42ef6eb01bfd52dea63882e5e15903bc145ba40372f85b34014104f5aa8
ea9a29b5629c4e833d5431e2313cabd46f4dbcc69770fc513d7b35f980f7159a911f967dcf73cb15
a8c8bb7b382b66963d3a5244bdb2e7698afb319c882ffffffff1e5080555fba889f404285d50bc39
8b98385eb314e6424f76a1a3691baf7ff31010000000ffffffff0240420f00000000001976a9140
63bb1590a928aeb9441f81f4e1f6ffd8488f4c988ac87d7e500000000001976a914138a87f639de8
074c46e6200f7bbe57ea1db0d4e88ac00000000'

In [54]: mal_tx = tx[0:82] + '8d' + '4d4800' + tx[86:]

In [55]: mal_tx
Out[55]:
'0100000002adb501e00a8537bb093b8f66f01067af12ae5e6ff291900e9176fd6225a8493f01000
0008d4d4800 30450221009570bd0e03a55cf2e0defe08ab34693594b97146e838d44b1cec0b973b2
9dac402203fa1acde3626f06c42ef6eb01bfd52dea63882e5e15903bc145ba40372f85b34014104f
5aa8ea9a29b5629c4e833d5431e2313cabd46f4dbcc69770fc513d7b35f980f7159a911f967dcf73
cb15a8c8bb7b382b66963d3a5244bdb2e7698afb319c882ffffffff1e5080555fba889f404285d50
bc398b98385eb314e6424f76a1a3691baf7ff31010000000ffffffff0240420f00000000001976a
914063bb1590a928aeb9441f81f4e1f6ffd8488f4c988ac87d7e500000000001976a914138a87f63
9de8074c46e6200f7bbe57ea1db0d4e88ac00000000'

In [56]:
```

정상적인 Tx를 생성함

공격용 Tx를 생성함

History log | IPython console

(실습 파일 : 4-2.Transaction(P2PKH).py)

Transaction Malleability : 거래 데이터 조작 가능성 – Python 연습

- 조작된 Tx (mal_tx)를 decode (testnet.blockchain.info/decode-tx)하면 정상적으로 decode 됨.
- 조작된 Tx를 testnet으로 전송하면 (testnet.blockchain.info/pushtx) 아래와 같이 에러가 발생하고 전송이 거부됨. → 이 유형의 공격은 보완되었음.

BLOCKCHAIN WALLET BLOCK, HASH, TRANSACTION, I GET A FREE WALLET

Warning! This is the testnet3 blockchain. Testnet coins have no value.

This page will decode a raw transaction in hex format (i.e. characters 0-9, a-f) and display it in human readable format

0100000002adb501e00a8537bb093b8f66f01067af12ae5e6ff291900e9176fd6225a8493f010000008d4d4800304502210095 70bd0e03a55cf2e0defe08ab34693594b97146e838d44b1cec0b973b29dac402203fa1acde3626f06c42ef6eb01bfd52dea63882e5e15903bc145ba40372f85b34014104f5aa8ea9a29b5629c4e833d5431e2313cabd46f4dbcc69770fc513d7b35f980f7159a911f967dcf73cb15a8c8bb7b382b66963d3a5244bdb2e7698afb319c882ffffffff1e5080555fba889f404285d50bc398b98385eb314e6424f76a1a3691baf7ff3101000000000ffffffff0240420f00000000001976a914063bb1590a928aeb9441f81f4e1f6ffd8488f4c988ac87d7e500000000001976a914138a87f639de8074c46e6200f7bbe57ea1db0d4e88ac00000000

Submit Transaction

- decode-tx 에서는 정상적으로 decode 됨
- pushtx 에서 이 tx를 전송하려고 하면 아래와 같이 에러가 발생함. 이 유형의 Transaction Malleability는 보완 되었음을 알 수 있음.

Validation Error: BitcoindException(super=com.neemre.btcdcli4j.core.BitcoindException: Error #-26: 64: non-mandatory-script-verify-flag (Data push larger than necessary), code=-26)


```
{
   "lock_time":0,
   "size":301,
   "inputs":[
      {
         "prev_out":{
            "index":1,
            "hash":"3f49a82562fd76910e9091f26f5eae12af6710f0668f3b09bb37850ae001b5ad"
         },
         "script":"4d4800304502210095 70bd0e03a55cf2e0defe08ab34693594b97146e838d44b1cec0b973b29dac402203fa1acde3626f06c42ef6eb01bfd52dea63882e5e15903bc145ba
40372f85b34014104f5aa8ea9a29b5629c4e833d5431e2313cabd46f4dbcc69770fc513d7b35f980f7159a911f967dcf73cb15a8c8bb7b382b66963d3a5244bdb2e7698afb319c882"
      },
```

다중 서명 : Multi Signature (Multisig)

- 지금까지는 지갑 주소 한 개에 개인키가 한 개 대응되고 전자서명 값도 한 개인 경우만 살펴 보았음 (지갑의 주인이 한 명인 개인용 지갑으로 생각할 수 있음).
- 지갑 주소 한 개를 여러 명이 공동으로 사용하는 경우에는 여러 명의 전자서명이 필요함. 여러 명의 서명이 모두 들어가야 거래가 가능한 경우도 있을 수 있고, 최소 몇 명 이상의 서명만 있으면 거래가 가능한 경우도 있을 수 있음.
- 이런 유형으로는, 동업자가 지갑을 공동으로 사용하는 경우, 사업체에서 부서 별로 관리하는 지갑으로 결제 서명이 필요한 경우 등 다양한 사례가 있을 수 있음.
- 다중 서명이 필요한 지갑은 여러 개의 개인키와, 여러 개의 공개키가 있고, 지갑 주소는 1 개가 사용됨.
- 다중 서명은 일반적으로 M-of-N 방식으로 사용됨. N은 전체 키의 개수이고, M은 서명이 필요한 개수임. (N의 최댓값은 15임, OP_1 ~ OP_15)
- Multi Signature의 Script는 아래와 같은 구조로 되어 있으며, Pay-to-Script-Hash (P2SH) 방식으로 구현할 수 있음.

Script 구조

- Locking Script (M-of-N signature) : OP_M <Public Key 1> <Public Key 2> …… <Public Key N> OP_N OP_CHECKMULTISIG (단, M, N <= 15)
- Unlocking Script : <Sig-1> <Sig-2> …… <Sig-M>
- Combined Script : <Sig-1> <Sig-2> …… <Sig-M> OP_M <Public Key 1> <Public Key 2> … <Public Key N> OP_N OP_CHECKMULTISIG

- Script를 이와 같이 구성하면 Stack을 이용한 Script Language를 이용하여 서명을 검증할 수 있음.
- 그러나 개발 과정에서 프로그램에 Bug가 있어, 위의 Combined Script를 Stack에서 꺼낼 때 한 개를 더 꺼내는 현상이 발생하였음. 이 오류를 방지하기 위해 (임시 방편으로) 아래와 같이 첫 부분에 OP_0 (empty array) 를 추가로 Stack에 Push해야 함. → 현재는 이것이 합의 규칙임

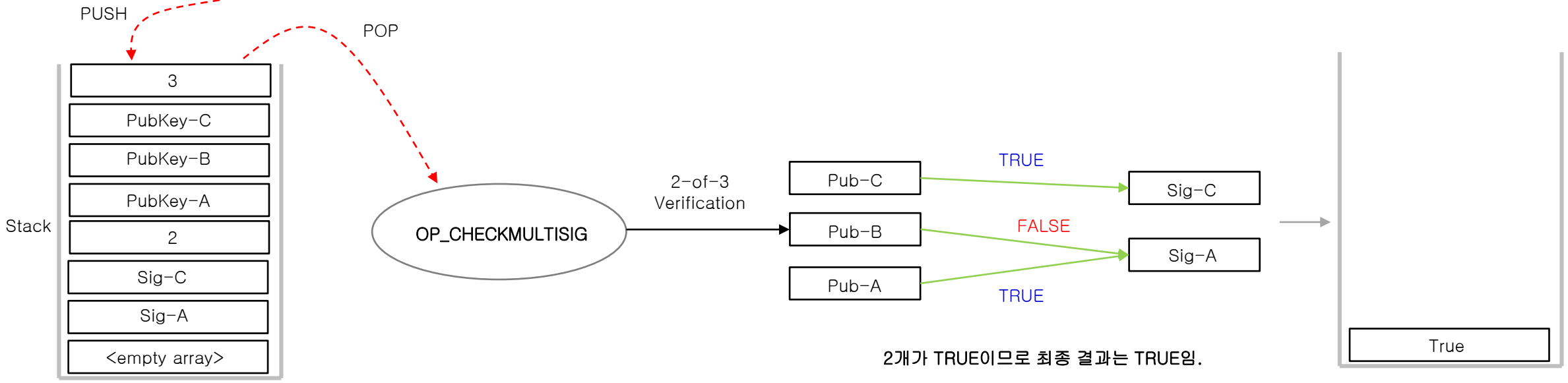
- Unlocking Script : <OP_0> <Sig-1> <Sig-2> …… <Sig-M>
- Combined Script : <OP_0> <Sig-1> <Sig-2> …… <Sig-M> OP_M <Public Key 1> <Public Key 2> … <Public Key N> OP_N OP_CHECKMULTISIG

다중 서명 : Multi Signature (Multisig) – 예시 : 2-of-3 Multisig

- Locking Script에 3개의 공개키가 제시되고, Unlocking Script에 2개의 Signature가 제시됨.
- 제시된 공개키 3개로 2개의 Signature를 검증할 때 최소 2개 이상이 TRUE이면 이 거래가 유효한 것으로 판단함. (3개 중 2개가 유효 : 2-of-3 Multisig)

2-of-3 Multisig Script 구성 및 검증 절차

- Locking Script (2-of-3 signature) : OP_2 <Public Key A> <Public Key B> <Public Key C> OP_3 OP_CHECKMULTISIG
- Unlocking Script : <Sig-A> <Sig-C>
- Combined Script : OP_0 <Sig-A> <Sig-C> OP_2 <Public Key A> <Public Key B> <Public Key C> OP_3 OP_CHECKMULTISIG



Pay-to-Script Hash : P2SH

- P2SH는 2012년 BIP-16 제안으로 개발되었으며, 비트코인 스크립트 언어를 활용하여 다양한 조건으로 거래를 생성할 수 있는 매우 강력한 기능으로 현재 널리 사용되고 있음.
- P2PKH (Pay-to-Public-Key-Hash) 방식은 수신자의 공개키 (해시)로 송금하는 방식임. Transaction output에 상대방의 공개키 해시를 기록함.
- P2PKH의 ScriptPubKey : OP_DUP OP_HASH160 <PubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
- P2SH 방식은 수신자의 Script Hash로 송금하는 방식임. Transaction output에 상대방의 Script Hash를 기록함.
- P2SH의 ScriptPubKey : OP_HASH160 <Redeem Script Hash> OP_EQUAL
- P2PKH는 개인키로 공개키를 만들고 공개키로 지갑 주소를 만들지만, P2SH는 스크립트 (Redeem Script)라는 걸 먼저 만들고 이 스크립트의 해시 값을 Base58 Encoding으로 지갑 주소를 만듦 (BIP-13). Redeem Script는 매우 다양한 조건 (ex : if ~ than ~ else)으로 거래를 생성할 수 있는 스크립트임.
- Multi Signature의 경우 Locking Script에 여러 개의 공개키를 하나의 redeemScript로 만들고, 이 스크립트를 사용하는 지갑 주소를 생성하면 P2SH을 이용하여 Multi Signature를 쉽게 구현할 수 있음. (P2SH은 Multi Signature 만을 위한 것이 아니라, 더 복잡한 형태의 거래를 수행할 수 있음)
- P2PKH 방식의 주소는 '1'로 시작하고 (ex : 1Loe7BnXch8G3iQPVC6fu8iEeHEF85urFX), P2SH 방식은 '3'으로 시작함 (ex : 3LchZzVvjxHn8aboeyxYmXE71vMDJbHJYp)

Multi Signature에 P2SH를 적용한 예시

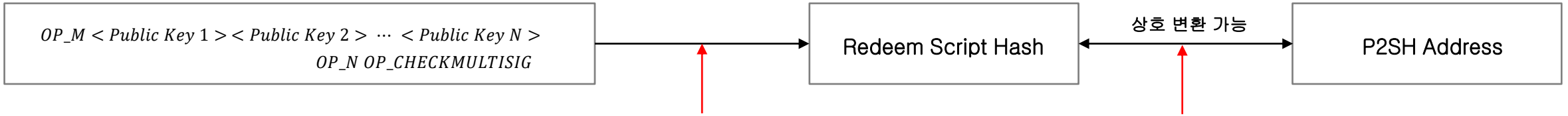
- Redeem Script (M-of-N signature) : OP_M <Public Key 1> <Public Key 2> …… <Public Key N> OP_N OP_CHECKMULTISIG (단, M, N <= 15)
- Locking Script : OP_HASH160 <Redeem Script hash> OP_EQUAL
- Unlocking Script : <Sig-1> <Sig-2> …… <Sig-M> <Redeem Script>

Pay-to-Script Hash : P2SH – 지갑 주소 생성 (BIP-13)

- P2SH용 Address는 아래와 같이 Redeem Script → Redeem Script Hash → Address 순으로 생성함. 생성된 Address는 '3'으로 시작함.
- Address에는 Redeem Script의 정보가 Hash로 요약되어 있음. 즉, Address 자체가 Script에 대한 정보를 담고 있음.
- Multi Signature용 Redeem Script를 만들기 위해서는 먼저 개인키와 공개키를 N 개씩 만들어야 함.
- P2SH는 Multi Signature 이외에도 다양하고 복잡한 스크립트에 활용할 수 있음. → 강력한 기능임.
- P2SH 주소로 송금할 때 송금자 (sender)는 scriptPubKey (Locking script)에 HASH160 <Redeem Script Hash> EQUAL 를 기록함. 송신자는 수신자의 Address로 Redeem Script Hash를 알 수 있음 (변환 가능)

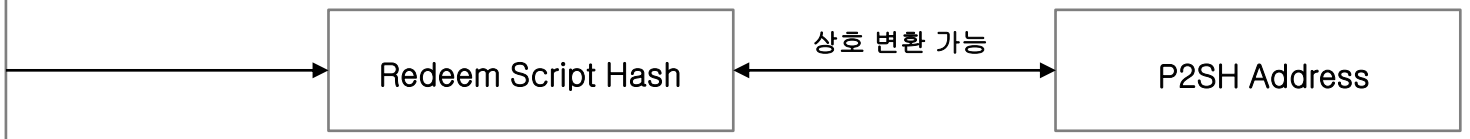
* Multi Signature용 Redeem Script 예시

(ex : 3LchZzVvjxHn8aboeyxYmXE71vMDJbHJYp)



* 복잡한 형태의 Redeem Script 예시

```
IF
  IF
    2
  ELSE
    < 30 days > CHECKSEQUENCEVERIFY DROP
    < Abdul the Lawyer's Pubkey > CHECKSIGVERIFY
    1
  ENDIF
  < Mohammed's Pubkey > < Saeed's Pubkey > < Zaira's Pubkey > 3 CHECKMULTISIG
ELSE
  < 90 days > CHECKSEQUENCEVERIFY DROP
  < Abdul the Lawyer's Pubkey > CHECKSIG
ENDIF
```

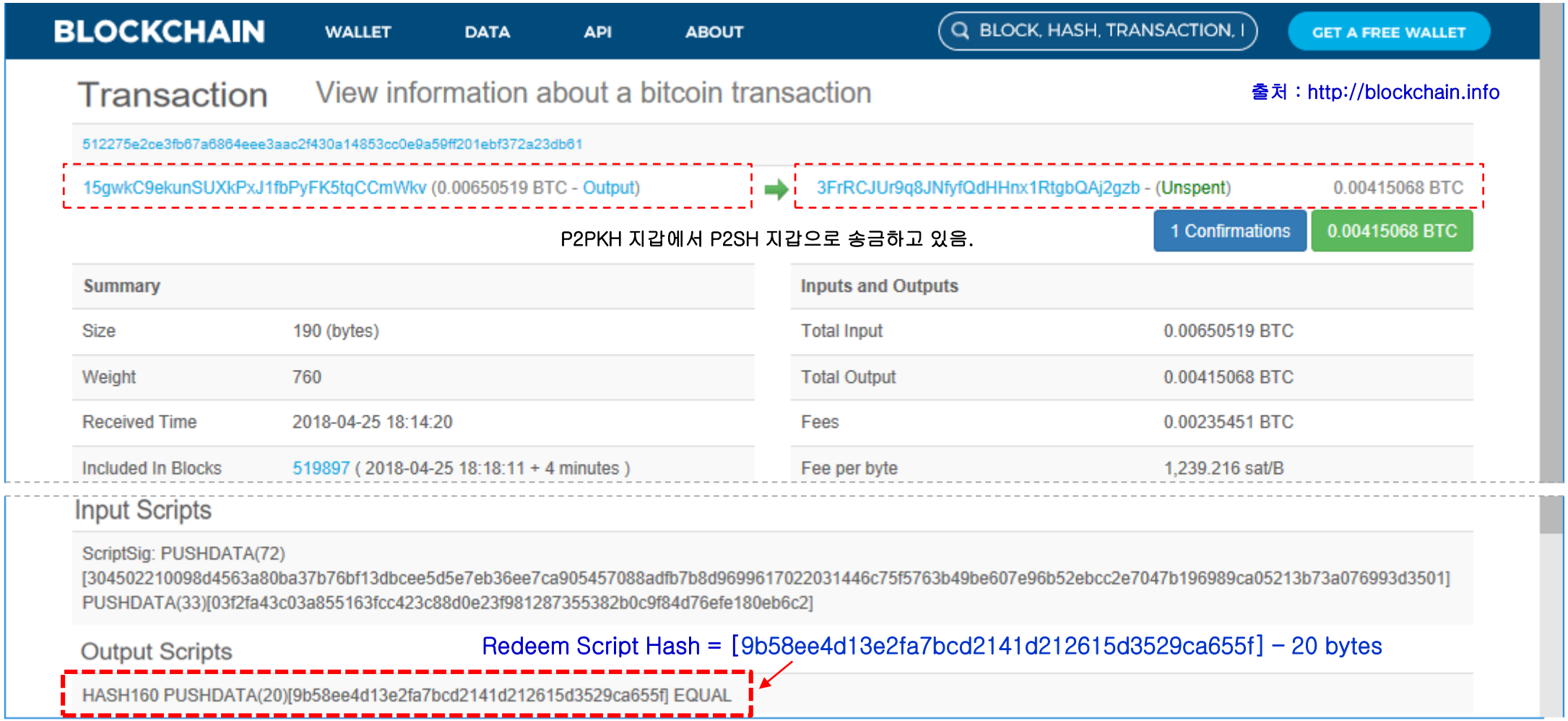


주소 자체가 Redeem Script 정보를 포함하고 있음.
(주소로 Script를 알아낼 수는 없음)

* Script 출처 : Mastering Bitcoin (2nd edition by Andreas Antonopoulos) P.168

P2PKH → P2SH 거래 예시

- 아래 예시는 P2PKH 지갑에서 P2SH 지갑으로 송금하는 거래임. P2PKH 지갑은 수신자의 Address를 20 byte (160 bit)의 Redeem Script Hash로 변환하여 Transaction output에 기록함 (송신자는 Redeem Script를 몰라도 됨). P2SH 지갑은 향후 자신의 Redeem Script를 사용하여 이 UTXO를 사용할 수 있음.

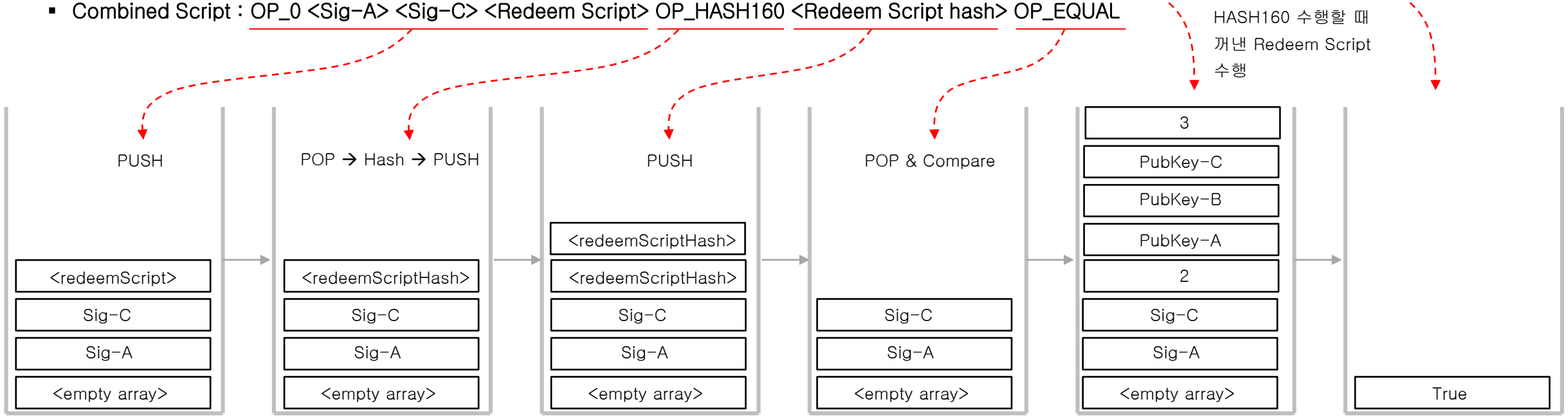


P2SH : 2-of-3 Multisig 적용 예시

- N개의 개인키와 공개키를 만들고 아래와 같이 2-of-3 multisig로 Redeem Script를 만든 후 이 Script로 지갑 주소를 생성함.
- 다른 사람이 이 지갑으로 송금할 때 ('3'으로 시작하면) output의 Locking Script를 아래와 같이 구성하고, 이 지갑이 다른 사람에게 송금할 때 input의 Unlocking Script는 아래와 같이 두 개의 Signature와 Redeem Script를 기록함.

2-of-3 Multisig Script 구성 및 검증 절차

- Redeem Script (2-of-3 signature) : OP_2 <Public Key A> <Public Key B> <Public Key C> OP_3 OP_CHECKMULTISIG
- Locking Script : OP_HASH160 <Redeem Script hash> OP_EQUAL
- Unlocking Script : OP_0 <Sig-A> <Sig-C> <Redeem Script>
- Combined Script : OP_0 <Sig-A> <Sig-C> <Redeem Script> OP_HASH160 <Redeem Script hash> OP_EQUAL



P2SH : 2-of-3 Multisig 적용 예시 – Python 연습

(실습 파일 : 4-4.TX(MultiSig_and_P2SH.py)

- 3 개의 개인키와 공개키를 만들고 Redeem Script를 생성한 후 Script Hash 값으로 testnet의 P2SH용 지갑 주소를 생성함.

```python
# P2SH을 이용하여 Multi Signature Transaction을 생성한다
#
# Multi Signature (2-of-3) 지갑에서 P2PKH 지갑으로 송금한다.
#
# Transaction 시험은 testnet에서 수행한다.
# 3d-party API 서버 (testnet) : https://testnet.blockchain.info
#
# 참조 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
# WARNING !!! 기능 시험용 코드이므로 mainnet에서 실제 비트코인 사용 금지.
#
# 2018.4.25 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------
import bitcoin.main as btc
from bitcoin.bci import history
from bitcoin.transaction import mk_multisig_script, scriptaddr, multisign, mk
from urllib.request import urlopen
from urllib.parse import urlencode
url = "https://testnet.blockchain.info/"

# n-개의 개인키를 만든다
def getPrivKey(n):
    seed = "MultiSig를 시험하기위해 만든 seed"
    key = []
    for i in range(n):
        privKey = btc.sha256(seed)
        key.append(privKey)
        seed = privKey
    return key

# 개인키 만큼 공개키를 만든다
def getPubKey(privKey):
```

```
In [15]: privKey
Out[15]:
['cc04e5dd26a217e8812f6fe1c7a9429c03b97fdf93df845c6009915806f288ff',
 'd6d07868186cf524bebbb3a0b72b83901225182c939885527a606a1a4b367f63',
 '565fccebf4e3646b830d31938fafb3a4f1cd372f11c815b782e34e1d70048091']

In [16]: pubKey
Out[16]:
['0462871af71bb0ef4d52ea7ac74ed3bf37442e9e68dcd45f2f3b8389fd7afaab2cf5c422b8eb0a
303ec84af5d232a06cd3acd67ec8d22eb4a920d7a43fb7aac01d',
 '045ba8018420d04758004fbae38124a2906d61e93e65aa9efa8eb09ad9e617aba16a7a6ded463c
40ddb9fc1091dcc2ac9a4fc8a26b52c75304d7a42839665e468f',
 '045317dfa41ba8d9879544475c2e555574b8d3936e53a559d98a94c16f60fca645b351e6d9c974
c613ce5fb951d66531003a886a24c6c23395e7b011d604b0a88b']

In [17]: script
Out[17]:
'52410462871af71bb0ef4d52ea7ac74ed3bf37442e9e68dcd45f2f3b8389fd7afaab2cf5c422b8e
b0a303ec84af5d232a06cd3acd67ec8d22eb4a920d7a43fb7aac01d41045ba8018420d04758004fb
ae38124a2906d61e93e65aa9efa8eb09ad9e617aba16a7a6ded463c40ddb9fc1091dcc2ac9a4fc8a
26b52c75304d7a42839665e468f41045317dfa41ba8d9879544475c2e555574b8d3936e53a559d98
a94c16f60fca645b351e6d9c974c613ce5fb951d66531003a886a24c6c23395e7b011d604b0a88b5
3ae'

In [18]: addr
Out[18]: '2NCAudjRxMQo8LNEML7aRPUDNEGZP2EJVRy'

In [19]:
```

Redeem Script

History log | IPython console

(실습 파일 : 4-4.TX(MultiSig_and_P2SH.py)

P2SH : 2-of-3 Multisig 적용 예시 – Python 연습

- P2SH 지갑에서 P2PKH 지갑으로 송금하는 Transaction을 생성함.

```
# P2SH을 이용하여 Multi Signature 1
#
# Multi Signature (2-of-3) 지갑에서
#
# Transaction 시험은 testnet에서 수
# 3d-party API 서버 (testnet) : ht
#
# 참조 : pybitcointools (https://p
# WARNING !!! 기능 시험용 코드이므로
#
# 2018.4.25 아마추어 퀀트 (조성현)
# --------------------------------
import bitcoin.main as btc
from bitcoin.bci import history
from bitcoin.transaction import mk
from urllib.request import urlopen
from urllib.parse import urlencode
url = "https://testnet.blockchain.

# n-개의 개인키를 만든다
def getPrivKey(n):
    seed = "MultiSig를 시험하기위해
    key = []
    for i in range(n):
        privKey = btc.sha256(seed)
        key.append(privKey)
        seed = privKey
    return key

# 개인키 만큼 공개키를 만든다
def getPubKey(privKey):
```

Multi Signature가 완성된 Transaction Data

```
In [19]: tx2
Out[19]:
'0100000001b5c5378010aa50f8198771fda09087286976ccedaa3f3389f4fd11c169e1a74401000000fd5d010048304502210 0e7a50c5b7c8a6a58269f
0ac983921baa53cd6b61ea9b1c5c92896533854c892b022070ae94576c14ac8ad62782c587ea54bc7fe10290533594d3ae350fa6cea9fbfe01473044022
0577f9d5150091ef8512c860e02c80869232948d3faad592743b48429bd4877c7022056b8a10c55844eabb82e43b46e563959f26e6d930d8d5955b8d975
d57d3d13b3014cc952410462871af71bb0ef4d52ea7ac74ed3bf37442e9e68dcd45f2f3b8389fd7afaab2cf5c422b8eb0a303ec84af5d232a06cd3acd67
ec8d22eb4a920d7a43fb7aac01d41045ba8018420d04758004fbae38124a2906d61e93a65aa9efa8eb09ad9e617aba16a7a6ded463c40ddb9fc1091dcc2
ac9a4fc8a26b52c75304d7a42839665e468f41045317dfa41ba8d9879544475c2e555574b8d3936e53a559d98a94c16f60fca645b351e6d9c974c613ce5
fb951d66531003a886a24c6c23395e7b011d604b0a88b53aeffffffff0280841e00000000001976a914138a87f639de8074c46e6200f7bbe57ea1db0d4e
88aca0e409030000000017a914cf996794dc65a5903e7bddb73f8c192cbc6732418700000000'

In [20]: deserialize(tx2)
Out[20]:
{'ins': [{'outpoint': {'hash': '44a7e169c111fdf489333faaedcc7669288790a0fd718719f850aa108037c5b5',
    'index': 1},
   'script':
'00483045022100e7a50c5b7c8a6a58269f0ac983921baa53cd6b61ea9b1c5c92896533854c892b022070ae94576c14ac8ad62782c587ea54bc7fe10290
533594d3ae350fa6cea9fbfe014730440220577f9d5150091ef8512c860e02c80869232948d3faad592743b48429bd4877c7022056b8a10c55844eabb82
e43b46e563959f26e6d930d8d5955b8d975d57d3d13b3014cc952410462871af71bb0ef4d52ea7ac74ed3bf37442e9e68dcd45f2f3b8389fd7afaab2cf5
c422b8eb0a303ec84af5d232a06cd3acd67ec8d22eb4a920d7a43fb7aac01d41045ba8018420d04758004fbae38124a2906d61e93a65aa9efa8eb09ad9e
617aba16a7a6ded463c40ddb9fc1091dcc2ac9a4fc8a26b52c75304d7a42839665e468f41045317dfa41ba8d9879544475c2e555574b8d3936e53a559d9
8a94c16f60fca645b351e6d9c974c613ce5fb951d66531003a886a24c6c23395e7b011d604b0a88b53ae',
   'sequence': 4294967295}],
 'locktime': 0,
 'outs': [{'script': '76a914138a87f639de8074c46e6200f7bbe57ea1db0d4e88ac',
   'value': 2000000},
  {'script': 'a914cf996794dc65a5903e7bddb73f8c192cbc67324187',
   'value': 50980000}],
 'version': 1}

In [21]:
```

Locking Script

History log | IPython console

(실습 파일 : 4-4.TX(MultiSig_and_P2SH.py)

P2SH : 2-of-3 Multisig 적용 예시 – Python 연습

- Signature가 완성된 Transaction Data를 분석함 (Decoding).

Description		Hex Data
	Version	01000000
	input count	01
	previous tx hash	b5c5378010aa50f8198771fda09087286976ccedaa3f3389f4fd11c169e1a744
	output index	01000000
	scriptSig length	fd5d01 --> 0xfd = 3 bytes, 0x015d = 349 bytes
scriptSig	OP_0	00
	Sig-A length	48 --> 72 bytes
	Sig-A	3045022100e7a50c5b7c8a6a58269f0ac983921baa53cd6b61ea9b1c5c92896533854c892b022070ae94576c14ac8ad62782c587ea54bc7fe10290533594d3ae350fa6cea9fbfe01
	Sig-C length	47 --> 71 bytes
	Sig-C	30440220577f9d5150091ef8512c860e02c80869232948d3faad592743b48429bd4877c7022056b8a10c55844eabb82e43b46e563959f26e6d930d8d5955b8d975d57d3d13b301
	OP_PUSHDATA1	4c
	Redeem Script length	c9 --> 201 bytes
	Redeem Script	52410462871af71bb0ef4d52ea7ac74ed3bf37442e9e68dcd45f2f3b8389fd7afaab2cf5c422b8eb0a303ec84af5d232a06cd3acd67ec8d22eb4a920d7a43fb7aac01d41045ba8018420d04758004fbae38124a2906d61e93a65aa9efa8eb09ad9e617aba16a7a6ded463c40ddb9fc1091dcc2ac9a4fc8a26b52c75304d7a42839665e468f41045317dfa41ba8d9879544475c2e555574b8d3936e53a559d98a94c16f60fca645b351e6d9c974c613ce5fb951d66531003a886a24c6c23395e7b011d604b0a88b53ae
	Sequence	ffffffff
	output count	02
output-1	value	80841e0000000000 --> 0.02 BTC
	scriptPubKey length	19 --> 25 bytes
	scriptPubKey	76a914138a87f639de8074c46e6200f7bbe57ea1db0d4e88ac
output-2	value	a0e4090300000000 --> 0.5098 BTC
	scriptPubKey length	17 --> 23 bytes
	scriptPubKey	a914cf996794dc65a5903e7bddb73f8c192cbc67324187
	locktime	00000000

(실습 파일 : 4-4.TX(MultiSig_and_P2SH.py)

P2SH : 2-of-3 Multisig 적용 예시 - Python 연습

- Signature가 완성된 Transaction Data의 Redeem Script와 scriptPubKey를 분석함 (Decoding).
- output-1은 P2PKH 지갑으로 송금하는 거래이고, output-2는, 자신에게 다시 보내는 거래이므로, P2SH 지갑으로 보내는 거래임.

Description		Hex Data
Redeem Script	OP_2	52
	PubKey-A length	41 --> 65 bytes (1 + 32 + 32)
	PubKey-A	0462871af71bb0ef4d52ea7ac74ed3bf37442e9e68dcd45f2f3b8389fd7afaab2cf5c422b8eb0a303ec84af5d232a06cd3acd67ec8d22eb4a920d7a43fb7aac01d
	PubKey-B length	41
	PubKey-B	045ba8018420d04758004fbae38124a2906d61e93a65aa9efa8eb09ad9e617aba16a7a6ded463c40ddb9fc1091dcc2ac9a4fc8a26b52c75304d7a42839665e468f
	PubKey-C length	41
	PubKey-C	045317dfa41ba8d9879544475c2e555574b8d3936e53a559d98a94c16f60fca645b351e6d9c974c613ce5fb951d66531003a886a24c6c23395e7b011d604b0a88b
	OP_3	53
	OP_CHECKMULTISIG	ae

Description		Hex Data
scriptPubKey (output-1)	OP_DUP	76
	OP_HASH160	a9
	pubKeyHash length	14 --> 20 bytes
	pubKeyHash	138a87f639de8074c46e6200f7bbe57ea1db0d4e
	OP_EQUALVERIFY	88
	OP_CHECKSIG	ac
scriptPubKey (output-2)	OP_HASH160	a9
	Redeem Script Hash length	14 --> 20 bytes
	Redeem Script Hash	cf996794dc65a5903e7bddb73f8c192cbc673241
	OP_EQUAL	87

P2SH : 2-of-3 Multisig 적용 예시 – Python 연습

(실습 파일 : 4-4.TX(MultiSig_and_P2SH.py)

- TX를 testnet.blockchain.info/pushtx로 전송함. 정상 전송되었고, 이 거래는 블록 1,294,486 (testnet)에 기록되었음. (잔고가 줄어 들었음)

```
1  # P2SH을 이용하여 Multi Signature Transaction을 생성한다
2  #
3  # Multi Signature (2-of-3) 지갑에서 P2PKH 지갑으로 송금한다.
4  #
5  # Transaction 시험은 testnet에서 수행한다.
6  # 3d-party API 서버 (testnet) : https://testnet.blockchain.info
7  #
8  # 참조 : pybitcointools (https://pypi.python.org/pypi/bitcoin)
9  # WARNING !!! 기능 시험용 코드이므로 mainnet에서 실제 비트코인 사용 금
10 #
11 # 2018.4.25 아마추어 퀀트 (조성현)
12 # ------------------------------------------------------------------
13 import bitcoin.main as btc
14 from bitcoin.bci import history
15 from bitcoin.transaction import mk_multisig_script, scriptaddr, mul
16 from urllib.request import urlopen
17 from urllib.parse import urlencode
18 url = "https://testnet.blockchain.info/"
19
20 # n-개의 개인키를 만든다
21 def getPrivKey(n):
22     seed = "MultiSig를 시험하기위해 만든 seed"
23     key = []
24     for i in range(n):
25         privKey = btc.sha256(seed)
26         key.append(privKey)
27         seed = privKey
28     return key
29
30 # 개인키 만큼 공개키를 만든다
31 def getPubKey(privKey):
```

```
In [50]: getUtxo(addr)
Out[50]:
[{'address': '2NCAudjRxMQo8LNEML7aRPUDNEGZP2EJVRy',
  'block_height': 1294291,
  'output': '44a7e169c111fdf489333faaedcc7669288790a0fd718719f850aa108037c5b5:1',
  'value': 52990000}]

In [51]: sendTx(tx2)
Transaction Submitted

In [52]: getUtxo(addr)
Out[52]:
[{'address': '2NCAudjRxMQo8LNEML7aRPUDNEGZP2EJVRy',
  'block_height': None,
  'output': '146f8ce40354d3685d3385bf9c46961d7182ee21c141600382be51908eb52bdf:1',
  'value': 50980000}]

In [53]: getUtxo(addr)
Out[53]:
[{'address': '2NCAudjRxMQo8LNEML7aRPUDNEGZP2EJVRy',
  'block_height': 1294486,
  'output': '146f8ce40354d3685d3385bf9c46961d7182ee21c141600382be51908eb52bdf:1',
  'value': 50980000}]

In [54]:
```

TX 전송 이전의 UTXO

TX 전송. 정상 전송 되었음.

TX 전송 이후의 UTXO (아직 Unconformation 상태임)

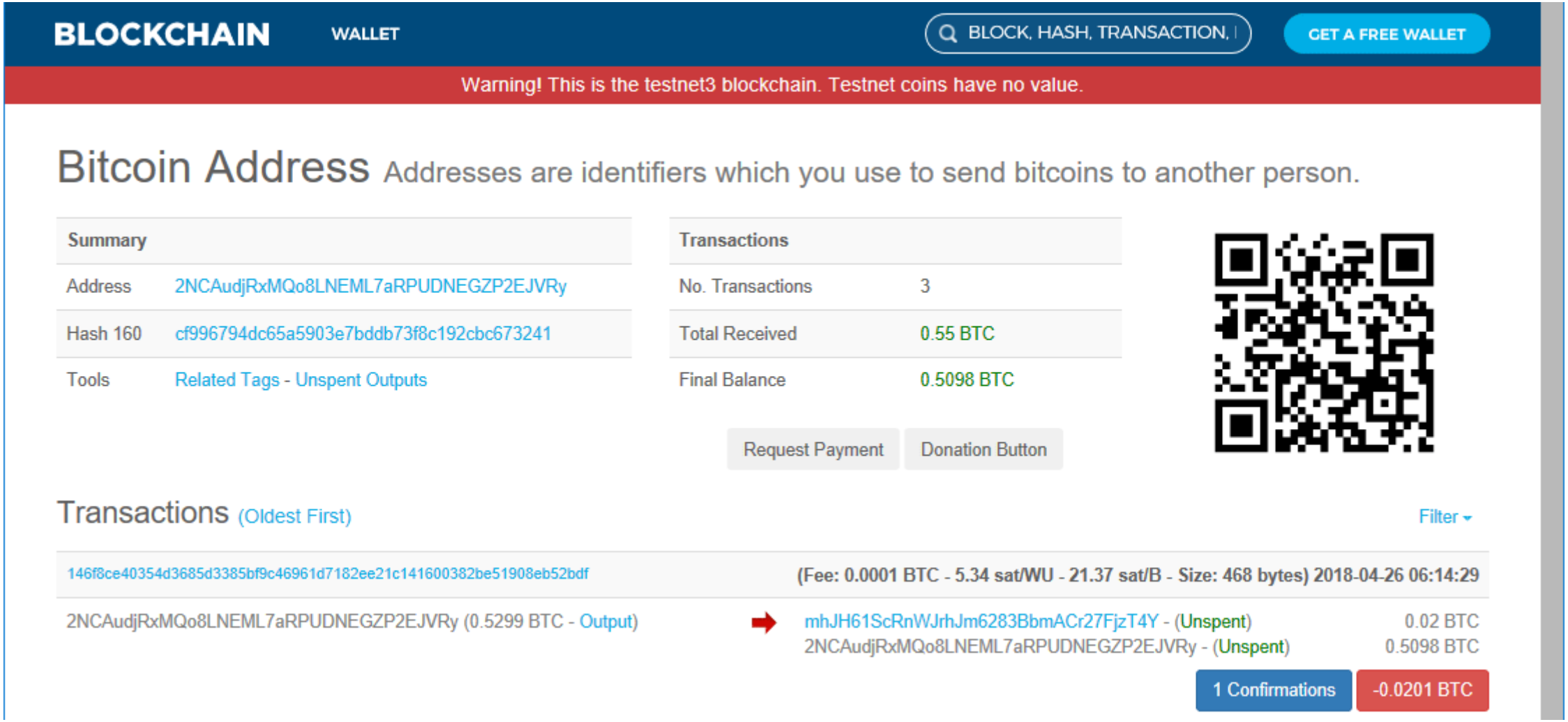
TX 전송 이후의 UTXO (Conformation 상태임)

History log | IPython console

(실습 파일 : 4-4.TX(MultiSig_and_P2SH.py)

P2SH : 2-of-3 Multisig 적용 예시 – Python 연습

- TX 승인 결과를 확인함. P2PKH 지갑으로 0.02 BTC가 송금되었고, 다시 자신의 지갑 (P2SH)으로 0.5089 BTC가 입금되었음. 수수료는 0.0001 BTC 임.

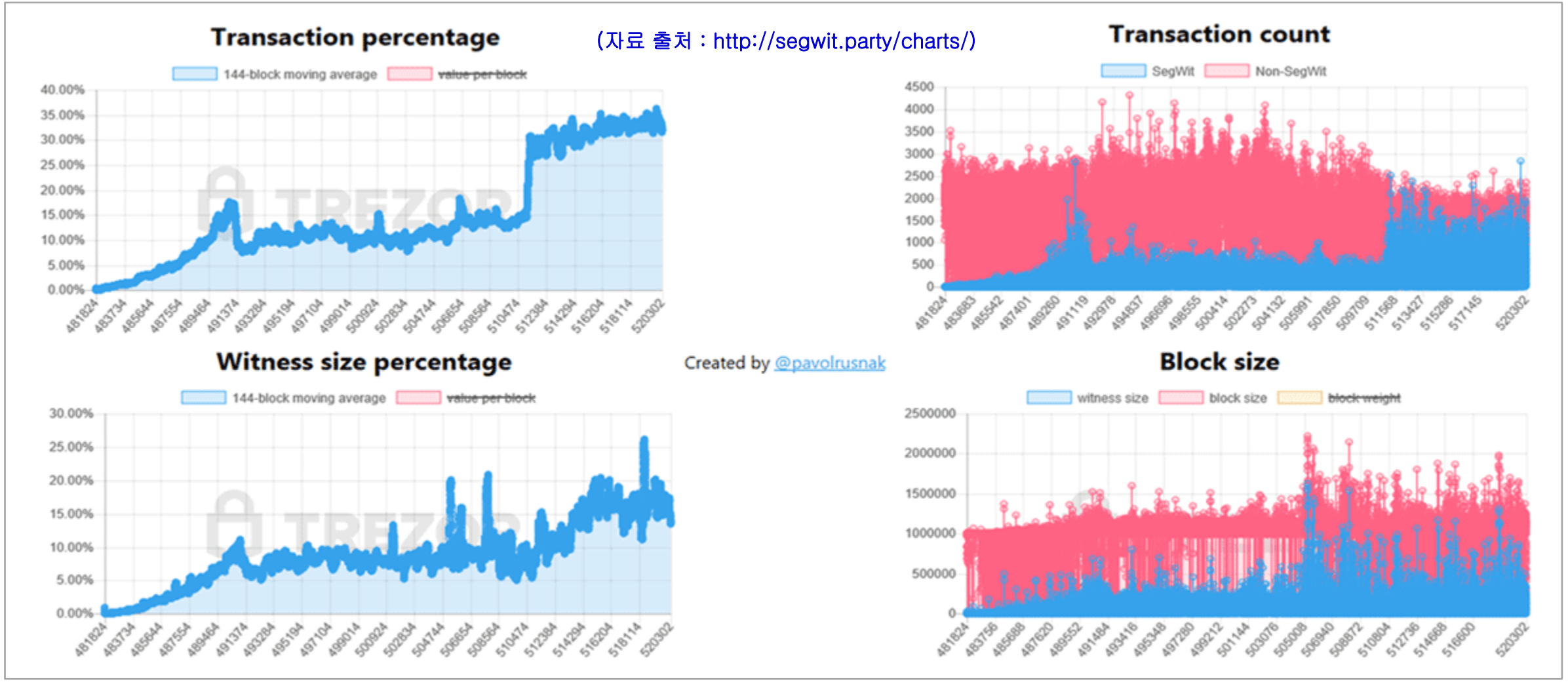


Segregated Witness : SegWit

- SegWit은 비트코인의 확장성 (Scalability) 문제를 완화시키고 Transaction Malleability 문제를 근본적으로 해결하기 위해 2015년 Pieter Wuille 박사에 의해 제안되어 (BIP-141, BIP-143, BIP-144), 2017년 비트코인 네트워크에 적용되었음.
- 확장성 문제를 해결하기 위해서는 현재 1MB의 제한을 가진 블록의 크기를 늘릴 필요가 있음 (노드 전체의 합의가 필요함).
- SegWit은 Transaction의 전자서명 (Signature) 부분을 분리하여 새로운 Transaction 구조로 변경하는 방식임. 기존의 [Input (Signature 포함) + Output] 구조를 [Input (Signature 제외) + Output + Signature (Witness라 함)] 구조로 변경하였음.
- 변경된 Transaction이 블록에 기록될 때는 Input과 Output의 Tx는 기존 처럼 기록되고, Witness 부분은 별도 공간에 저장됨. 기존 Tx에서 Witness 부분이 차지하는 비율은 약 75%로, Tx에서 Witness를 제외하면 Tx의 크기를 대폭 줄일 수 있음.
- 크기가 작아진 Tx를 이용하면 블록 안에 더 많은 Tx를 기록할 수 있음. 단, 이 Tx는 전자서명 부분이 없으므로 거래를 인증할 때는 Witness 부분을 사용해야 함.
- 한 블록 안에 더 많은 Tx를 기록하므로 확장성 문제를 완화할 수 있고, 전자서명 부분이 분리되므로 Transaction Malleability 문제를 해결할 수 있음.
- 블록 크기의 제한은 byte 단위가 아닌 weight 단위를 사용하고, 최대 4 M weight까지 확장됨. Byte 단위로 환산하면 평균 약 1.8 MB 정도로 확장 된다고 함. (참고 자료 : http://learnmeabitcoin.com/faq/segregated-witness & 블록 #493,182)
- Tx의 구조가 변경되므로 합의가 필요한 사항이나, 비트코인 코어 개발팀은 이전 구조를 갖는 노드들과 (S/W upgrade 이전, Non-SegWit 버전) 공존할 수 있도록 Soft Fork 방식으로 개발하였음 (Backward Compatibility). 각 노드들이 점진적으로 S/W를 upgrade (SegWit 버전) 하도록 함.
- 2017년 8월 SegWit이 Soft Fork 방식으로 적용되었을 때 일부 Major급 Miner들이 (Bitmain) 이 방식에 반대하였고 (97%는 찬성하였으나, 3% 정도가 반대했다고 함), 블록의 크기를 8MB로 확장하도록 Hard Fork 방식을 도입하여 비트코인 캐시 (Bitcoin Cash)를 탄생시켰음.
- 비트코인 네트워크에서는 SegWit이 적용되어 현재 운용 중에 있고, SegWit Transaction은 점차 증가하고 있음.
- SegWit은 Non-SegWit 버전들과의 호환성 (Soft Fork 방식) 문제를 해결하기 위해 다소 복잡해진 측면이 있지만 좋은 기능으로 평가받고 있음.

Segregated Witness : SegWit 거래 사용량

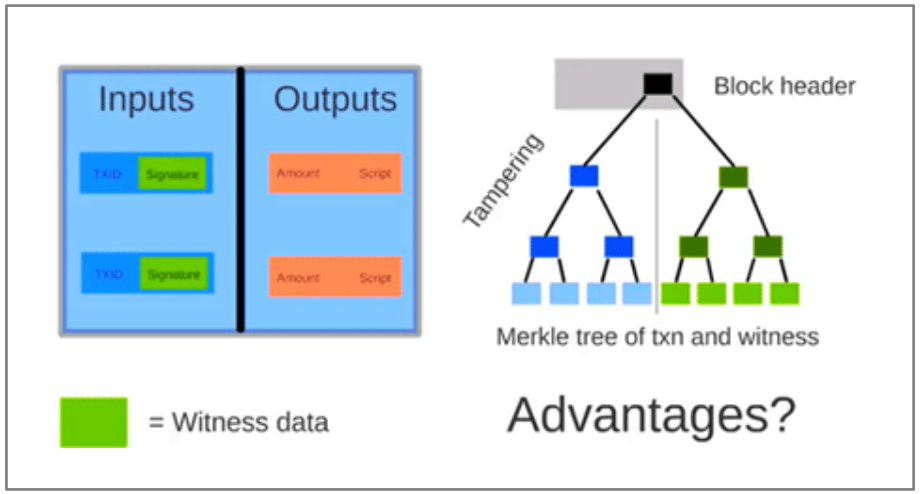
- 아래 결과는 블록 414,824 (2017.8.24) 부터 현재 (2018.4) 까지 한 블록 내에 포함된 SegWit 거래로, 점차 증가 추세에 있음을 보여 주고 있음



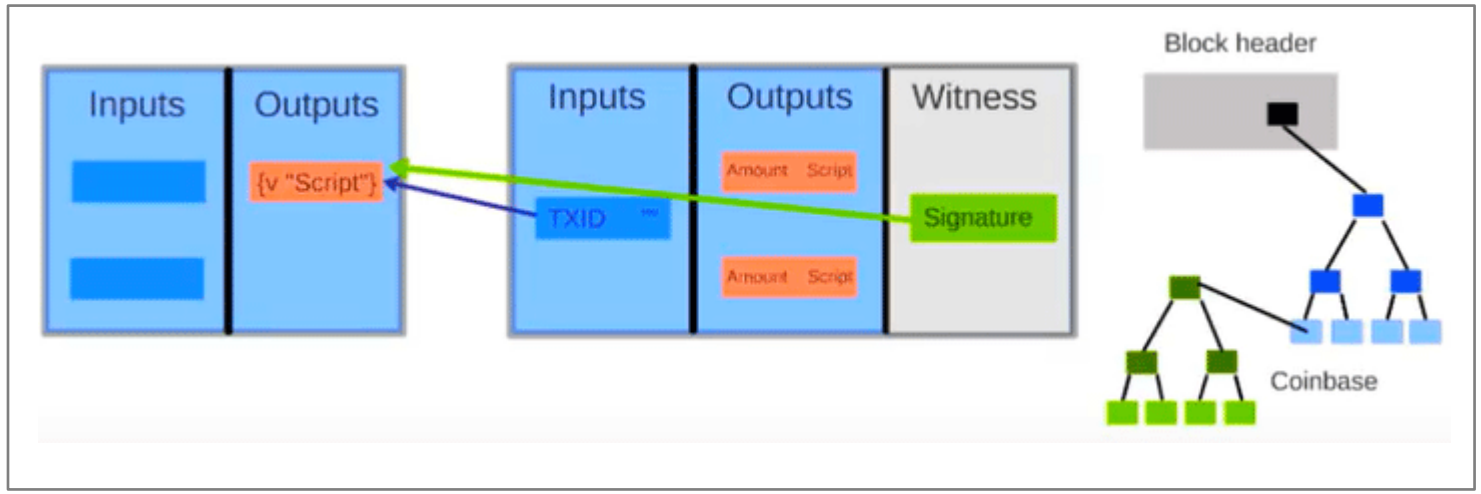
Segregated Witness : SegWit – Merkle Root

- Transaction (Tx)의 위, 변조는 Merkle Tree로 방지할 수 있음. Witness 부분을 기존 Tx에서 분리하여 별도로 기록하면 Tx는 기존 Merkle Tree로 보호할 수 있지만 분리된 Witness는 별도의 보호 장치가 필요함. 이를 위해 아래 두 경우를 생각해 볼 수 있음 (Case-A와 B)
- Case-A는 Merkle Tree를 두 부분으로 나누어 한 쪽은 Tx Hash를 기록하고 다른 한 쪽은 Witness Hash를 기록하는 방식임. 이 방식은 현재 블록 구조에서 Soft Fork 방식으로는 구현할 수 없다고 함 (Pieter Wuille)
- Case-B는 기존의 Tx용 Merkle Tree는 그대로 사용하면서, Witness용 Merkle Tree를 새로 만들어서 Coinbase Tx의 output에 붙이는 방식임.
- Case-B는 현재 블록 구조에서 Soft Fork 방식으로 구현 가능하다고 함 (다소 번거롭고 깔끔하지 못한 측면은 있지만 Soft Fork를 위한 방식임).
- SegWit은 Case-B 방식을 채택하였음. Witness 데이터 중 일부가 바뀌면 Witness용 Merkle Root가 바뀌고, Coinbase Tx의 Hash 값이 바뀌므로 최종 Merkle Root의 Hash 값이 바뀜. Witness의 일부분이 변경되면 기존 Merkle Root로 검출이 가능하므로, Witness 데이터도 보호할 수 있음.

Case-A



Case-B



- 자료 출처 : Pieter Wuille의 2015년 프리젠테이션 : Segregated witness and its impact on scalability (https://www.youtube.com/watch?v=NOYNZB5BCHM)

(자료 출처 : http://blockchain.info & BIP-141)

Segregated Witness : SegWit – Merkle Root

- Witness용 Merkle Root는 아래와 같이 Coinbase Tx의 두 번째 output에 기록되어 있음.
- Miner들은 SegWit용 블록을 생성할 때 Tx로부터 Witness를 분리하고, 별도의 Merkle Tree를 만들어서 Coinbase Tx에 연결시켜야 함.

69b532a34078d41e53f0da2244b5b1c9709964d2692909aae0627ae54c331d84

No Inputs (Newly Generated Coins) → 1C1mCxRukix1KfegAY5zQQJV7samAciZpv - (Unspent) 12.64733717 BTC

이 부분에 Witness Merkle Root가 기록되어 있음. → Unable to decode output address - (Unspent) 0 BTC

Summary	
Size	243 (bytes)
Weight	864
Received Time	2018-04-30 03:36:02

CoinBase

033ff10704238fe65a622f4254432e434f4d2ffabe6d6daaac2a18da5e
(decoded) ?◆#◆◆Zb/BTC.COM/◆◆mm◆◆*◆^j◆Bt◆◆_◆◆◆~

Output Scripts

DUP HASH160 PUSHDATA(20)[78ce48f88c94df3762da89dc8498205373a8ce6f] EQUALVERIFY CHECKSIG

RETURN PUSHDATA(36)[aa21a9ed268794c3c22604b2d5d2a2bc0db9a87f683348fa41c2f79d5d04335d1d0a8fab] ← Coinbase Tx 두 번째 output에 Witness Merkle Root가 기록되어 있음.

(decoded) ◆!◆◆&◆◆◆◆&◆◆H◆ ◆◆h3H◆A◆◆◆]3] ◆◆

Commitment structure

* BIP-141 내용

A new block rule is added which requires a commitment to the `wtxid`. The `wtxid` of coinbase transaction is assumed to be `0x0000....0000`.

A `witness root hash` is calculated with all those `wtxid` as leaves, in a way similar to the `hashMerkleRoot` in the block header.

The commitment is recorded in a `scriptPubKey` of the coinbase transaction. It must be at least 38 bytes, with the first 6-byte of `0x6a24aa21a9ed`, that is:

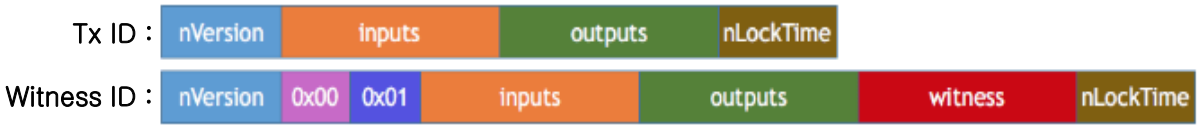
```
1-byte - OP_RETURN (0x6a)
1-byte - Push the following 36 bytes (0x24)
4-byte - Commitment header (0xaa21a9ed)
32-byte - Commitment hash: Double-SHA256(witness root hash|witness reserved value)

39th byte onwards: Optional data with no consensus meaning
```

Segregated Witness (SegWit) : Transaction 구조 (BIP-144)

- SegWit용 Transaction 구조는 아래와 같이 구성되어 있음. marker (0x00)와 flag (0x01)가 추가되었고, txin의 ScriptSig의 서명 부분이 Witness로 분리됨.
- Tx ID는 Witness를 제외한 부분의 Hash 값이고 (txid), Witness ID는 Witness 부분을 포함한 Hash 값임 (wtxid). (세부 내용은 BIP-144 참조)

- SegWit Transaction 구조

Field Size	Name	Type	Description
4	version	int32_t	Transaction data format version
1	marker	char	Must be zero
1	flag	char	Must be nonzero
1+	txin_count	var_int	Number of transaction inputs
41+	txins	txin[]	A list of one or more transaction inputs
1+	txout_count	var_int	Number of transaction outputs
9+	txouts	txouts[]	A list of one or more transaction outputs
1+	script witnesses	script witnesses[]	The witness structure as a serialized byte array
4	lock_time	uint32_t	The block number or timestamp until which the transaction is locked



- 자료 출처 : https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki

- SegWit Transaction 구조 예시
  (txid = c586389e5e4b3acb9d6c8be1c19ae8ab2795397633176f5a6442a261bbdefc3a)

02000000 : nVersion
00 : Marker
01 : Flag

---- txin ----
01 : input count
40d43a99926d43eb0e619bf0b3d83b4a31f60c176beecfb9d35bf45e54d0f742 : previous output
01000000 : output index
17 : Script length
16 00 14 a4b4ca48de0b3fffc15404a1acdc8dbaae226955
Script - PUSHDATA(0x16) [OP_0 <20 bytes>]
ffffffff : sequence

---- txout ----
01 : output count
00e1f50500000000 : value
17 : script length
a9 14 4a1154d50b03292b3024370901711946cb7cccc3 87
Script - OP_HASH160 [20 byte] OP_EQUAL

---- witness ----
02483045022100 8604ef8f6d8afa892dee0f31259b6ce02dd70c545cfcfed8148179971876c54a0
22076d771d6e91bed212783c9b06e0de600fab2d518fad6f15a2b191d7fbd262a3e0121039d25a
b79f41f75ceaf882411fd41fa670a4c672c23ffaf0e361a969cde0692e8

00000000 : nLockTime

Base32 주소 (Bech32 주소) : BIP-173

- Bech32는 2017년 3월 Pieter Wuille 박사가 BIP-173으로 제안한 새로운 주소 체계임. BIP-173은 Draft 상태에서 2018년 1월 Proposed 상태로 바뀌었음.
- 기존의 주소 체계는 Base58Check 인코딩 방식이었으며, 아래와 같은 문제점이 있음. 이 문제를 위해 Base32 인코딩 방식을 제안하였음.
  - Base58 needs a lot of space in QR codes, as it cannot use the alphanumeric mode.
  - The mixed case in base58 makes it inconvenient to reliably write down, type on mobile keyboards, or read out loud.
  - The double SHA256 checksum is slow and has no error-detection guarantees. ← Bech32는 오류 검출 능력이 좋음.
  - Most of the research on error-detecting codes only applies to character-set sizes that are a prime power, which 58 is not.
  - Base58 decoding is complicated and relatively slow.
- Bech32 주소는 90 character 까지 가능하고, human-readable part (hrp : 임의 문자열)와 separator, 그리고 data part로 구성됨.
- SegWit 주소 형식은, hrp = 'bc' (mainnet), 'tc' (testnet)를 사용하고, Witness 버전과, Witness program을 사용함. P2WPKH의 경우 Witness program은 160 비트의 공개키 해시 값임.

  Examples : All examples use public key 0279BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798. The P2WSH examples use key OP_CHECKSIG as script.

  - Mainnet P2WPKH: bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4
  - Testnet P2WPKH: tb1qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsx
  - Mainnet P2WSH: bc1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3qccfmv3
  - Testnet P2WSH: tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sl5k7
- 상세 내역은 BIP-173 문서 및 Pieter Wuille의 Presentation : https://www.youtube.com/watch?v=NqiN9VFE4CU 참조.

Base32 주소 (Bech32 주소) : Python 연습

(실습 파일 : 4-5.segwit_addr.py)

- Bech32 주소는 Base58Check 방식이 아닌 Base32Check 방식으로, Pieter Wuille 박사가 SegWit용 주소로 제안한 것임. (상세내용은 BIP-173 참조)

```python
# BIP-173 format의 bech32 주소를 생성한다.
# BIP-173 : Base32 address format for native v0-16 witness outputs
#
# 참고 : https://github.com/sipa/bech32/blob/master/ref/python/segwit_addr.py
#        pybitcointools (https://pypi.python.org/pypi/bitcoin)
#
# 2018.4.28 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
import binascii
import bitcoin.main as btc
import bitcoin.segwit_addr as bech32

# 개인키를 생성한다
while (1):
    privKey = btc.random_key()                      # 256 bit Random number를 생성
    dPrivKey = btc.decode_privkey(privKey, 'hex')   # 16진수 문자열을 10진수 숫자로
    if dPrivKey < btc.N:                            # secp256k1 의 N 보다 작으면 O
        break

# 개인키로 공개키를 생성한다. Compressed format.
pubKey = btc.privkey_to_pubkey(privKey)
cPubKey = btc.com

# 공개키로 160-bit
witprog = btc.bin

# BIP-173 주소를
mainnetAddr = bec
testnetAddr = bec

# 결과
```

Name	Size	Type	Date Modified
4-4.TX(MultiSig_and_P2SH).py	3 KB	py File	2018-04-26 오후 3...
4-5.segwit_addr.py	1 KB	py File	2018-04-28 오전 2...

IPython console

Console 1/A

```
공개키 : 02abdf6f1c75537fe5db1dbe196ca19543bcc2a54429e9446ff542c5fd9765ac5d
Bech32 주소 (Mainnet P2WPKH) : bc1qglf463483thz7dj0um4kznhjtkax7gey82q4cj
Bech32 주소 (Testnet P2WPKH) : tb1qglf463483thz7dj0um4kznhjtkax7geydvmxrp


BIP-173 문서의 Example 확인
==========================

공개키 : 0279BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
Bech32 주소 (Mainnet P2WPKH) : bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4
Bech32 주소 (Testnet P2WPKH) : tb1qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsx
```

BIP-173 문서 예제의 결과와 잘 일치함

Segwit Address Check

https://bc-2.jp/tools/bech32demo/index.html

Message: OK

SegwitAddr: bc1qn8lk8k0sfy4mjpctyk4w2j4kpzhfaffuufvw59

SegwitAddr: bc1qn8lk8k0sfy4mjpctyk4w2j4kpzhfaffuufvw59

Segwit Address Check

Message: Invalid

Error 검출이 용이함

SegwitAddr: bc1qn8lk8k0sfk4mjpctyk4w2j3kpzhfaffuufvw59

SegwitAddr: bc1qn8lk8k0sfk4mjpctyk4w2j3kpzhfaffuufvw59

Segregated Witness (SegWit) : Backward Compatibility (Soft Fork : BIP-141)

- SegWit 노드와 Non-SegWit 노드가 (일정 기간) 공존하려면 Non-SegWit 노드가 SegWit Tx를 받아도 유효한 거래로 판단할 수 있어야 함.
- Non-SegWit 노드가 SegWit Tx를 이해할 수는 없어도 새로운 Scheme을 무시하거나, 어떤 형태로든 유효한 거래로 판단하여 Relay할 수 있어야 함.
- Full 노드가 Non-SegWit 노드들로 Tx를 보낼 때는 (getdata의 MSG_TX 요청시) Witness 부분을 빼고 Input과 Output 부분만 보냄 (서명 부분이 없음).
- Non-SegWit Wallet이나 SPV 노드들은 Witness 부분이 빠진 Tx를 받으므로 거래를 완전히 인증할 수는 없지만 "Anyone can spend" 거래로 처리함.
- 예를 들어 오른쪽 예시에서 Witness 부분이 없으면 아래의 스크립트로 검증함.

$$0 \ < 20 - byte - keyhash > \ OP - HASH160$$
$$< 20 - byte - scripthash > \ OP - EQUAL$$

- 위의 검증 결과는 항상 TRUE 이므로 무조건 Valid한 거래로 취급됨 (Anyone can spend).
- Soft Fork를 위해서는 지갑 주소 유형별로 모든 경우에 대해 Anyone can spend가 성립하도록 scriptSig 부분을 생성해야 함. (상세내용은 BIP-141 참조)
- Anyone can spend 거래는 Non-SeqWit 노드들에 참조될 뿐 Mining 되지는 못함. 이 거래는 Valid하지만 Non-standard이므로 Miner들이 블록에 포함시키지 않음.

P2WPKH nested in BIP16 P2SH

The following example is the same P2WPKH, but nested in a BIP16 P2SH output.

```
witness:      <signature> <pubkey>
scriptSig:    <0 <20-byte-key-hash>>
              (0x160014{20-byte-key-hash})
scriptPubKey: HASH160 <20-byte-script-hash> EQUAL
              (0xA914{20-byte-script-hash}87)
```

The only item in scriptSig is hashed with HASH160, compared against the 20-byte-script-hash in scriptPubKey, and interpreted as:

```
0 <20-byte-key-hash>
```

The public key and signature are then verified as described in the previous example.

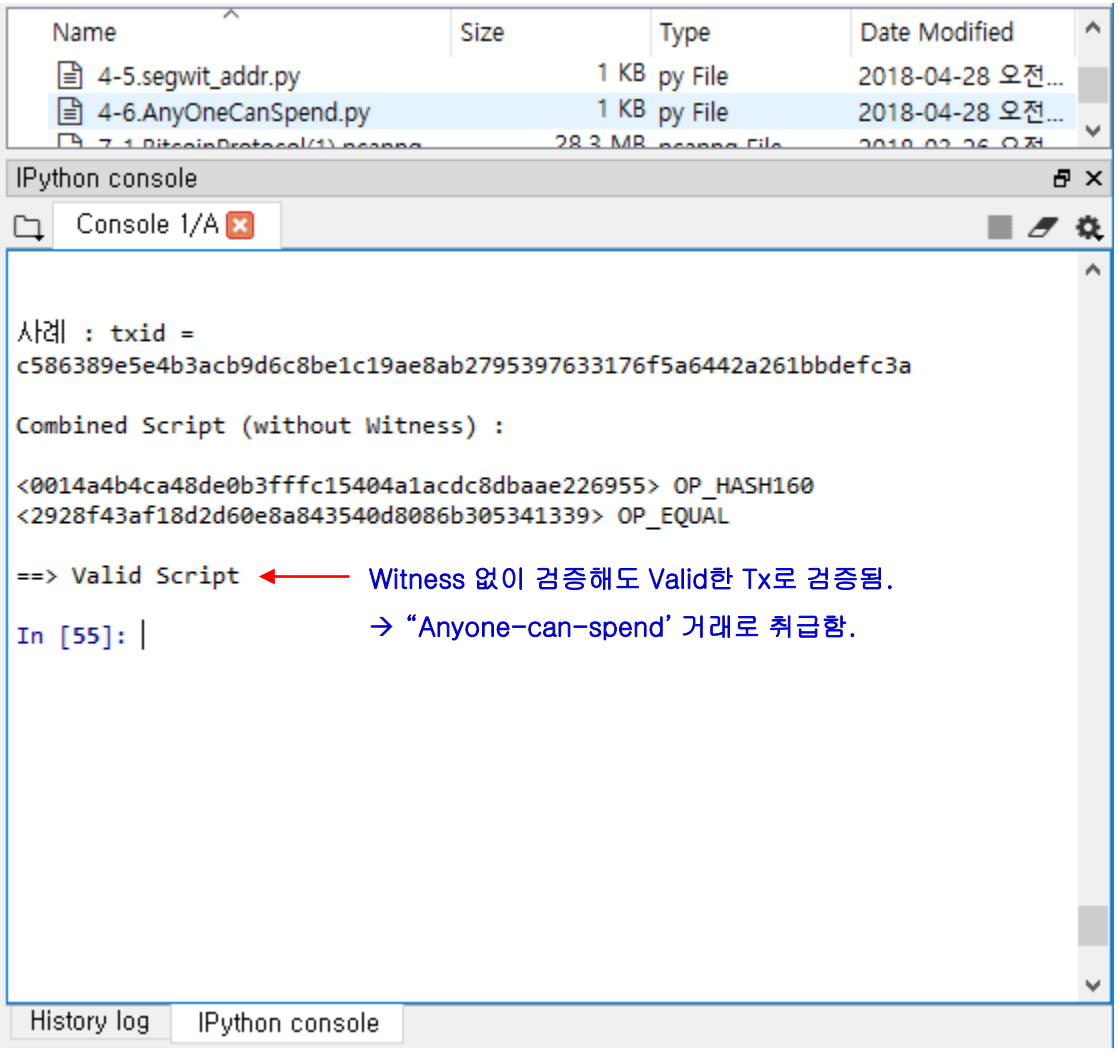
Comparing with the previous example, the scriptPubKey is 1 byte bigger and the scriptSig is 23 bytes bigger. Although a nested witness program is less efficient, its payment address is fully transparent and backward compatible for all Bitcoin reference client since version 0.6.0.

(실습 파일 : 4-6.AnyoneCanSpend.py)

Segregated Witness (SegWit) : Anyone-can-spend Python 연습

- 아래 예시에서 해당 Tx의 input에 있는 ScriptSig와 UTXO의 output에 있는 ScriptPubKey 스크립트를 검증함 (Witness 부분은 빼고 검증함).

```
# Non-SegWit 노드에서 Witness 데이터가 없어도 "Anyone can spend"로
# 처리되어, 이 TX가 valid하게 인식됨을 확인한다.
#
# Non-SegWit 노드가 Full 노드에게 getdata [MSG_TX]를 요구하면, Full 노드는 해당
# TX가 SegWit TX이면 Witness 부분을 빼고 Non-SegWit 노드에게 전송한다.
# Non-SegWit 노드는 Witness 부분이 없어도, 서명 부분이 없어도, "Anyone can spend"로
# 확인되어 Valid한 TX로 처리한다. (Soft Fork, for Backward compatibility)
#
# txid = c586389e5e4b3acb9d6c8be1c19ae8ab2795397633176f5a6442a261bbdefc3a
# 사례를 분석한다.
#
# 참조 : https://bitcoincore.org/en/segwit_wallet_dev/
#
# 2018.4.30 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
import binascii
import bitcoin.main as btc

# txid = c586389e5e4b3acb9d6c8be1c19ae8ab2795397633176f5a6442a261bbdefc3a
scriptSig = '0014a4b4ca48de0b3fffc15404a1acdc8dbaae226955'

# utxo = 42f7d0545ef45bd3b9cfee6b170cf6314a3bd8b3f09b610eeb436d92993ad440 : 1
# OP_HASH160 <script hash> OP_EQUAL
scriptHash = '2928f43af18d2d60e8a843540d8086b305341339'

# combined script
print ('\n\n사례 : txid = c586389e5e4b3acb9d6c8be1c19ae8ab2795397633176f5a6442a261b
print("\nCombined Script (without Witness) :")
print("\n<%s> OP_HASH160 <%s> OP_EQUAL" % (scriptSig, scriptHash))

# Validity check : OP_HASH160
check = btc.bin_hash160(binascii.unhexlify(scriptSig))
```



(실습 파일 : 4-7.SegWit(거래예시).xlsx)

Segregated Witness (SegWit) : Backward Compatibility 거래 예시

- Soft Fork를 위해서는 주소 유형별로 Anyone-can-spend 가 성립하도록 ScriptSig를 구성해야 함. ScriptPubKey는 기존 노드가 하는 일로 변경할 수 없음.
- 아래 예시는 실제 블록체인에 기록된 주소 유형별 SegWit Transaction임. 각 거래의 UTXO를 확인하여 Witness 없이 거래를 검증할 수 있음. → Exercise
- 첫 번째 예시는 P2SH 주소 끼리 SegWit 거래이고, 두 번째 예시는 P2SH에서 bech32 주소로의 SegWit 거래임..

txid : c586389e5e4b3acb9d6c8be1c19ae8ab2795397633176f5a6442a261bbdefc3a	
From	To
35SegwitPieWKVHieXd97mnurNi8o6CM73	38Segwituno6sUoEkh57ycM6K7ej5gvJhM
ScriptSig: PUSHDATA(22)[0014a4b4ca48de0b3fffc15404a1acdc8dbaae226955] **Witness**: 02483045022100864ef8f6d8afa892dee0f31259b6·····	HASH160 PUSHDATA(20)[4a1154d50b03292b3024370901711946cb7cccc3] EQUAL

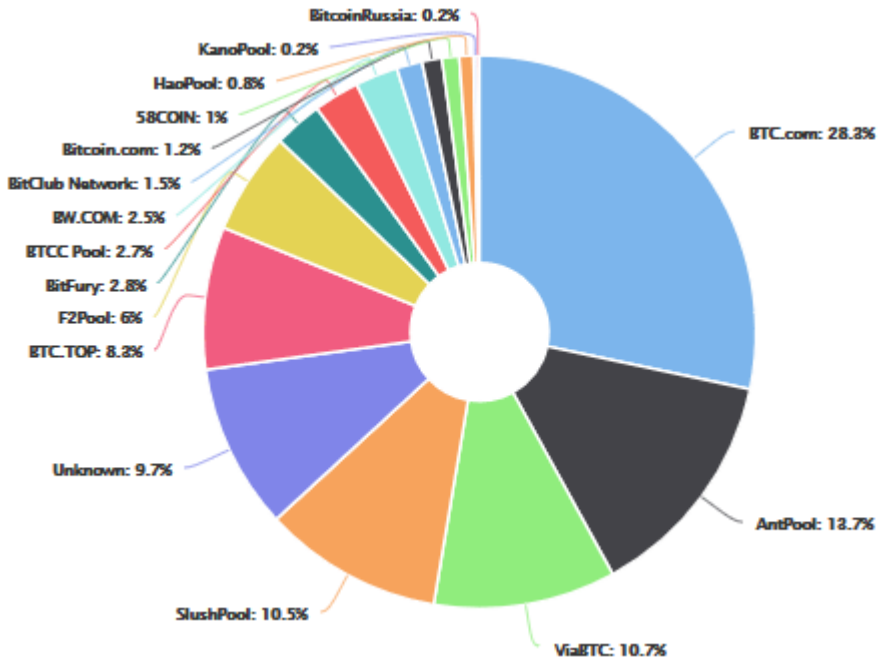
txid : 4d4803d4217f5ba71a8de414b98e5a55e337dd9f4424a39fe1e7485c8d178f5d	
From	To
3JEAszztBTBtsGgxT75mX1QMmvp3FbtLZy	bc1qwqdg6squsna38e46795at95yu9atm8azzmyvckulcc7kytlcckxswvvzej
ScriptSig: 0[] PUSHDATA(72)[3045022100a7984bfed3203175·····	"script":"0020701a8d401c84fb13e6baf169d59684e17abd9fa216c8cc5b9fc63d622ff8c58d"
	0[] PUSHDATA(32)[701a8d401c84fb13e6baf169d59684e17abd9fa216c8cc5b9fc63d622ff8c58d]

txid : 9aeb5738c16f8c10fdadd6db0530a2f8f7713c6e9bfc465b59794f86e15221ad	
From	To
bc1qwqdg6squsna38e46795at95yu9atm8azzmyvckulcc7kytlcckxswvvzej	1GQfDPVGRL2u78h8zEy5JMCahgJ4TrjN7G
ScriptSig: **Witness**: 040047304402205e05ceec0184a997620cb55·····	DUP HASH160 PUSHDATA(20)[a9035a5c2d874c1c66d8c6d37f5988279a5a6a04] EQUALVERIFY CHECKSIG

txid : aecd8f4c118c1919cd87d9e1e8f1f0fa670d226ca58e95323928594d287ad6f5	
From	To
bc1qzjeg3h996kw24zrg69nge97fw8jc4v7v7yznftzk06j3429t52vse9tkp9	3KW7Wtg7m1EuwFTpW7wJ6Mm4CdANEM3B5G
ScriptSig: **Witness**: 040048304502210abc61f1d33c28640e7901b9831f889fb2·····	HASH160 PUSHDATA(20)[c3625b4e90c1ce165820fc09957b87dbc4cac70c] EQUAL
	bc1qwqdg6squsna38e46795at95yu9atm8azzmyvckulcc7kytlcckxswvvzej
	0[] PUSHDATA(32)[701a8d401c84fb13e6baf169d59684e17abd9fa216c8cc5b9fc63d622ff8c58d]

5. 채굴 (Mining)

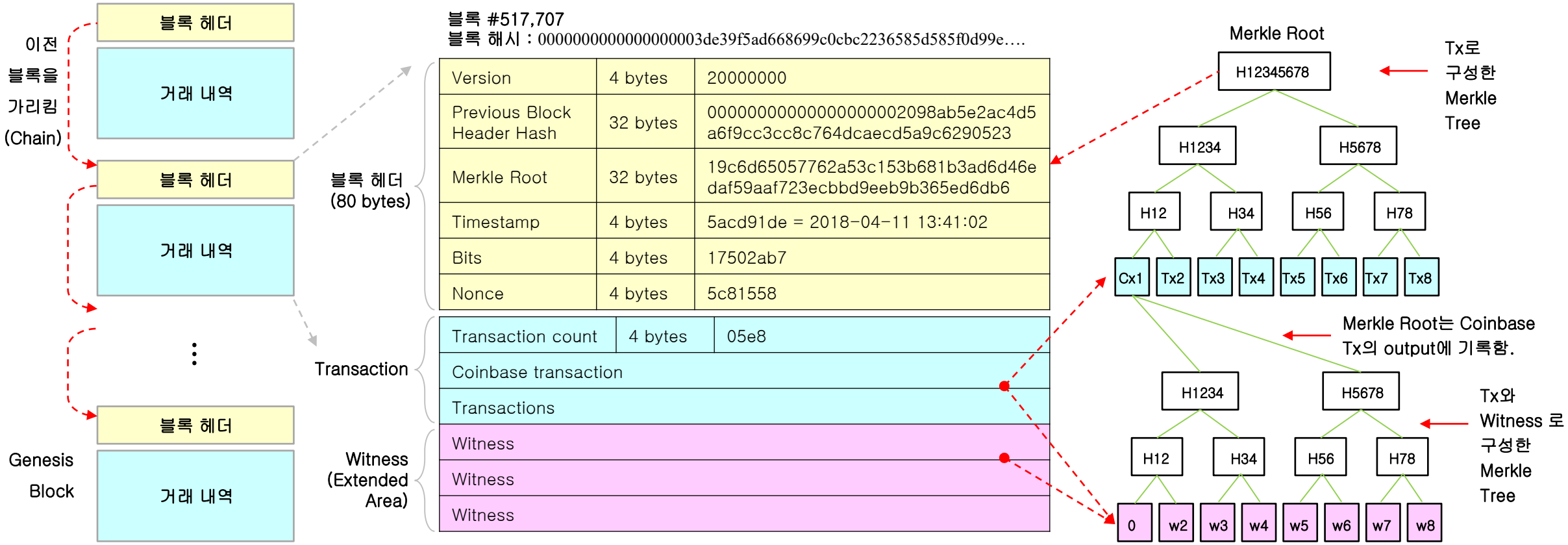
5-1. 블록 헤더 구조
5-2. 블록 버전 (Version) : BIP-9
5-3. 해시 난이도 (Target bits & Difficulty)
5-4. 해시 난이도 조절 (Retarget)
5-5. Merkle Tree (Merkle Root)
5-6. Nonce & Extra Nonce
5-7. Mining 절차
5-8. Hash Power (GPU, ASIC)
5-9. 비트코인 발행량 (미국 달러 발행량과 비교)
5-10. Transaction Fee와 최적 Block 크기
5-11. Solo mining vs. Pool mining
5-12. 블록체인의 일시적 불일치 (Fork)
5-13. Block height, Depth, Confirmation
5-14. Hard Fork와 Soft Fork
5-15. Hard Fork와 네트워크 분리



blockchain.info/pools

블록 헤더 구조

- 블록체인의 블록은 Header와 Body 부분으로 구성되어 있고, 헤더에는 이전 블록의 해시 값이 기록되어 있어, 블록들은 서로 체인으로 연결됨.
- 한 블록의 크기는 1 MB로 제한되었으나, SegWit 적용 이후 약 1.8 MB 까지 확장될 수 있음. SegWit 적용 이후에는 바이트 단위가 아닌 블록 weight라는 단위로 측정하도록 변경되었음.
- Body에는 Transaction (Tx) 데이터가 기록되어 있음. Tx 데이터 중 서명 (Signature, or Witness) 부분은 따로 분리되어 별도의 확장된 공간에 기록됨.
- 서명 부분을 포함한 Tx로 1차 Merkle Tree를 구성하고, 서명 부분이 없는 Tx들로 최종 Merkle Tree를 구성함.



블록 헤더 구조

- 블록 헤더는 아래와 같이 버전, 이전 블록의 해시 값, 머클 트리, 블록 생성 시각, nBits, nonce의 6개 항목으로 구성되어 있고, 총 길이는 80 바이트임.
- 현재 블록 헤더의 해시 값은 기록되어 있는 것이 아니라, 80 바이트 데이터를 double-SHA256으로 계산해서 사용함.
- 6개 항목은 모두 little-endian 방식임. (예시에 보이는 것과 바이트 순서가 반대임)

Bytes	Name	Data Type	Description	예시 (블록 #520608)
4	version	int32_t	The block version number indicates which set of block validation rules to follow.	20000000
32	previous block header hash	char[32]	A SHA256(SHA256()) hash in internal byte order of the previous block's header. This ensures no previous block can be changed without also changing this block's header.	0000000000000000028a9837a638d6ab0b0aa51cd97c87cefd7ca0b5ca55201
32	merkle root hash	char[32]	A SHA256(SHA256()) hash in internal byte order. The merkle root is derived from the hashes of all transactions included in this block, ensuring that none of those transactions can be modified without modifying the header. See the merkle trees section below.	5299b7778e8409227af85b00ca4f006717a5d7e90470012779441deea32b67ce
4	time	uint32_t	The block time is a Unix epoch time when the miner started hashing the header (according to the miner). Must be strictly greater than the median time of the previous 11 blocks. Full nodes will not accept blocks with headers more than two hours in the future according to their clock.	5ae7502d (= 2018-04-30 17:19:41)
4	nBits	uint32_t	An encoded version of the target threshold this block's header hash must be less than or equal to. See the nBits format described below.	1745fb53
4	nonce	uint32_t	An arbitrary number miners change to modify the header hash in order to produce a hash less than or equal to the target threshold. If all 32-bit values are tested, the time can be updated or the coinbase transaction can be changed and the merkle root updated.	d96e2432 (= 3647874098)

* 참조 : https://bitcoin.org/en/developer-reference#block-headers

(실습 파일 : 5-1.BlockHeader.py)

블록 헤더 구조 : Python 연습

- 실제 블록 헤더 데이터 (80 byte)를 파싱하고, 블록 해시 값을 계산하여 blockchain.info 사이트 결과와 비교함.

```python
# 블록 헤더 데이터를 파싱하고, 해시 값을 계산한다.
#
# 2018.5.1 : 아마추어 퀀트 (조성현)
# ----------------------------------------
import binascii
import hashlib
from datetime import datetime

# big-endian <--> little-endian으로 변환한다
# 단, x는 hexa-string을 입력한다.
def reverse(x):
    return ''.join(reversed([x[i:i+2] for i in range(0, len(x), 2)]))

def doubleSha256(header):
    return hashlib.sha256(hashlib.sha256(header).digest()).digest()

def makeHeader(version, prevHash, merkle, timeStamp, nBits, nonce):
    h = str(hex(nonce))[2:]
    s = reverse(version) + reverse(prevHash) + reverse(merkle) + reverse(timeSta
    return s

# block #520608의 header
# blockchain.info/block/ 에서 format=hex로 변환하여 블록 데이터를 받고, 앞 부분 80
# hash = 0000000000000000000362055c948f121449ae827599dc88882ffc107a70fca1b
header = '000000200152a55c0bcad7ef7cc897cd51aab0b06a8d637a83a928000000000000000000

# Header hash를 계산하고 blockchain.info의 결과와 비교해 본다
h = doubleSha256(binascii.unhexlify(header))
print("Header Hash = ", reverse(h.hex()))

# header를 decode한다.
version = reverse(header[0:0 + 4 * 2])        # 4 bytes
```

Name	Size	Type	Date Modified
4-5.segwit_addr.py	1 KB	py File	2018-04-28 오전 11:37
4-6.AnyoneCanSpend.py	2 KB	py File	2018-04-30 오전 2:49
4-7.SegWit(거래예시).xlsx	11 KB	xlsx File	2018-04-30 오후 2:20
5-1.BlockHeader.py	3 KB	py File	2018-05-01 오전 3:22

File explorer | Profiler

IPython console

Console 1/A

```
Header =
000000200152a55c0bcad7ef7cc897cd51aab0b06a8d637a83a928000000000000000000ce672b
a3ee1d447927017004e9d7a51767004fca005bf87a2209848e77b799522d50e75a53fb45173224
6ed9
```

Parsing

```
   Version = 20000000
  prevHash = 0000000000000000028a9837a638d6ab0b0aa51cd97c87cefd7ca0b5ca55201
  currHash = 0000000000000000362055c948f121449ae827599dc88882ffc107a70fca1b
merkleRoot = 5299b7778e8409227af85b00ca4f006717a5d7e9047001279441deea32b67ce
time stamp = 5ae7502d [2018-04-30 17:19:41]
     nBits = 1745fb53
     nonce = 3647874098
```

현재 블록 헤더의 해시 값

Assemble

```
Header =
000000200152a55c0bcad7ef7cc897cd51aab0b06a8d637a83a928000000000000000000ce672b
a3ee1d447927017004e9d7a51767004fca005bf87a2209848e77b799522d50e75a53fb45173224
6ed9

In [4]:
```

History log | IPython console

(실습 파일 : 5-2.BlockVersion.xlsx)

블록의 버전 : Version (BIP-9)

- BIP-9 표준은 Soft Fork가 진행될 때 Miner들이 해당 기능을 지원할 준비가 되었는지를 Version에 표시하도록 권고하고 있음 (강제 사항은 아님).
- 32 bit Version의 처음 3 비트는 001로 시작하고, 나머지 29개 비트는 Soft Fork가 진행될 때 해당 bit를 세트함. SegWit의 경우는 비트 1 이었으므로 version = 0x20000002 이었음. 특정 비트가 세트된 블록이 얼마나 많은지를 파악하면, 얼마나 많은 Miner들이 해당 Soft Fork를 지원하는지 파악할 수 있음. 동시에 29개 Soft Fork 진행 가능.
- 아래 예시에서 2016.11.1~2017.11.15 까지 0x20000002로 생성된 블록의 변화를 관찰할 수 있음. 현재는 Soft Fork가 없는 0x20000000 으로 Mining되고 있음.

BIP-141 (SegWit)

Deployment

This BIP will be deployed by "version bits" BIP-9 with the name "segwit" and using bit 1.

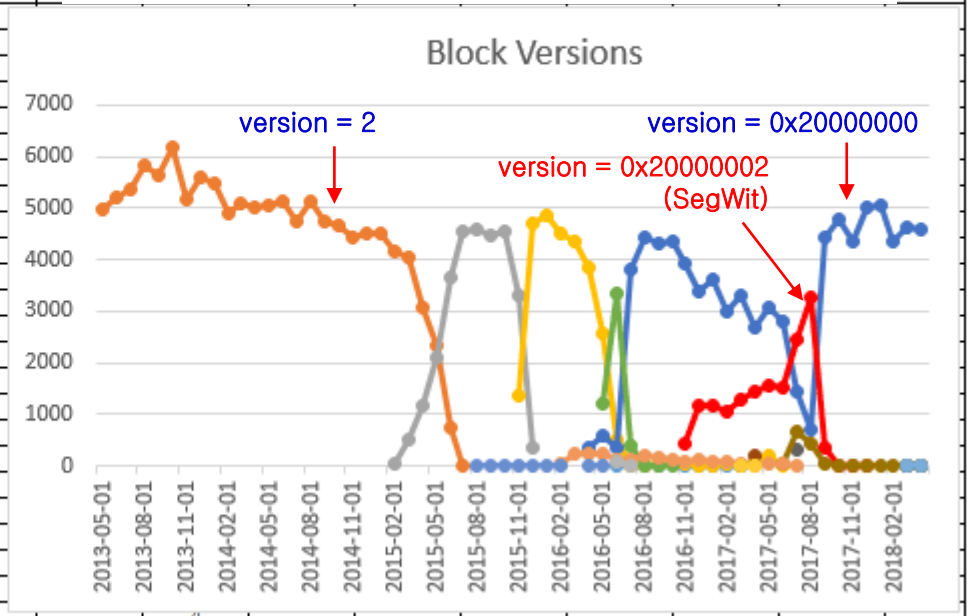
For Bitcoin mainnet, the BIP-9 starttime will be midnight 15 november 2016 UTC (Epoch timestamp 1479168000) and BIP-9 timeout will be midnight 15 november 2017 UTC (Epoch timestamp 1510704000).

version -->	8000004	2	3	4	20000000	20000001	20000002	20000004	20000010	20000012	20000F00	20FFF000	20FFFF00	30000000	30000001	30000007	BIP101
2015-11-01			3295	1350													4
2015-12-01			364	4692													7
2016-01-01				4866													1
2016-02-01				4514													1
2016-03-01				4347													
2016-04-01				3847	376												
2016-05-01				2554	598	1217											
2016-06-01	6			516	363	3362											
2016-07-01	9			6	3805	390											
2016-08-01	1			2	4453	2											
2016-09-01	1			45	4335	3											
2016-10-01				40	4369	1											
2016-11-01	4			37	3942		448										
2016-12-01	4			9	3382		1185										
2017-01-01	5			13	3606		1183										
2017-02-01	8			65	3012		1065										
2017-03-01	1			2	3307	1	1281										
2017-04-01				100	2669		1435	212									
2017-05-01				184	3058		1573										
2017-06-01				23	2815		1539										
2017-07-01					1448		2462										
2017-08-01					723		3265										
2017-09-01					4453		347										
2017-10-01					4793		20										
2017-11-01					4343		27										
2017-12-01					5021		2										
2018-01-01					5049												
2018-02-01					4365												
2018-03-01					4647					3	1	2	2				
2018-04-01					4602					1	2	4	1				

None

SegWit

* 자료 출처 : https://data.bitcoinity.org/bitcoin/block_version/5y



해시 난이도 : nBits (Target Bits) & Difficulty

- Target Bits는 블록 헤더의 해시 값에 대한 상한 값을 의미함. 즉, 헤더의 해시 값은 Target Bits 보다 작아야 함.
- Target Bits는 아래와 같이 Compact 형식으로 구성됨. 처음 2 바이트는 byte 길이를 나타내고 (exponents), 다음 3 바이트는 coefficient를 의미함.
- Compact 형식의 Target Bits는 아래 공식 (1)로 Normal 형식으로 변환할 수 있음.
- Difficulty는 Satoshi가 생성한 최초 블록 (Genesis block)의 난이도에 비해 현재 난이도가 얼마나 어려워졌는지를 측정한 척도임.
- Difficulty 척도는 식 (2)로 측정함. 아래 예시의 Target = 0x1745fb53은 Genesis 블록보다 약 $4 * 10^{12}$ 배 어려워졌음을 의미함.
- Target Bits는 누가 정해주는 것이 아니라 Miner가 제시하고 Full Node가 검증하는 방식으로 운영됨 (Decentralization).→ Retarget 개념이 필요함.
- Target Bits 제시가 잘못 되었거나, 헤더의 해시 값이 제시된 Target Bits 보다 크면 Full Node는 이 블록을 Reject 함.

Target Bits

Target Bits (Compact Format) = $0x17\ \ 45fb53$

Coefficient (0x45fb52)

byte 길이 (Exponents)
= 0x17 = 23 bytes

$0x45fb53000000000000000000000000000000000000$

$$(1)\ target = coefficient\ *\ 2^{\{8\ *(exponents\ -3)\}}$$

Difficulty

Genesis Block의 Target Bits = $0x1d00ffff$

Genesis Block의 Difficulty = 1 (기준)

$$(2)\ \text{Difficulty} = \frac{Genesis\ target}{target}$$

$$\text{Difficulty} = \frac{0x00ffff000000000000000000000000000000000000000000000000}{0x45fb53000000000000000000000000000000000000}$$

$$(2)\ Difficulty\ =\ 402059196164.954$$

(실습 파일 : 5-3.TargetBits.py)

해시 난이도 : nBits (Target Bits) & Difficulty – Python 연습

- 실제 블록 #520,608 의 nBits와 Difficulty를 계산해 보고, 헤더의 해시 값이 Target bits보다 작은지 확인함.

```
# target bits로부터 target value와 difficulty를 계산한다.
# bits는 4byte의 compact format임. 이것을 32byte의 target value로 변환한다.
# block hash는 target value보다 작아야 한다.
#
# ex : bits = 0x1745fb53 (블록 #520608의 bits)
#
#      0x17 = exponents : target value는 총 23 byte. 앞의 9byte는 0.
#      0x45fb53 = coefficient
#      target value = 0x45fb530000000000000000000000000000000000000000 (23 bytes)
#      target value = 0x0000000000000000045fb53000000..... (32 bytes)
#                     앞 부분에 00이 18개 와야함. <-- miner가 풀어야할 숙제임.
#
# target value 계산 방법 :
# target value = coefficient * 2 ** (8 * (exponent - 3))
#
# difficulty는 genesis block의 target value를 기준으로 현재 target value가 몇 배
# 어려워 졌는지를 척도화 한 것임.
#
# difficulty = genesis block의 target value / 현재의 target value
#
# 2018.5.1 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------------
def targetValue(bits):
    # bit의 왼쪽 1바이트 (exponents)를 추출한다.
    exponents = bits >> 24

    # bits의 오른쪽 3바이트 (coefficient)를 추출한다.
    coefficient = bits & 0x007fffff

    # target value를 계산한다
    target = coefficient << 8 * (exponents - 3)

```

```
Name                        Size      Type        Date Modified
4-7.SegWit(거래예시).xlsx    11 KB     xlsx File   2018-04-30 오후 ...
5-1.BlockHeader.py           2 KB      py File     2018-05-01 오후 ...
5-2.BlockVersion.xlsx        21 KB     xlsx File   2018-05-02 오전 ...
5-3.TargetBits.py            2 KB      py File     2018-05-02 오전 ...

File explorer  Profiler

IPython console
Console 1/A

 Target Bits = 0x1745fb53
Target Value = 0x45fb530000000000000000000000000000000000000000
  Difficulty = 4022059196164.954

Block Hash =  0x362055c948f121449ae827599dc88882ffc107a70fca1b
Block hash is less than target value. --> valid.

In [14]:

History log  IPython console
```

해시 난이도 조절 : Retarget

- Target Bits는 블록 생성 시간 간격이 평균 약 10 분이 되도록 조절해야 함. 조절은 Miner가 수행하고, 검증은 Full Node가 수행함.
- Target Bits는 2016 블록마다 조절함. 10 분마다 블록이 생성된다면 2 주 (weeks) 동안 총 2016개의 블록이 생성됨.
- 최종 블록 생성 시각과, 2016개 이전 블록의 생성 시각을 측정하면 2016개 블록이 생성된 평균 시간을 측정할 수 있음
- Miner는 블록 번호가 2016의 배수가 될 때마다 지난 2 주 동안의 평균 시간을 측정하여 10 분보다 작으면 난이도를 더 어렵게 조절하고, 10 분보다 크면 난이도를 더 쉽게 조절함.
- 아래 예시에서 블록 #520127 까지 생성되었고, Miner는 새로운 블록 #520128을 생성하려고 함. 520128은 2016의 배수이므로 Target bits를 조절해야 함.
- 블록 #518112 ~ #520127 까지 총 1,154,642 초 걸렸고, 블록 당 평균 시간은 9.55 분임. (실제는 2015개 블록 생성 시간임)
- 아래 예시는 최근 2 주간 블록 생성 시간이 평균 9.55 분 이므로 난이도를 약간 높게 조절해야하기 때문에 0x1749500d → 1745fb53 으로 조절하였음.

$$New\ Target = Old\ Target * \frac{1154642}{14 * 24 * 60 * 60}$$

← 2 weeks 동안 2015 블록이 생성된 시간 (초 단위)

← 2 weeks (초 단위)

block	% 2016	difficulty	bits (decimal)	bit (hex)	timeStamp	time	timeStamp (dec)	Total elapsed (Sec)	Average elapsed (Min)
518110	2014	3,511,060,552,899.72	391129783	17502AB7	5ad15f0a	2018-04-14 01:53:14	1523670794		
518111	2015	3,511,060,552,899.72	391129783	17502AB7	5ad165da	2018-04-14 02:22:18	1523672538		
518112	0	3,839,316,899,029.67	390680589	1749500D	5ad16764	2018-04-14 02:28:52	1523672932	1154642	9.55
518113	1	3,839,316,899,029.67	390680589	1749500D	5ad16c02	2018-04-14 02:48:34	1523674114		
518114	2	3,839,316,899,029.67	390680589	1749500D	5ad1713d	2018-04-14 03:10:53	1523675453		
520126	2014	3,839,316,899,029.67	390680589	1749500D	5ae304e2	2018-04-27 11:09:22	1524827362		
520127	2015	3,839,316,899,029.67	390680589	1749500D	5ae305b6	2018-04-27 11:12:54	1524827574		
520128	0	4,022,059,196,164.95	390462291	1745FB53	5ae3085d	2018-04-27 11:24:13	1524828253		

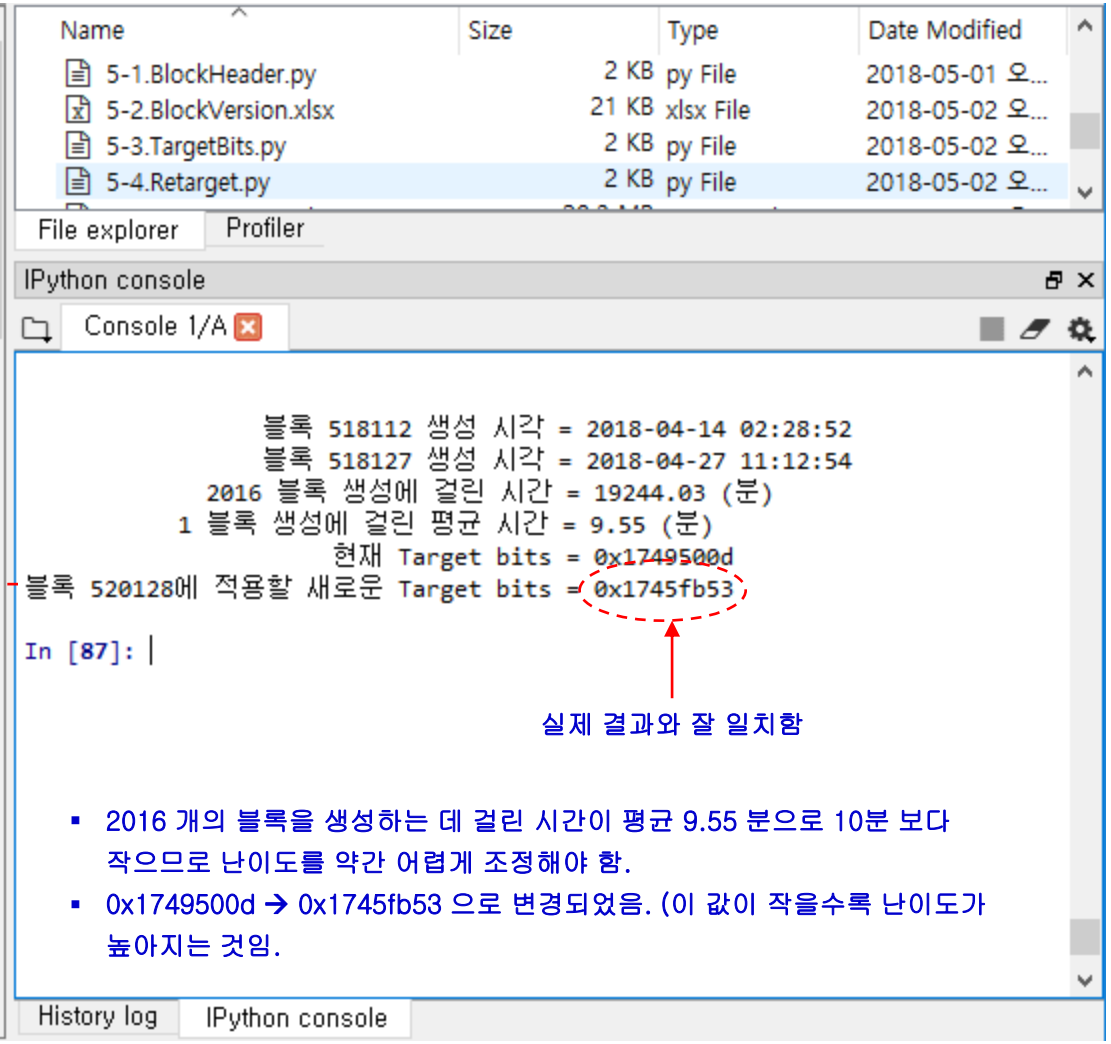
2016 블록

(실습 파일 : 5-4.ReTarget.py)

해시 난이도 조절 : Retarget – Python 연습

- 이전 예시의 블록 #520,128의 Target Bits (0x1745fb53)를 검증함. Full 노드는 Target Bits 를 검증하고, 잘못 지정되었으면 이 블록을 Reject 함.

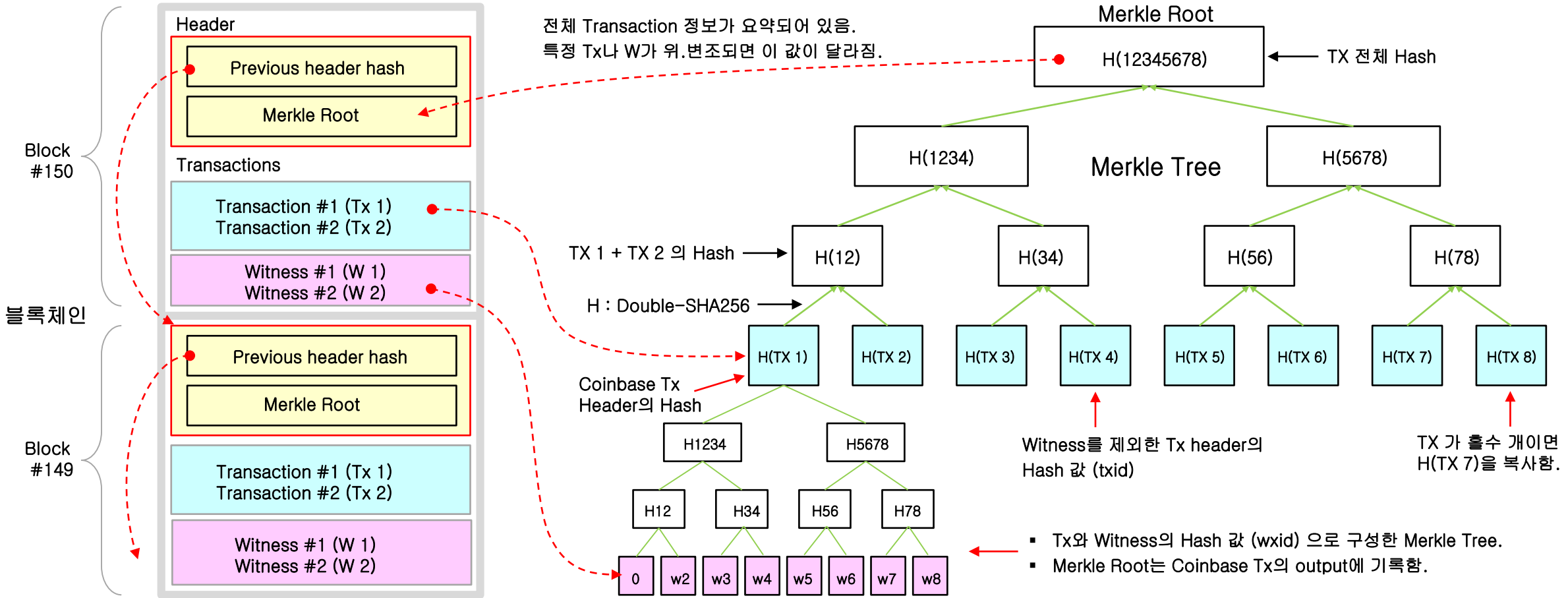
```
# 2016 블록 주기로 Target bits를 변경한다.
#
# 현재 블록 height는 520127 이고, Miner는 520128 블록을 생성하려고 한다.
# 520128 % 2016 = 0 이므로 이 블록의 Target bits를 변경해야 한다.
# 기존의 0x174950d 를 얼마로 변경해야 할까?
#
# block   % 2016 difficulty            bits         timeStamp    time
# 518110  2014   3,511,060,552,899.72  0x17502AB7   0x5ad15f0a   2018-04-14 01:53:14
# 518111  2015   3,511,060,552,899.72  0x17502AB7   0x5ad165da   2018-04-14 02:22:18
# 518112  0      3,839,316,899,029.67  0x1749500D   0x5ad16764   2018-04-14 02:28:52
# 518113  1      3,839,316,899,029.67  0x1749500D   0x5ad16c02   2018-04-14 02:48:34
# 518114  2      3,839,316,899,029.67  0x1749500D   0x5ad1713d   2018-04-14 03:10:53
# 520126  2014   3,839,316,899,029.67  0x1749500D   0x5ae304e2   2018-04-27 11:09:22
# 520127  2015   3,839,316,899,029.67  0x1749500D   0x5ae305b6   2018-04-27 11:12:54
# 520128  0      4,022,059,196,164.95  0x1745FB53   0x5ae3085d   2018-04-27 11:24:13
#
# 참조 : Bitcoin Core의 src/pow.cpp - CalculateNextWorkRequired()
#
# 2018.5.2 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------------
from datetime import datetime

def UncompactForm(bits):
    # bit의 왼쪽 1바이트 (exponents)를 추출한다.
    exponents = bits >> 24

    # bits의 오른쪽 3바이트 (coefficient)를 추출한다.
    coefficient = bits & 0x007fffff

    # target value를 계산한다
    target = coefficient << 8 * (exponents - 3)
    return target
```



- 2016 개의 블록을 생성하는 데 걸린 시간이 평균 9.55 분으로 10분 보다 작으므로 난이도를 약간 어렵게 조정해야 함.
- 0x1749500d → 0x1745fb53 으로 변경되었음. (이 값이 작을수록 난이도가 높아지는 것임.

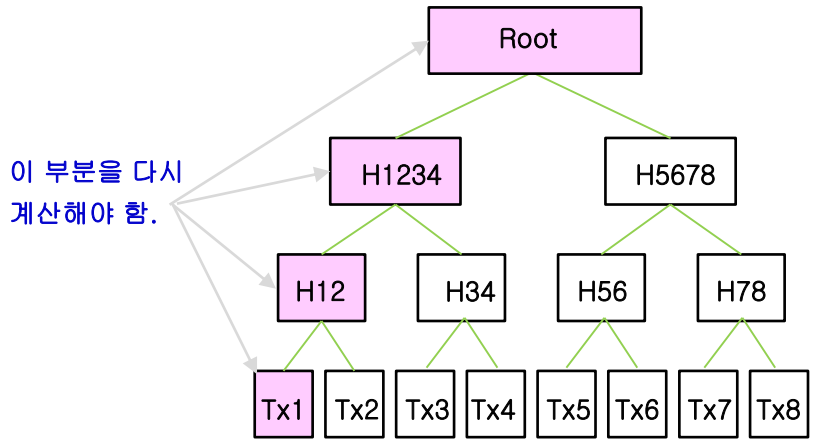
Merkle Tree : Merkle Root

- Merkle Tree는 Transaction (Tx)과 Witness (W)들을 Hash function (H : double-SHA256)으로 요약한 것임. Tx나 W가 일부라도 변경되면 Merkle Root가 달라지므로 Tx나 W는 변경될 수 없음 (블록에 기록된 이후에는 변경될 수 없음). Tx와 Witness는 위조, 변조가 불가함.
- Witness 데이터로 별도의 Merkle Tree를 구축하여 Merkle Root를 Coinbase Tx의 2번째 Output에 기록함. (Coinbase Tx의 output은 최소 2개 임).
- Merkle Tree는 SPV 노드가 자신의 거래가 특정 블록에 포함되었는지 Merkle 인증할 때도 사용됨. (P2P 프로토콜 부분에서 자세히 다룸.)



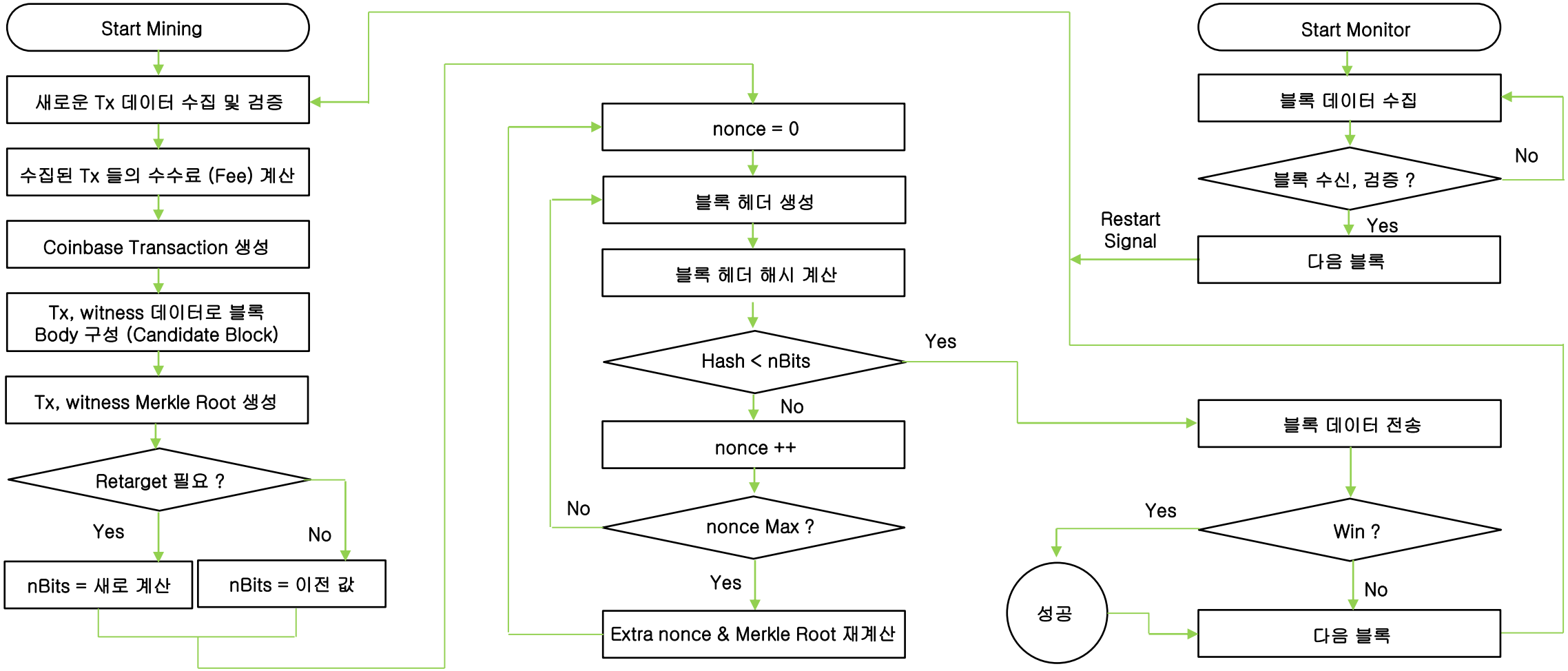
Nonce 와 Extra nonce

- Nonce 값을 0 ~ 0xffffffff (약 42 억) 로 변경해 가면서, Header의 해시 값이 Target bits 보다 작은 nonce 값을 찾음 (해를 찾는다고 표현함).
- 최근에는 Miner 들의 경쟁이 심해지고 ASIC의 등장으로 Hash power가 높아지면서 난이도가 계속 높아지고 있음. (42 억 번 수행 시간이 1초 미만임).
- 난이도가 높아져서 Nonce 값의 범위를 전부 뒤져도 해를 찾지 못하는 경우가 발생할 수도 있음. 이런 경우에는 변경할 수 있는 다른 값을 변경해야 함.
- timestamp를 약간 변경할 수도 있으나 한계가 있음. Full 노드는 블록의 timestamp가 2시간 이내임을 체크하므로, 2시간 이내로만 변경 가능함.
- timestamp를 약간 변경하고 Nonce 값을 다시 0 ~ 0xffffffff 으로 변경하면 해를 찾는 범위가 넓어질 수는 있음.
- 다른 방법으로는 Coinbase Tx의 Input 값을 변경할 수 있음. Input의 Script에는 Miner가 자유롭게 기록할 수 있는 부분이 있는데, 이 부분을 변경해 가면서 해를 찾을 수 있음. 이 값을 변경하면서 또 Nonce 값을 변경할 수 있기 때문에 거의 무한한 범위에서 해를 찾을 수 있음.
- Coinbase Tx의 Input Script를 변경하면 Merkle Root를 다시 계산해야 함.
- 참고 : Genesis 블록의 Coinbase input에 Satoshi 기록한 Script : “The Times 03/Jan/2009 Chancellor on brink of second bailout for banks”



Mining 절차

- Mining은 아래 절차로 수행됨 (개념적으로 간단히 표현함). 합의 규칙에 맞는 블록 헤더의 해시 값을 찾으면서, 동시에 다른 Miner가 먼저 성공했는지 모니터링 함.



(실습 파일 : 5-5.Mining.py)

Mining : Python 연습

- 주어진 헤더 정보로 Nonce 값을 변경해 가면서 Target Bits 보다 작은 헤더의 해시 값을 찾아 보고, Hash power를 측정함.

```python
# Block Header의 Hash 값이 Target bits 보다 작게 만드는 nonce 값을 찾는다.
#
# 시간이 오래 걸리므로 이미 알고 있는 nonce 값 주변을 검색해 본다
# 그리고 별도로 10 만번을 돌려서 Hash Power를 측정해 본다.
#
# 2018.5.2 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
import binascii
import hashlib
import time
from datetime import datetime

# big-endian <--> little-endian으로 변환한다
# 단, x는 hexa-string을 입력한다.
def reverse(x):
    return ''.join(reversed([x[i:i+2] for i in range(0, len(x), 2)]))

def doubleSha256(header):
    return hashlib.sha256(hashlib.sha256(header).digest()).digest()

def makeHeader(version, prevHash, merkle, timeStamp, nBits):
    s = reverse(version) + reverse(prevHash) + reverse(merkle) + reverse
    return s

def UncompactForm(bits):
    # bit의 왼쪽 1바이트 (exponents)를 추출한다.
    exponents = bits >> 24

    # bits의 오른쪽 3바이트 (coefficient)를 추출한다.
    coefficient = bits & 0x007fffff

    # target value을 계산한다
```

```
3647874085 b835a745be7bc4db8151ca171fc93d4d8c881207bc4c85717b2a119461a86063
3647874086 062c98a424f5c3b46bee2309a58a56a87a8d0cd2a04da1b6f6021b9d167642d8
3647874087 40a47f68f552a3435985a6497f061a169045d0612f4c779ca00110a8aff0ab16
3647874088 b9cc84eafb50e091c144a57dd31b431dee33de9f51ac0d22c5ddb60aadc2ede3
3647874089 11f94e5cbac75bfd812e7e079b2623226215df5d37564bd255b873e84b2fb6b6
3647874090 19c309d11cd2eb2ea63c786687e7b0eaaedfc0317839ddf5816d93db29374260
3647874091 2d5dd3a8f84c3c84f78e51d6d269f06d1e0314cfd8c5e988b5ef567f10e1ad93
3647874092 776b757e5bda98a63db6f441807e02ca56141f8ff2df3d1615a685e991a6a540
3647874093 fc4c3596d84869bc1ddd60112df9e58df9399e0c27622fb4d33a1cf68512ae5e
3647874094 b4259288f826999b249bdd744b5e0c62c7570c070799f01353d3fabce24fa85f
3647874095 b04155aa4d67dcbd37c3e9103601c2f9a3d0ff7b5c7413836df33f12db90b399
3647874096 fb846d6de642103320b88ffe89f9281a71cb03f80acc37dc6bc3d226247487ae
3647874097 0cf21bad3299ea52f7155794ed76973ba8ef4297e78c0785dd4c1af7b51d3290
3647874098 00000000000000000000362055c948f121449ae827599dc88882ffc107a70fca1b

   Version = 20000000
  prevHash = 0000000000000000000028a9837a638d6ab0b0aa51cd97c87cefd7ca0b5ca55201
merkleRoot = 5299b7778e8409227af85b00ca4f006717a5d7e90470012779441deea32b67ce
time stamp = 5ae7502d [2018-04-30 17:19:41]
     nBits = 1745fb53
     nonce = 3647874098 <-- Found
Header =
000000200152a55c0bcad7ef7cc897cd51aab0b06a8d637a83a928000000000000000000ce672ba3ee1d
447927017004e9d7a51767004fca005bf87a2209848e77b799522d50e75a53fb451732246ed9

Hash = 00000000000000000000362055c948f121449ae827599dc88882ffc107a70fca1b


Hash power 측정 :
Elpased time (10 만회) = 1.22 (sec)
Hash power (Python & CPU & i7 processor) = 84200.42 (hash/sec)

In [91]:
```

History log | IPython console

Hash Power (GPU와 ASIC)

- Miner 들은 Target Bits 보다 작은 블록 헤더의 해시 값을 다른 Miner들 보다 먼저 찾기 위해 무한 경쟁을 펼치고 있음 (Arms race).
- Miner 들은 비트코인 초창기인 2010 ~ 2011에는 CPU 를 사용하다가 점차 GPU (Graphics Process Unit)를 사용하게 됨.
- 2013년 부터는 SHA256 Hash 전용 반도체 소자인 FPGA (Field Programmable Gate Array)가 등장하다가 ASIC (Application Specific Integrated Circuit)을 사용하게 됨. FPGA는 프로그래밍이 가능하여 오류를 수정할 수 있어 초기 개발비가 저렴하나 속도가 늦고 (ASIC에 비해) 전력 소비가 많은 반면, ASIC은 특정 용도의 기능을 (Hash 전용 기능)을 칩에 고정해 놓은 것으로 수정은 불가하지만 속도가 빠르고 소비 전력이 낮은 장점이 있음.
- ASIC의 등장으로 Miner들이 Hash 값을 계산하는 능력 (Hash Power)이 크게 증가하였고, 해시 난이도 (Difficulty)도 크게 증가 하였음.
- Hash Power는 초 당 계산 가능한 Hash 개수로 표시함 (Hash / sec).

- Mining hardware cluster

* 출처 : https://cryptona.co/bitcoin-mining-work/

- Bitcoin double SHA256 ASIC mining hardware

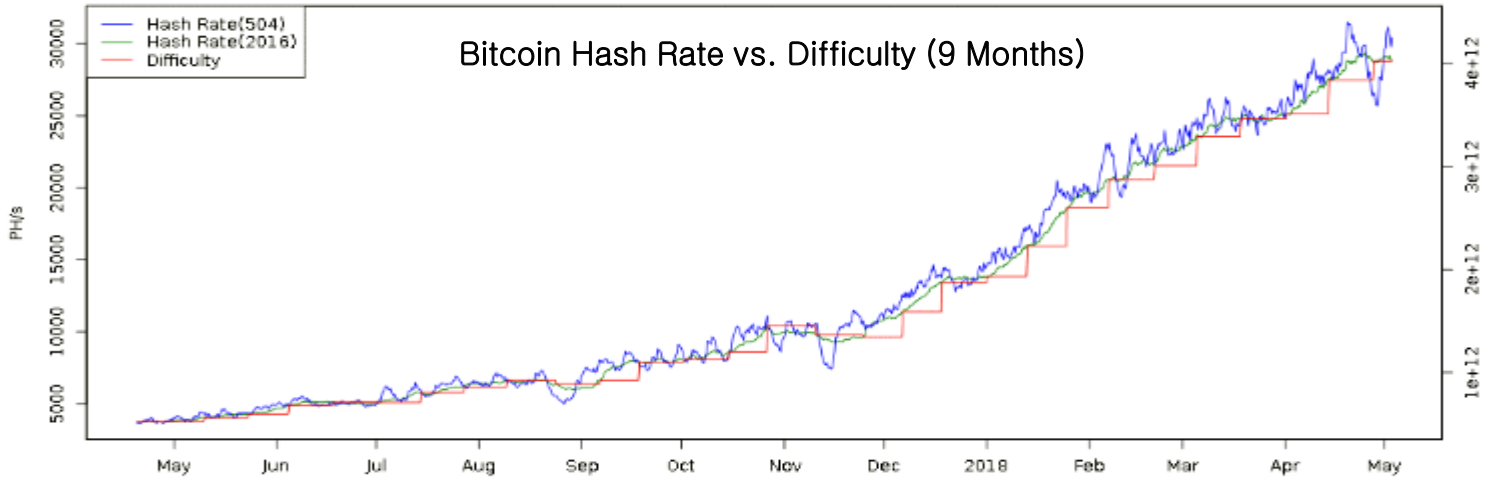
Product	Advertised Mhash/s	Mhash/J	Mhash/s /$	Watts	Price (USD)	Currently shipping	Comm ports
Ebit E10	18,000,000	11100	3440	1620	5230	Yes	Ethernet
AntMiner S9	14,000,000	10182	5833	1,375	2,400	Yes	Ethernet
Ebit E9++	14,000,000	10500	3600	1330	3880	Yes	Ethernet
WhatsMiner M3	11,500,000			1785		Yes	Ethernet
Avalon821	11,000,000	9170	3800	1200	2900	Bulk only	Ethernet
WhatsMiner M2	9,200,000			2046		DAed	Ethernet
Ebit E9+	9,000,000	6900	6428	1300	1400	Yes	Ethernet
Avalon761	8,800,000	6670	4730	1320	1860	Yes	Ethernet
AntMiner S5+	7,722,000	2247	3347	3,436	2,307	No	Ethernet
Avalon741	7,300,000	6350	5035	1150	1450	Yes	Ethernet
Ebit E9	6,300,000	7140	4468	882	1410	No	Ethernet
Avalon721	6,000,000	6000		1000		No	Ethernet
Spondooliestech SP35 Yukon	5,500,000	1506	2460	3650	2235	DAed	Ethernet

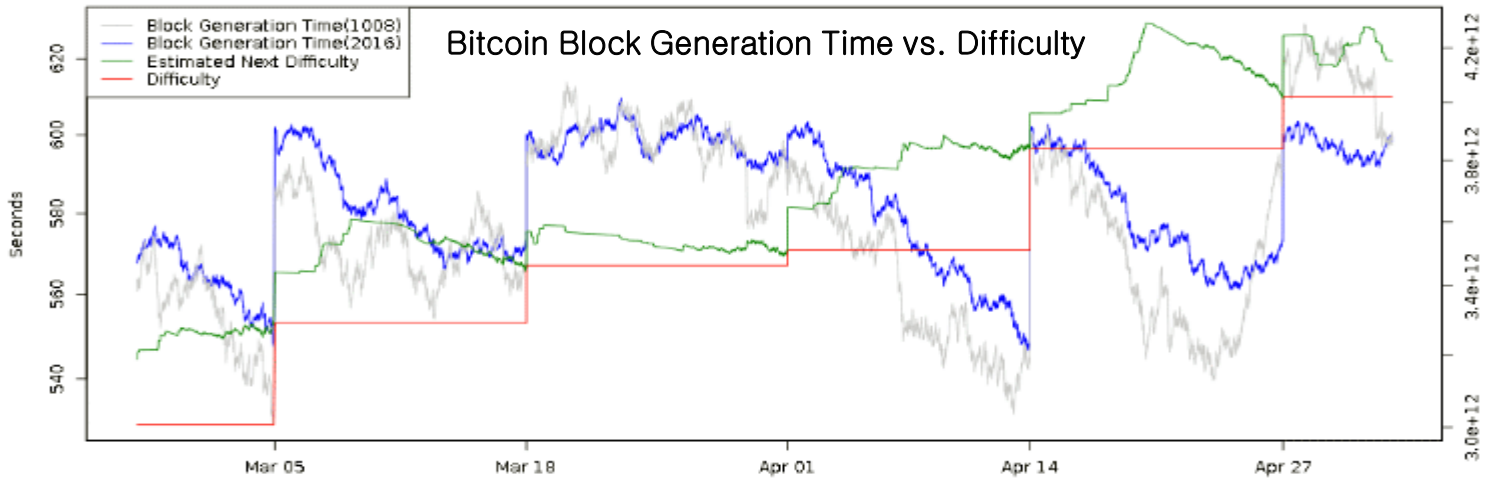
* 출처 : https://en.bitcoin.it/wiki/Mining_hardware_comparison

* 자료 출처 : https://bitcoinwisdom.com/bitcoin/difficulty

Hash Power 와 Difficulty

- 아래 사례는 최근 9 개월 간 Hash Power와 Difficulty 증가 추세를 보여 주고 있음. Difficulty가 증가할 때 블록 생성 시간도 늦어지지만 점차 낮아지면서 다시 Difficulty가 증가하는 모습이 반복되고 있음. 우측 자료는 Difficulty와 (추정된) Hash Rate을 보여 주고 있음.





Date	Difficulty	Change	Hash Rate
Apr 27 2018	4,022,059,196,164	4.76%	28,791,021,184 GH/s
Apr 14 2018	3,839,316,899,029	9.35%	27,482,900,867 GH/s
Apr 01 2018	3,511,060,552,899	1.40%	25,133,150,415 GH/s
Mar 18 2018	3,462,542,391,191	5.23%	24,785,843,885 GH/s
Mar 05 2018	3,290,605,988,754	9.42%	23,555,075,176 GH/s
Feb 20 2018	3,007,383,866,429	4.62%	21,527,692,255 GH/s
Feb 06 2018	2,874,674,234,415	10.43%	20,577,719,706 GH/s
Jan 25 2018	2,603,077,300,218	16.84%	18,633,553,122 GH/s
Jan 13 2018	2,227,847,638,503	15.36%	15,947,554,580 GH/s
Jan 01 2018	1,931,136,454,487	3.10%	13,823,613,194 GH/s
Dec 18 2017	1,873,105,475,221	17.74%	13,408,211,263 GH/s
Dec 06 2017	1,590,896,927,258	18.11%	11,388,083,790 GH/s
Nov 24 2017	1,347,001,430,558	-1.28%	9,642,211,820 GH/s
Nov 10 2017	1,364,422,081,125	-6.09%	9,766,913,694 GH/s
Oct 26 2017	1,452,839,779,145	21.39%	10,399,832,230 GH/s
Oct 15 2017	1,196,792,694,098	6.49%	8,566,975,802 GH/s
Oct 02 2017	1,123,863,285,132	1.85%	8,044,926,758 GH/s
Sep 18 2017	1,103,400,932,964	19.58%	7,898,451,536 GH/s
Sep 06 2017	922,724,699,725	3.89%	6,605,120,681 GH/s
Aug 24 2017	888,171,856,257	-3.80%	6,357,781,793 GH/s
Aug 09 2017	923,233,068,448	7.32%	6,608,759,726 GH/s
Jul 27 2017	860,221,984,436	6.92%	6,157,708,817 GH/s
Jul 14 2017	804,525,194,568	13.53%	5,759,015,666 GH/s
Jul 02 2017	708,659,466,230	-0.43%	5,072,782,052 GH/s
Jun 17 2017	711,697,198,173	4.85%	5,094,526,985 GH/s

Hash Power 와 Difficulty

- 아래 사례는 불과 1초 만에 블록이 생성된 경우임. Hash Power가 대단히 빠르다는 것을 알 수 있음.

블록 #521463

요약	
거래 수	794
출력 합계	13,918.95185795 BTC
예상된 거래량	6,977.95943844 BTC
거래 수수료	0.26660917 BTC
높이	521463 (주요 체인)
타임 스탬프	2018-05-06 10:40:29
수신 시간	2018-05-06 10:40:29
릴레이된 곳	BitFury
난이도	4,022,059,196,164.95
Bits	390462291
크기	602.997 kB
무게	2181.579 kWU
번역	0x20FFFF00
해시 난수	4083044925
블록 보상	12.5 BTC

블록 #521464

요약	
거래 수	1910
출력 합계	990.47528007 BTC
예상된 거래량	123.67821107 BTC
거래 수수료	0.03935835 BTC
높이	521464 (주요 체인)
타임 스탬프	2018-05-06 10:40:30
수신 시간	2018-05-06 10:40:30
릴레이된 곳	SlushPool
난이도	4,022,059,196,164.95
Bits	390462291
크기	1178.668 kB
무게	3993.1 kWU
번역	0x20000000
해시 난수	1894345261
블록 보상	12.5 BTC

- 서로 다른 주체 (BitFury, SlushPool)에 의해 1초 간격으로 블록이 생성되었음.
- Previous hash를 이용해야 하기 때문에 동시에 시도된 것이 아니라, 순차적으로 시도된 것임. 이전 블록이 생성된 이후에 다음 블록이 생성된 것임.
- 두 블록에 포함된 거래들도 모두 다름. (794개, 1910 개)
- BitFury가 블록을 생성하고, 전파하고, Full 노드가 승인하고, SlushPool이 Tx를 모으고, 블록을 생성하는데 1초 밖에 걸리지 않았음.
- 단, 타임 스탬프는 Miner가 기록한 것으로 약간의 오차는 있을 수 있음.
- 오차로 시간이 역전되는 경우도 있음 (ex : 블록 521570과 521571)

비트코인 발행량

- 2009년 1월 3일 Satoshi가 최초 블록을 생성한 보상으로 50 BTC를 받음. 최초 비트코인 발행량 = 50 BTC.
- 블록은 약 10 분 마다 생성되어 1년에 약 52,560 블록이 생성됨 (365 * 24 * 6). 4년에는 약 210,000 블록이 생성됨.
- Miner의 보상은 최초 50 BTC에서 시작하여, 210,000 블록 마다 (약 4년 마다) 절반으로 줄어듦. 50 BTC → 25 BTC → 12.5 BTC → 6.25 BTC → ….
- 블록 0 ~ 209999 까지는 50 BTC, 210000 ~ 419999 까지는 25 BTC, 420000 부터는 12.5 BTC를 받음. 현재 (2018년 4월)는 12.5 BTC 임.
- 비트코인의 연간 총 발행량은 52,560 블록 * Miner의 보상 (현재 12.5 BTC)으로 계산할 수 있음.
- Miner의 보상은 4년 마다 절반으로 줄어들어 약 2,140 년이 되면 없어짐 (1 Satoshi 미만). 이 때까지 발행될 총 비트코인은 약 2천 100만 BTC 임.
- Miner의 정규 보상은 점차 감소하지만, 거래가 증가하면 거래 수수료 (Fee)가 증가하여 Mining의 총 보상은 유지될 수도 있음.
- 실물 화폐 (ex : 미국 달러)는 매년 발행량이 증가하지만 (2008년 금융위기 이전까지 연간 약 10% 증가), 비트코인은 발행량이 점차 감소함. → 디플레이션 화폐
- 비트코인의 디플레이션 화폐 특성은 자본주의 화폐의 특성과 반대이므로, 비트코인이 실제 화폐로 유통되면 실물 경제에 어떤 영향을 미칠지에 대해서는 의견이 분분한 상태임. 비트코인의 총 발행 비율이 감소하면서 코인 당 실물 화폐 교환 가치가 상승하기 때문에 큰 문제가 없다는 견해도 있고, 반대 견해도 있음.
- 비트코인 지갑의 개인키를 분실하거나, 지갑 주인이 불의의 사고를 당해 지갑을 Access할 수 없을 때 해당 비트코인은 아무도 사용할 수 없으므로 사장될 수 있음. 이러한 이유로 비트코인의 실제 유통량은 오히려 감소할 수밖에 없음. → 디플레이션 특성이 가속화 됨.
- 2017년 중반까지 사장된 비트코인은 약 4M 정도로 추정한 보고도 있음.

* 자료 출처 : http://fortune.com/2017/11/25/lost-bitcoins/

	Total as of mid-2017	Percent Lost	Lost Bitcoins
Mined Coins	604,388	0%	-
Transactional	6,066,664	2%	121,333
Out of circulation ("Hodlers")	5,110,898	50%	2,555,449
Strategic Investors	3,557,539	2%	71,151
Original Coins ("Satoshi Coins")	1,041,715	100%	1,041,715
Total	16,381,204		***3,789,648***

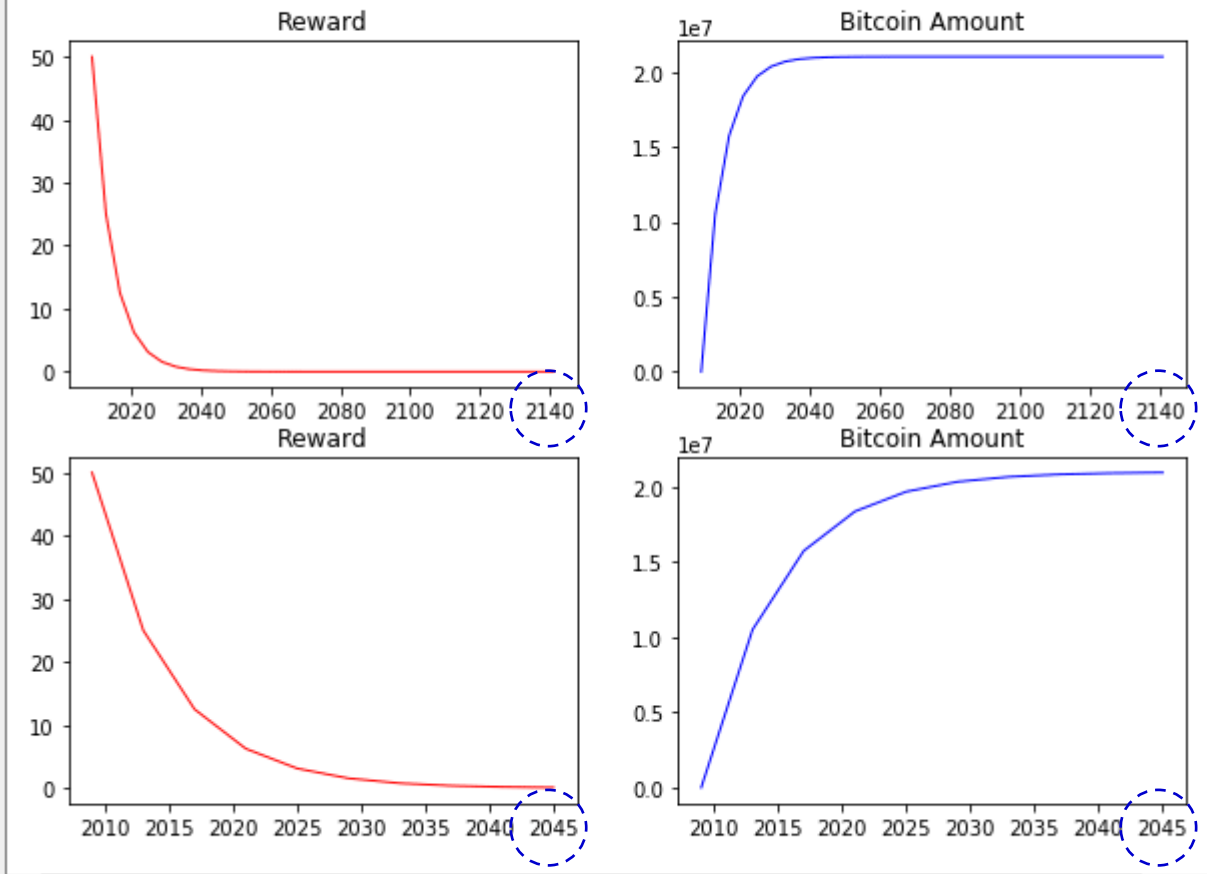
(실습 파일 : 5-6.MiningReward.py)

비트코인 발행량 확인 – Python 연습

- 비트코인은 2,140년 까지 약 2천 100만 BTC가 발행됨. 이 때의 블록 Height는 약 6백 9십만 블록임.

```python
# Bitcoin 총 발행량을 계산한다.
#
# 2009년 1월 50 BTC에서 시작하고, 210000 블록 (약 4년) 마다 절반으로 줄어든다
#
# 2018.5.3 : 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
import matplotlib.pyplot as plt

blockYear = 210000      # 1 년간 생성될 블록 수 (근사치 : 365 * 24 * 6)
Reward = [50]           # 초기 Reward = 50 BTC
Year = [2009]           # Mining 초기 연도
blockHeight = [0]       # 초기 블록 번호
Bitcoin = [50]          # 초기 발행량

# 4 년씩 지나가면서 blockHeight, Reward, 비트코인 발행량을 측정함.
for year in range(2010, 9999):
    blockHeight.append(blockHeight[-1] + blockYear - 1)
    Bitcoin.append(Bitcoin[-1] + Reward[-1] * blockYear)
    Reward.append(Reward[-1] / 2)
    Year.append(Year[-1] + 4)

    if Reward[-1] < 1e-8:    # 1 Satoshi
        break

print("\n\n* Total Bitcoin = %.2f until %d year" % (Bitcoin[-1], Year[-1]))
print("* Block Height =", blockHeight[-1])

fig = plt.figure(figsize=(10, 7))
p1 = fig.add_subplot(2,2,1)
p2 = fig.add_subplot(2,2,2)
p3 = fig.add_subplot(2,2,3)
p4 = fig.add_subplot(2,2,4)
```

```
* Total Bitcoin = 21000050.00 until 2141 year
* Block Height = 6929967
```



History log | IPython console

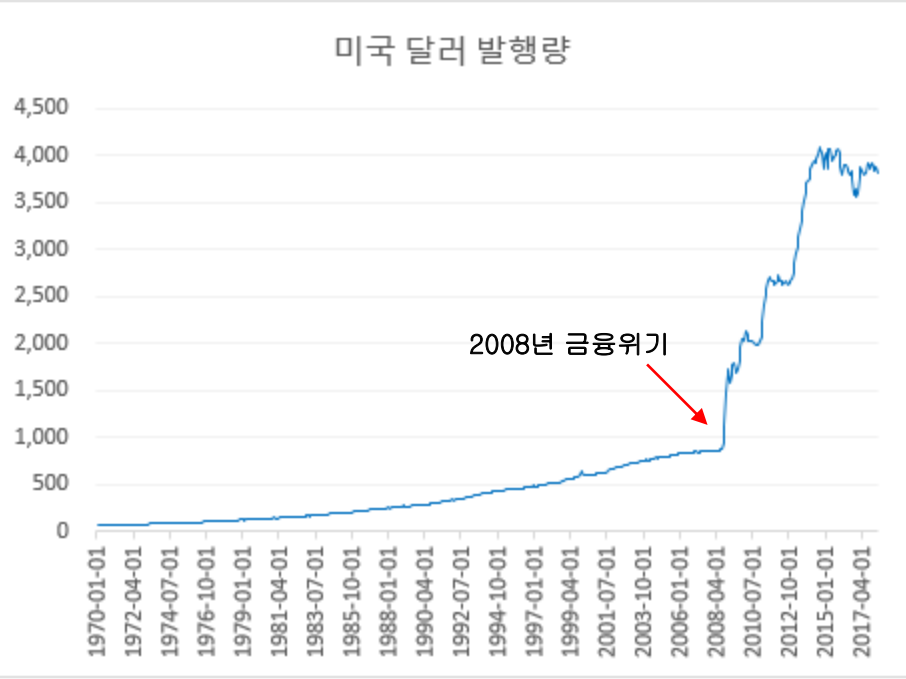
(실습 파일 : 5-7.비트코인발행량.py)

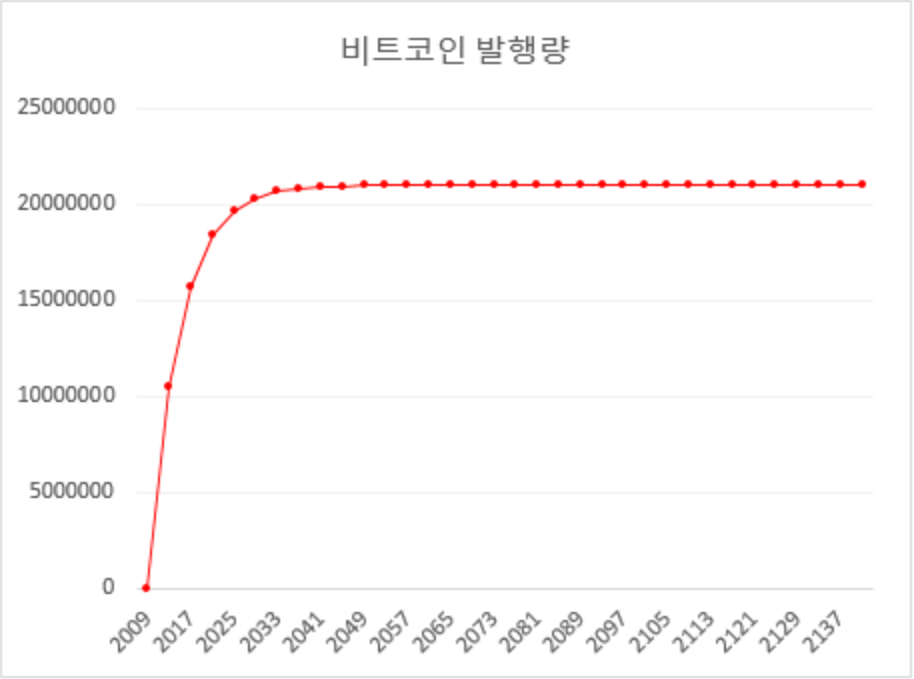
참고 : 미국 달러 발행량과 비트코인 발행량 비교

- 미국 달러는 1918년 이후 2008년 금융위기 이전까지 연 평균 약 5% ~ 6%의 증가율로 발행되어 왔음. 200년 금융위기 이후에는 3 차례에 걸친 양적완화 정책 (QE3)으로 많은 양의 달러가 발행되었음. → 인플레이션 화폐
- 비트코인은 2009년 50 BTC를 시작으로 2140년 까지 총 2,100만 BTC 까지만 발행될 것임. → 디플레이션 화폐
- 비트코인은 정부나 특정 주체에 의해 통제되는 화폐 시스팀에 반대하는 Cypherpunk 들의 철학이 담겨져 있음.
- 자본주의 경제에서 어떤 형태의 화폐 시스템이 적합할 지에 대해서는 의견이 분분한 상태임.

* 자료 출처 : https://fred.stlouisfed.org/series/AMBNS

달러 발행량		비트코인 발행량	
date	AMBNS	year	bitcoin
1970-01-01	62.909	2009	50
1970-02-01	61.920	2013	10,500,050
1970-03-01	61.870	2017	15,750,050
1970-04-01	62.662	2021	18,375,050
1970-05-01	63.168	2025	19,687,550
1970-06-01	63.546	2029	20,343,800
1970-07-01	64.337	2033	20,671,925
1970-08-01	64.498	2037	20,835,988
1970-09-01	64.969	2041	20,918,019
1970-10-01	65.330	2045	20,959,034
1970-11-01	65.940	2049	20,979,542
1970-12-01	67.420	2053	20,989,796
1971-01-01	67.398	2057	20,994,923
1971-02-01	66.714	2061	20,997,487
1971-03-01	66.950	2065	20,998,768
1971-04-01	67.585	2069	20,999,409
1971-05-01	68.342	2073	20,999,730
1971-06-01	68.779	2077	20,999,890





Transaction Fee와 최적 Block 크기

- Miner 들은 서로 속도 경쟁을 벌임 (Tx 수집/검증 속도, Hash 계산 속도 + 전파 속도). 블록의 크기를 작게 할수록 속도 경쟁에는 유리하지만, 거래 수수료를 많이 얻지 못하고, 반대로 블록의 크기를 크게 할수록 거래 수수료는 많이 얻을 수 있지만, 속도 경쟁에서 질 확률이 증가함 (Risk 비용이 커짐.)
- Miner들은 Risk 대비 수수료를 최대로 하려는 전략을 구사할 것임.
- User들은 높은 수수료를 지불하면 Waiting time이 작아지지만 수수료 비용이 증가하고, 반대로 낮은 수수료를 지불하면 Waiting time이 커져서 Miner가 선택하지 않을 Risk 비용이 증가함.
- User들도 Risk 대비 수수료를 최소화하려는 전략을 구사할 것임.
- Miner들 간의 경쟁이나, Miner와 User 간의 경쟁은 게임 이론으로 설명할 수 있으며, 경쟁의 결과로 모두가 최적인 전략을 선택하면, 시장은 Nash 균형을 이루게 됨.
- Miner는 MemPool에서 Tx를 선택할 때 바이트 당 수수료가 큰 Tx를 우선적으로 선택하고, 점차 수수료가 낮은 Tx들로 자신이 결정한 블록 크기 (Q)를 채울 것임.
- 우측은 Miner가 Tx를 선택하는 모습을 가정한 것임.
- Tx Size를 밑변으로 하고, 수수료를 높이로 하는 직각 삼각형을 생각하면 빗변의 기울기는 바이트 당 수수료임.
- Miner는 기울기가 큰 Tx를 우선적으로 선택하고 점차 기울기가 작은 Tx를 선택할 것임.
- x-축은 블록 크기이고 y-축은 누적 수수료임. 블록 크기가 증가할수록 누적 수수료는 증가하나, 기울기는 점차 감소할 것임 (체감적 증가).

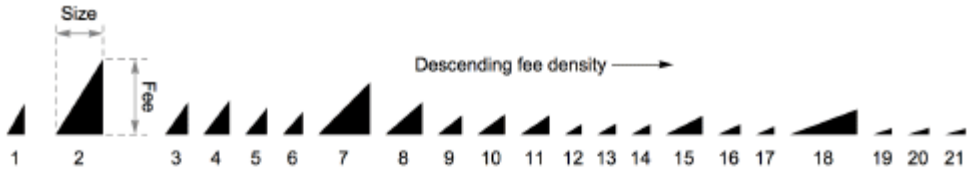


Fig. 2. A transaction can be visualized as a right triangle: its height represents its fee, its width represents its size in bytes, and its slope represents its fee density. To construct the mempool demand curve, we first sort the transactions in mempool in order of descending fee density.

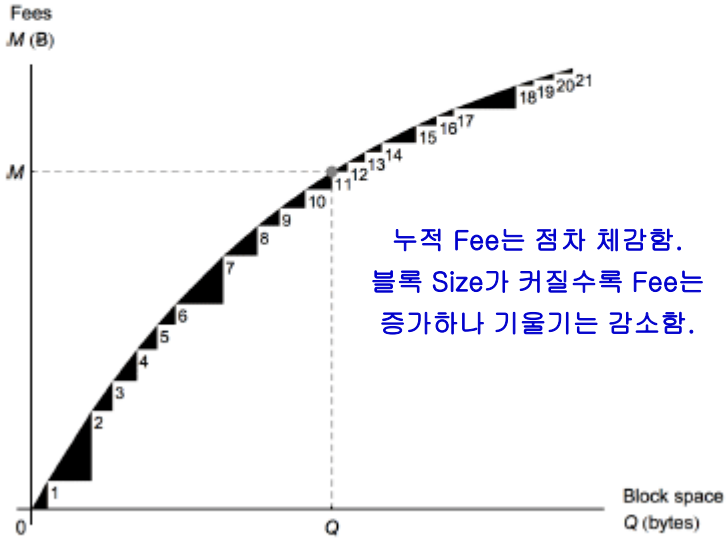
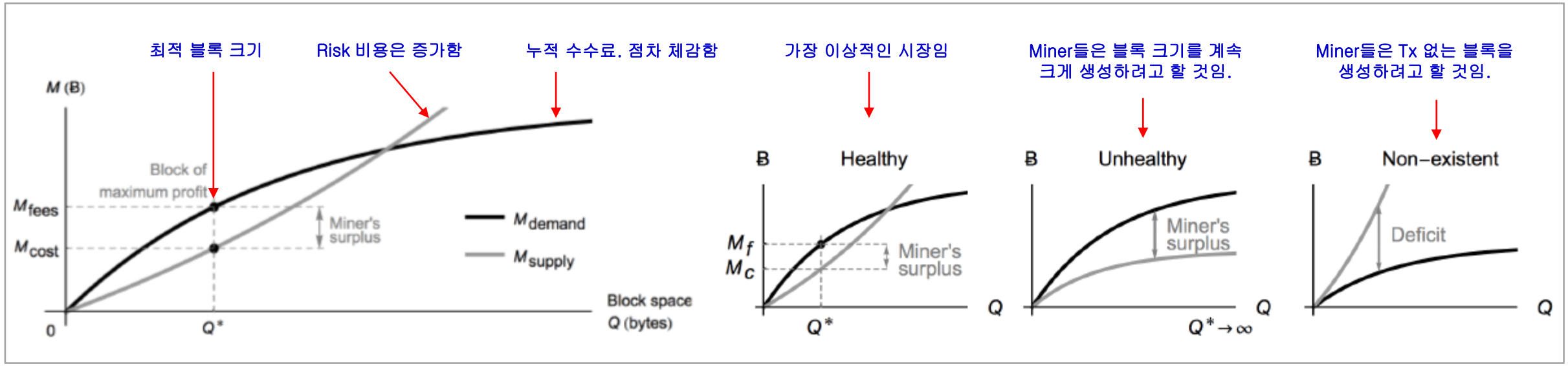


Fig. 3. The mempool demand curve describes the *maximum* fees a miner can claim from his mempool, M, as a function of the quantity of block space, Q, he might produce. To construct this curve, the triangles representing the sorted transactions in mempool are stacked vertex-to-vertex as shown.

* 출처 : Peter R. Rizun, 2015, A Transaction Fee Market Exists Without a Block Size Limit

Transaction Fee와 최적 Block 크기

- Miner가 블록 크기 (Q)를 크게 할수록 누적 수수료는 증가하지만 Risk 비용도 증가함.
- Miner는 누적 수수료 곡선과 Risk 비용 곡선의 차이가 최대가 되는 지점에서 최적 블록 크기 (Q^*)를 결정할 것임. 이 지점이 비용 대비 이득이 가장 큰 지점임.
- Risk 비용 곡선의 기울기가 일정하거나 (직선) 점차 증가하면 (지수적 증가) 최적 Q^* 가 존재하여 시장이 균형을 이룰 수 있지만 (Healthy), 비용 곡선의 기울기가 점차 감소하면 (체감적) Miner는 최적 블록 크기는 존재하지 않음 (발산함, Unhealthy). 만약, 비용 곡선이 누적 수수료 곡선보다 항상 크다면 Miner는 블록 크기 = 0, 즉, Tx는 없고 Coinbase Tx 만 있는 블록을 생성하려고 할 것임 (Non-existent).
- 최적 Q^* 가 존재하는 시장이라면, 블록의 크기를 제한하지 않아도 (현재는 1 MB 임) 됨. Nash 균형에 의해 시장에서 Q^* 가 자동으로 결정될 수 있음.
- Miner들과, Miner와 User의 경쟁 측면에서는 블록의 크기를 제한하지 않고 시장에 맡기는 것이 좋을 수도 있으나, 다른 측면 (Security 측면)에서는 블록의 크기를 제한하는 것이 안전하다고 함. 얼마로 제한하는 것이 좋은지는 계속 연구 과제임.



* 출처 : Peter R. Rizun, 2015, A Transaction Fee Market Exists Without a Block Size Limit

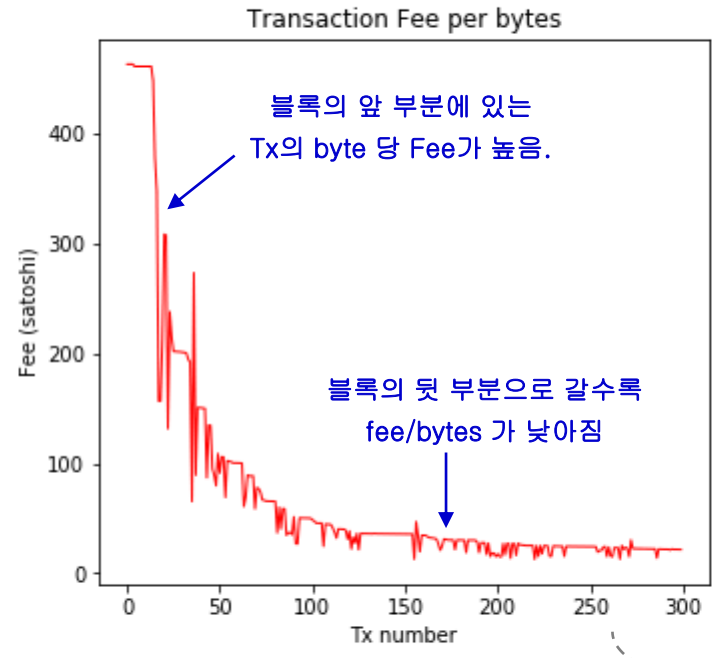
(실습 파일 : 5-8.FeePerBytes.py)

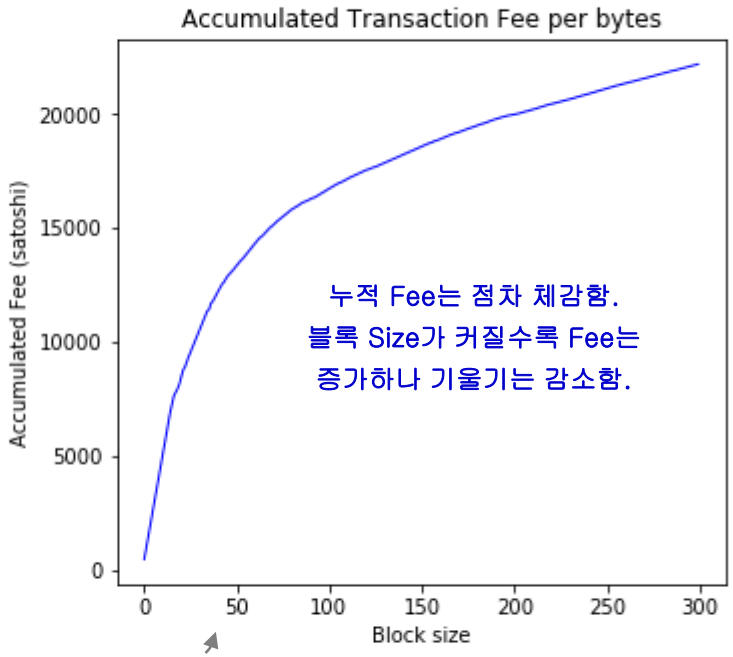
Miner의 Tx 선택 전략 확인 : Python 실습

- Miner 들은 MemPool에서 바이트 당 수수료 (Fee per bytes)가 높은 Tx을 우선적으로 선정하여 블록에 포함시키는 것을 확인할 수 있음.

```
# Miner가 Tx를 선택하는 기준을 확인한다.
# - Miner는 byte 당 fee가 높은 Tx를 우선적으로 블록에 포함
# - 한 블록 안에 있는 Tx를 순차적으로 읽어가면서 fee per b
# 참조 : https://blockchain.info
#
# 2018.5.11
# 아마추어 퀀트 (조성현)
# ------------------------------------------------------
import requests
import matplotlib.pyplot as plt

# 마지막 블록의 해시 값을 읽어온다
url = 'https://blockchain.info/latestblock?format=json'
resp = requests.get(url=url)
data = resp.json()

# 마지막 블록을 읽어온다
url = 'https://blockchain.info/block/' + data['hash'] +
resp = requests.get(url=url)
data = resp.json()

# Transaction (Tx) 부분을 분석한다
tx = data['tx']
nTx = len(tx)

# Tx를 순차적으로 읽어가면서 fee / bytes를 계산한다
txFee = []
byteFee = []
accByteFee = []
for i in range(1, nTx):
    txIn = tx[i]['inputs']
    txOut = tx[i]['out']
```

```
Block Height = 522539
Number of Transactions = 1105
Total Fee = 0.127536 BTC
```





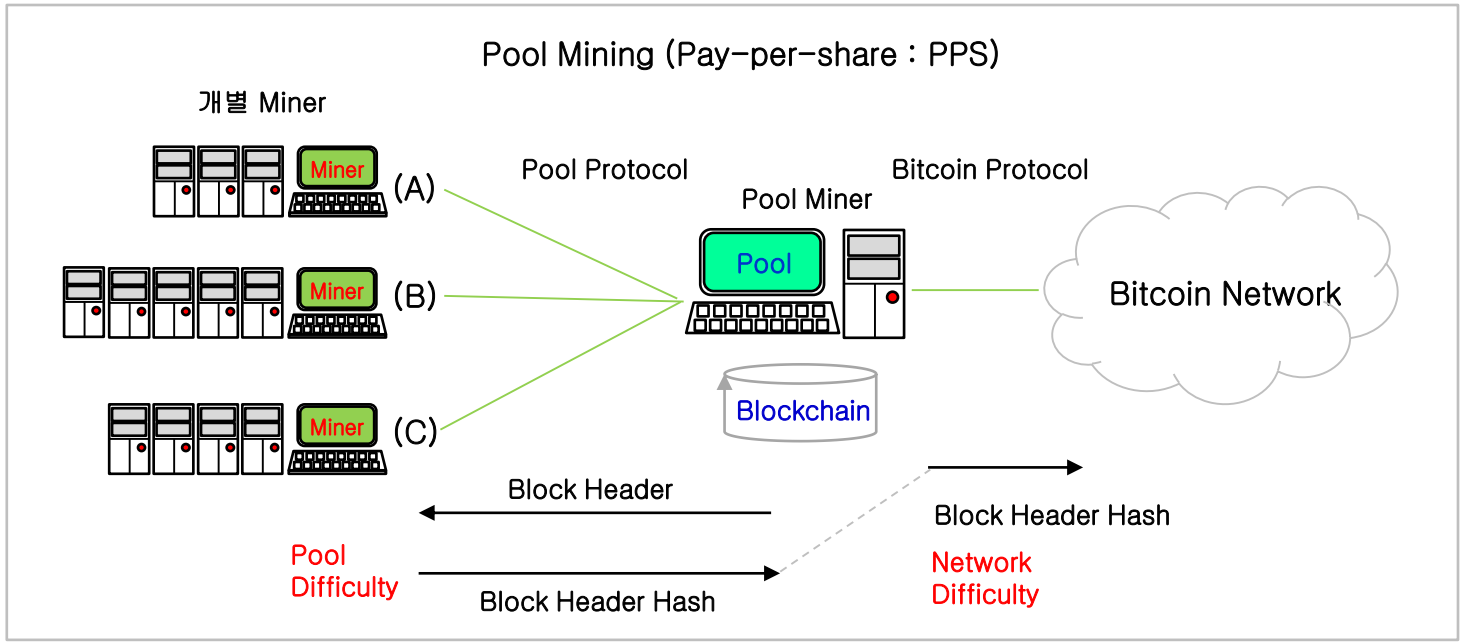

```
In [24]:
```

누적

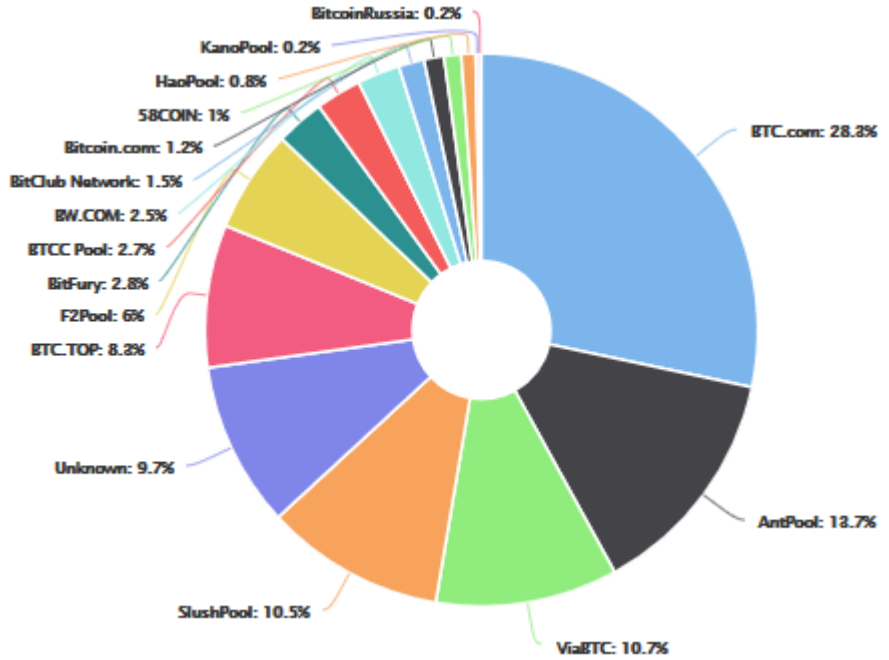
History log | IPython console

Solo Mining vs. Pool Mining

- Mining 경쟁이 극도로 심해지고 승자만이 독식하는 구조로 (winner-takes-all), 개별적으로 Mining 경쟁에 참여하는 것은 (Solo mining) 효율적이지 못함.
- Pool mining 방식은 pool miner가 여러 miner들을 모집하여 집단으로 mining을 수행하고 공헌도에 따라 보상을 나누는 방식임.
- 개별 miner들은 누구나 pool에 참여할 수 있음. Pool miner는 시스템 관리, 블록체인 관리, S/W 개발, 업그레이드, 유지보수, mining 서비스를 제공하고, 개별 miner들은 mining 장비 운영을 제공함 (시설비, 운용비, 전기료 발생). 공헌도에 따라 mining 보상을 share하는 시스템을 pay-per-share (PPS) 방식이라 함.
- Pool miner는 개별 miner들에게 block header와 비트코인 difficulty보다 쉬운 수준의 difficulty (Pool difficulty) 제공하고, 개별 miner 들은 pool difficulty에 맞는 Hash 값을 계산하여 지속적으로 pool miner에게 제공함. 장비를 많이 보유한 miner (B)는 pool difficulty 해답을 많이 제공하게 될 것임.
- Pool miner는 개별 miner들이 제공한 해답들 중 network difficulty를 통과하는 해답이 있으면 비트코인 네트워크로 전송함.
- Pool miner가 mining에 성공하면 개별 miner 들에게 공헌도에 따라 보상을 배분함. (A)가 pool difficulty를 통과하는 해답을 2,000 번, (B)는 5,000번, (C)는 3,000 번 제시했다면, (A)에게는 20%, (B)에게는 50%, (C)에게는 30%의 보상을 지급함.

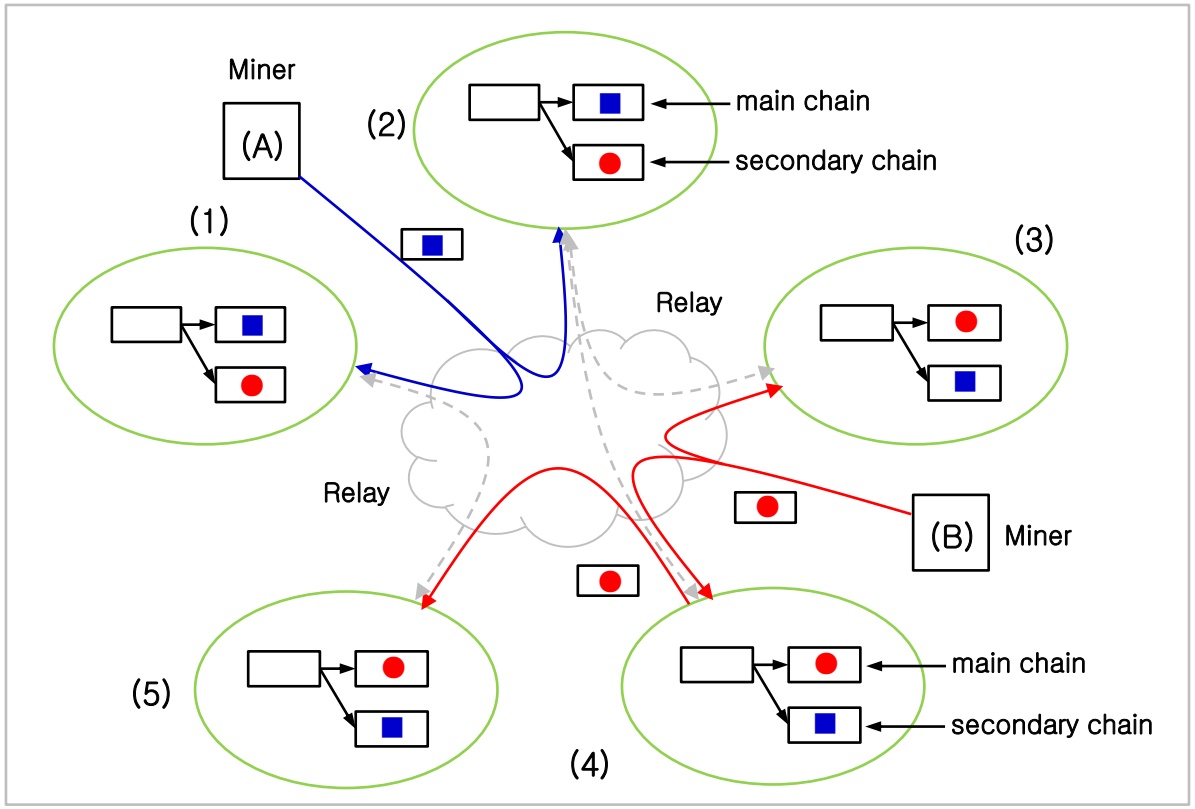


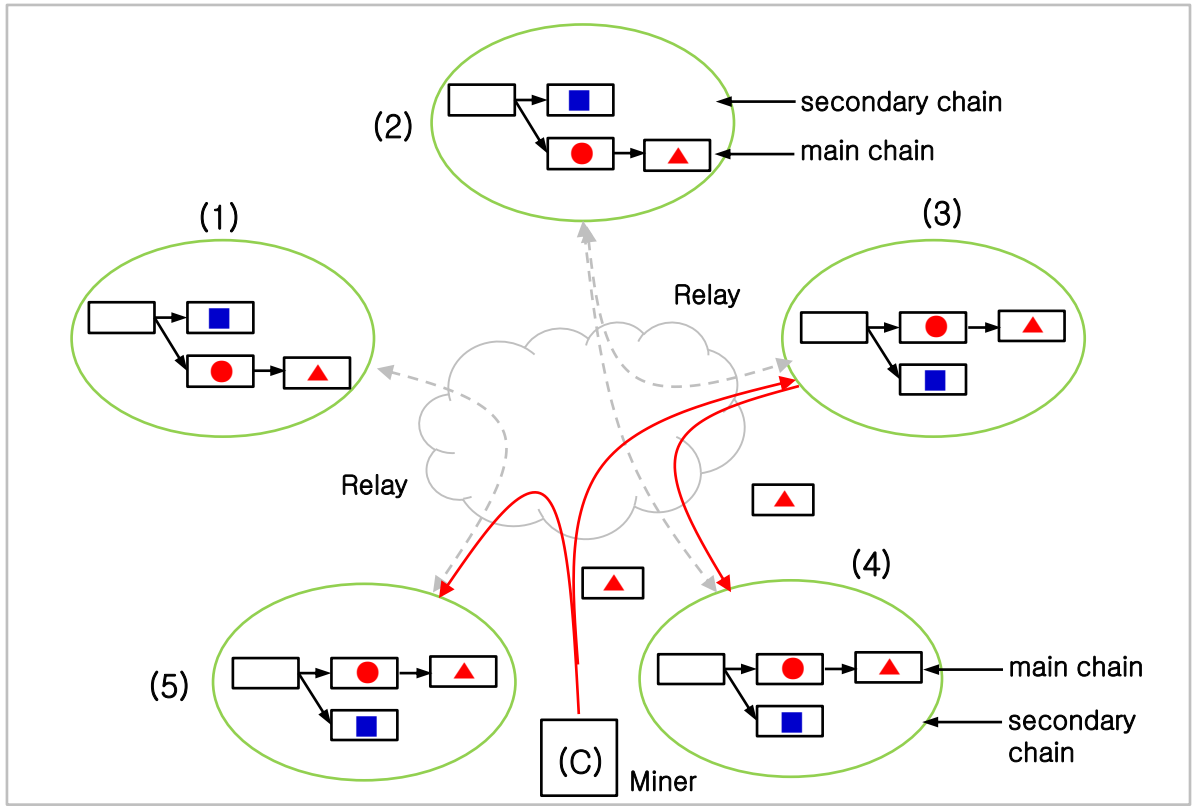
Pool Miner 비중 (blockchain.info/pools 참조)



블록체인의 일시적 불일치 : 블록체인 Fork

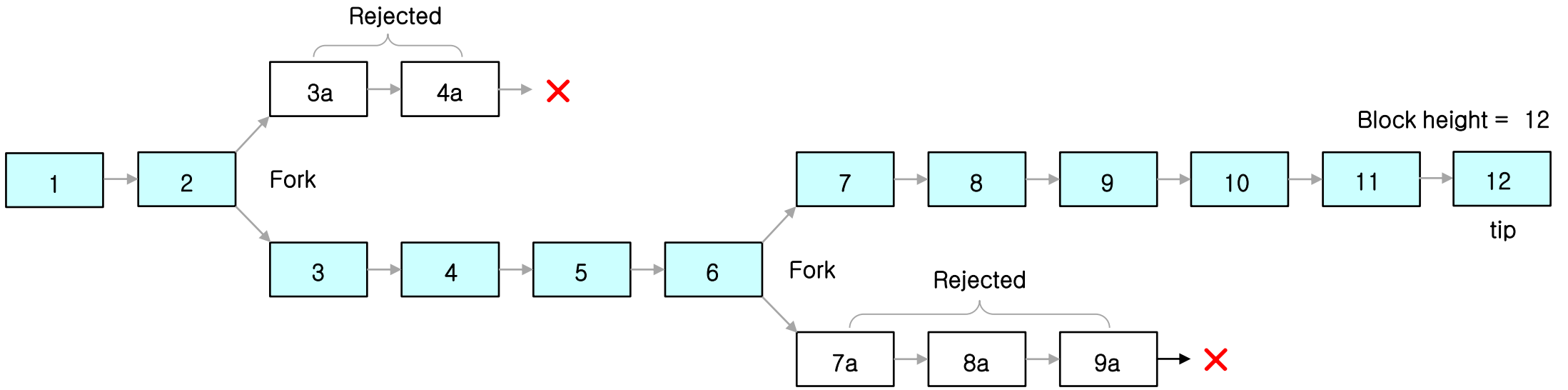
- 여러 Miner들이 거의 동시에 블록을 생성하면 블록체인이 일시적으로 분리될 수 있음 (main chain과 secondary chain). 또한, 연속된 두 블록이 짧은 시간 간격으로 생성되고, 어떤 노드가 나중 블록을 먼저 받으면 이 블록은 parent가 없는 고아 블록 (Orphan block) 될 수도 있음 (임시 저장 후 parent를 받으면 처리함).
- 일시적으로 분리된 체인은 나중에 생성될 블록들이 parent로 선택하는 chain이 살아 남게 됨 (긴 체인이 살아 남고, 짧은 체인은 사라짐).
- 아래 좌측 예시는 miner (A)와 (B)가 블록 생성하였고, 이 블록을 받은 인근 노드들의 블록체인이 분리되었음. 노드 (1), (2)와 (3), (4), (5)의 main chain이 다름.
- 아래 우측 예시에서 miner (C)는 동그란 블록을 previous hash (parent)로 선택하였으므로, 모든 노드들이 이 체인을 main chain으로 결정하게 됨.
- 이런 과정은 miner들이 다수결로 결정하는 방식이므로 매우 합리적이라 할 수 있음.





Block Height, Depth, Confirmation

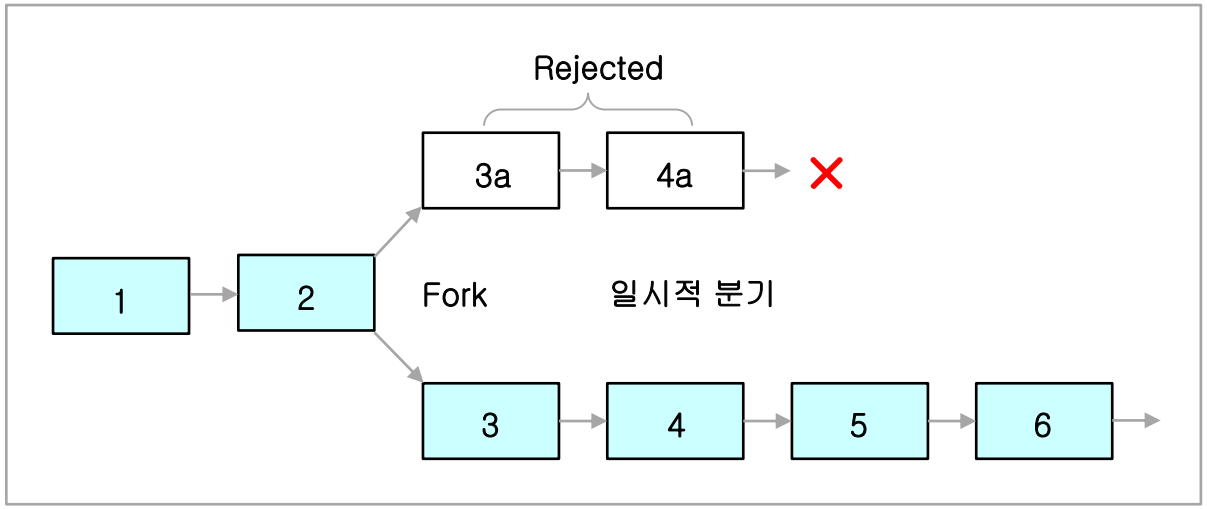
- 블록체인은 Decentralization 특성으로 자율적으로 운영되기 때문에 일시적인 분기 (Fork)가 발생하다가 체인들 간의 경쟁으로 다시 균형으로 돌아옴.
- Miner들은 안전한 보상을 위해 분기된 체인 중에 가장 긴 체인을 선택해서 그 뒤에 블록을 생성함. Previous header hash (parent)를 선택하는 것임.
- Miner들이 더 이상 선택하지 않는 체인은 자연히 사장되고, 사장된 블록들은 모두 reject 처리됨. 그 안에 있던 모든 Tx 들도 블록체인에 기록되지 못함.
- 블록체인의 마지막 블록을 block tip이라 하고, 이 블록 번호를 block height라 함. 아래 예시에서 block height는 12 임.
- 현재 블록 tip이 최종 확정된 블록은 아님. 블록 번호 12 이전에 분기가 있고 이 체인보다 더 긴 체인이 발생한다면 현재 보이는 블록 12 도 reject될 수 있음.
- 블록 12가 최종 확정되려면 이 뒤에 일정량의 블록이 추가되어야 함. 약 6 블록 정도가 추가되면 확정되었다고 보고 있음.
- 이런 이유 때문에 miner의 coinbase 거래는 100 블록이 지나야 소비할 수 있음.
- 일반 사용자 지갑의 거래도 해당 거래가 기록된 블록 이후에 6개 블록이 쌓여야 거래가 최종 인증되었다고 판단함.
- 사용자 지갑의 거래가 블록 9에 기록되어 있고, 현재 main chain의 tip이 12라면, 해당 거래 뒤에 3개의 블록이 추가된 상태임. 추가된 블록 수를 block depth라 함.
- Block depth를 confirmations 라고도 함. 특정 거래의 confirmation이 6보다 커야 해당 거래가 최종 인증되었다고 판단함.
- SPV 노드 (혹은 지갑)의 경우는 자신의 거래가 특정 블록에 정상 기록되었는지만 확인이 가능하며 block depth로 거래가 최종 인증되었는지 확인함.



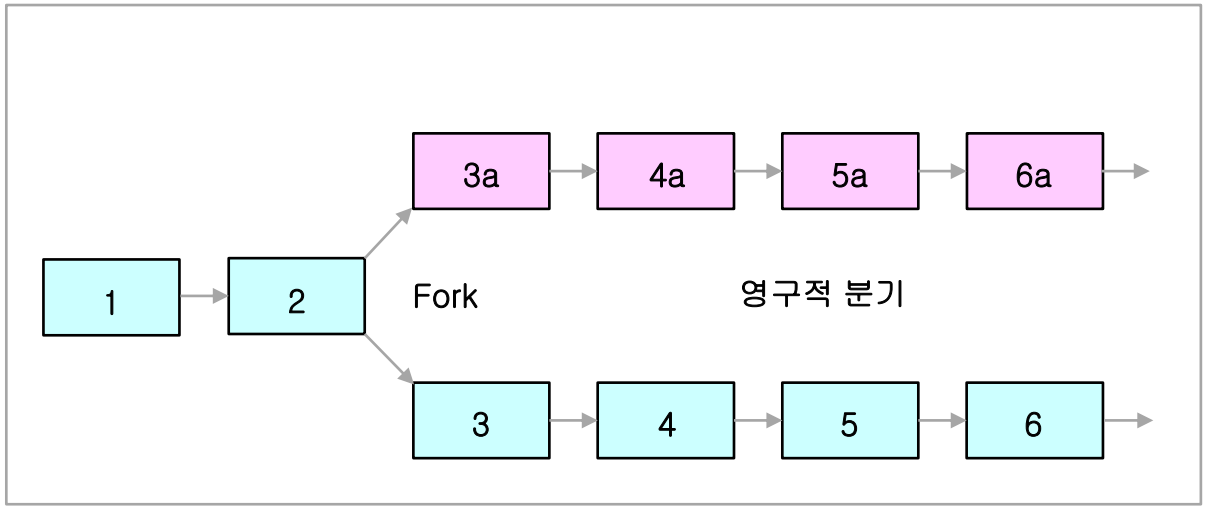
Hard Fork 와 Soft Fork

- 블록체인은 Miner들에 의해 일시적으로 분기가 발행할 수도 있지만, 개발자 들에 의해 S/W가 업그레이드될 때도 발생할 수 있음.
- 비트코인은 통제 시스팀 없이 각 노드들이 합의 규칙에 따라 자율적으로 운영되기 때문에, 합의 규칙을 변경하는 S/W 업그레이드는 대단히 어려운 일임.
- 다수의 노드들이 합의 규칙 변경에 동의해야 가능한 일임. 일부 노드들이 반대하고, 일부 Miner들이 규칙이 변경되기 이전 상태로 블록을 생성하면 블록체인에 분기가 발생함. 블록체인 분기가 영구적으로 유지되면 Hard Fork라 하고, 일시적으로 분기가 발생하다가 다시 한 체인으로 유지되면 Soft Fork라 함.
- 대부분 개발자들은 Hard Fork를 원하지 않음. 그 이유는 Hard Fork가 발생하면 비트코인 네트워크 자체가 분리되는 것이므로 Risk가 매우 크기 때문임.
- 개발자들은 Soft Fork 형태로 S/W를 업그레이드하기 위해, 이전 버전에서도 새로운 기능이 호환되도록 (Backward Compatibility) 하여 블록체인이 분기하지 않도록, 매우 다양한 노력을 기울이고 있음 (ex : BIP-9, BIP 문서에 Backward compatibility 방안 제시 등).
- Miner들이 분리되거나, 개발팀도 분리될 수 있으며, 이런 경우 Hard Fork가 발생하고, 비트코인 네트워크 자체가 분리됨.
- Miner들은 BIP-9 권고에 의해 업그레이드된 블록을 생성할 준비가 되었다는 의사를 표시하고, Miner들의 95%가 동의하면 업그레이드가 Activated 됨.
- 그 동안 여러 차례 Soft Fork가 성공적으로 진행 되어왔음 (ex : SegWit).

Soft Fork

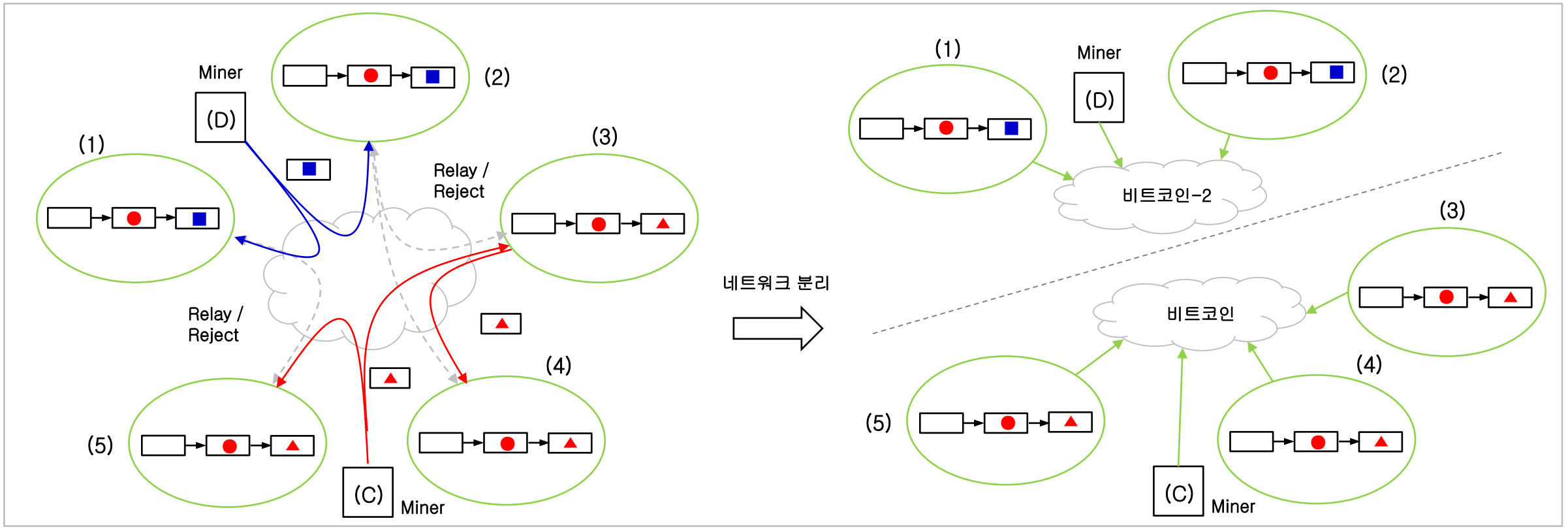


Hard Fork



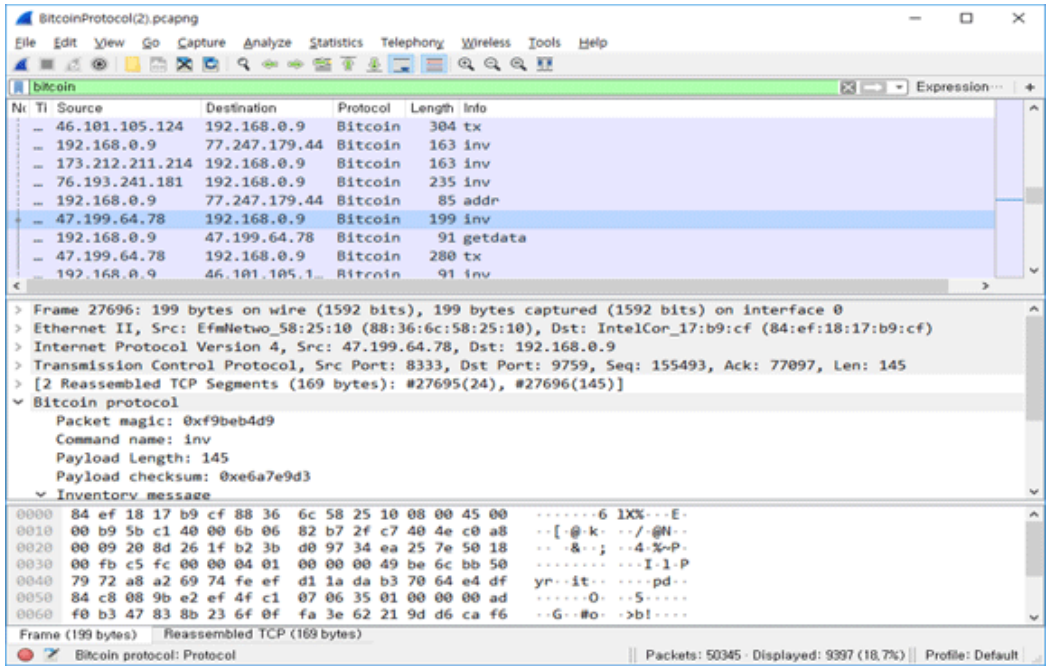
Hard Fork 와 네트워크 분리

- Hard Fork가 발생하면 아래와 같이 네트워크가 분리됨. 서로 호환이 되지 않으므로 공존할 수 없고, 완전히 다른 네트워크가 됨.
- 대표적인 사례로 2017년 Segregated Witness (SegWit) 기능이 Soft Fork로 진행되었을 때 일부 Miner 들이 (ex : Bitmain) 동의하지 않아 블록체인이 분리되었고, 네트워크가 분리되었음. 하나는 Soft Fork에 동의한 기존의 비트코인 네트워크이고, 다른 하나는 Hard Fork 방식의 비트코인 캐시 (Cash) 임.
- 네트워크가 분리되면 블록체인이 분기되기 이전 블록에 기록된 모든 거래 내역은 양 쪽 네트워크에서 모두 유효함. 양쪽에서 사용할 수 있는 UTXO 임.



6. 비트코인 P2P 프로토콜

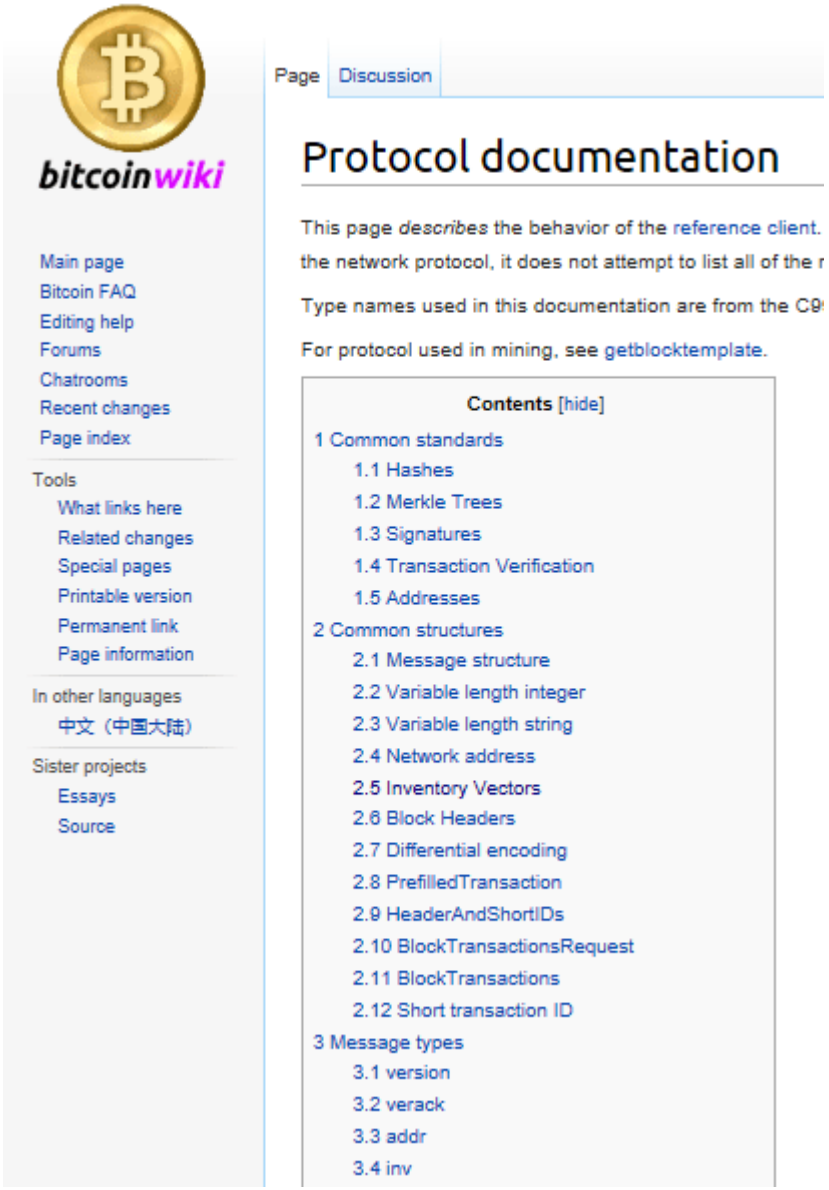
6-1. 비트코인 프로토콜 개요
6-2. Packet Analyzer : Wireshark
6-3. Version, VerAck 메시지 교환
6-4. Getaddr, Addr 메시지 교환
6-5. Ping, Pong 메시지 교환
6-6. 블록 데이터 동기화 (Block-first, Header-first 방식)
6-7. 신규 블록 데이터 Relay (Compact Block Relay)
6-8 거래 (Transaction) 메시지 Relay
6-9. Reject 메시지 (BIP-61)
6-10. SPV (Simplified Payment Verification) 프로토콜
6-11. Bloom Filter (BIP-37)
6-12. Bloom Filter와 Merkle Path 검증
6-13. 기타 메시지 (Feefilter, mempool, notfound)
6-14. Penalty 부여 및 노드 차단



비트코인 프로토콜 개요

- 비트코인 네트워크의 노드들은 TCP 소켓을 통해 서로 통신함 (TCP 포트 번호 : mainnet = 8333, testnet = 18333).
- 프로토콜의 세부 내용은 https://en.bitcoin.it/wiki/Protocol_documentation (우측 화면)을 참조함.
- Bitcoin core의 경우 처음 비트코인 네트워크에 참여하면 아래와 같은 절차를 거침.

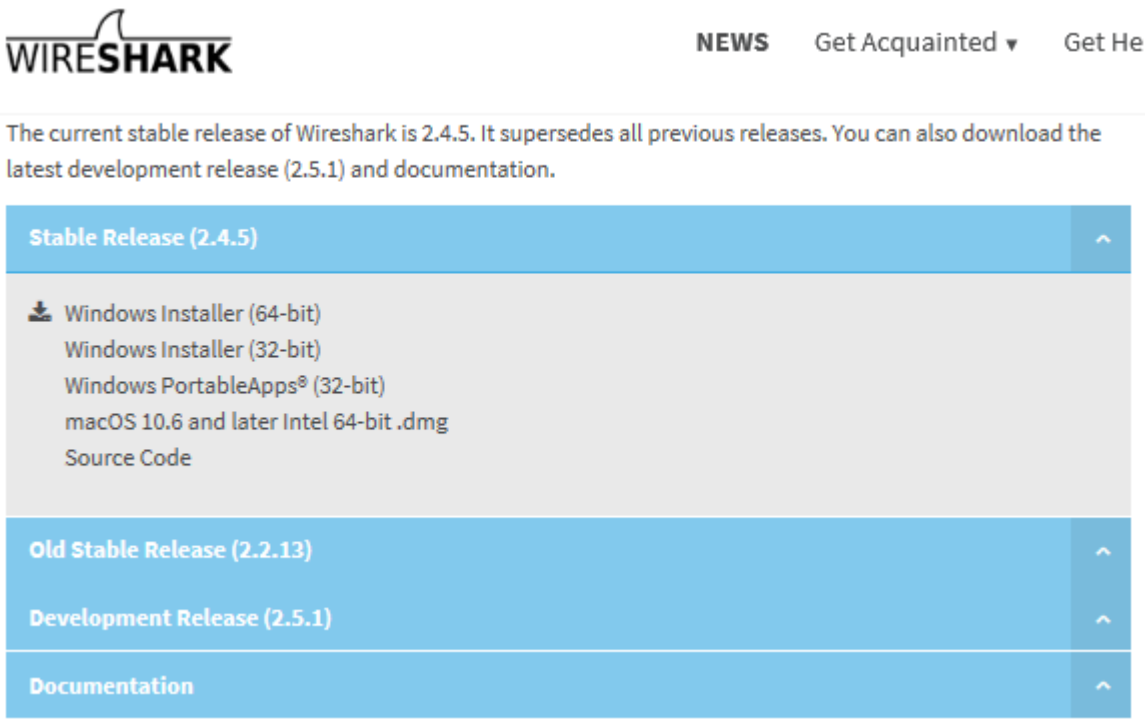
1) 자체 내장된 DNS seed를 조화하여 다른 노드들의 네트워크 주소를 알아냄.
2) 알아낸 주소로 Version 메시지를 보냄. Version 메시지로 자신이 사용하는 프로토콜 버전 정보, 블록체인의 블록수 등을 보내면, 상대방도 Version 메시지를 보내 응답함.
3) 다른 노드들의 주소를 알아보기 위해 이미 알고 있는 노드로 getaddr 메시지를 보내면, 상대방은 addr 메시지로 자신이 알고 있는 노드들의 주소를 알려줌.
4) 알아낸 주소의 노드들과 Version 메시지를 교환하고 서로 통신함. 노드들끼리는 주기적으로 ping/pong 메시지를 주고 받으면서 서로 통신이 가능한 상태인지 확인함.
5) 다른 노드들과 블록체인 데이터를 동기화함. Getheaders, headers, getdata, block 메시지를 주고 받으면서 부족한 블록 데이터를 다운로드 함 (Initial Block Download : IBD).
6) 블록체인 데이터가 동기화된 후에는 Compact Block Relay 방식으로 새로 발생하는 블록 데이터를 주고 받음.
7) 각 노드에서 보내오는 Transaction 내역을 검증하고 다른 노드로 Relay함. Inv, getdata, tx 메시지를 주고 받으면서 Relay함.



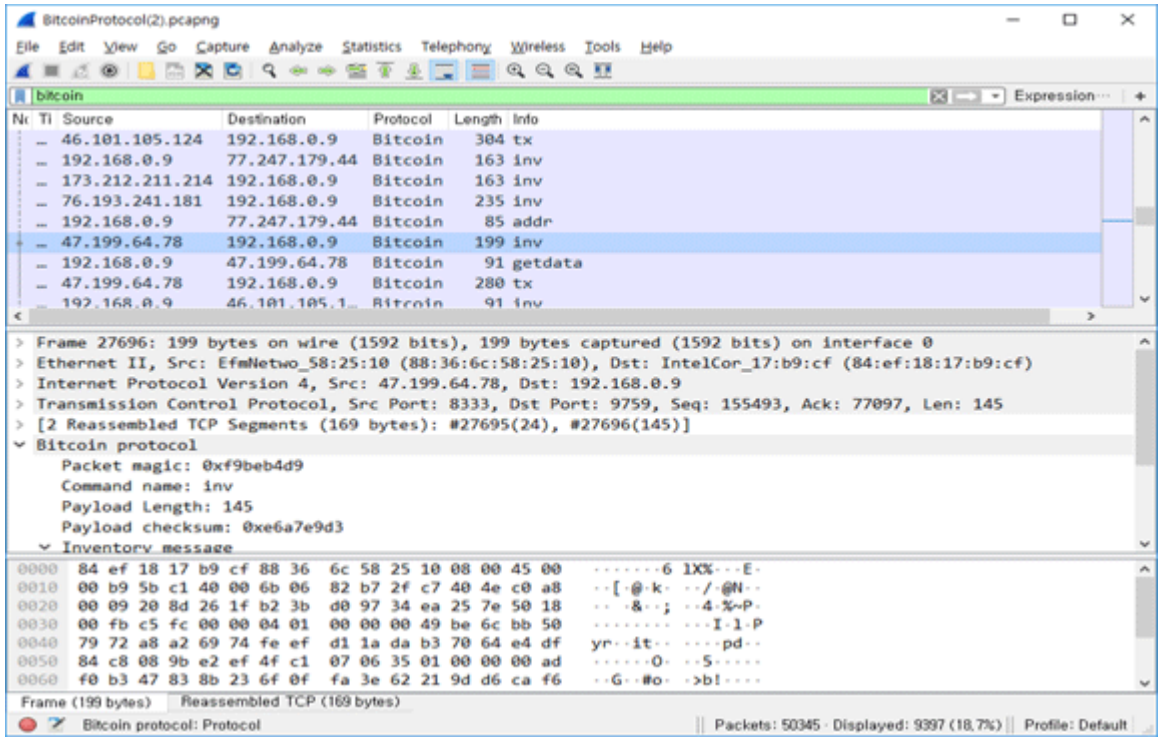
Packet Analyzer : Wireshark

- 오픈 소스인 Wireshark를 사용하면 비트코인 네트워크 상에서 전달되는 TCP/IP 패킷을 관찰, 분석할 수 있음.
- Wireshark의 최신 버전은 Bitcoin protocol을 지원하지만, BIP (Bitcoin Improvement Proposal)에 의해 새로 생성되는 Bitcoin protocol들을 완전히 지원하는 것은 아님. Wireshark는 아래 사이트에서 다운로드 받을 수 있고, 실행 화면에서 Bitcoin으로 filtering (녹색 부분)하면 Bitcoin 관련 메시지들을 분석할 수 있음.
- 참고 : https://www.wireshark.org/docs/dfref/b/bitcoin.html ← Wireshark의 비트코인 메시지

- 다운로드 : https://www.wireshark.org/download.html



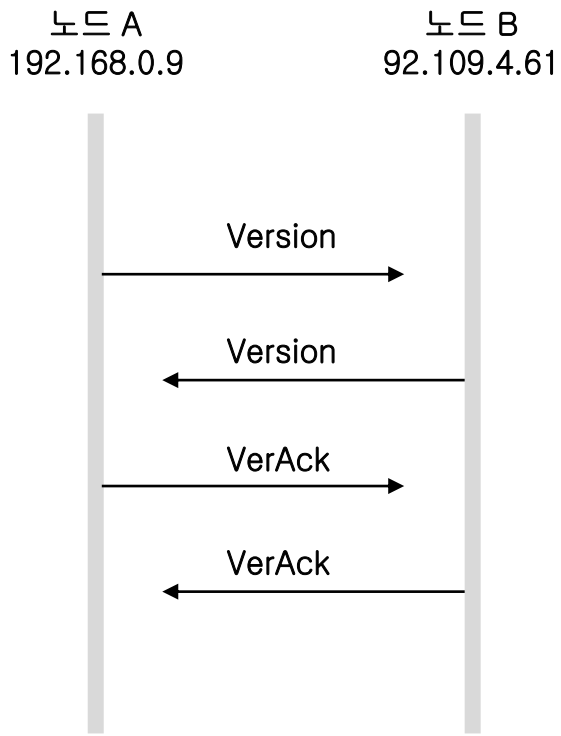
- Wireshark 실행 화면

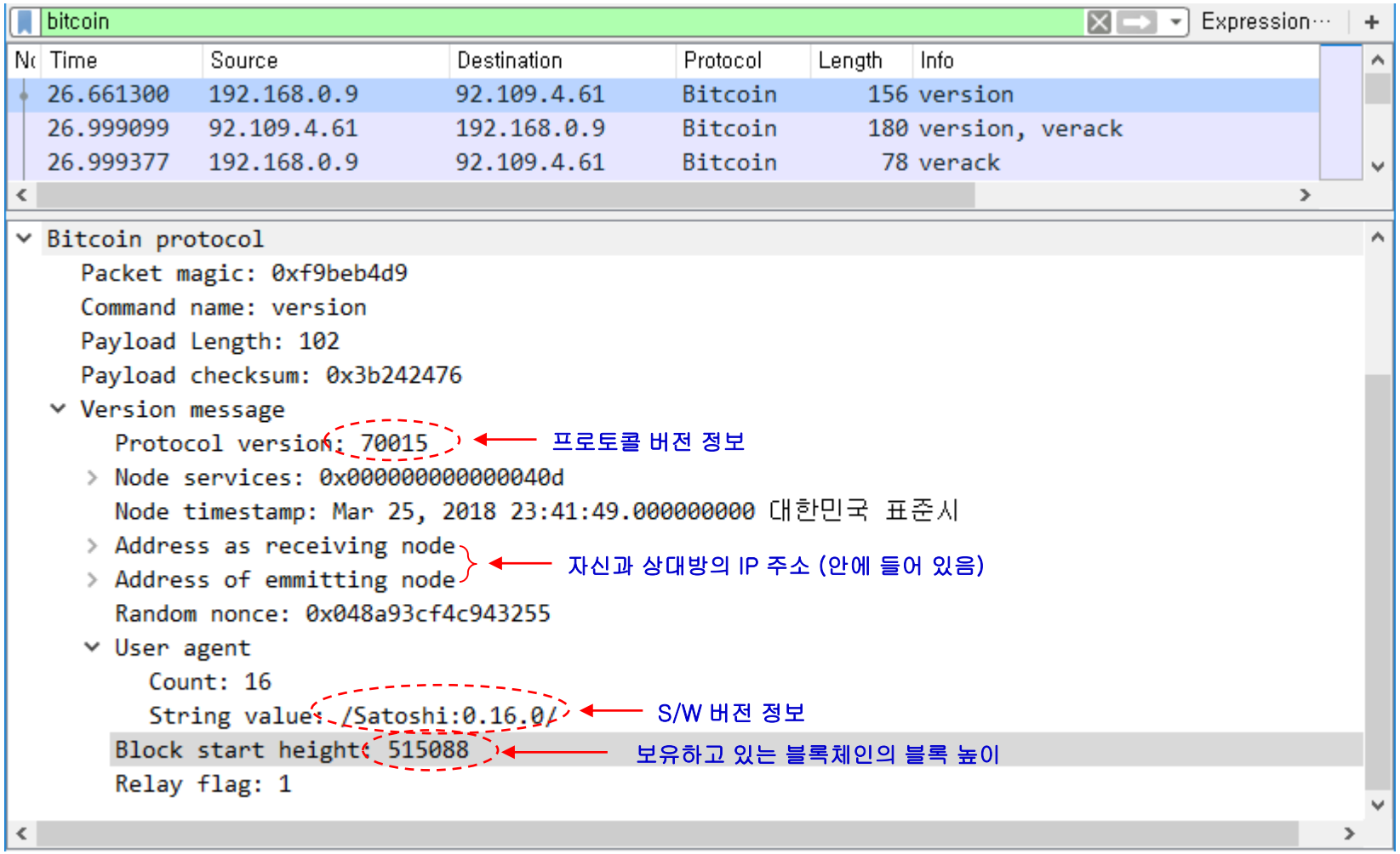


(실습 파일 : 6-1.BitcoinProtocol(1).pcapng)

Version, VerAck 메시지 교환

- 상대방과 프로토콜 버전, S/W 버전, IP 주소, 블록 높이 등의 정보를 교환함. Version 메시지를 보내면 상대방도 Version 메시지를 보내고 서로 확인했다는 의미로 VerAck 메시지를 보냄. 버전에 따라 교환할 메시지가 다르기 때문에 상대측의 버전 정보를 알아야 함 (호환성). 상대측이 구 버전을 사용하고 있으면 새로운 기능에 대한 메시지를 주고 받을 수 없음.
  (ex : SegWit 등)

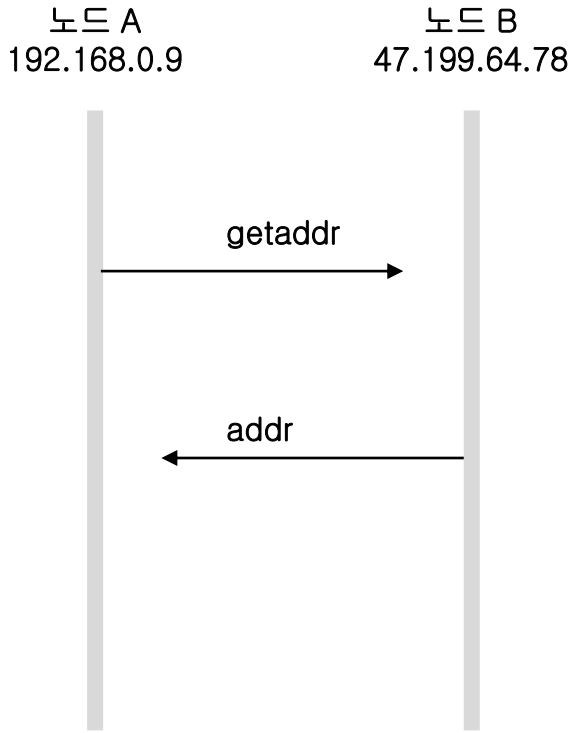




(실습 파일 : 6-1.BitcoinPrototol(1).pcapng)

Getaddr, Addr 메시지 교환

- 노드 A는 getaddr 메시지를 통해 노드 B에게 B가 알고 있는 다른 노드들의 주소를 요청함.
- 노드 B는 addr 메시지를 통해 자신이 알고 있는 다른 노드들의 주소를 A에게 알려 줌.



bitcoin and ip.addr==47.199.64.78 Expression…

No.	Time	Source	Destination	Protocol	Length	Info
534	42.827368	192.168.0.9	47.199.64.78	Bitcoin	78	getaddr
10852	53.968947	47.199.64.78	192.168.0.9	Bitcoin	857	addr

```
Bitcoin protocol
    Packet magic: 0xf9beb4d9
    Command name: addr
    Payload Length: 30003
    Payload checksum: 0xa372ef47
  ∨ Address message
      Count: 1000
    > Address: bb5aa85a0d0000000000000000000000000000000000ffff...
    > Address: cee1b55a0d040000000000002a0023c4ff871d00c8069ad6...
    > Address: 5fe0b55a0d0000000000000000000000000000000000ffff...
    > Address: 31b7a45a0d0000000000000000000000000000000000ffff...
    > Address: e3ffb55a0d040000000000000200100009d386ab834872f81...
    ∨ Address: 72fbb35a0d0000000000000000000000000000000000ffff...
      > Node services: 0x000000000000000d
        Node address: ::ffff:66.113.74.130
        Node port: 8333
        Address timestamp: Mar 23, 2018 03:52:34.000000000 대한민국 표준시
    ∨ Address: b314b55a010000000000000000000000000000000000ffff...
```

노드 B는 자신이 알고 있는 주소 1,000개를 보냄,

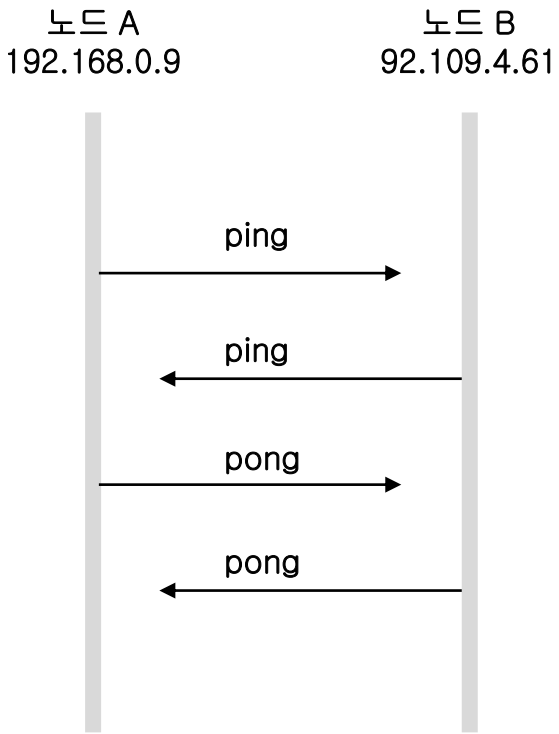
1,000개 주소 목록

1,000개 주소 중 하나

(실습 파일 : 6-1.BitcoinPrototol(1).pcapng)

Ping, Pong 메시지 교환

- 각 노드들은 주기적으로 ping, pong 메시지를 보내 자신이 네트워크 상에 연결되어 있음을 알려 줌.
- Ping 메시지는 랜덤하게 부여된 nonce 값을 보내고, pong 메시지는 상대방이 보낸 nonce 값을 되돌려 줌. Nonce 값으로 내가 보낸 Ping 메시지에 대한 응답임을 확인할 수 있음.



bitcoin

Expression…

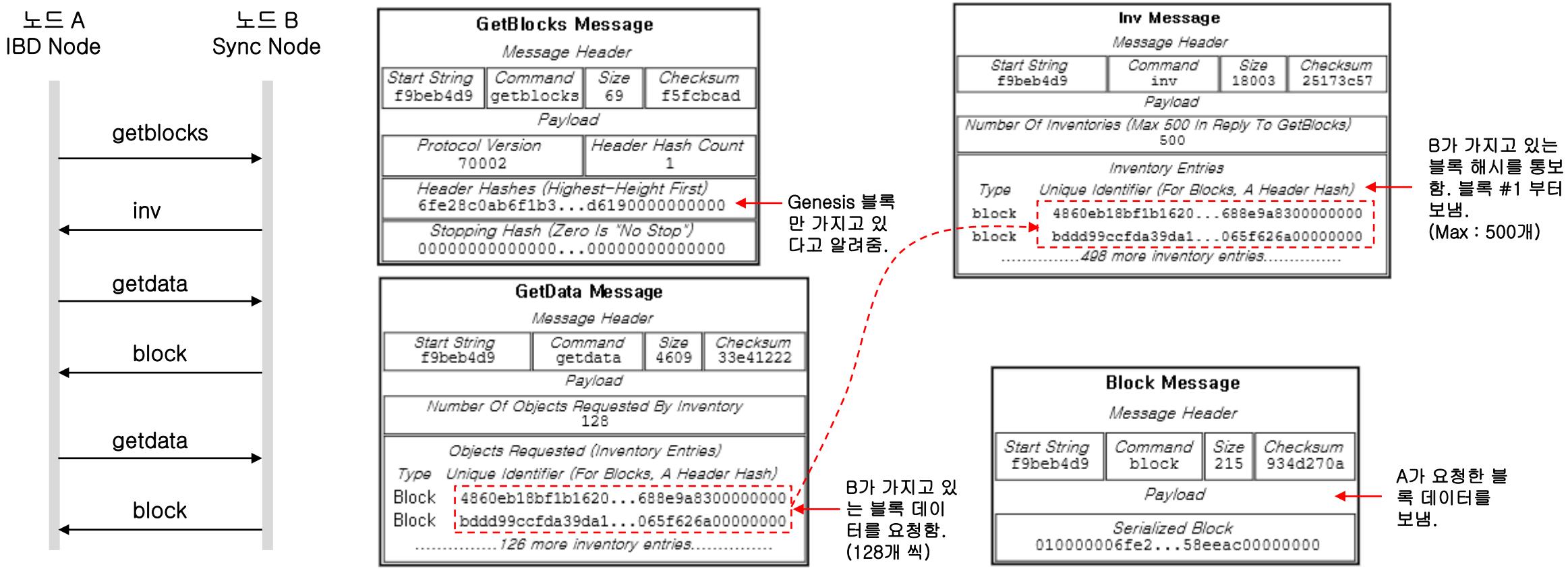
No.	Time	Source	Destination	Protocol	Length	Info
80	27.019422	192.168.0.9	92.109.4.61	Bitcoin	62	ping
82	27.332318	92.109.4.61	192.168.0.9	Bitcoin	78	[unknown command]
83	27.332506	92.109.4.61	192.168.0.9	Bitcoin	120	[unknown command]
84	27.332507	92.109.4.61	192.168.0.9	Bitcoin	86	ping
85	27.332508	92.109.4.61	192.168.0.9	Bitcoin	109	addr
88	27.332806	92.109.4.61	192.168.0.9	Bitcoin	1083	getheaders, [unkr
93	27.358265	92.109.4.61	192.168.0.9	Bitcoin	62	pong
99	27.416011	192.168.0.9	92.109.4.61	Bitcoin	62	pong

```
> Frame 93: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface 0
> Ethernet II, Src: EfmNetwo_58:25:10 (88:36:6c:58:25:10), Dst: IntelCor_17:b9:cf (84:ef:18:17:b9:cf)
> Internet Protocol Version 4, Src: 92.109.4.61, Dst: 192.168.0.9
> Transmission Control Protocol, Src Port: 8333, Dst Port: 5132, Seq: 1405, Ack: 297, Len: 8
> [2 Reassembled TCP Segments (32 bytes): #92(24), #93(8)]
v Bitcoin protocol
    Packet magic: 0xf9beb4d9
    Command name: pong
    Payload Length: 8
    Payload checksum: 0x38952598
  v Pong message
      Random nonce: 0xf97c684d21ff4a49
```

← 상대방이 보낸 nonce 값으로 응답함.

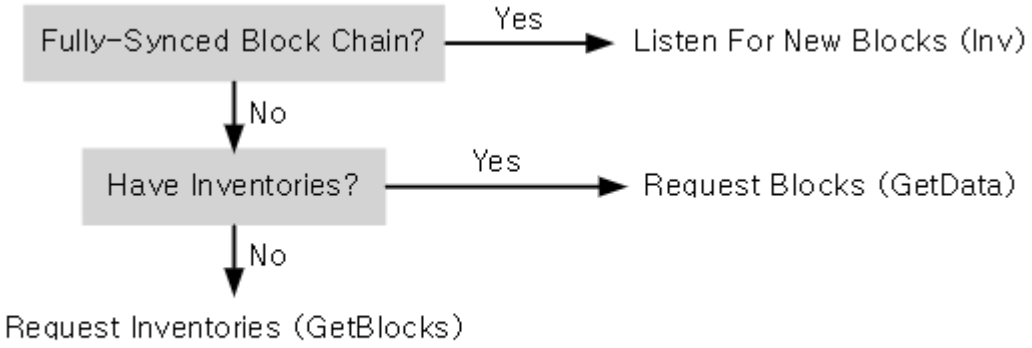
블록 데이터 동기화 (Getblocks, inv, getdata, block 메시지) : Block-first 방식 – Bitcoin core 0.9.0 이전에 사용한 방식

- Full Node는 새로 발생하는 Transaction이나 Miner가 생성한 새로운 블록을 검증하기 전에 다른 노드들과 블록체인 데이터를 동기화해야 함.
- 초기 데이터 동기화 과정을 Initial Block Download (IBD or initial sync)라 함. IBD 방식은 Block-first 방식 (이전 버전)과 Header-first 방식 (현재 버전)이 있음.
- 아래 과정은 Block-first 방식으로 IBD Node가 Genesis 블록만 가지고 있을 때, Sync Node로부터 전체 블록을 다운로드 받는 예시임.
- 자료 출처 : Bitcoin Developer Guide (https://bitcoin.org/en/developer-guide#initial-block-download)

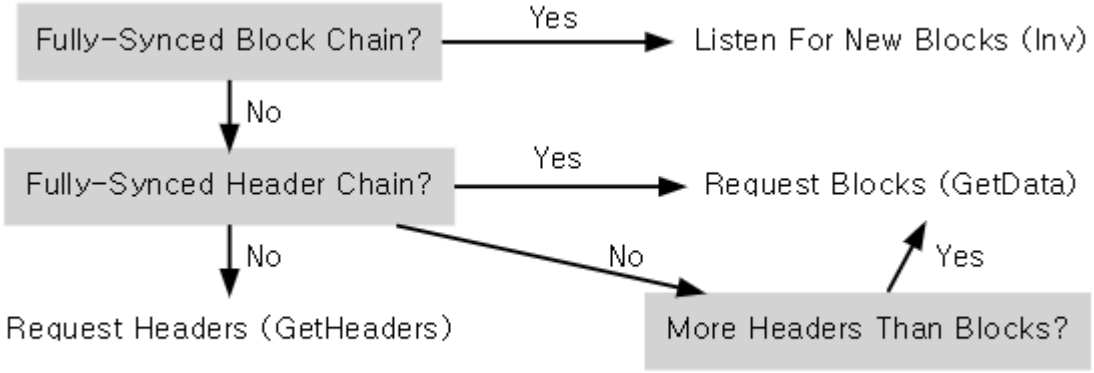


블록 데이터 동기화 : Block-first 방식과 Header-first 방식

- Block-first 방식은 단순하다는 장점이 있지만 전적으로 Sync 노드에 의존하기 때문에 Sync 노드의 전송 속도가 늦다면 다운로드 속도가 늦다는 단점이 있음.
- Sync 노드가 블록체인 데이터 전체를 가지고 있지 않거나, 다운로드 중 연결이 끊어지면, IBD 노드는 다른 Sync 노드를 찾아서 재 요청해야 하는 단점이 있음.
- 또한, Sync 노드가 부적절한 블록 데이터를 전송한다면 이를 곧 바로 검증하기 어렵기 때문에 나중에 다른 Sync 노드로부터 다시 다운로드 받아야할 수도 있음. 예를 들어 Sync 노드가 악의적 목적이거나 실수로 블록 데이터를 순차적으로 보내지 않고 중간 블록을 빼거나 순서를 바꿔서 보내면 IBD 노드에서는 Parent가 없는 블록 (고아 블록 : Orphan block)들을 블록체인에 등록할 수 없고 임시 저장 장소에 보관하게 되어 디스크나 메모리 점유 공격을 받을 수도 있음.
- 이러한 단점을 보완하기 위해 Bitcoin core 버전 0.10.0 부터는 Header-first 방식을 사용함.
- 자료 출처 : Bitcoin Developer Guide (https://bitcoin.org/en/developer-guide#initial-block-download)



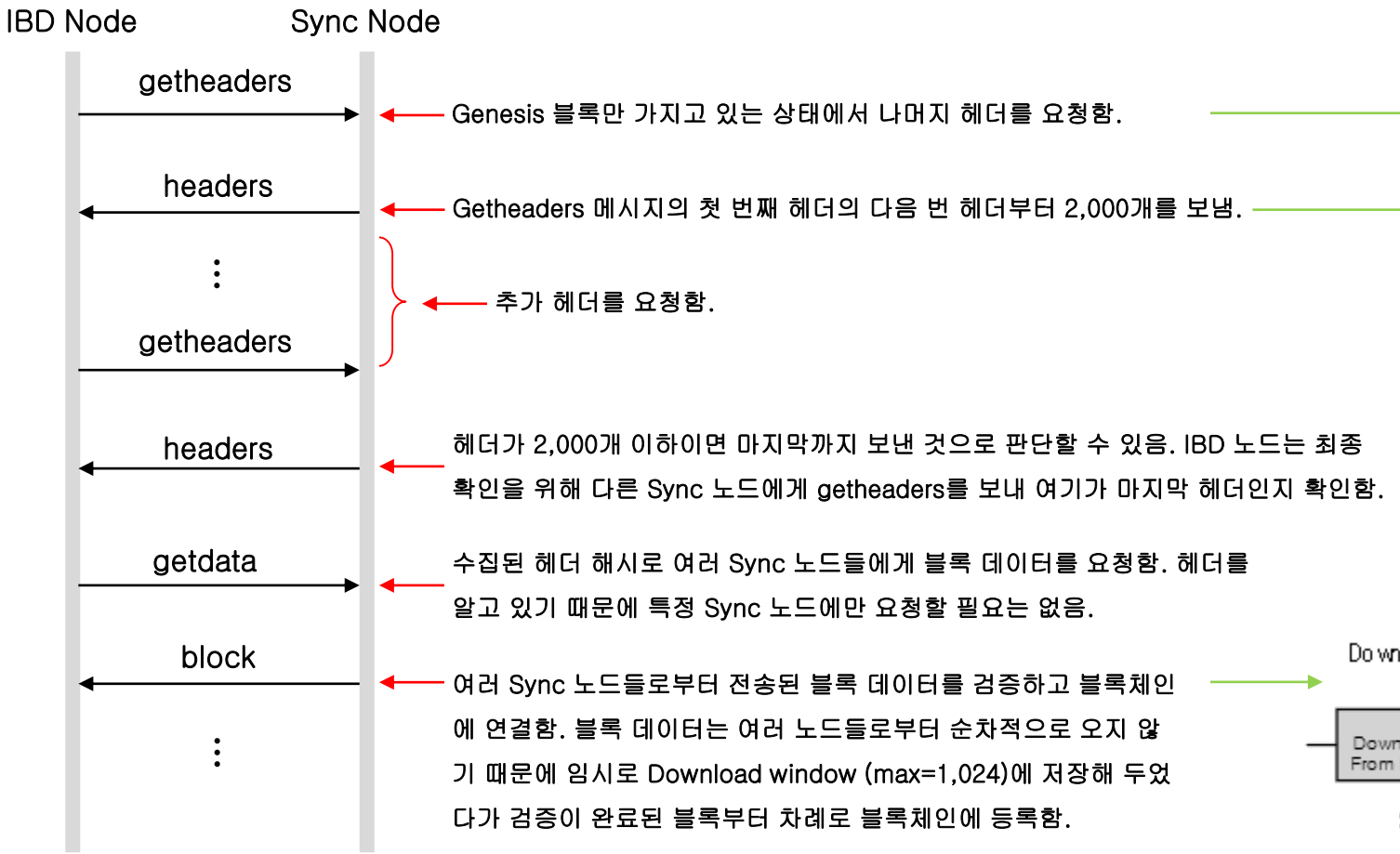
Overview Of Blocks-First Initial Blocks Download (IBD)

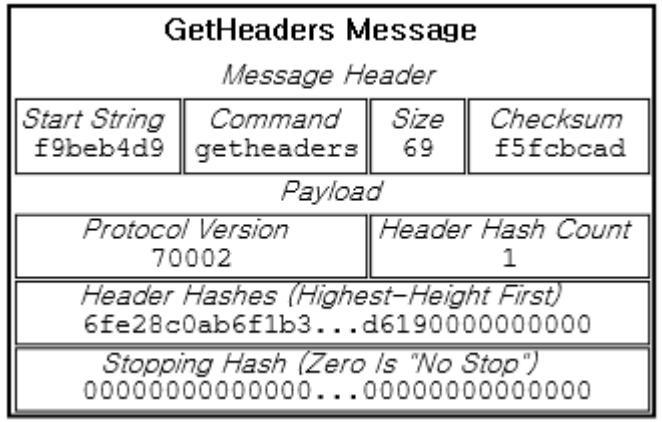


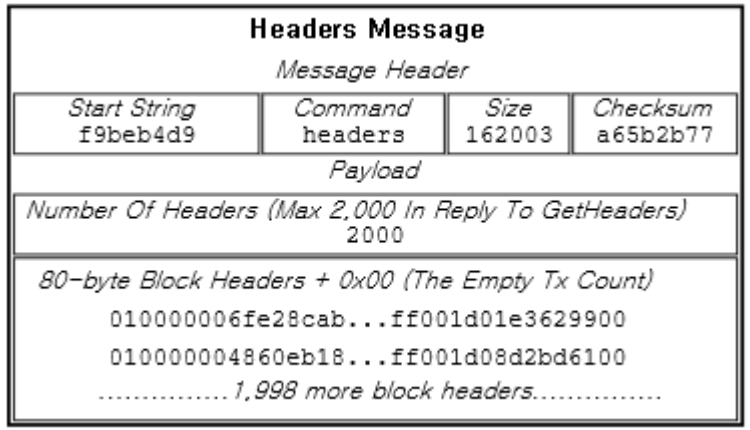
Overview Of Headers-First Initial Blocks Download (IBD)

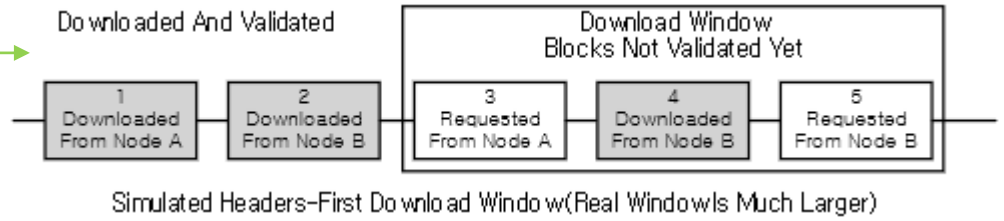
블록 데이터 동기화 : Header-first 방식 - Bitcoin core 0.10.0 부터 사용되는 방식

- Header-first 방식은 Sync 노드로부터 블록의 헤더 정보만 먼저 받고, 헤더 정보를 이용하여 다른 여러 Sync 노드들로부터 동시에 블록 데이터를 받음 (특정 노드에 의존하지 않음).
- 자료 출처 : Bitcoin Developer Guide (https://bitcoin.org/en/developer-guide#initial-block-download)





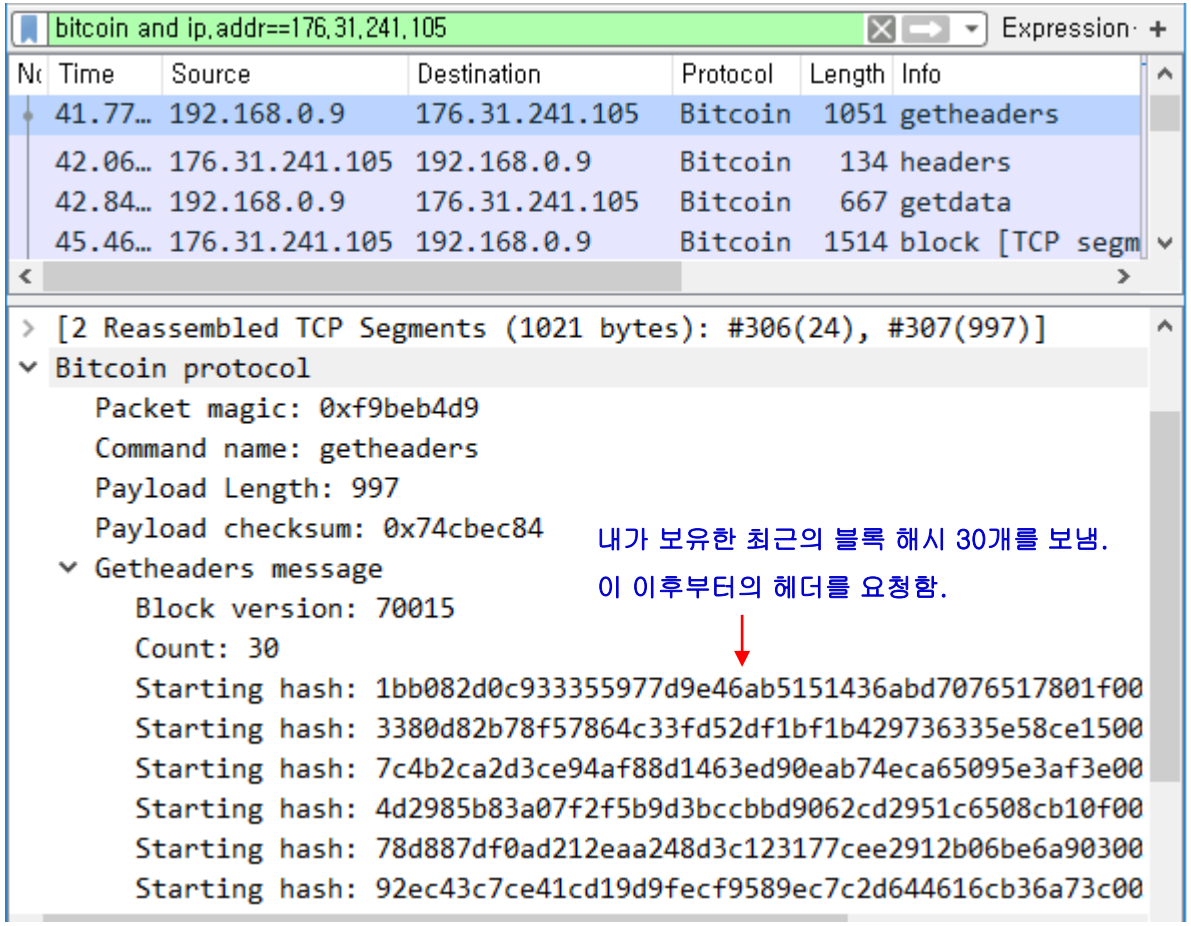


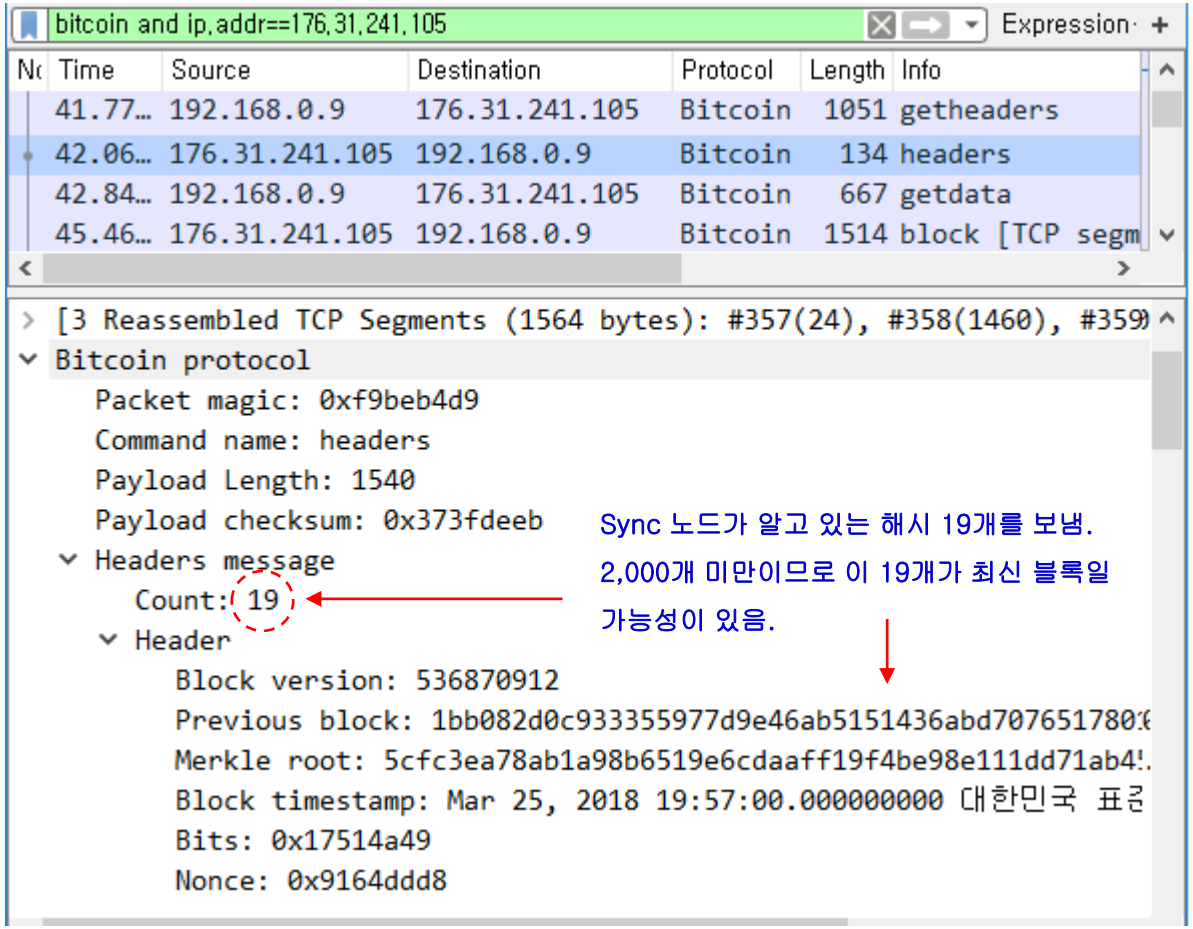


(실습 파일 : 6-1.BitcoinPrototol(1).pcapng)

블록 데이터 동기화 : Header-first 방식

- IBD 노드 (192.168.0.9)가 Sync Node (176.31.241.105)에게 getheaders 메시지를 통해 자신이 알고 있는 최근의 블록 헤더 해시 30 개를 보냄.
- Sync 노드는 getheaders를 받고 IBD가 알고 있는 블록 이후의 블록이 자신의 블록체인에 존재하는지 검색하고 IBD 노드가 가지고 있지 않은 19개의 블록 헤더를 보냄. Sync 노드는 한 번에 최대 2,000개 블록 헤더까지 보낼 수 있는데 19개만 보낸 걸로 봐서 여기까지가 자신이 알고 있는 최고의 블록임.

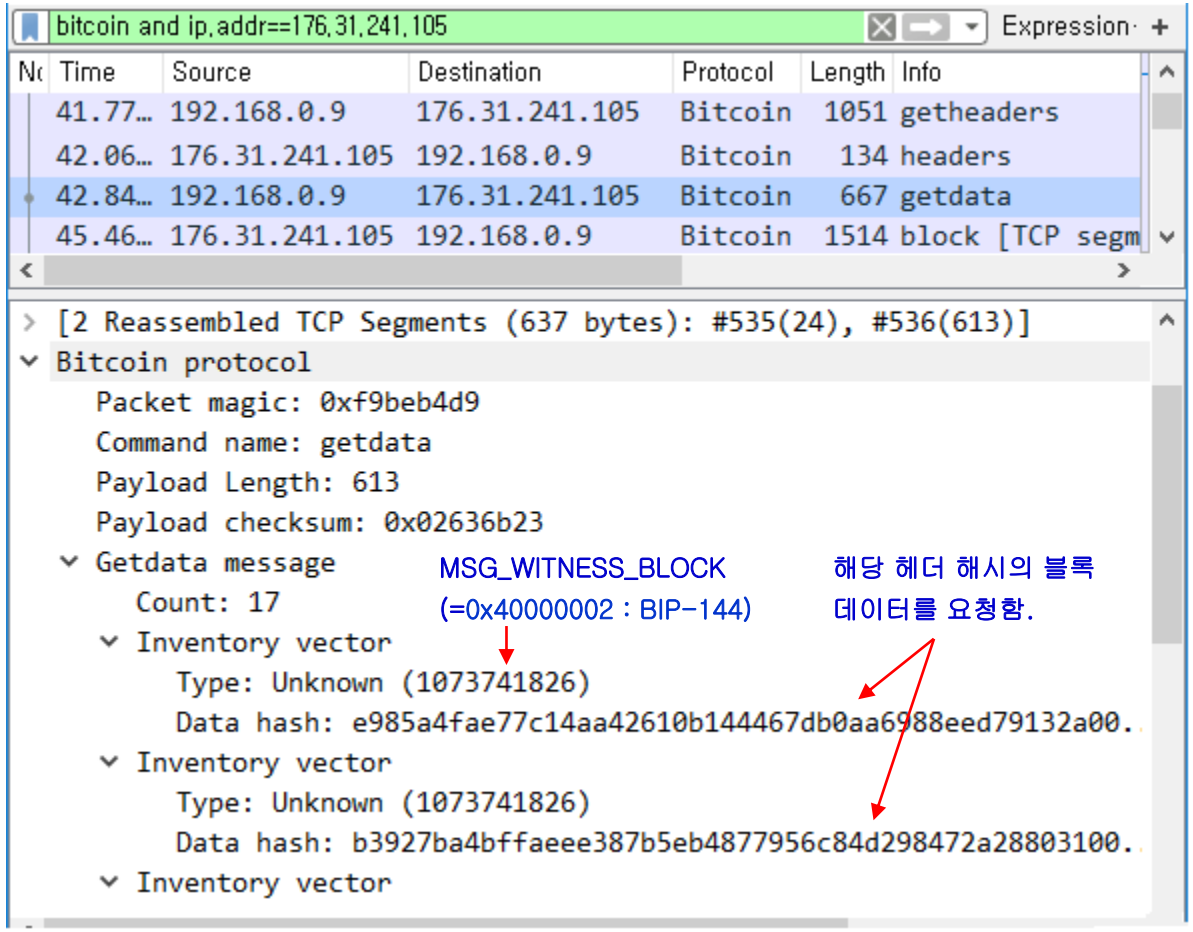


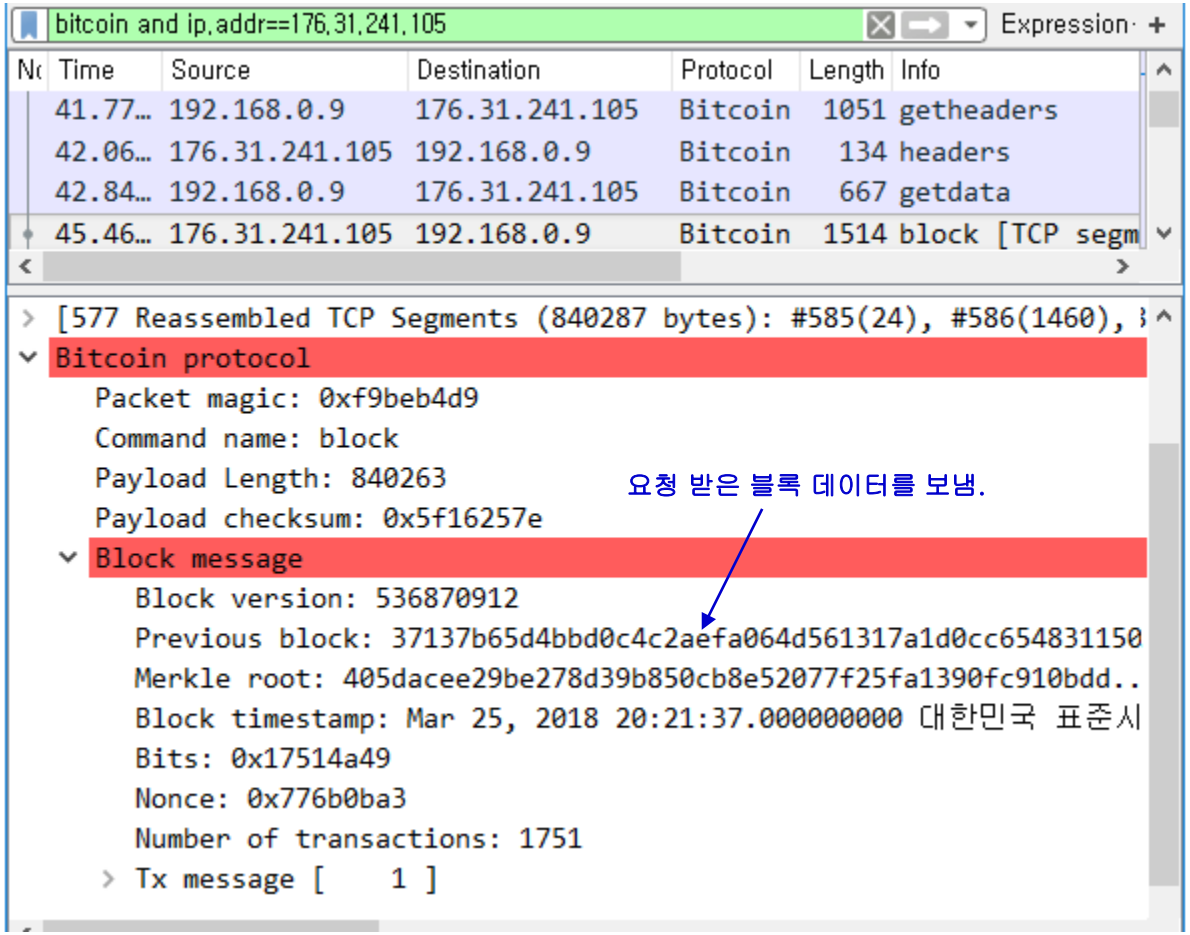


(실습 파일 : 6-1.BitcoinPrototol(1).pcapng)

블록 데이터 동기화 : Header-first 방식

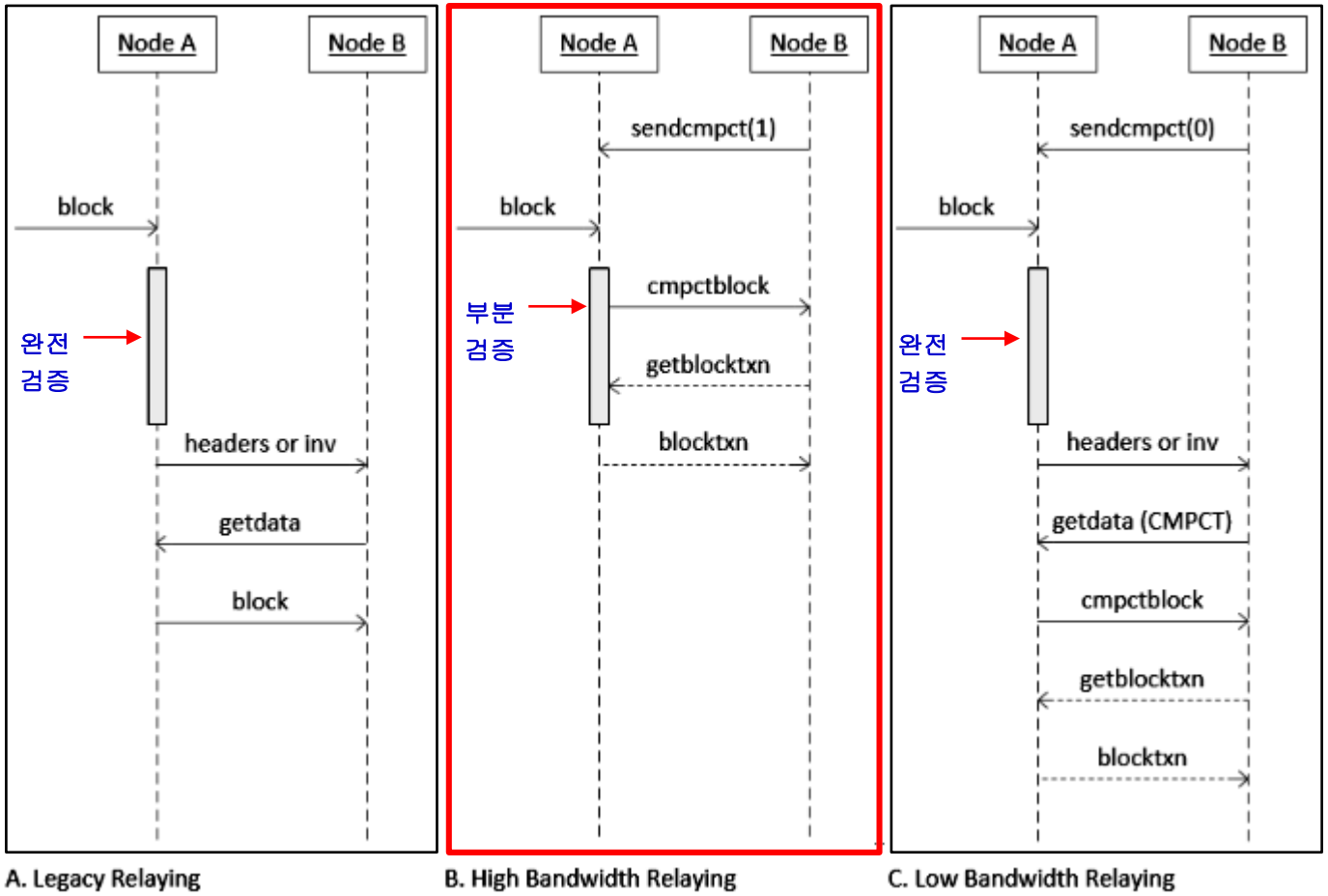
- IBD 노드가 headers 메시지로 블록체인의 헤더들을 수집한 후에는 getdata 메시지를 통해 각 헤더에 해당하는 실제 블록 데이터를 요청함. Getdata 요청은 여러 Full 노드들에게 동시에 요청할 수 있음. SPV (Simplified Payment Verification) 노드는 getheaders-headers 메시지만 교환함.
- Getdata를 받은 Sync 노드는 해당 헤더의 실제 블록을 IBD 노드에게 전송함.





신규 블록 데이터 Relay : Compact Block Relay 방식 (BIP152)

- IBD가 완료된 후 Miner에 의해 지속적으로 발생하는 블록 데이터는 Compact block relay 방식으로 전달됨. Bitcoin Core 0.13.0 부터 이 방식이 적용됨.
- 자료 출처 : BIP152 (https://github.com/bitcoin/bips/blob/master/README.mediawiki)



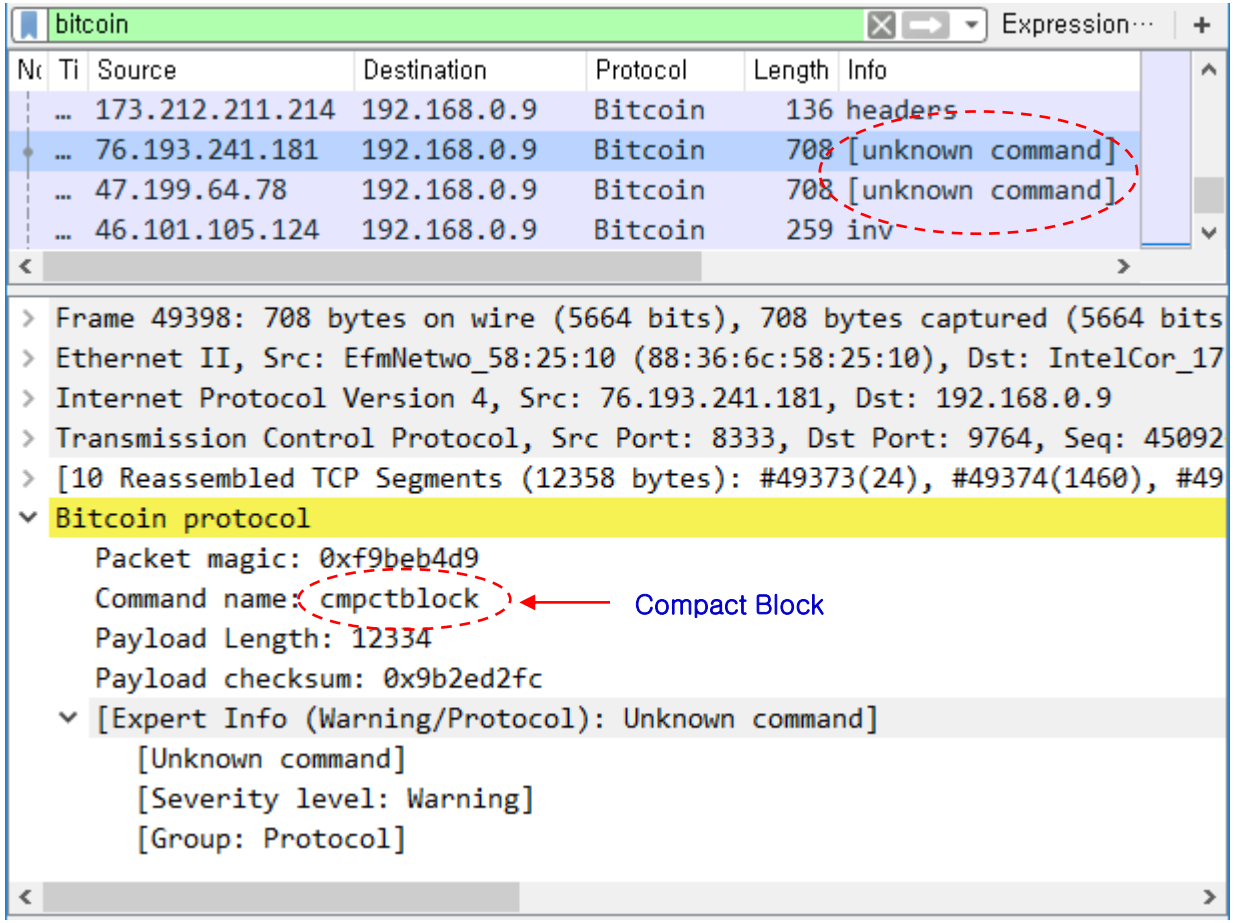
High Bandwidth Relaying

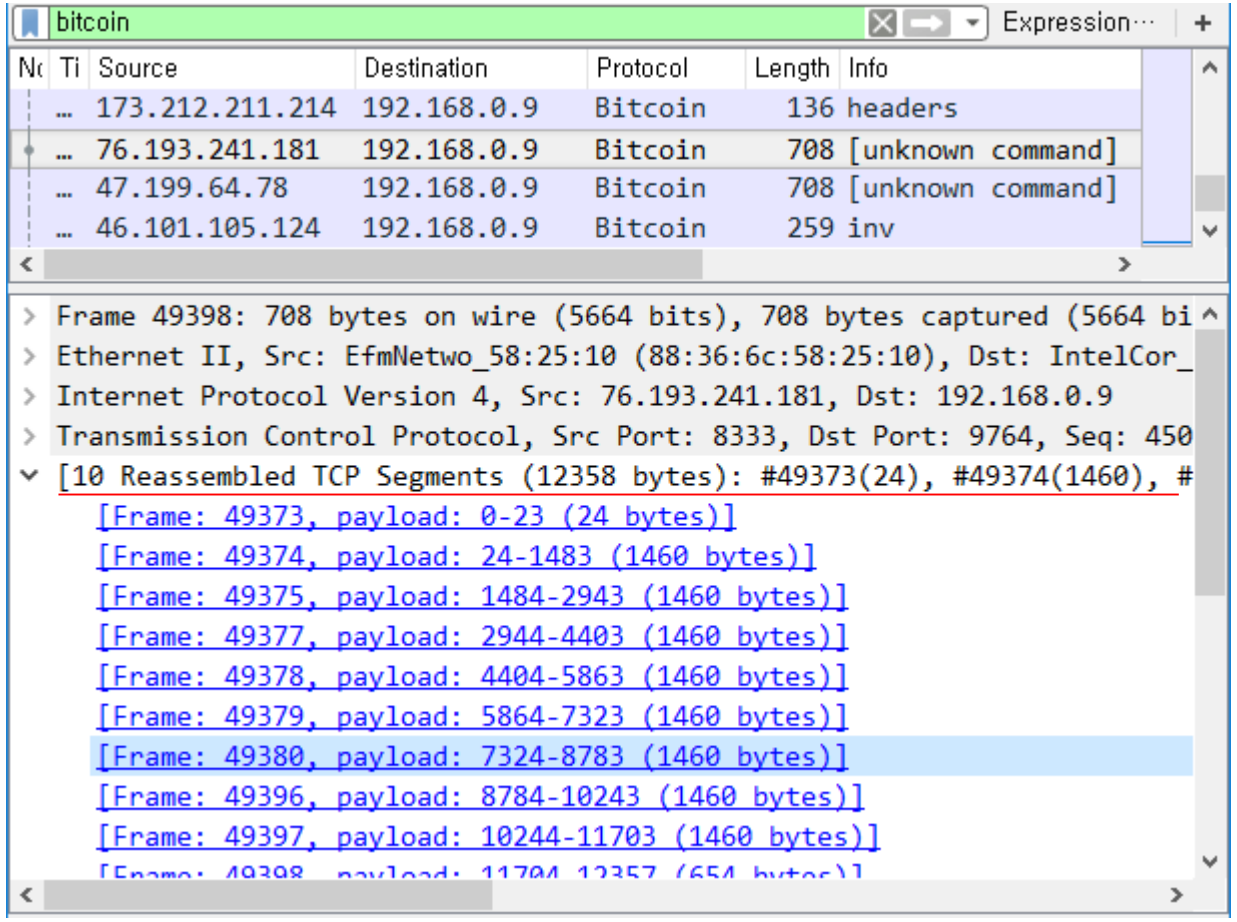
- 노드 B는 A에게 sendcmpt(1) (send compact) 메시지를 보내서 block 메시지는 발생 즉시 받고 싶다고 통보함. (inv, getdata 등의 메시지 교환없이 그냥 블록 데이터를 보내달라고 요청함)
- 노드 A가 새로운 블록 데이터를 받으면 기본적인 검증만 완료하고 cmpctblock 메시지를 통해 즉시 노드 B에게 블록 데이터를 보냄. 블록 데이터를 받자마자 보내므로 일부 TX 데이터는 아직 없을 수도 있음.
- 노드 B가 cmpctblock 메시지를 받으면 블록 데이터를 재구성하고 (reconstruction) 부족한 TX 데이터가 있으면 노드 A에게 getblocktxn 메시지를 보내서 빠진 TX 데이터를 추가로 요청함.
- 노드 A가 getblocktxn 메시지를 받으면 추가로 보낼 TX 데이터가 있으면 blocktxn 메시지를 통해 노드 B에게 TX 데이터를 마저 보냄.

(실습 파일 : 6-2.BitcoinProtocol(2).pcapng)

신규 블록 데이터 Relay : Compact Block Relay 방식 (BIP152)

- 노드 A (76.193.241.181)은 자신이 받은 블록 데이터를 cmpctblock 메시지를 통해 노드 B (192.168.0.9)에게 전달함. Getblocktxn, blocktxn은 잡히지 않았음.
- Cmpctblock 메시지는 아직 wireshark에서 지원하지 않아 [unknown command]로 처리되고 있음.
- 세부 블록 데이터는 TCP Segments 되어 들어 왔음 (아래 우측 화면). 이 데이터들이 블록에 수록된 거래 내역 (TX)들 일 것임.

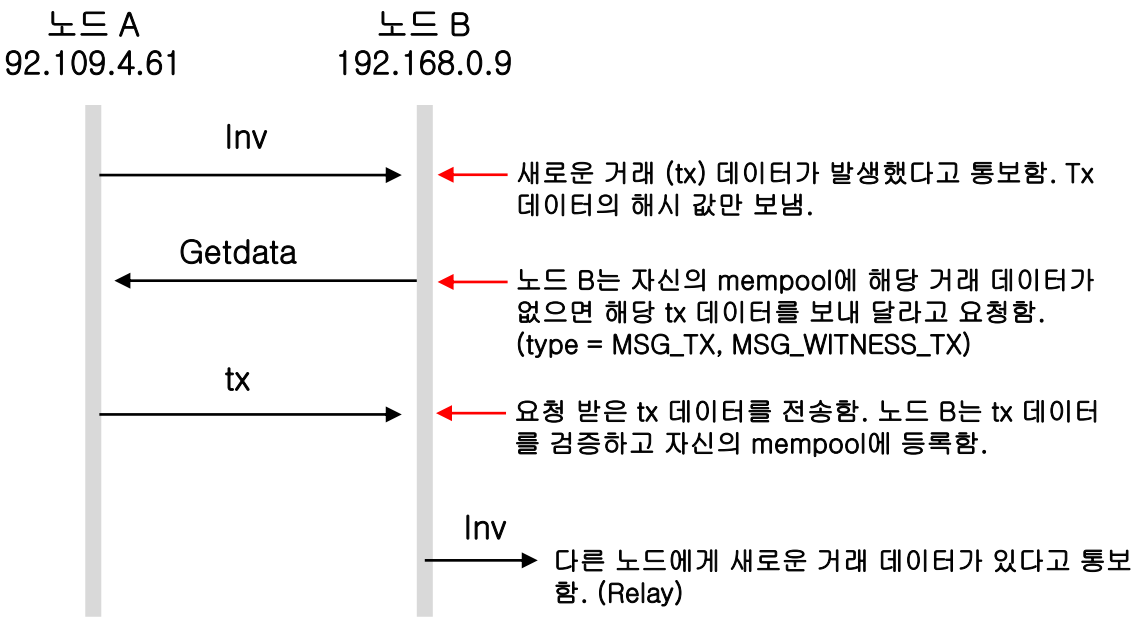


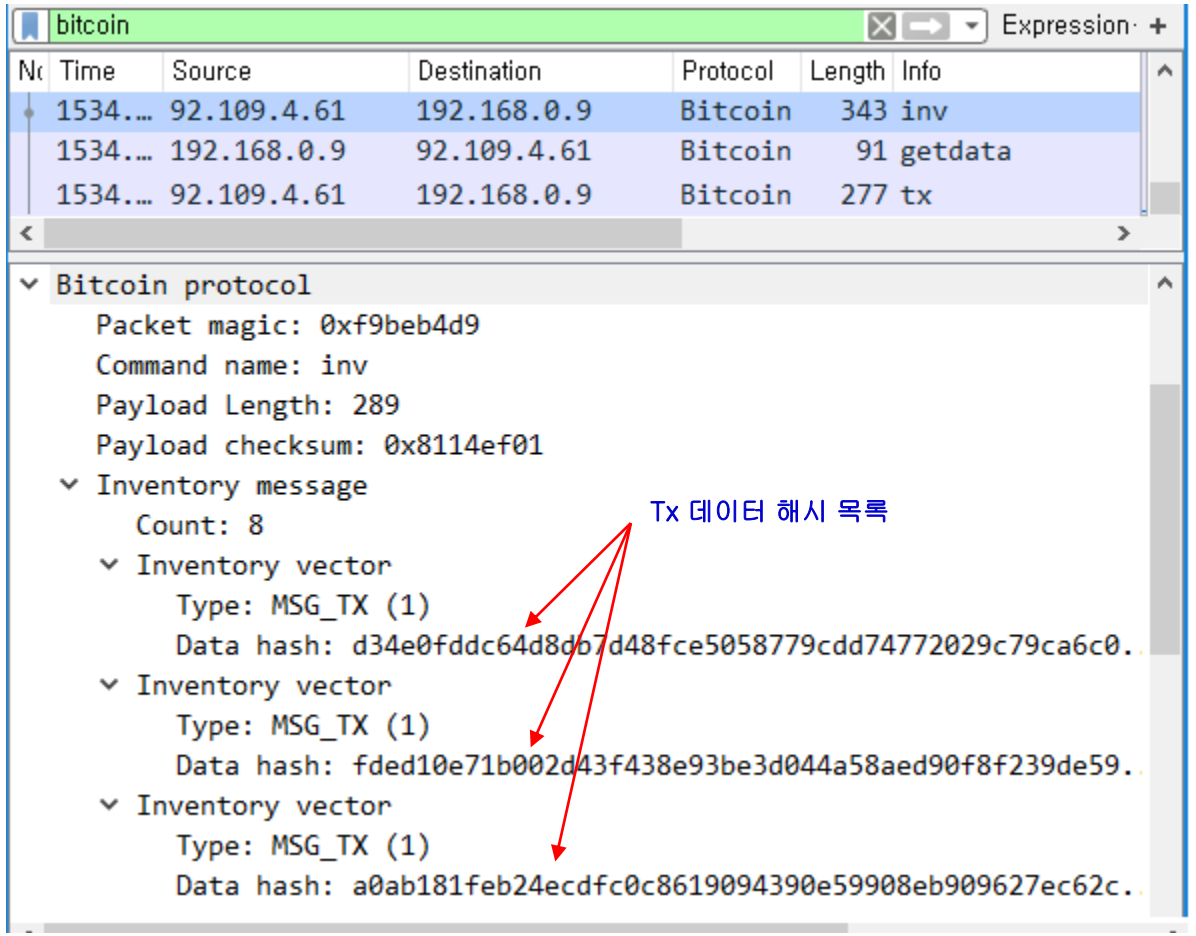


(실습 파일 : 6-2.BitcoinProtocol(2).pcapng)

거래 (Transaction) 메시지 Relay : Inv, getdata, tx 메시지 교환

- 노드 A가 신규로 거래를 발행하거나, 다른 노드들로부터 자신이 몰랐던 새로운 거래 데이터 (tx)를 받으면 노드 B에게 Inv 메시지를 통해 새로운 거래 데이터가 발생했다고 통보함. Inv 메시지로는 새로운 거래 데이터의 해시 값만 보냄. (inventory vector)
- 노드 B는 Inv 메시지 안에 있는 tx 해시 값을 보고 자신이 모르는 tx 이면 getdata 메시지를 통해 tx 데이터를 보내달라고 요청하고, 노드 A는 tx 메시지로 tx 데이터를 전송함. 노드 B는 tx 데이터를 검증하고 다른 노드에게 Inv 메시지로 새로운 tx 데이터가 발생했다고 알려줌. (Relay)
- getdata의 type은 MSG_TX, MSG_WITNESS_TX 이며 SegWit 업그레이드로 인해 MSG_TX는 점차 없어지고 있음. MSG_TX (Old node)로 요청하면 Witness 데이터는 빼고 보냄 (Anyone can spend로 인증됨).

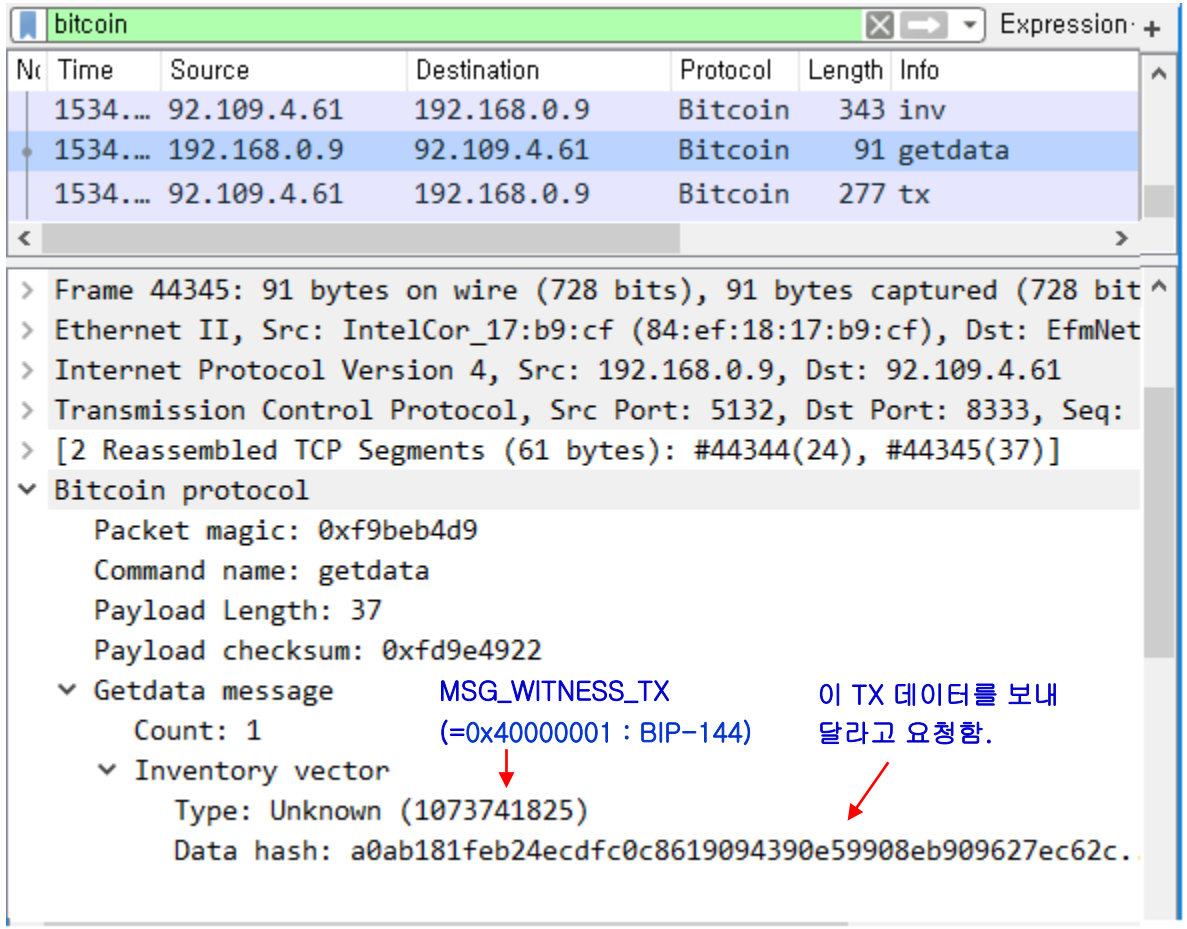


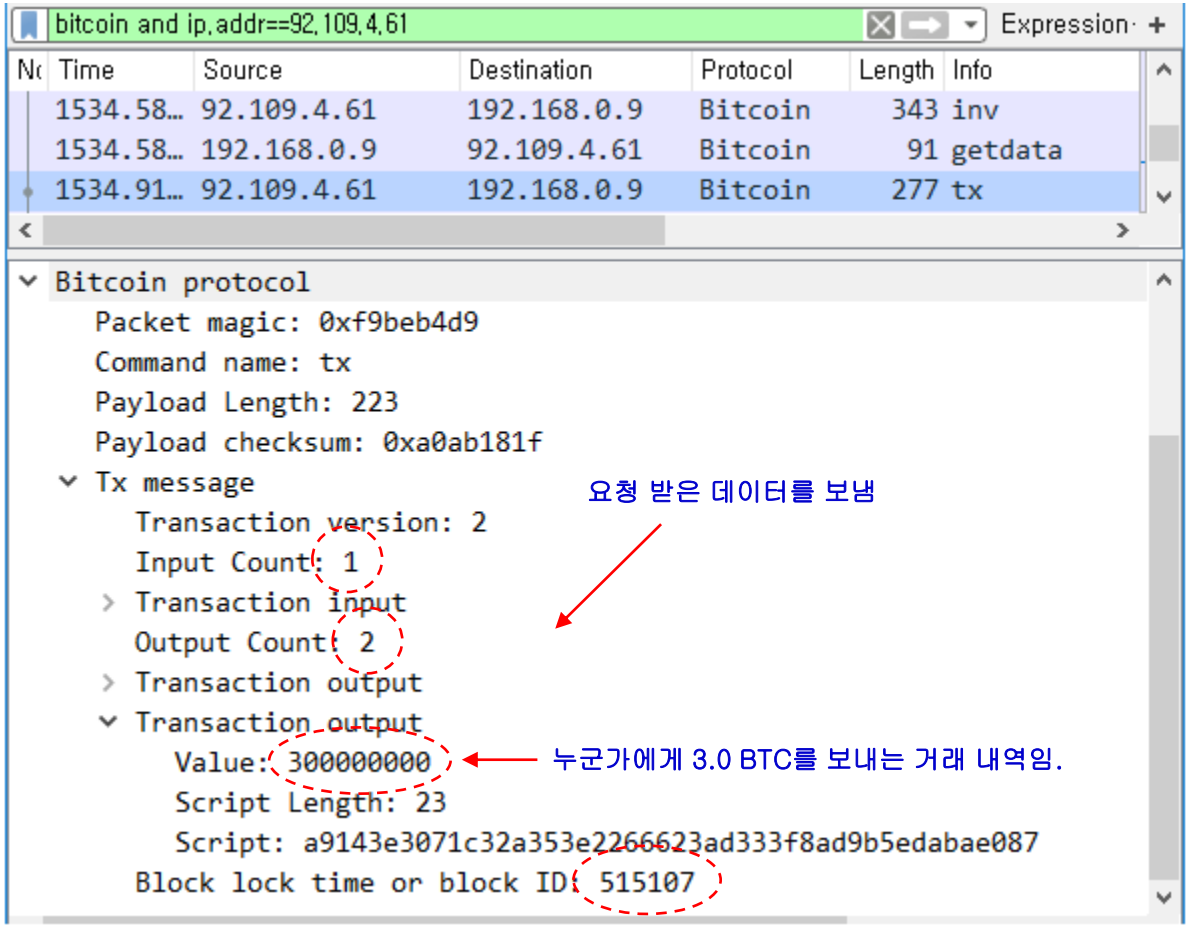


(실습 파일 : 6-2.BitcoinPrototol(2).pcapng)

거래 (Transaction) 메시지 Relay : Inv, getdata, tx 메시지 교환

- 노드 B (192.109.4.61)가 노드 A (92.109.4.61)에게 getdata 메시지를 통해 tx 데이터 1개를 보내달라고 요청하고 있음.
- 노드 B는 해당 tx 데이터를 전송함. 이 거래는 input 1개, output 2개인 거래이고, 두 번째 output은 누군가에게 3.0 BTC (300000000 satoshi)를 보내는 거래임.
- Lock time은 5e8 이하이므로 블록 ID를 의미하고 Miner는 블록체인의 highest 블록 ID가 515,107를 지났으면 이 거래를 승인 대상에 포함시킴.

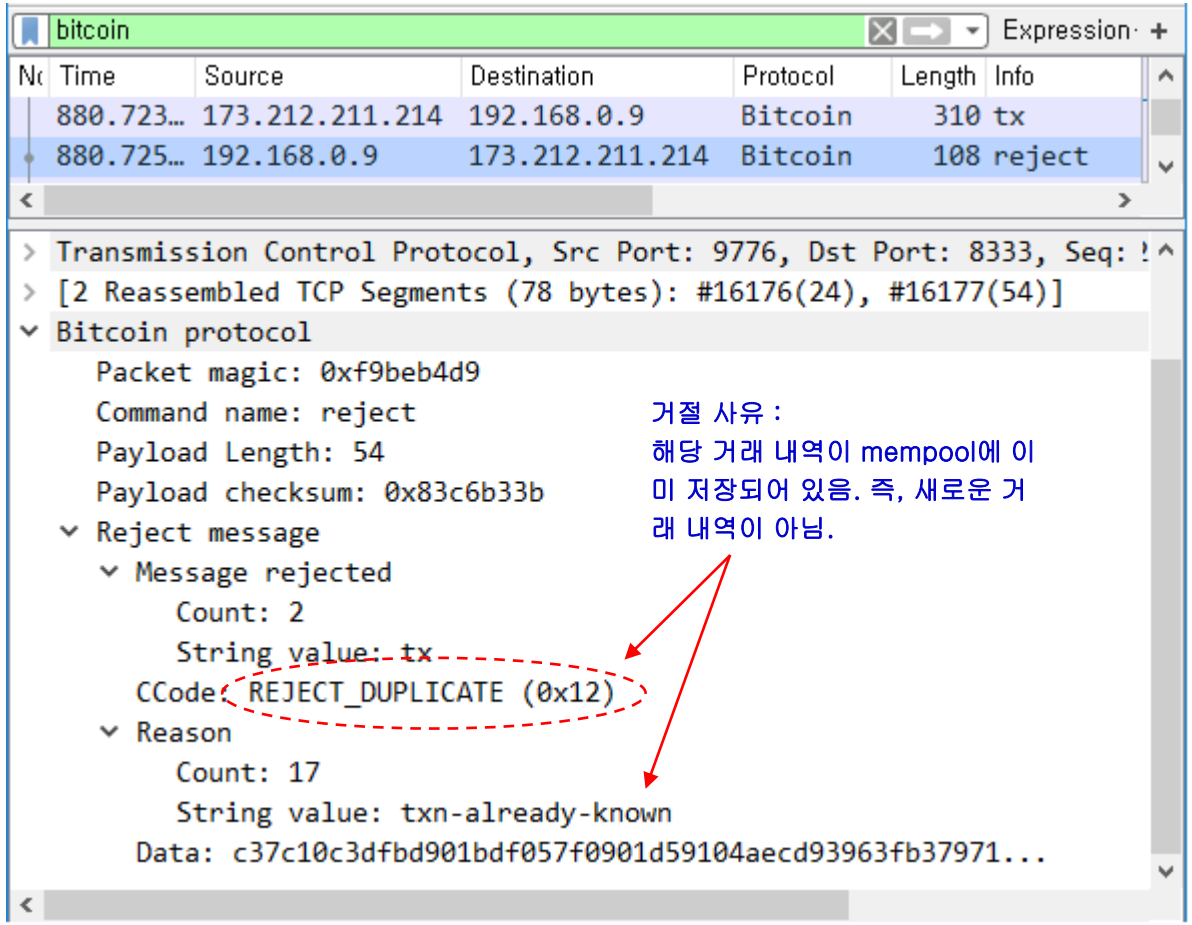


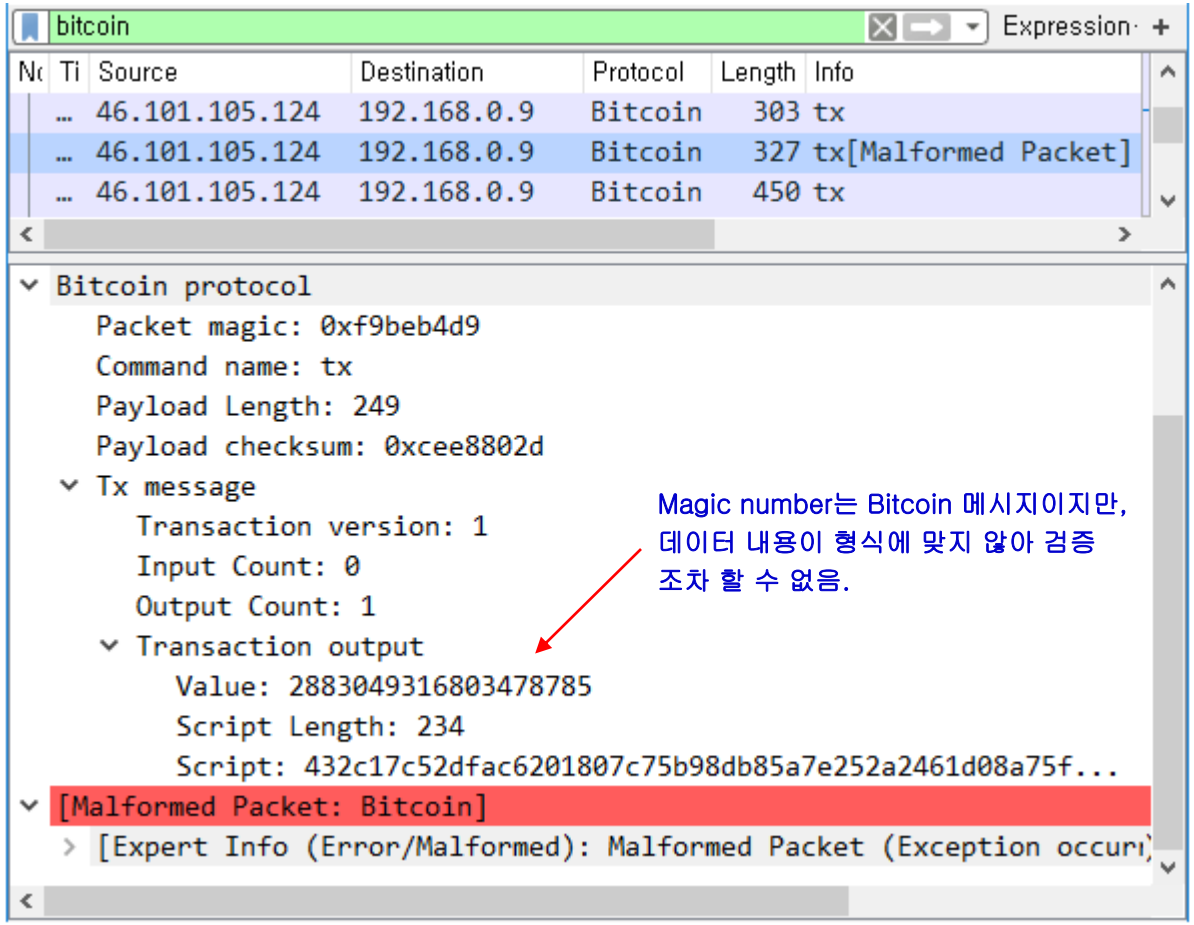


(실습 파일 : 6-2.BitcoinPrototol(2).pcapng)

Reject 메시지 (BIP-61)

- 각 노드는 TX 나 블록 메시지를 받으면 내용의 타당성을 검증하고 다른 노드로 Relay 하는데, 검증 과정에서 잘못된 내용이 발견되면 발신자 노드에게 Reject 메시지를 통해 무엇이 문제인지 알려줌. Reject를 받은 발신자 노드는 이 정보를 참고용으로 사용함. 특정 노드가 악의적인 목적으로 모든 정상 데이터에 대해 Reject를 보낼 수도 있으므로 Reject를 받은 발신자 노드는 자기가 보낸 데이터를 재확인하거나, Reject를 무시하거나 등의 행위를 할 수 있음.
- 아래 오른쪽 그림처럼 데이터의 형식이 잘못된 메시지들은 검증 조차 할 수 없으므로 무시함.
- Reject 기능은 Bitcoin Core 0.9.0 부터 적용되었음. (참고 : https://github.com/bitcoin/bitcoin/pull/3185)





(실습 파일 : 6-3.protocolMsg.py)

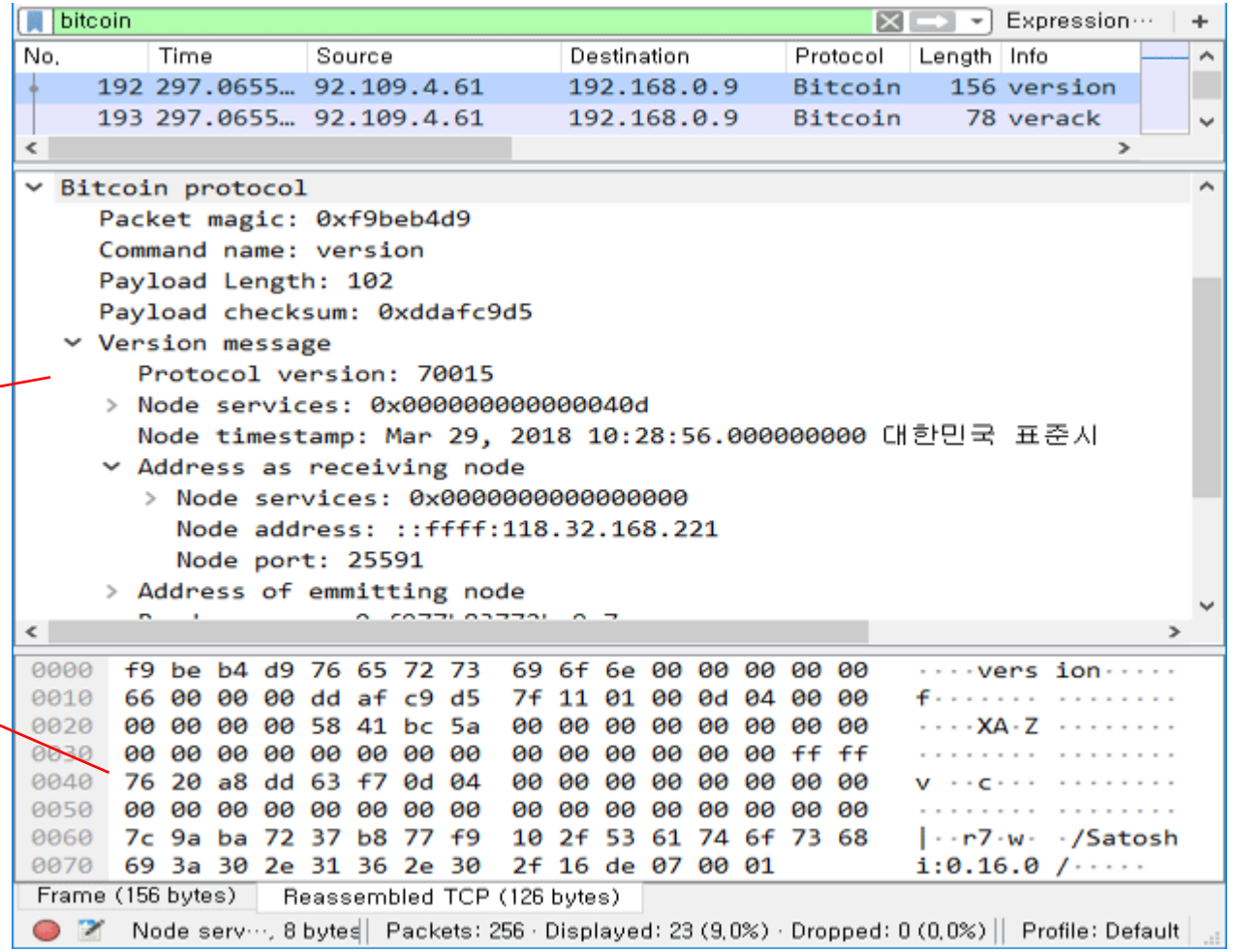
Python 실습 : 비트코인 노드들과 Version, VerAck 메시지 교환

- Wireshark 결과를 참조하면 프로토콜 메시지를 쉽게 만들 수 있음.
- 아래 그림은 Wireshark의 Version, VerAck 메시지를 참조하여 Python 으로 Version, VerAck 메시지 패킷을 구성하고 실제 노드에게 전송하는 예시임.

```
37      return (nodeService + filler + nodeAddress + nodePort)
38
39  # Version 메시지 생성
40  # 참조 : Wireshark analyzer & https://en.bitcoin.it/wiki/Protocol_docur
41  def msgVersion(ip):
42      timestamp = int(time.time())
43      addrYou = verAddr(socket.inet_aton(ip), tcpPort)
44      addrMe = verAddr(b'', 0)
45      nonce = random.getrandbits(64)
46      userAgent = swVersion
47      blockHeight = 0
48
49      msg = struct.pack('<LQQ26s26sQB16sLB',
50                        protocolVersion,
51                        services,
52                        timestamp,
53                        addrYou,
54                        addrMe,
55                        nonce,
56                        16,
57                        userAgent,
58                        blockHeight,
59                        1)
60      return assembleMessage(b'version', msg)
61
62  # Version Ack 메시지 생성
63  def msgVerAck():
64      return assembleMessage(b'verack', b'') # payload 없음
65
66  def sendVersion(destIP):
67      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

참조

Network byte order에 맞게 패킷을 조립함



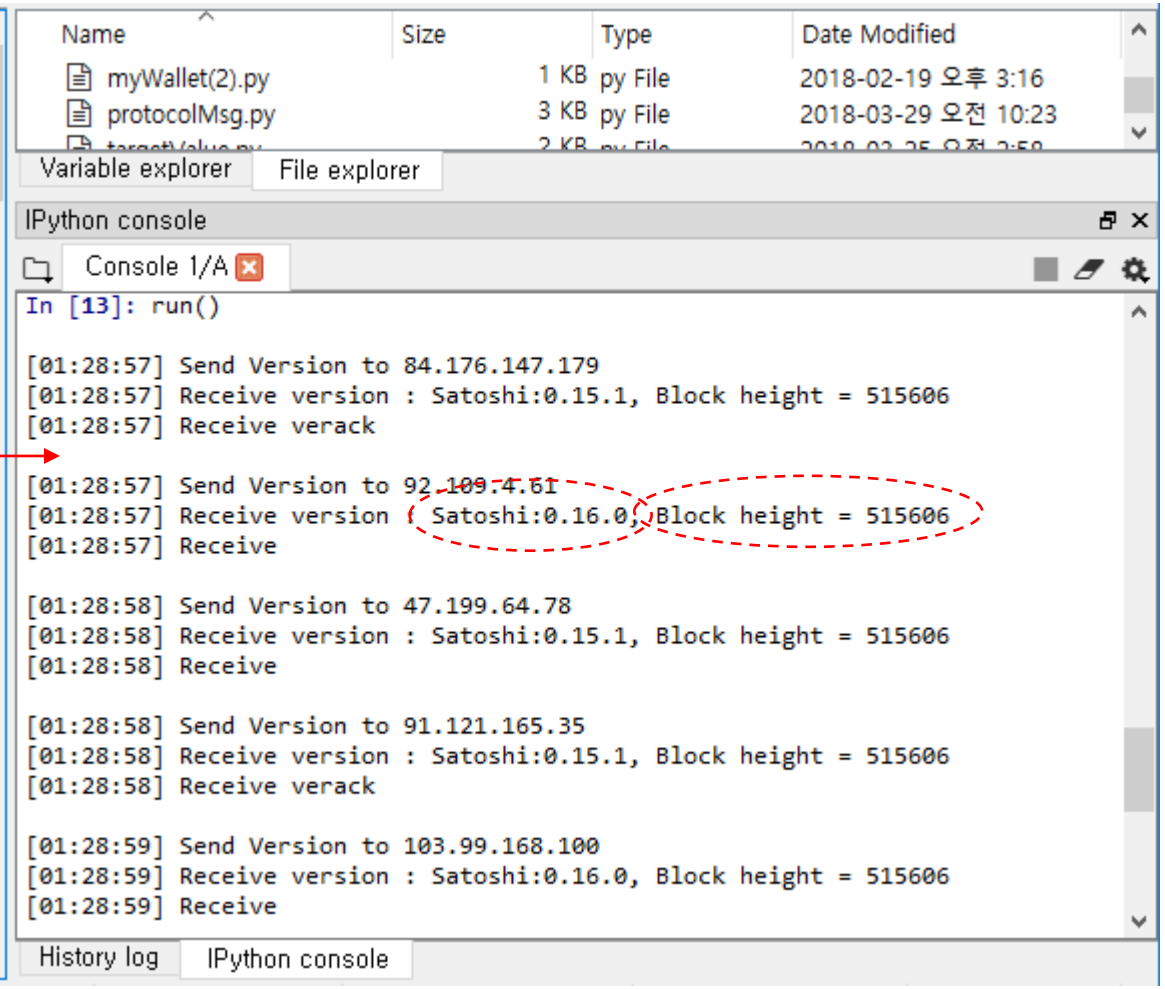
(실습 파일 : 6-3.protocolMsg.py)

Python 실습 : 비트코인 노드들과 Version, VerAck 메시지 교환

- 비트코인 네트워크 상의 노드 5개를 선정하여 Version 메시지를 보내고, 상대방의 Version, VerAck를 받고, VerAck를 보냄.
- 상대방 노드가 정상적으로 동작 중인지 확인할 수 있고, 상대방 노드가 사용하는 Protocol version과 Block height를 알 수 있음.

```
1  # Bitcoin Protocol 연습
2  # Bitcoin 네트워크의 노드들과 Version, VerAck 메시지를 교환한다
3  # Version 메시지를 통해 상대 노드의 protocol version과 block height를 
4  #
5  # 2018.3.29
6  # 아마추어 퀀트 (조성현)
7  # ------------------------------------------------------------------
8
9  import socket
10 import hashlib
11 import struct
12 import time
13 import random
14
15 magic = 0xd9b4bef9
16 protocolVersion = 70015
17 swVersion = b'/Satoshi:0.16.0/'     # 문자열 길이는 16으로 고정함
18 NODE_NETWORK = 1 # Node Service (bitcoin/src/protocol.h 참조)
19 services = NODE_NETWORK
20 tcpPort = 8333
21
22 def assembleMessage(command, payload):
23     checksum = hashlib.sha256(hashlib.sha256(payload).digest()).diges
24     header = struct.pack('L12sL4s', magic, command, len(payload), che
25     return  header + payload
```

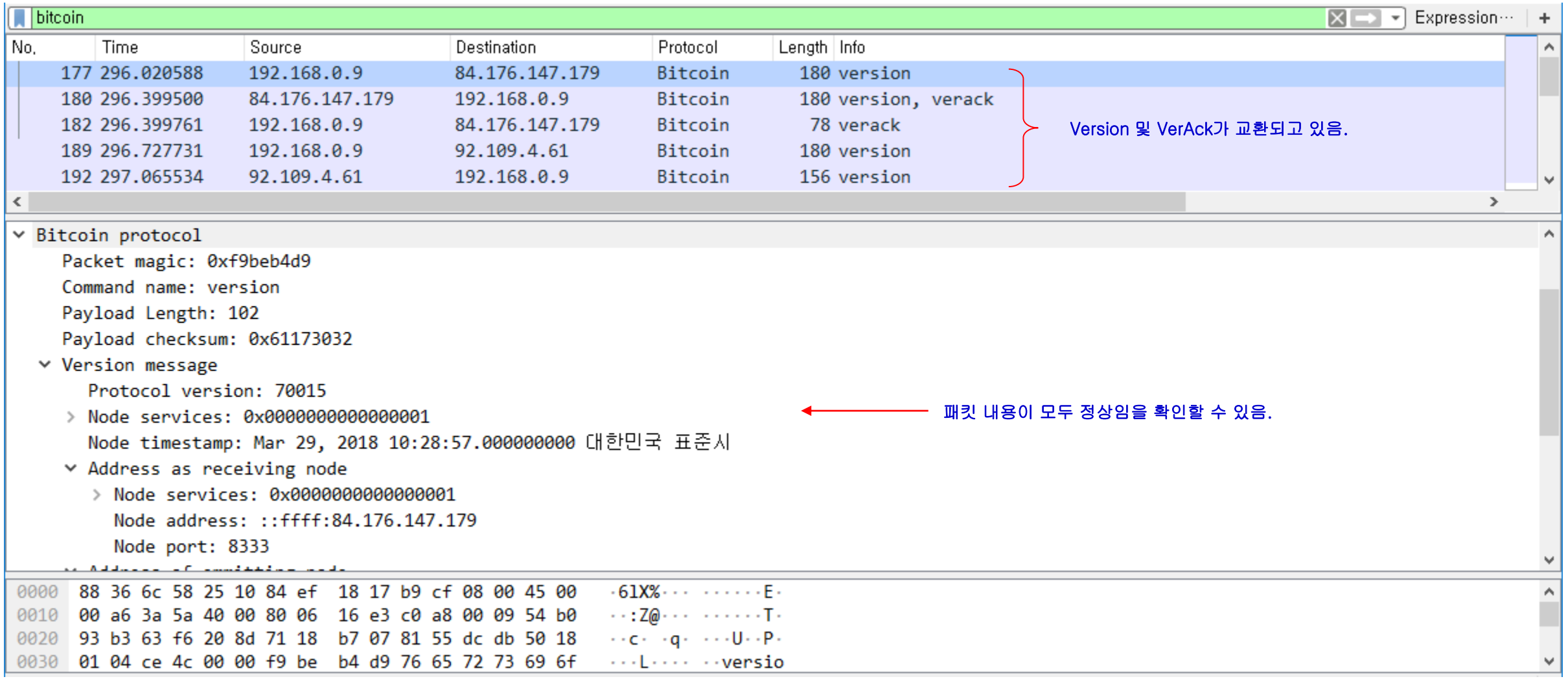
S/W는 0.15.1 과 0.16.0이 주로 사용되고 있고, 5개 모두 515,606 개의 블록을 가지고 있음.



(실습 파일 : 6-4.BitcoinPrototol(3).pcapng)

Python 실습 : 비트코인 노드들과 Version, VerAck 메시지 교환

- 메시지 교환 과정을 Wireshark로 관찰함. 오류 패킷 없이 모두 정상임을 확인할 수 있음.

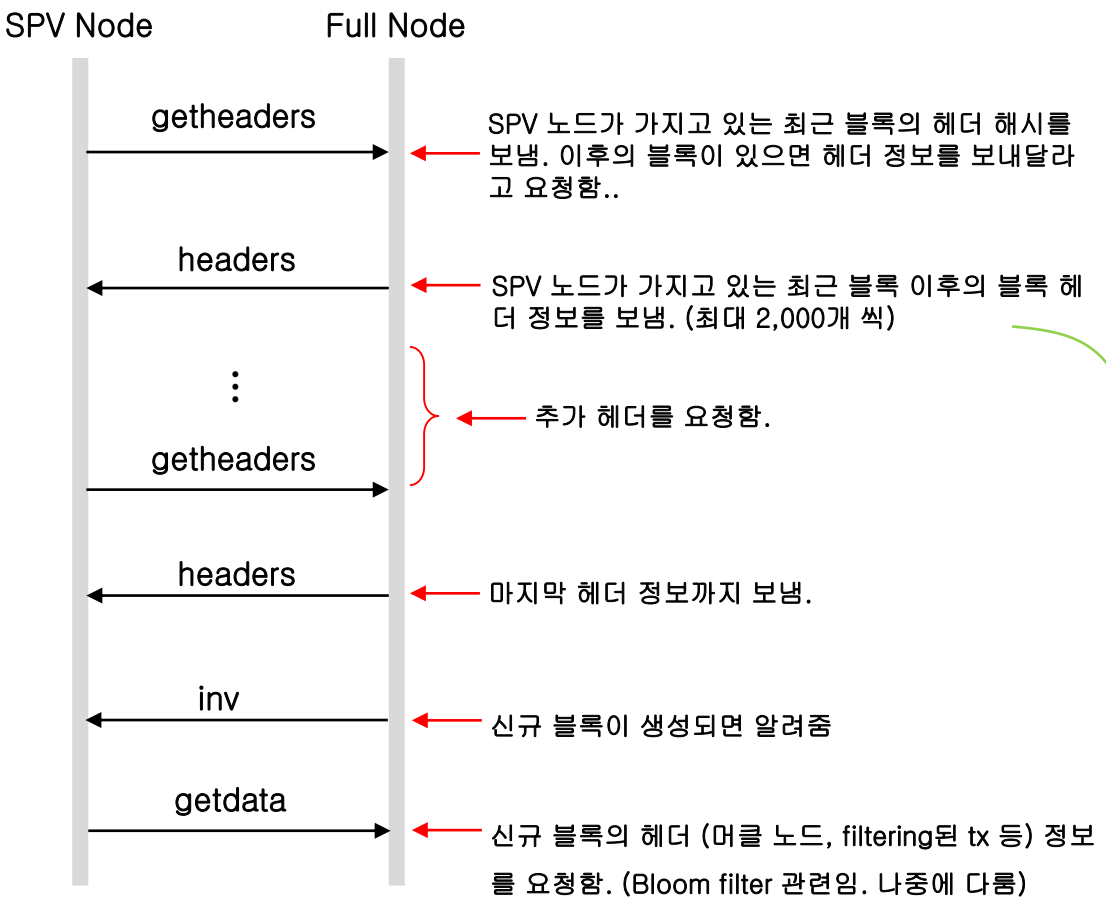


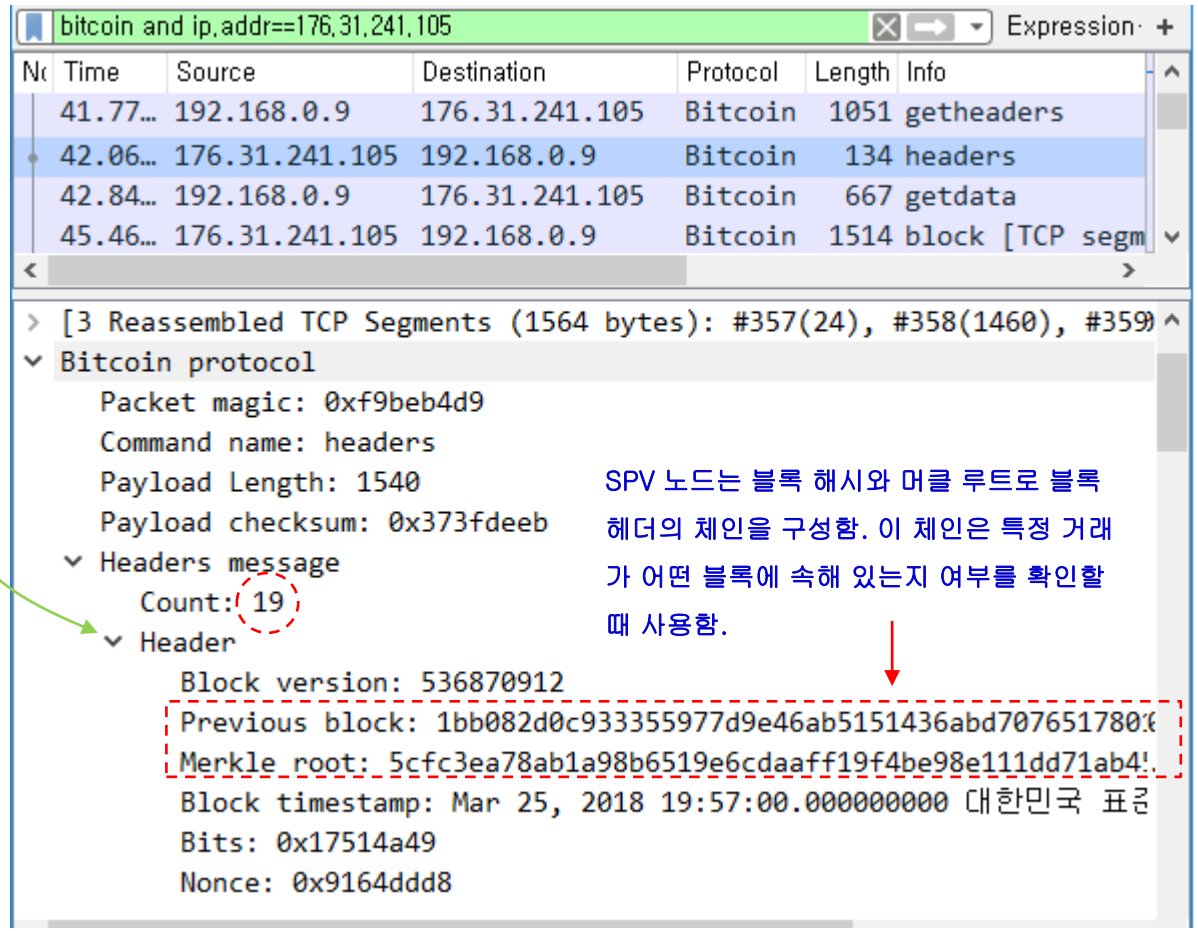
SPV 노드 : Simplified Payment Verification

- Full 노드는 블록체인 데이터 전체를 다운로드 받아 운용하고, 모든 거래의 Unspent output (UTXO) DB를 운용하므로 모드 거래의 유효성을 검증할 수 있음.
- Full 노드는 약 200GB (2018년 4월 현재)의 블록체인 데이터를 보관해야 하므로 스마트 기기 (스마트폰, 노트패드) 같은 소형 장비에 적용하기 곤란함.
- SPV (혹은 Lightweight) 노드는 블록체인의 헤더 정보와 Merkle tree root 정보만 다운로드 받아 운용하는 방식으로 큰 데이터 저장 공간이 필요 없어 (Full 노드의 약 1/1,000) 소형 장비에 적용하기 용이함. (ex : 안드로이드 Bitcoin Wallet App)
- SPV 노드는 거래 내역 (Transaction : TX)이 기록된 블록 데이터가 없기 때문에 거래나 블록의 유효성을 검증할 수 없고, 다른 노드로 Relay할 수 없음.
- SPV 노드는 Full 노드로부터 선별적으로 자신의 지갑과 관련된 거래 내역을 받을 수 있음. SPV 노드는 Full 노드에게 자신의 주소로 발생한 거래 내역만 전송해 달라고 요청할 수 있고, 특정 블록에 속한 자신의 주소와 관련된 거래 내역을 요청할 수 있음.
- SPV 노드가 자신의 주소와 관련된 거래 내역을 네트워크에 요청하면 IP 주소가 노출되어 지갑 주소의 주체가 어떤 장비인지 알려지게 되어 네트워크 공격의 대상이 될 수 있음. 예를 들어 DoS 공격을 받을 수도 있고, 심하면 지갑 S/W에서 관리하는 개인키까지도 유출될 수 있는 위험이 있음.
- 또한, 거래 내역을 요청하는 Full 노드가 악의적인 목적으로 잘못된 거래 내역을 보내면 잔고 계산이 부정확해져서 Miner에게 큰 Fee를 지불할 수도 있음.
- SPV 노드는 Full 노드에 비해 보안상 취약점이 많을 수밖에 없음.
- SPV 노드는 Bloom Filter (BIP-37)를 사용하여 보안 취약점을 어느 정도 보완할 수 있음. → 취약점이 완전히 해결되는 것은 아님.
- Bloom Filter를 사용하면 SPV 노드 자신의 주소 이외의 다른 주소들과 관련된 거래 내역들도 같이 요청하여 자신의 주소가 직접적으로 노출되는 것을 방지할 수 있음. Full 노드는 Bloom Filter를 통과하는 모든 거래 내역을 SPV 노드로 보내주기 때문에 그 중 어느 것이 SPV 노드에 속하는지 판단하기 어려움. SVP 노드는 여러 거래 내역 중에 자신의 주소에 해당하는 것만 이용하고 나머지는 버림.
- SPV 노드는 Bloom Filter를 사용하여 아직 Mining 되지 않은 거래 내역을 받아볼 수 있고, 이미 Mining되어 특정 블록에 속한 거래 내역도 받아볼 수 있어서 자신에게 발생한 거래를 확인할 수 있음.
- SPV 노드는 신규 블록이 발생할 때 해당 블록의 헤더 정보를 수집하고 (getheaders), Full 노드에게 신규 블록에 포함된 거래 내역 중 Bloom Filter를 통과하는 것들과 Merkle branch 정보를 요청함 (getdata). 자신의 거래 내역이 신규 블록에 있는지 여부는 Merkle brench를 통해 확인할 수 있음.
- 신규 블록에 자신의 거래 내역이 있다면 SPV 노드는 자신의 거래 내역이 성공적으로 Mining 되었다는 것을 확인할 수 있고 (block height), 향후 블록이 추가로 6개 이상 Mining 되면 (block depth) 자신의 거래가 블록체인에 등록되었다는 것을 최종 확인할 수 있음.

SPV 노드 : 블록 헤더 체인 구축

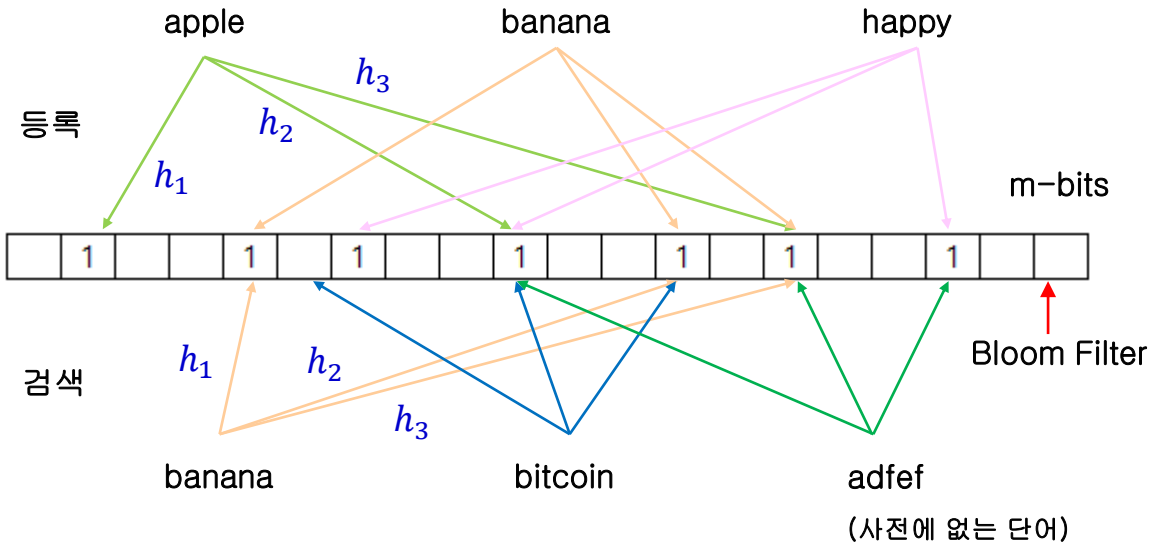
- SPV 노드가 비트코인 네트워크에 처음 접속하면 Full 노드가 가지고 있는 블록체인의 모든 블록 헤더 정보를 받아 블록 헤더 데이터를 보관함.
- 블록 헤더 정보는 SPV 자신의 지갑과 관련된 거래가 특정 블록에 속해 있는지 여부를 증명할 때 사용됨.





Bloom Filter

- Bloom Filter는 어떤 원소가 특정 집합에 속해 있는지 여부를 확인할 때 사용되는 유용한 자료구조임 (확률적 자료구조)
- 사전 (Dictionary) 집합 A = {apple, banana, happy, …} 가 있을 때 banana, bitcoin, adfef 라는 단어가 사전 집합 A에 들어 있는지 여부를 확인함.
- m-bits의 Filter 배열과 k개의 hash function (h_1, h_2, h_3, $k = 3$)를 준비함. 각 hash function은 특정 원소에 대해 1 ~ m 까지의 자연수를 구해주고 (ex : h_1(banana) = 5, h_2(banana) = 13, h_3(banana) = 15), 그 결과에 해당하는 Filter의 비트 (ex : 5, 13, 15번 비트)를 1로 설정함.
- 집합 A의 모든 원소에 대한 hash 값을 Filter에 등록해 두면, 임의의 단어가 Filter에 등록되어 있는지 확인할 수 있음. 예를 들어 banana가 Filter에 등록되어 있는지 확인하려면 동일 hash function을 사용하여 해당 비트가 1로 설정되어 있는지 확인하면 됨. 5, 13, 15 비트 모두 1이므로 banana는 집합 A에 속해 있음.
- Hash function은 Collision이 발생할 수 있으므로 어떤 단어의 hash 값이 모두 1이라도 그 단어가 집합 A에 들어 있다고 확신할 수는 없음 (False positive error > 0). 그러나 해당 단어의 hash 값 중에 하나라도 1이 아니면 (0 이면) 그 단어는 집합 A에 속해 있지 않는다고 확신할 수 있음. (False negative error = 0)



- 단어 banana는 집합 A에 속해 있을 가능성이 있음 (불확실성. 실제로 있음).
- 단어 bitcoin은 집합 A에 속해있지 않음 (확실성)
- 단어 adef는 집합 A에 속해 있을 가능성이 있음. (실제는 없음)

- 최적 hash function 개수 (k)와 Filter의 비트 개수 (m) 결정 :
  n은 집합 A의 원소 개수이고, p는 희망하는 False positive error 임

$$m = -\frac{n\,ln(p)}{(ln(2))^2} \qquad k = \frac{m}{n}\,ln(2)$$

- 공식 참조 : https://en.wikipedia.org/wiki/Bloom_filter

(실습 파일 : 6-5.bloomFilter.py)

Bloom Filter : Python 실습 – Bloom Filter 구축과 파라메터 (k, m) 최적화

- False Positive error가 원하는 수준이 되도록 hash function 개수 (k)와 Bloom Filter의 비트수 (m)를 설정하고, 실제 error를 측정하여 확인함.

```python
# Bloom Filter 연습
# 임의의 난수 100개를 생성해서 집합 A에 넣고, 임의의 난수들을 발생하면서 해당 난수가
# 집합 A에 속하는지 확인한다.
#
# False positive 비율 (확률)이 원하는 수준이 되도록 Bloom filter 파라메터 (k, m)를
# 결정하고, 실제 False positive 비율을 측정해 본다.
#
# k = Number of hash functions
# m = Number of Bloom filter bits
#
# Bitcoin Core는 Murmur3 hash function을 사용함.
# 참고 : 1. bitcoin/src/hash.cpp --> unsigned int MurmurHash3(...)
#        2. https://en.wikipedia.org/wiki/MurmurHash
#        3. https://bitcoin.org/en/developer-examples#creating-a-bloom-filter
#
# 2018.3.30
# 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
from bitarray import bitarray
import math
import random
import mmh3     # pip install mmh3 : Murmur3 hash function

# Bloom filter의 최적 파라메터를 결정한다
# n = number of elements
# prob = Desired False positive probability
def optParameter(n, prob):
    m = -1.44 * n * math.log(prob, 2)
    k = math.ceil(m * math.log(2) / n)
    return k, math.ceil(m)

# element를 Bloom filter에 등록한다
def add(BloomFilter, element, k, m):
    for i in range(k):
```

```
Name                    Size      Type      Date Modified
3-5.bloomFilter.py      3 KB      py File   2018-04-01 오전 3:17
digitalSigniture.py     1 KB      py File   2018-02-20 오전 2:07

Variable explorer  File explorer

IPython console
Console 1/A

In [2]: run(prob = 0.1)

Desired FP rate =  0.1
Number of hash functions (k) =  4
Number of Bloom filter bit (m) =  479
Actual FP rate (p) =  0.11749

In [3]: run(prob = 0.3)

Desired FP rate =  0.3
Number of hash functions (k) =  2
Number of Bloom filter bit (m) =  251
Actual FP rate (p) =  0.2993

In [4]: run(prob = 0.7)

Desired FP rate =  0.7
Number of hash functions (k) =  1
Number of Bloom filter bit (m) =  75
Actual FP rate (p) =  0.76135

In [4]:

History log  IPython console
```

False Positive 희망값

False Positive 실측값

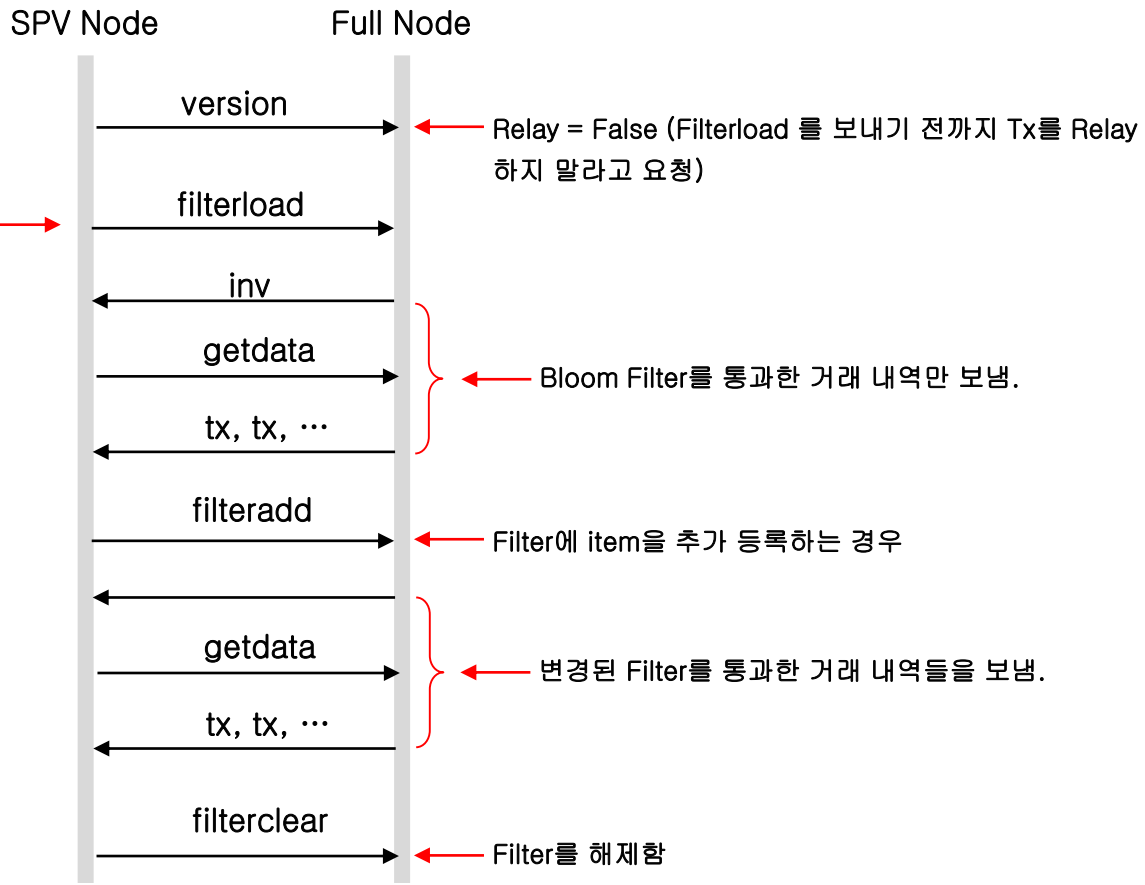
두 값이 잘 일치하고 있음.

SPV 노드 : Bloom Filter – BIP 37 (Connecting Bloom Filter)

- SPV 노드의 Privacy 노출 위험을 방지하기위해 Bloom Filter가 도입됨 (BIP 37). 완전히 해결되는 것은 아니고 개선 방법이 필요함 (계속 논의되고 있음).
- SPV 노드는 filterload 메시지를 통해 자신이 관심있는 거래 내역만 받을 수 있음. Filterload를 통해 Bloom Filter를 Full 노드에게 전달함.
- Full 노드는 Bloom Filter를 통과한 Tx 들만 SPV 노드로 Relay함. SPV 노드는 자신이 관심있는 Tx 만 처리하고, 나머지는 무시함.

m = 16
k = 11

```
02 ......... Filter bytes: 2
b50f ....... Filter: 1010 1101 1111 0000
0b000000 ... nHashFuncs: 11
00000000 ... nTweak: 0/none
00 ......... nFlags: BLOOM_UPDATE_NONE
```



- 사전에 자신의 거래 정보 (public key hash 등) 를 포함한 Bloom Filter를 제작하고 (k, m, filter bit array) Full 노드에게 보냄.
- Full 노드는 이 Filter를 통과하는 거래 내역만 SPV 노드에게 Relay 함.
- SPV 노드는 관심있는 거래 내역만 참조하고 나머지는 무시함.
- 무시하는 거래가 많아질수록 (Loose filter) SPV 노드의 Privacy 노출 정도는 감소하지만 Relay 되는 Tx 들이 많아져서 처리할 Traffic이 늘어남. 반면에 무시하는 거래가 적어질수록 (Strict filter) SPV 노드가 처리할 Traffic은 감소하지만 Privacy 노출 정도는 증가함.
- SPV 노드는 Privacy와 Traffic 처리 사이의 Trade-off 를 고려하여 Filter를 설정함 (SPV 노드의 bandwidth를 고려함).
- Full 노드는 Filter를 통과한 Tx 들과 연계된 Tx 들도 Relay 할 수 있도록 Filter를 add 할 수도 있음 (nFlags로 조절함)

SPV 노드 : Bloom Filter 설정 – BIP 37 (Connecting Bloom Filter)

- Filteradd 메시지에는 Bloom Filter를 구성하는 정보가 들어 있음. 초기에 SPV 노드는 Full 노드에게 Bloom Filter 정보를 전달함.
- nFlag가 설정되어 있으면, Full 노드는 Filter를 통과하는 거래들의 output 정보를 Filter에 추가함. 이것은 SPV 노드가 자신의 UTXO set을 관리하기 위해 필요함.
- Full 노드가 자동으로 Filter를 업데이트하면 False positive error가 점차 증가하므로 SPV 노드는 이를 지속적으로 관찰하여 새로운 Filter를 보내야함.

* Filterload format

Bytes	Name	Data Type	Description
Varies	nFilterBytes	compactSize uint	Number of bytes in the following filter bit field.
Varies	filter	uint8_t[]	A bit field of arbitrary byte-aligned size. The maximum size is 36,000 bytes.
4	nHashFuncs	uint32_t	The number of hash functions to use in this filter. The maximum value allowed in this field is 50.
4	nTweak	uint32_t	An arbitrary value to add to the seed value in the hash function used by the bloom filter.
1	nFlags	uint8_t	A set of flags that control how outpoints corresponding to a matched pubkey script are added to the filter. See the table in the Updating A Bloom Filter subsection below.

← 아래 filter 의 바이트 수

← Full 노드에서 거래 내역을 필터링할 filter. Filter의 비트 수 = m.

← Hash function 개수. SPV 노드에서 결정해서 Full 노드에게 알려줌.
Full 노드는 SPV 노드와 동일한 Hash function (murmur3 32-bit), k, m을 사용함.

← Murmur3 hash function의 추가 seed 값. ([1] 참조)

- 입금 Tx1 발생. Full 노드가 보내줌. SPV 노드는 UTXO 등록.
- Full 노드는 Tx1의 output에 있는 Public key script를 자동으로 Filter에 add함.
- Tx1을 사용하는 Tx2 발생. Tx2도 filtering 되어 SPV로 보내짐. SVP는 Tx1의 output을 Spent로 설정할 수 있음.

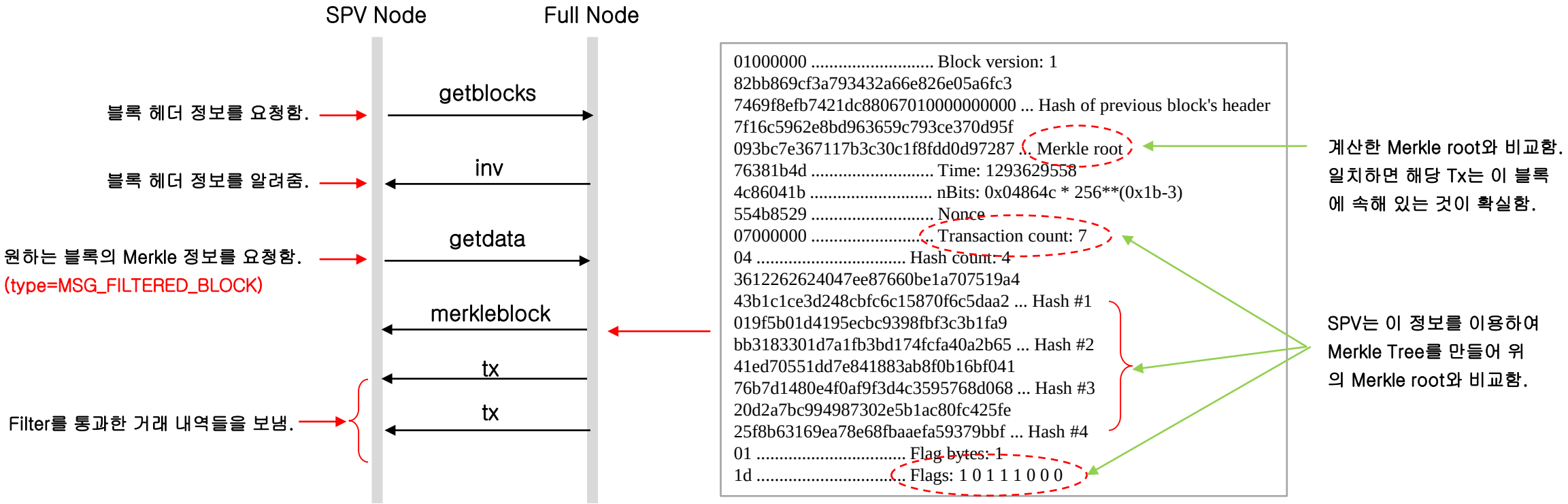
[1] seed = nHashNum * 0xfba4c795 + nTweak
nHashNum : 1 ~ nHashFuncs (sequence number)
0xfba4c795 : nHashNum 이 조금 변해도 seed가 크게 변하도록 최적화된 상수값
nTweak : SPV 노드가 추가적으로 요청하는 seed 값

[2] Filtering 대상 : 1. 현재 TXID
2. 현재 TXID의 input이 가리키는 TXID와 output index.
3. Unlock Script (ScriptSig)
4. Lock Script (ScriptPub)

Value	Name	Description
0	BLOOM_UPDATE_NONE	The filtering node should not update the filter.
1	BLOOM_UPDATE_ALL	If the filter matches any data element in a pubkey script, the corresponding outpoint is added to the filter.
2	BLOOM_UPDATE_P2PUBKEY_ONLY	If the filter matches any data element in a pubkey script and that script is either a P2PKH or non-P2SH pay-to-multisig script, the corresponding outpoint is added to the filter.

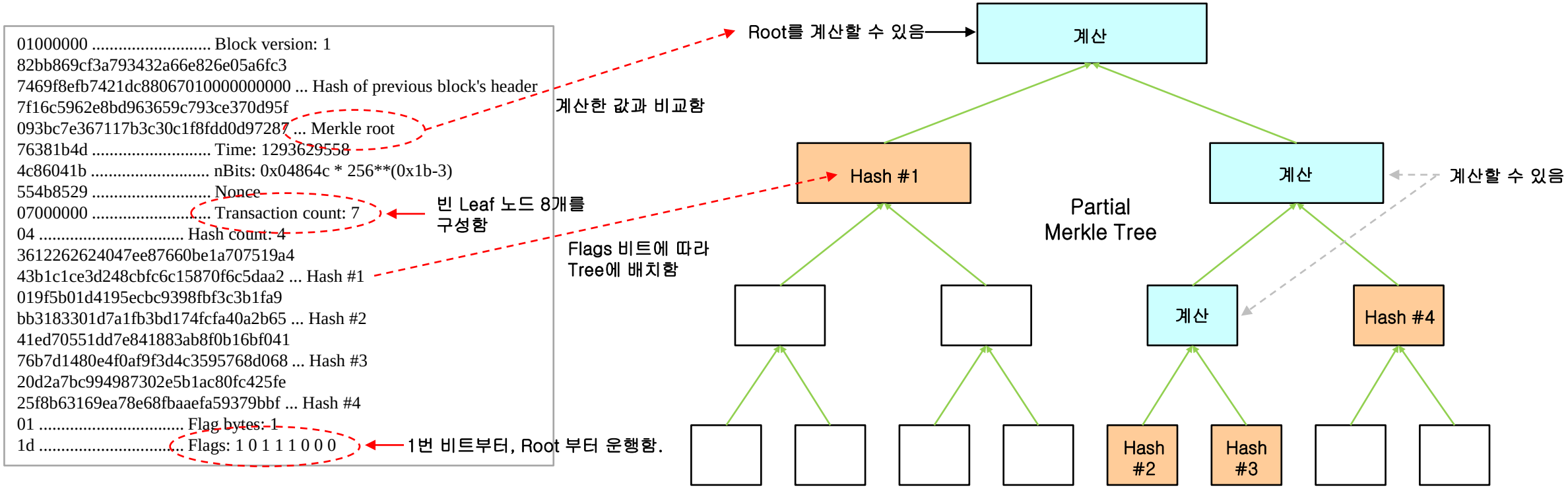
SPV 노드 : Merkle Block – BIP 37 (Connecting Bloom Filter)

- SPV 노드는 getdata 메시지를 통해 Full 노드로부터 특정 블록의 Merkle block을 받을 수 있음.
- Full 노드는 SPV 노드가 Merkle Tree를 만들어 Merkle root를 계산할 수 있도록, 관련 재료들 (merkleblock)과, Filter를 통과한 거래 내역들 (tx)을 보냄.
- SPV 노드는 Full 노드가 보낸 재료를 이용하여 간단한 형태의 Partial Merkle Tree를 만들고 Merkle root를 계산한 후 블록헤더의 Merkle root와 비교함.
- Merkle root가 일치하면 Filtering된 Tx 가 이 블록에 등록되어 있다는 것이 증명되는 것임.



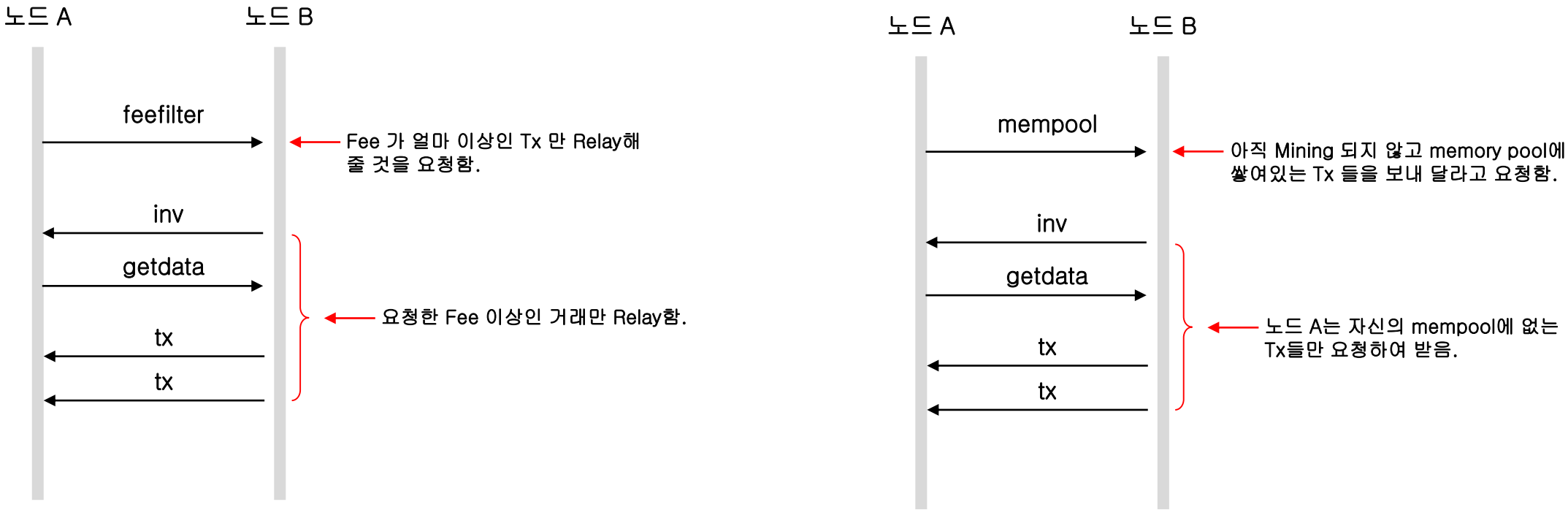
SPV 노드 : Merkle Root 계산 – BIP 37 (Connecting Bloom Filter)

- SPV 노드는 Full 노드에게 받은 merkleblock을 이용하여 아래와 같이 Partial Merkle Tree를 구축할 수 있고, Merkle root를 계산할 수 있음.
- Transaction count를 이용하여 하위 단의 (Leaf 노드) 노드 8개를 배치하고 초기화한 다음, Flags의 첫 번째 비트부터, Root에서 시작하여 Hash # 들을 배치함.
- 비트 "1" & non-leaf 이면 나중에 Hash 계산이 필요하다는 의미이고, "1" & leaf 이면 Filter를 통과한 Tx라는 의미임 (Matched Tx). "0" & non-leaf 이면 Hash # 정보를 배치하라는 의미이고, "0" & leaf 이면 Filter를 통과하지 않은 Tx 라는 의미임 (Unmatched Tx).
- 첫 번째 비트 1로, root에서 시작해서 왼쪽으로 가고 (1 & non-leaf), 두 번째 비트 0으로 Hash #1을 배치함. Hash #1 이후 하위 노드들의 정보는 알 필요 없음.
- 세 번째 비트 1로 우측 노드로 가고, 네 번째, 다섯 번째 비트 1로 왼쪽으로 가서 Hash #2를 배치함 (Matched Tx). 여섯 번째 0은 Unmatched Tx 이고 Hash #3을 배치함. 일곱 번째 비트 0으로 Hash #4를 배치하면 Root를 계산할 수 있음.



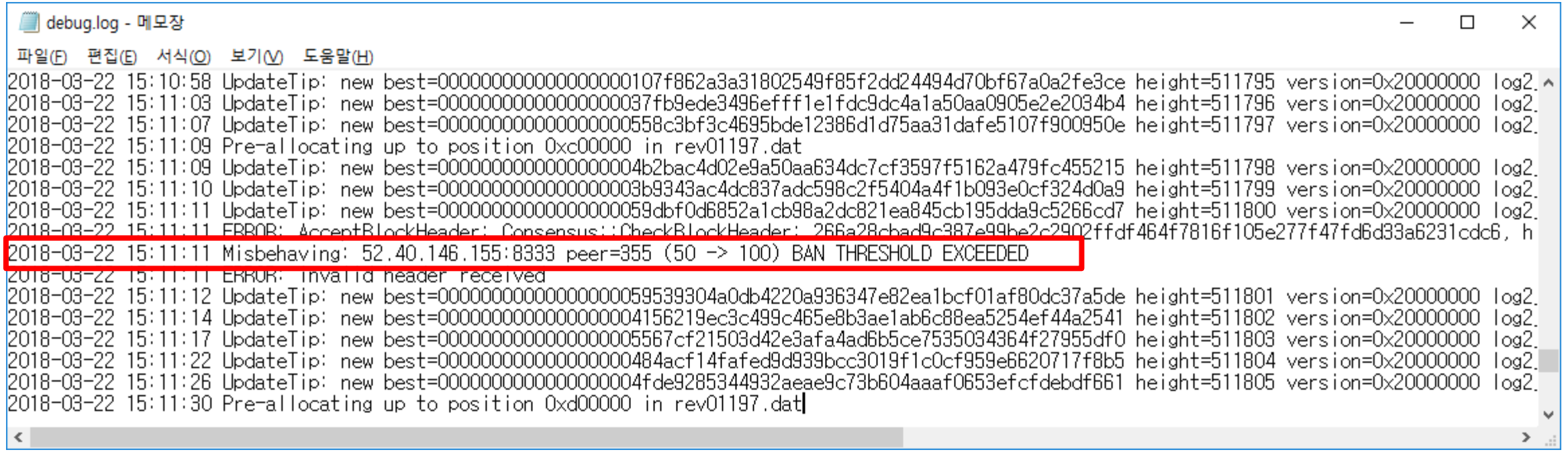
기타 메시지 : Feefilter, mempool, notfound

- 노드 A는 Feefilter를 이용하여 Fee가 얼마 이상인 Tx만 Relay해 줄 것을 요청하고, 노드 B는 Fee가 그 이상인 것만 Relay 함 (must는 아님).
- Feefilter는 Spam Tx를 방지하기 위해 사용할 수도 있고, Miner가 mining할 때도 유용하게 사용할 수도 있음.
- 만약 노드 A가 SPV이고 Bloom Filter가 세팅된 상태에서 feefilter를 요청하면 Bloom Filter가 우선함. 이것은 Merkle 증명이 더 중요하기 때문임.
- Mempool 메시지는 아직 Mining 되지 않고 memory pool에 쌓여있는 Tx 들을 받고 싶을 때 사용함. 노드 A가 mempool을 요청하면 노드 B는 자신의 memory pool에 저장된 Tx 목록들을 보내주고, 노드 A는 자신의 memory pool에 없는 Tx 들만 getdata를 요청하여 받음.
- Mempool 은 노드가 네트워크에 초기 접속할 때 유용할 수 있고, Miner 들이 Tx 들을 모을 때 유용하게 사용할 수 있음.
- Notfound 메시지는 getdata의 응답으로 요청한 데이터 (Tx, block, merkleblock) 가 없을 때 보내짐.



Penalty 부여 및 노드 차단

- 특정 노드가 악의적인 목적으로 잘못된 메시지를 송출하면 이를 받은 각 노드들은 해당 노드에게 벌점 (Penalty)을 부여함.
- 이것은 특정 노드의 부정 행위 (DoS 공격 등)로부터 비트코인 네트워크를 보호하기 위한 조치임.
- 예를 들어, 특정 노드가 부정확한 Alert 메시지를 보내면 (Alert 메시지는 2016년 Bitcoin Core에서 제외되었음) 10 점의 벌점을 부과하고, 거래 내역에 잘못된 전자서명이 있으면 매우 심각한 행위로 판단하여 100 점의 벌점을 부과함.
- 어떤 노드의 벌점이 100 점을 초과하면 해당 노드는 24 시간 동안 네트워크에서 격리 조치됨. (BAN THRESHOLD EXCEEDED)
- 아래 예시는 (Block chain data 폴더의 debug.log 파일) 노드 52.40.146.155 가 벌점 50점에서 100점으로 늘어나면서 차단된 사례임.
- 참고로 UpdateTip은 해당 블록을 다운로드 받는 과정을 보여주고 있음.



7. Bitcoin Core 설치 및 블록체인 데이터 탐색

7-1. Bitcoin Core 설치

7-2. Bitcoin Core 실행 (서버, 클라이언트)

7-3. Bitcoin Core 데이터베이스

7-4. 블록체인 데이터 탐색

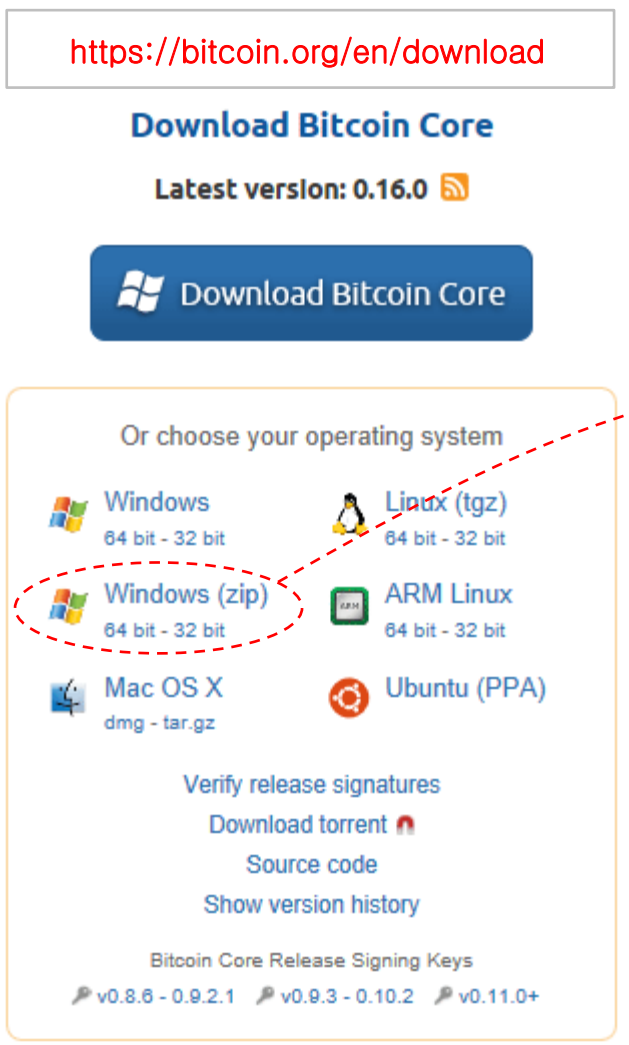
7-5. Bitcoin Core API : JSON-RPCs

7-6. JSON-RPC와 Python 연동 시험

7-7. Python 연동 실습 : 블록 데이터, 거래 데이터 조회 등

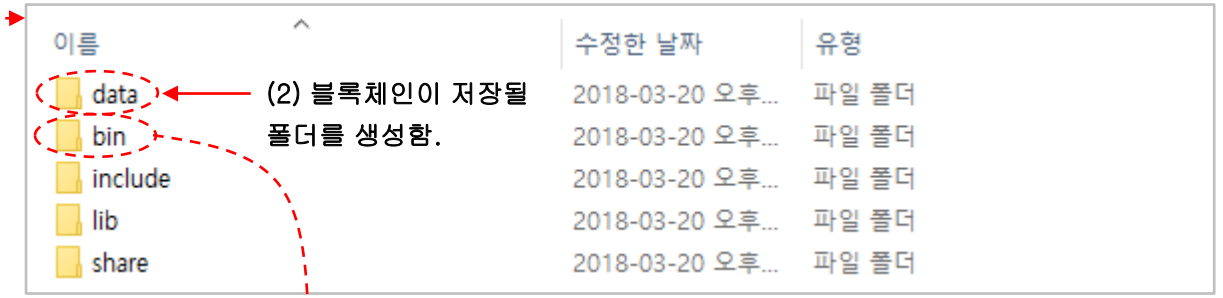
Bitcoin Core 설치

https://bitcoin.org/en/download



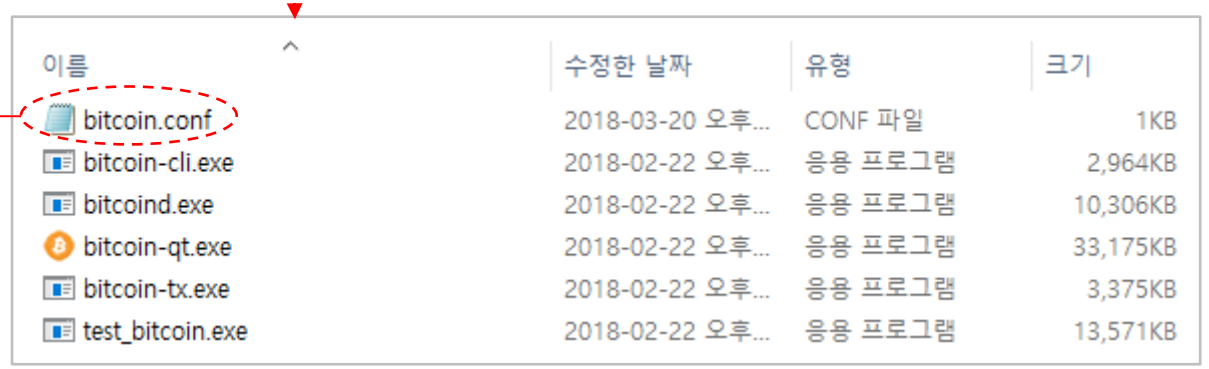
- Bitcoin Core는 기본적으로 Full node로 동작하고 블록체인 전체를 구축하므로, 약 200 GB 이상의 공간이 있는 (2018년 3월 현재 175GB) 디스크 폴더에 설치해야 함.
- (1) Bitcoin Core 프로그램을 다운받아 압축 해제한 후 (2) 블록체인 데이터가 저장될 폴더를 생성하고, (3) configuration (bitcoin.conf) 파일에 rpcuser, rpcpassword, 블록체인 데이터 폴더를 지정함. (ex : d:₩bitcoin ₩data)
- bitcoin.conf 파일은 C:₩Users₩<username>₩AppData₩Roaming₩Bitcoin 에도 복사함. 이것은 bitcoind.exe, bitcoin-cli.exe 프로그램이 그 곳에서 configuration 파일을 찾기 때문임 (default path).
- txindex = 1 은 블록내의 Tx 들을 조회할 때 필요함. Tx Hash로 검색하기 위해서는 Indexing 이 필요함.

(1) 다운로드 후 압축 해제



(2) 블록체인이 저장될 폴더를 생성함.

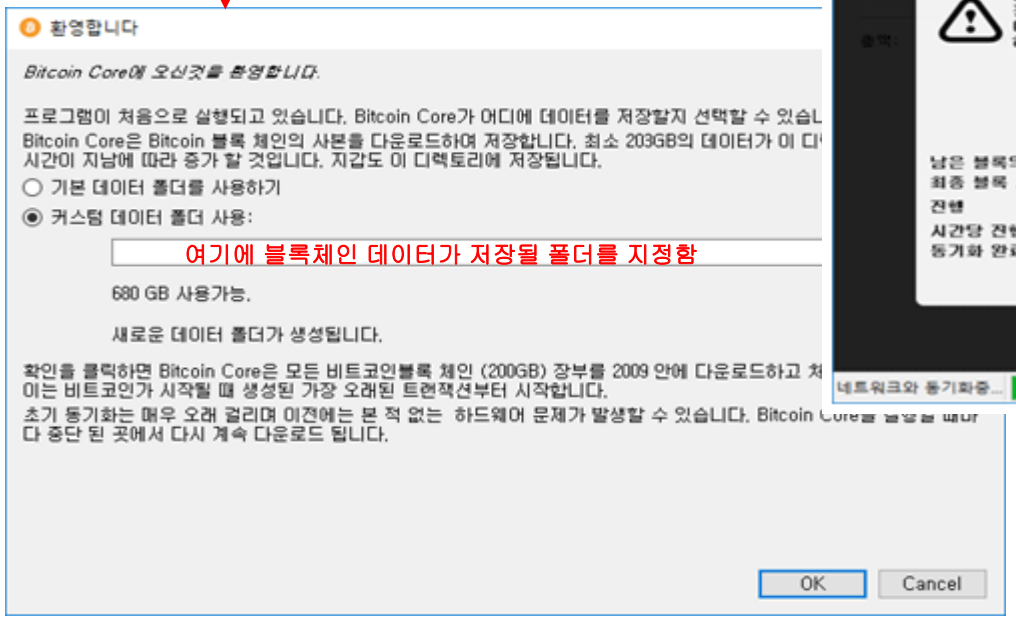
(3) configuration 파일 생성 후 C:₩Users₩<username>₩AppData₩Roaming₩Bitcoin 에 복사함.

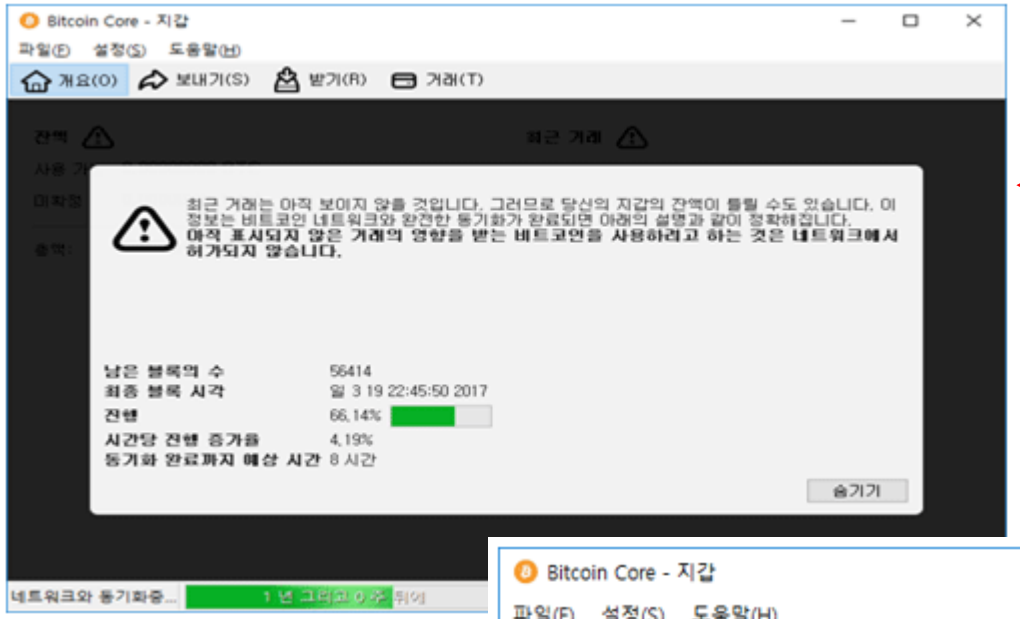



```
rpcuser=<username>
rpcpassword=<password>
datadir=d:₩Bitcoin₩data
txindex = 1
```

Bitcoin Core 실행

- bitcoin-qt.exe -server -datadir=d:₩bitcoin₩data





← 블록체인 데이터 수신 중 (2017년 3월 19일 데이터까지 전송된 상태임. 약 66.14%)

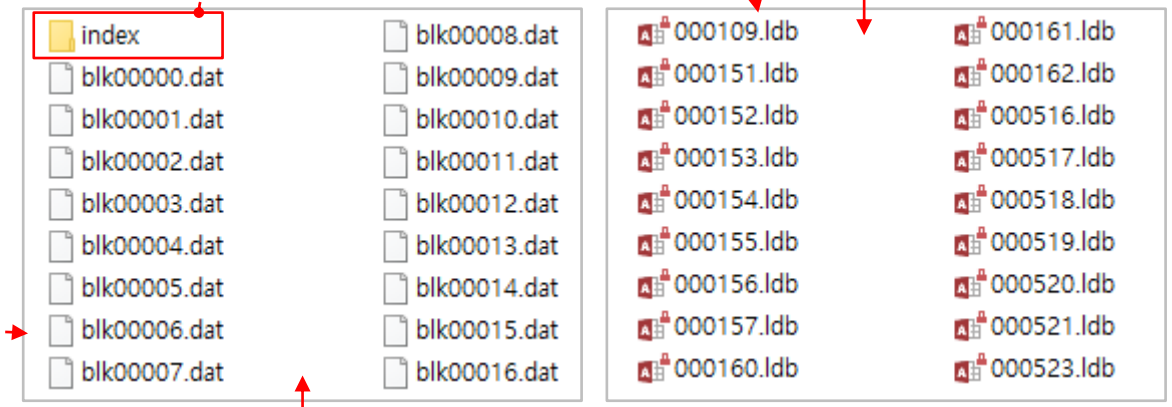
- 처음에는 블록체인 데이터가 없으므로 주변의 Full node로부터 데이터를 받아와야 함. (초기 상태는 Genesis block만 있음)
- DNS seed를 통해 주변의 Full node 접속 주소를 파악한 후 Full node에게 블록 데이터를 요청함.
- getblocks, inv, getdata, block 메시지를 통해 블록 데이터가 전송됨.
- 초기 블록체인 데이터 구축은 약 3~4일 정도 소요됨 (2018년 3월 현재)

Bitcoin Core : 데이터베이스

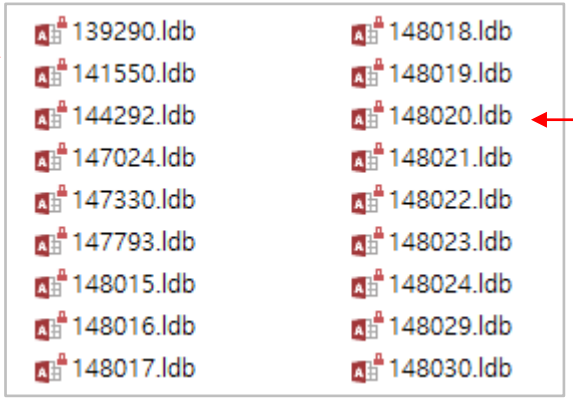
- Bitcoin Core 데이터는 아래와 같이 블록체인 데이터, 인덱스 DB, UTXO set, 지갑 데이터 등으로 구성되어 있음.
- 참고 : https://en.bitcoin.it/wiki/Bitcoin_Core_0.11_(ch_2):_Data_Storage
- 데이터베이스는 Google의 LevelDB를 사용함. (LevelDB is an open source on-disk key-value store written by Google fellows Jeffrey Dean and Sanjay Ghemawat. Inspired by Bigtable, LevelDB is hosted on GitHub under the New BSD License and has been ported to a variety of Unix-based systems, Mac OS X, Windows, and Android. https://en.wikipedia.org/wiki/LevelDB)

- Bitcoin Core의 데이터베이스 폴더

Indexing (pointer) 정보로 블록체인 Raw 데이터를 고속으로 검색할 수 있음.



블록체인 Raw 데이터. 블록 데이터, Transaction 데이터가 모두 기록되어 있음.



- LevelDB 데이터베이스. Unspent Transaction Output (UTXO) Set와 이전 Transaction들의 Chain 데이터베이스.
- 신규 Block과 Transaction을 검증할 때 사용됨.
- 신규 Block이 생성될 때마다 UTXO set이 변경됨. Spent output은 삭제되고, 새로운 Unspent Output은 등록됨.

Bitcoin Core : 블록체인 데이터 탐색

- Bitcoin Core는 지갑 생성, 거래 등 다양한 기능을 수행하지만, 본 과정에서는 블록체인 탐색 (Blockchain explorer) 방법에 대해 살펴보기로 함.
- 블록체인을 탐색하려면 아래와 같이 Bitcoin Core 서버, 클라이언트 환경이 필요함.
- 서버는 bitcoin-qt -server -datadir=d:₩bitcoin₩data 로 실행해도 되고, 아래와 같이 bitcoind 를 실행해도 됨. bitcoind로 실행한 경우는 클라이언트 측에서 bitcoin-cli stop 명령으로 서버를 종료 시킴.

```
D:₩Bitcoin>bitcoind -datadir=d:₩bitcoin₩data -printtoconsole
2018-03-23 02:50:50 Bitcoin Core version v0.16.0 (release build)
2018-03-23 02:50:50 InitParameterInteraction: parameter interaction: -whitelistforcerelay=1 -> setting -whitelistrela
2018-03-23 02:50:50 Assuming ancestors of block 0000000000000000005214481d2d96f898e3d5416e43359c145944a909d242e0 have
2018-03-23 02:50:50 Setting nMinimumChainWork=0000000000000000000000000000000000000000000000000f91c579d57cad4bc5278cc
2018-03-23 02:50:50 Using the 'sse4' SHA256 implementation
2018-03-23 02:50:50 Using RdRand as an additional entropy source
2018-03-23 02:50:51 Default data directory C:₩Users₩seong₩AppData₩Roaming₩Bitcoin
2018-03-23 02:50:51 Using data directory d:₩bitcoin₩data
2018-03-23 02:50:51 Using config file d:₩bitcoin₩data₩bitcoin.conf
2018-03-23 02:50:51 Using at most 125 automatic connections (2048 file descriptors available)
2018-03-23 02:50:51 Using 16 MiB out of 32/2 requested for signature cache, able to store 524288 elements
2018-03-23 02:50:51 Using 16 MiB out of 32/2 requested for script execution cache, able to store 524288 elements
2018-03-23 02:50:51 Using 4 threads for script verification
2018-03-23 02:50:51 scheduler thread start
2018-03-23 02:50:51 HTTP: creating work queue of depth 16
2018-03-23 02:50:51 Config options rpcuser and rpcpassword will soon be deprecated. Locally-run instances may remove
2018-03-23 02:50:51 HTTP: starting 4 worker threads
.............

2018-03-23 02:51:01 net thread start
2018-03-23 02:51:01 dnsseed thread start
2018-03-23 02:51:01 addcon thread start
2018-03-23 02:51:01 opencon thread start
2018-03-23 02:51:01 msghand thread start
2018-03-23 02:51:02 Imported mempool transactions from disk: 1462 succeeded, 0 failed, 0 expire
2018-03-23 02:51:09 New outbound peer connected: version: 70015, blocks=514751, peer=1
2018-03-23 02:51:12 Loading addresses from DNS seeds (could take a while)
2018-03-23 02:51:13 140 addresses found from DNS seeds
2018-03-23 02:51:13 dnsseed thread exit
2018-03-23 02:51:15 New outbound peer connected: version: 70015, blocks=514751, peer=2
```

```
D:₩Bitcoin>bitcoin-cli -getinfo
{
  "version": 160000,
  "protocolversion": 70015,
  "walletversion": 159900,
  "balance": 0.00000000,
  "blocks": 514751,
  "timeoffset": -6,
  "connections": 9,
  "proxy": "",
  "difficulty": 3462542391191.563,
  "testnet": false,
  "keypoololdest": 1521473115,
  "keypoolsize": 1000,
  "paytxfee": 0.00000000,
  "relayfee": 0.00001000,
  "warnings": ""
}

D:₩Bitcoin>bitcoin-cli getblockcount
514751

D:₩Bitcoin>
```

현재 총 514,751 블록이 있음.

Bitcoin Core : 블록체인 데이터 탐색

- 특정 블록의 거래 내역 확인. (getblockhash → getblock → getrawtransaction)

```
D:₩Bitcoin>bitcoin-cli getblockhash 514747
000000000000000004a17ad7123b0eae9c15cabf7951ad24c44dba46ec20606

D:₩Bitcoin>bitcoin-cli getblock 000000000000000004a17ad7123b0eae9c15cabf7951ad24
{
  "hash": "000000000000000004a17ad7123b0eae9c15cabf7951ad24c44dba46ec20606",
  "confirmations": 5,
  "strippedsize": 952947,
  "size": 1133874,
  "weight": 3992715,
  "height": 514747,
  "version": 536870912,
  "versionHex": "20000000",
  "merkleroot": "61e5043078668d32cdb4bf96a072aa1b63054d748a0d45da30003f7fdd754f76",
  "tx": [
    "36c2cd3ee8e36b2706121236837bacca1d0caeb8ef9251dcdc02a5726f168660",
    "22301c85d444168e9e03e61ec260fb394cb8ccd73805d53fda0e55eb7eebbcec",
    "e7779a5632dbaf74e26ac584d0f6b10341579d990a05a1287981ec61bffaedda",
    "79710e2dee95a6cb39dfe9e0bb5ea7576622ca6577f62d6937ec0d3b0eed3565",
    "5c0df5529c629bb8b758a019852472bc6922c432e38b58000b231a00e74ee5f0",
    "f0bcad43666cddc74e08268ed610ae788dee465b53cc294d05bd0bebcf6f7f34",
    "c3c0c50850cfad0355c895dfe5478730f50d104feaee0a736b45d0083234840a",
    "51759f670a30d46723f0a2c0a76071898f69c029758d4c711b6c38edfa86dd8e",
     ......

    "66271f31ded696b89376c4256636334bd912c8607b4655b306923af0acb84a88",
    "b8f819634f964fa8e3c2ef0ca28efcca361a16d0a9cd92dced088ffe1619558d",
    "7e6edbf895de162b4cb4b8c8566c3a8d433a860c8fbba065caa62727de4f5894"
  ],
  "time": 1521770606,
  "mediantime": 1521765712,
  "nonce": 1611685489,
  "bits": "17514a49",
  "difficulty": 3462542391191.563,
  "chainwork": "0000000000000000000000000000000000000000000155308ae9c4c292400ffc54",
  "previousblockhash": "000000000000000048e6d5fc56c1eb43530236193bd8bec8101caf8
  "nextblockhash": "000000000000000005d9276e028d5d5bed13951c19ec762fd4b1b592cc0
}
```

이 거래내역을 조회함.

```
D:₩Bitcoin>bitcoin-cli getrawtransaction 22301c85d444168e9e03e61ec260fb394cb8ccd7
{
  "txid": "22301c85d444168e9e03e61ec260fb394cb8ccd73805d53fda0e55eb7eebbcec",
  "hash": "22301c85d444168e9e03e61ec260fb394cb8ccd73805d53fda0e55eb7eebbcec",
  "version": 1,
  "size": 339,
  "vsize": 339,
  "locktime": 0,
  "vin": [
    {
      "txid": "06c53a97795ec2e853692ca5776c1c1791003adb35a5bc121da11edfe26bd9da",
      "vout": 1,
      "scriptSig": {
        "asm": "3045022100c4fca31afccbc7c4dac7351d8f2722c43aef1bd11a062f9f72d0f49
        "hex": "483045022100c4fca31afccbc7c4dac7351d8f2722c43aef1bd11a062f9f72d0f
      },
      "sequence": 4294967295
    },
    {
      "txid": "728120b3b7ce8fa084ddfb97beb4883724aefa2cdea687a2feba016054dac071",
      "vout": 0,
      "scriptSig": {
        "asm": "3044022051567e98ee4cc1a3d404f17ee5aaef9e206a278b71354bb263918bc8f
        "hex": "473044022051567e98ee4cc1a3d404f17ee5aaef9e206a278b71354bb263918bc
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.03032331,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 a7f5e8adfee34be0589d43ef48d71444bb1ea3b7 OP_EQU
        "hex": "76a914a7f5e8adfee34be0589d43ef48d71444bb1ea3b788ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
```

Bitcoin Core API : Remote Procedure Calls (RPCs)

- Bitcoin Core는 JSON-RPC 인터페이스를 제공하므로, 외부 프로그래밍 언어 (ex : Python)와 연동 가능함.
- Block Chain RPCs :

(세부 내용 참조 : https://bitcoin.org/en/developer-reference#remote-procedure-calls-rpcs)

Methods	Description
GetBestBlockHash	returns the header hash of the most recent block on the best block chain.
GetBlock	gets a block with a particular header hash from the local block database either as a JSON object or as a serialized block. Updated in 0.13.0
GetBlockChainInfo	provides information about the current state of the block chain. Updated in 0.12.1
GetBlockCount	returns the number of blocks in the local best block chain.
GetBlockHash	returns the header hash of a block at the given height in the local best block chain.
GetBlockHeader	gets a block header with a particular header hash from the local block database either as a JSON object or as a serialized block header. New in 0.12.0
GetChainTips	returns information about the highest-height block (tip) of each local block chain.
GetDifficulty	returns the proof-of-work difficulty as a multiple of the minimum difficulty.
GetMemPoolAncestors	returns all in-mempool ancestors for a transaction in the mempool. New in 0.13.0
GetMemPoolDescendants	returns all in-mempool descendants for a transaction in the mempool. New in 0.13.0
GetMemPoolEntry	returns mempool data for given transaction (must be in mempool). New in 0.13.0
GetMemPoolInfo	returns information about the node's current transaction memory pool. Updated in 0.12.0
GetRawMemPool	returns all transaction identifiers (TXIDs) in the memory pool as a JSON array, or detailed information about each transaction in the memory pool as a JSON object. Updated in 0.13.0
GetTxOut	returns details about a transaction output. Only unspent transaction outputs (UTXOs) are guaranteed to be available.
GetTxOutProof	returns a hex-encoded proof that one or more specified transactions were included in a block. New in 0.11.0
GetTxOutSetInfo	returns statistics about the confirmed unspent transaction output (UTXO) set. Note that this call may take some time and that it only counts outputs from confirmed transactions—it does not count outputs from the memory pool.
PreciousBlock	treats a block as if it were received before others with the same work. New in 0.14.0
PruneBlockChain	prunes the blockchain up to a specified height or timestamp. New in 0.14.0
VerifyChain	verifies each entry in the local block chain database.
VerifyTxOutProof	verifies that a proof points to one or more transactions in a block, returning the transactions the proof commits to and throwing an RPC error if the block is not in our best block chain. New in 0.11.0

Bitcoin Core API : Remote Procedure Calls (RPCs)

Control RPCs

Methods	Description
GetInfo	prints various information about the node and the network. Deprecated
Help	lists all available public RPC commands, or gets help for the specified RPC. Commands which are unavailable will not be listed, such as wallet RPCs if wallet support is disabled.
Stop	safely shuts down the Bitcoin Core server.

Mining RPCs

Methods	Description
GetBlockTemplate	gets a block template or proposal for use with mining software.
GetMiningInfo	returns various mining-related information. Updated in 0.14.0
GetNetworkHashPS	returns the estimated current or historical network hashes per second based on the last n blocks.
PrioritiseTransaction	adds virtual priority or fee to a transaction, allowing it to be accepted into blocks mined by this node (or miners which use this node) with a lower priority or fee. (It can also remove virtual priority or fee, requiring the transaction have a higher priority or fee to be accepted into a locally-mined block.)
SubmitBlock	accepts a block, verifies it is a valid addition to the block chain, and broadcasts it to the network. Extra parameters are ignored by Bitcoin Core but may be used by mining pools or other programs.

Network RPCs

Methods	Description
AddNode	attempts to add or remove a node from the addnode list, or to try a connection to a node once. Updated in 0.14.0
ClearBanned	clears list of banned nodes. New in 0.12.0
DisconnectNode	immediately disconnects from a specified node. New in 0.12.0
GetAddedNodeInfo	returns information about the given added node, or all added nodes (except onetry nodes). Only nodes which have been manually added using the addnode RPC will have their information displayed. Updated in 0.14.0
GetConnectionCount	returns the number of connections to other nodes.
GetNetTotals	returns information about network traffic, including bytes in, bytes out, and the current time. Updated in 0.12.0
GetNetworkInfo	returns information about the node's connection to the network. Updated in 0.13.0
GetPeerInfo	returns data about each connected network node. Updated in 0.13.0
ListBanned	lists all banned IPs/Subnets. New in 0.12.0
Ping	sends a P2P ping message to all connected nodes to measure ping time. Results are provided by the getpeerinfo RPC pingtime and pingwait fields as decimal seconds. The P2P ping message is handled in a queue with all other commands, so it measures processing backlog, not just network ping.
SetBan	attempts add or remove a IP/Subnet from the banned list. New in 0.12.0
SetNetworkActive	disables/enables all P2P network activity. New in 0.14.0

Bitcoin Core API : Remote Procedure Calls (RPCs)

Raw Transaction PRCs

Methods	Description
CreateRawTransaction	creates an unsigned serialized transaction that spends a previous output to a new output with a P2PKH or P2SH address. The transaction is not stored in the wallet or transmitted to the network.
FundRawTransaction	adds inputs to a transaction until it has enough in value to meet its out value. New in 0.12.0, Updated in 0.14.0
DecodeRawTransaction	decodes a serialized transaction hex string into a JSON object describing the transaction. Updated in 0.13.0
DecodeScript	decodes a hex-encoded P2SH redeem script.
GetRawTransaction	gets a hex-encoded serialized transaction or a JSON object describing the transaction. By default, Bitcoin Core only stores complete transaction data for UTXOs and your own transactions, so the RPC may fail on historic transactions unless you use the non-default txindex=1 in your Bitcoin Core startup settings. Updated in 0.14.0
SendRawTransaction	validates a transaction and broadcasts it to the peer-to-peer network.
SignRawTransaction	signs a transaction in the serialized transaction format using private keys stored in the wallet or provided in the call.

Utility RPCs

Methods	Description
CreateMultiSig	creates a P2SH multi-signature address.
EstimateFee	estimates the transaction fee per kilobyte that needs to be paid for a transaction to be included within a certain number of blocks. Updated in 0.14.0
EstimatePriority	estimates the priority that a transaction needs in order to be included within a certain number of blocks as a free high-priority transaction. Deprecated
GetMemoryInfo	returns information about memory usage.
ValidateAddress	returns information about the given Bitcoin address. Updated in 0.13.0
VerifyMessage	verifies a signed message.

Bitcoin Core API : Remote Procedure Calls (RPCs)

Wallet RPCs

Methods	Description
AbandonTransaction	marks an in-wallet transaction and all its in-wallet descendants as abandoned. This allows their inputs to be respent. New in 0.12.0
AddWitnessAddress	adds a witness address for a script (with pubkey or redeemscript known). New in 0.13.0
AddMultiSigAddress	adds a P2SH multisig address to the wallet.
BackupWallet	safely copies wallet.dat to the specified file, which can be a directory or a path with filename.
BumpFee	replaces an unconfirmed wallet transaction that signaled RBF with a new transaction that pays a higher fee. New in 0.14.0
DumpPrivKey	returns the wallet-import-format (WIP) private key corresponding to an address. (But does not remove it from the wallet.)
DumpWallet	creates or overwrites a file with all wallet keys in a human-readable format.
EncryptWallet	encrypts the wallet with a passphrase. This is only to enable encryption for the first time. After encryption is enabled, you will need to enter the passphrase to use private keys.
GetAccountAddress	returns the current Bitcoin address for receiving payments to this account. If the account doesn't exist, it creates both the account and a new address for receiving payment. Once a payment has been received to an address, future calls to this RPC for the same account will return a different address. Deprecated
GetAccount	returns the name of the account associated with the given address.
GetAddressesByAccount	returns a list of every address assigned to a particular account. Deprecated
GetBalance	gets the balance in decimal bitcoins across all accounts or for a particular account.
GetNewAddress	returns a new Bitcoin address for receiving payments. If an account is specified, payments received with the address will be credited to that account.
GetRawChangeAddress	returns a new Bitcoin address for receiving change. This is for use with raw transactions, not normal use.
GetReceivedByAccount	returns the total amount received by addresses in a particular account from transactions with the specified number of confirmations. It does not count coinbase transactions. Deprecated
GetReceivedByAddress	returns the total amount received by the specified address in transactions with the specified number of confirmations. It does not count coinbase transactions.
GetTransaction	gets detailed information about an in-wallet transaction. Updated in 0.12.0
GetUnconfirmedBalance	returns the wallet's total unconfirmed balance.
GetWalletInfo	provides information about the wallet.
ImportAddress	adds an address or pubkey script to the wallet without the associated private key, allowing you to watch for transactions affecting that address or pubkey script without being able to spend any of its outputs.
ImportMulti	imports addresses or scripts (with private keys, public keys, or P2SH redeem scripts) and optionally performs the minimum necessary rescan for all imports. New in 0.14.0
ImportPrunedFunds	imports funds without the need of a rescan. Meant for use with pruned wallets. New in 0.13.0
ImportPrivKey	adds a private key to your wallet. The key should be formatted in the wallet import format created by the dumpprivkey RPC.

Bitcoin Core API : Remote Procedure Calls (RPCs)

Wallet RPCs

Methods	Description
ImportWallet	imports private keys from a file in wallet dump file format (see the dumpwallet RPC). These keys will be added to the keys currently in the wallet. This call may need to rescan all or parts of the block chain for transactions affecting the newly-added keys, which may take several minutes.
KeyPoolRefill	fills the cache of unused pre-generated keys (the keypool).
ListAccounts	lists accounts and their balances. Deprecated
ListAddressGroupings	lists groups of addresses that may have had their common ownership made public by common use as inputs in the same transaction or from being used as change from a previous transaction.
ListLockUnspent	returns a list of temporarily unspendable (locked) outputs.
ListReceivedByAccount	lists the total number of bitcoins received by each account. Deprecated
ListReceivedByAddress	lists the total number of bitcoins received by each address.
ListSinceBlock	gets all transactions affecting the wallet which have occurred since a particular block, plus the header hash of a block at a particular depth.
ListTransactions	returns the most recent transactions that affect the wallet. Updated in 0.12.1
ListUnspent	returns an array of unspent transaction outputs belonging to this wallet. Updated in 0.13.0
LockUnspent	temporarily locks or unlocks specified transaction outputs. A locked transaction output will not be chosen by automatic coin selection when spending bitcoins. Locks are stored in memory only, so nodes start with zero locked outputs and the locked output list is always cleared when a node stops or fails.
Move	moves a specified amount from one account in your wallet to another using an off-block-chain transaction. Deprecated
RemovePrunedFunds	deletes the specified transaction from the wallet. Meant for use with pruned wallets and as a companion to importprunedfunds. New in 0.13.0
SendFrom	spends an amount from a local account to a bitcoin address. Deprecated
SendMany	creates and broadcasts a transaction which sends outputs to multiple addresses.
SendToAddress	spends an amount to a given address.
SetAccount	puts the specified address in the given account. Deprecated
SetTxFee	sets the transaction fee per kilobyte paid by transactions created by this wallet.
SignMessage	signs a message with the private key of an address.
SignMessageWithPrivKey	signs a message with a given private key. New in 0.13.0
WalletLock	removes the wallet encryption key from memory, locking the wallet. After calling this method, you will need to call walletpassphrase again before being able to call any methods which require the wallet to be unlocked.
WalletPassphrase	stores the wallet decryption key in memory for the indicated number of seconds. Issuing the walletpassphrase command while the wallet is already unlocked will set a new unlock time that overrides the old one.
WalletPassphraseChange	changes the wallet passphrase from 'old passphrase' to 'new passphrase'.

(실습 파일 : 7-1.GetBlockChainInfo.py)

Bitcoin Core API : Remote Procedure Calls (RPCs) – Python 연습

- getblockchaininfo method로 현재 Block chain의 상태를 확인함.

```
1  # Bitcoin Core API : JSON-RPC 기능 시험
2  #
3  # Block chain의 현재 상태를 확인한다 (getblockchaininfo)
4  #
5  # 패키지 : https://github.com/petertodd/python-bitcoinlib (by Peter Todd)
6  #
7  # 2018.5.8 아마추어 퀀트 (조성현)
8  # ------------------------------------------------------------------------
9  from bitcoin.rpc import RawProxy
10 import pprint
11 pp = pprint.PrettyPrinter(indent=1)
12
13 # Bitcoin Core에 접속한다.
14 p = RawProxy()
15
16 # Blockchain 정보를 읽어온다
17 info = p.getblockchaininfo()
18
19 print("\nBlockchain Info :")
20 pp.pprint(info)
21
22
23
```

```
Blockchain Info :
{'bestblockhash': '00000000000000000003283d202148127d07759ea1d1c403869bee973016c12dd',
 'bip9_softforks': {'csv': {'since': 419328,
                            'startTime': 1462060800,
                            'status': 'active',
                            'timeout': 1493596800},
                    'segwit': {'since': 481824,
                               'startTime': 1479168000,
                               'status': 'active',
                               'timeout': 1510704000}},
 'blocks': 521632,
 'chain': 'main',
 'chainwork': '0000000000000000000000000000000000000001afec8b18ba5fafe1d2f950',
 'difficulty': Decimal('4022059196164.954'),
 'headers': 521632,
 'initialblockdownload': False,
 'mediantime': 1525705259,
 'pruned': False,
 'size_on_disk': 190837929245,
 'softforks': [{'id': 'bip34', 'reject': {'status': True}, 'version': 2},
               {'id': 'bip66', 'reject': {'status': True}, 'version': 3},
               {'id': 'bip65', 'reject': {'status': True}, 'version': 4}],
 'verificationprogress': Decimal('0.9999978769366572'),
 'warnings': ''}

In [46]: info['bestblockhash']
Out[46]: '00000000000000000003283d202148127d07759ea1d1c403869bee973016c12dd'

In [47]:
```

History log | IPython console

(실습 파일 : 7-2.GetBlock.py)

Bitcoin Core API : Remote Procedure Calls (RPCs) – Python 연습

- 블록체인의 마지막 블록의 내용을 조회함. (블록 헤더 + 거래 내역)

```python
# Bitcoin Core API : JSON-RPC 기능 시험
#
# Block chain의 마지막 블록의 내용을 조회한다.
#
# 패키지 : https://github.com/petertodd/python-bitcoinlib (by Pet
#
# 2018.5.8 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------
from bitcoin.rpc import RawProxy
import pprint
pp = pprint.PrettyPrinter(indent=1)

# Bitcoin Core에 접속한다.
p = RawProxy()

# 블록체인 tip (끝 부분)의 hash, height를 읽어온다
tip = p.getchaintips()

# 블록 height
height = tip[0]['height']
bHash = tip[0]['hash']

# 블록 번호 = height 의 블록 해시 값을 읽어온다
# bHash = p.getblockhash(height)

# 블록 정보를 읽어온다
block = p.getblock(bHash)
pp.pprint(block)
```

```
In [62]: block
Out[62]:
{'bits': '1745fb53',
 'chainwork': '0000000000000000000000000000000000000000001b00d775a56787890f18740',
 'confirmations': 1,
 'difficulty': Decimal('4022059196164.954'),
 'hash': '00000000000000000034d95f1c12643751a5a1e53c958fe3bad7c8d5ea4a4a57',
 'height': 521641,
 'mediantime': 1525711853,
 'merkleroot': 'eb5faa82bd823494200838878bb4d70f81b929fe484e2e3d6c10d51e9efb27cf',
 'nonce': 2431299782,
 'previousblockhash': '00000000000000000003c78ee53fa7976cbdb16217358f16b7f8c6027b5754520',
 'size': 1111304,
 'strippedsize': 960634,
 'time': 1525714046,
 'tx': ['7c20bc46fe8d592f3cdb5dbdf3d45ecde6d48b6c72b5c3738d2b151ffdb4081a',
  'a28eee28a2f968c548936214533476f44c8560389e862b6c42422555e6c090b6',
  '999560dccc8147937f98836cbc97ef022bb410db0c68a788b54f23019c50e0b1',
  '875fe2de585b6d03615c5819e6af885eed4eaf616e36aba868b36f57860de7d0',
  '6654d5af254d160cd5f69ed745b7cadc0e299c1c04dcdbc0a7f04e731b94c4c6',
  '5690dfcbfbce16086e0b218c715e102c0d710bc67c0d55f165b0ef1382f4a3d8',
  '25942ec97b54abdb95f92ff66d2d76b07d3374c80e36f235c6806643c9137bb5',
  '72454d168370b43d9b2942a9b3cdb5fd61f7dbaa1bdc74775598a5046c959314',
  '052a64144da40a69dfbc0a1931bd723dcc0402da7ba2b1e825b09357174cebe9',
  '1ed7583c5120dd5f63d3bb5aa0393ac94176d8e816b0dbcb91f9b096a0838d22',
  '3b8b4b002aedb93b83f2f53de1e45e320858aeb52ff3ca2cd6cd48924bb0cbc4',
  'fd687303b61d6ee497fb58f5e61cea8a3886fbced283cd7a0b08cc3e56afe4d7',
  'e68d2e27066e4a8bb6d1afd4b2e775bafd542d6d8318e6d4daa301cbd26e7200',
  '76b9f3af395f663d136480d3e01335724ae58755a42b63cc3a087ad38bc4131b',
  '11fcf1fba1cce61e57a4308a1aff19317ab37a668b5dd8f60141963a4222177a',
  'de5991ecf2aff8158fd2e9d5cba0de5bbf5894b406ed32011ea1180f66821764',
```

History log | IPython console

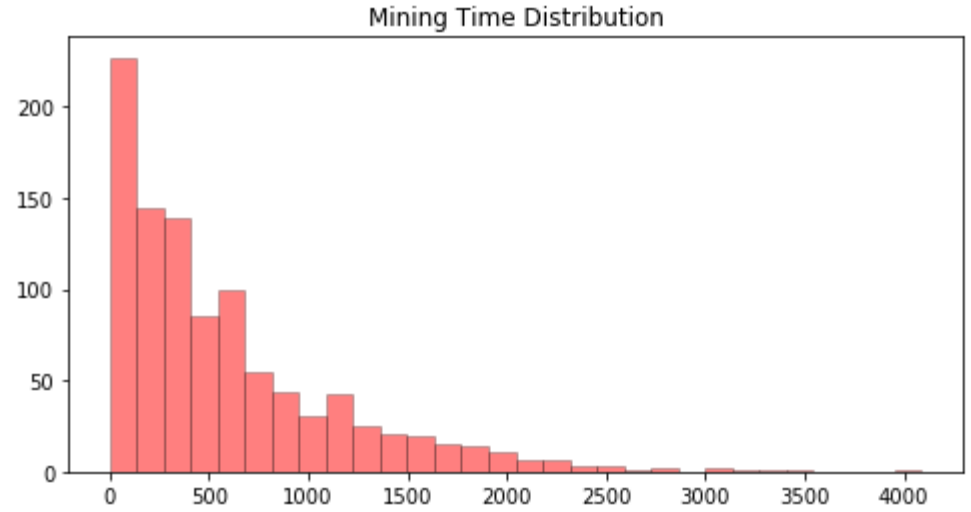
(실습 파일 : 7-3.GetHeader.py)

Bitcoin Core API : Remote Procedure Calls (RPCs) – Python 연습

- 블록체인의 최근 블록 헤더 1,000개를 조회하고, 블록 생성 시간의 분포를 확인함. 평균 = 586초, 표준편차 = 581초.

```
# Bitcoin Core API : JSON-RPC 기능 시험
#
# Block chain의 마지막 블록 헤더 1000개를 조회하고, 블록 생성 시간의 분포를
# 확인한다. 블록 생성 시간의 평균은 10분이겠지만 10분 이하나, 이상의 분포를
# 확인하고, 10분 이내에 거래가 Mining될 확률을 계산해 본다.
#
# 패키지 : https://github.com/petertodd/python-bitcoinlib (by Peter Todd)
#
# 2018.5.8 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
from bitcoin.rpc import RawProxy
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Bitcoin Core에 접속한다.
p = RawProxy()

# 블록체인의 블록 개수을 읽어온다
n = p.getblockcount()

# 최근 100개 블록의 헤더를 읽어서 생성 시간을 조회한다.
header = []
for i in range(n - 999, n+1):
    bHash = p.getblockhash(i)
    hdr = p.getblockheader(bHash)
    height = hdr['height']
    btime = hdr['time']
    bhash = hdr['hash']
    header.append([height, btime, bhash])
```

```
995   521638   1525712780   000000000000000000104159377 60dda189ea41f636823...
996   521639   1525712955   00000000000000000000213b22c53b4beaca8dd4c2906f4...
997   521640   1525713801   000000000000000000003c78ee53fa7976cbdb16217358f1...
998   521641   1525714046   00000000000000000034d95f1c12643751a5a1e53c958f...
999   521642   1525715117   000000000000000000000665dc6731c3f31eafb225c8ffc1...

[1000 rows x 3 columns]
총 1000 개 블록 헤더를 읽어왔습니다.
평균 Mining 시간 = 586 (초)
표준편차 = 581 (초)
```




```
10분 이내에 내 거래가 Mining될 확률 = 64.05 (%)

In [80]:
```

History log | IPython console

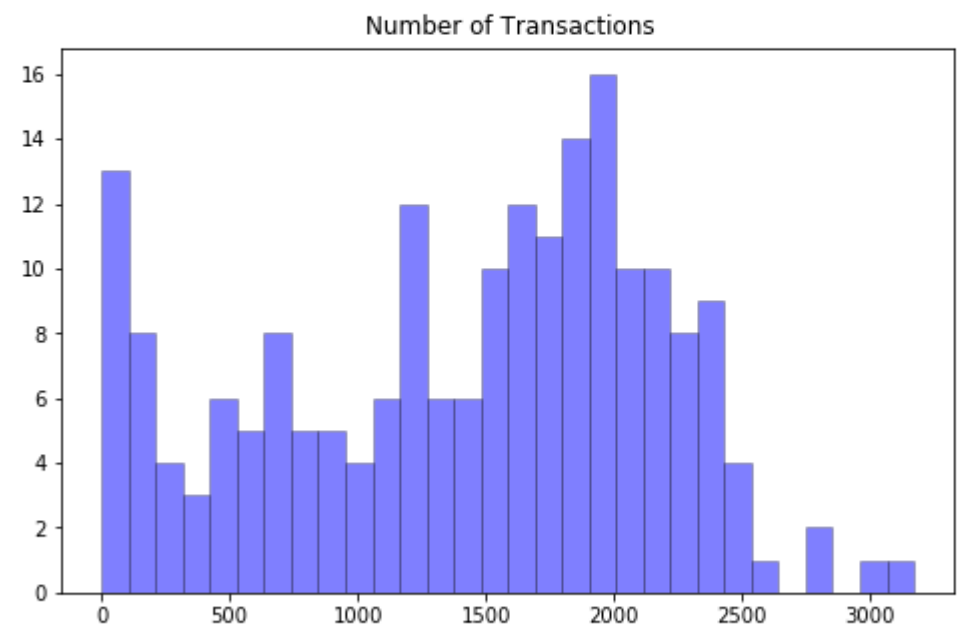
(실습 파일 : 7-4.TxDistribution.py)

Bitcoin Core API : Remote Procedure Calls (RPCs) – Python 연습

- 블록체인의 최근 블록 200개를 조회하고, 블록에 포함된 Transaction의 분포를 확인한다. 평균 Transaction = 1,415 개

```python
# Bitcoin Core API : JSON-RPC 기능 시험
#
# Block chain의 마지막 블록 헤더 200개를 조회하고, 블록 당 Transaction 개수의
# 분포를 확인한다.
#
# 패키지 : https://github.com/petertodd/python-bitcoinlib (by Peter Todd)
#
# 2018.5.8 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
from bitcoin.rpc import RawProxy
import matplotlib.pyplot as plt
import numpy as np

# Bitcoin Core에 접속한다.
p = RawProxy()

# 블록체인의 블록 개수을 읽어온다
n = p.getblockcount()

# 최근 200개 블록을 읽어서 Transaction 개수를 확인한다
nTx = []
for i in range(n - 199, n+1):
    bHash = p.getblockhash(i)
    block = p.getblock(bHash)
    nTx.append(len(block['tx']))

print(nTx)
print("\nTransaction 평균 = %.2f" % np.mean(nTx))

# Histogram을 그려 본다
plt.figure(figsize=(8,5))
```

```
452, 1039, 1680, 701, 186, 83, 1418, 1832, 979, 1, 286, 453, 1990, 2370, 2010,
1870, 1069, 1921, 668, 1526, 525, 1756, 79, 515, 1328, 1172, 1139, 1624, 18,
1409, 1877, 1634, 1991, 1911, 1569, 2150, 1208, 1987, 657, 749, 1222, 1119, 1150,
1731, 1799, 1475, 1584, 1594, 2061, 1862, 2143, 2309, 1729, 1779, 1996, 2034,
2017, 2347, 2805, 2435, 2022, 2972, 2370, 1870, 2332, 2074, 1943, 1766, 2113,
2136, 1542, 1230]

Transaction 평균 = 1415.12
```




```
In [4]:
```

History log | IPython console

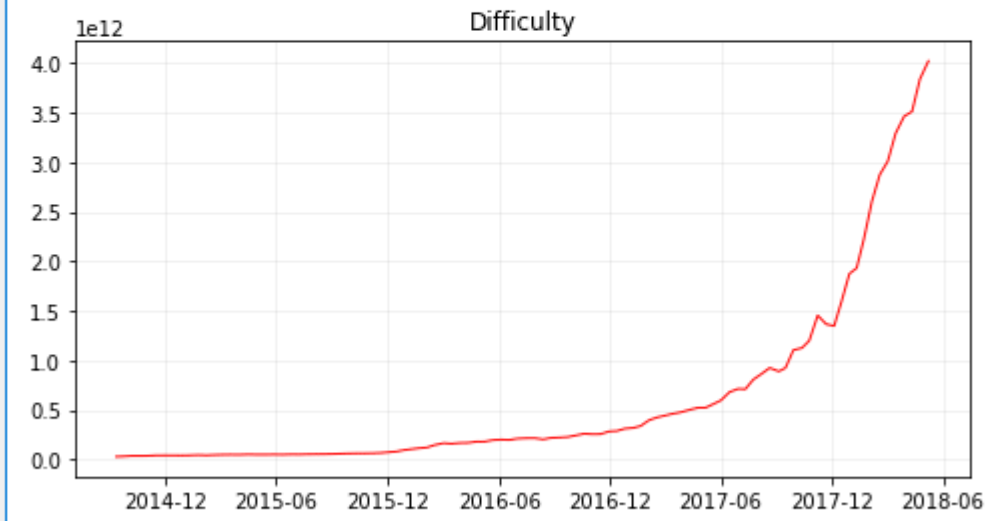
(실습 파일 : 7-5.Difficulty.py)

Bitcoin Core API : Remote Procedure Calls (RPCs) – Python 연습

- 블록체인의 최근 Difficulty의 변화를 관찰함. Difficulty는 2016 블록마다 변하므로, 최근 블록부터 2016 블록 이전의 Difficulty를 100개 관찰함.

```python
# Bitcoin Core API : JSON-RPC 기능 시험
#
# Block chain의 최근 Difficulty의 변화를 관찰한다. Difficulty는 2016 블록마다
# 바뀌므로, 최근 블록부터 2016 이전 블록의 Difficulty를 100 개 관찰한다.
#
# 패키지 : https://github.com/petertodd/python-bitcoinlib (by Peter Todd)
#
# 2018.5.8 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
from bitcoin.rpc import RawProxy
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime

# Bitcoin Core에 접속한다.
p = RawProxy()

# 블록체인의 블록 개수을 읽어온다
n = p.getblockcount()

# 최근 100개 블록의 헤더를 읽어서 생성 시간을 조회한다.
difficulty = []
i = 0
while True:
    bHash = p.getblockhash(n)
    hdr = p.getblockheader(bHash)

    dt = datetime.utcfromtimestamp(hdr['time'])
    st = dt.strftime('%Y-%m-%d')

    difficulty.append([st, hdr['difficulty']])

    # 약 2주 전 블록을 읽어온다. Difficulty는 2016 블록 단위로 바뀌기 때문
```

```
          Time         Difficulty
91   2018-01-10  1931136454487.716
92   2018-01-22  2227847638503.628
93   2018-02-04  2603077300218.593
94   2018-02-17  2874674234415.941
95   2018-03-02  3007383866429.732
96   2018-03-15  3290605988755.001
97   2018-03-29  3462542391191.563
98   2018-04-11   3511060552899.72
99   2018-04-24  3839316899029.672
100  2018-05-08  4022059196164.954
```




```
In [70]:
```

History log | IPython console

(실습 파일 : 7-6.MemPool.py)

Bitcoin Core API : Remote Procedure Calls (RPCs) – Python 연습

- 현재 MemPool에 저장된 Transaction의 상세 데이터를 조회함.

```python
# Bitcoin Core API : JSON-RPC 기능 시험
#
# 현재 MemPool에 있는 Transaction을 조회한다.
#
# 패키지 : https://github.com/petertodd/python-bitcoinlib (by Peter Todd)
#
# 2018.5.8 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
from bitcoin.rpc import RawProxy
import pprint
pp = pprint.PrettyPrinter(indent=1)

# Bitcoin Core에 접속한다.
p = RawProxy()

# MemPool에 있는 Tx 개수를 확인한다
mem = p.getmempoolinfo()
print("\n\nNumber of Tx :", mem['size'])

# MemPool에 있는Txid를 조회한다.
memTxid = p.getrawmempool()

# MemPool에 있는 Tx 데이터 10개 만 상세 데이터를 조회한다.
i = 0
for tx in memTxid:
    # MemPool에 있는 Txid의 상세 데이터를 읽어온다.
    memTx = p.getmempoolentry(tx)
    rawTx = p.getrawtransaction(tx)
    print("\nTXID :", tx)
    pp.pprint(memTx)
    print("\nRaw TX :")
    print(rawTx)
```

```
Number of Tx : 3824

TXID : f5bbdd4483a2e73a3acd139e8fe4d89072cb9ca0cb223dc4a6fbb4a4e3e02f3b
{'ancestorcount': 1,
 'ancestorfees': 200000,
 'ancestorsize': 225,
 'depends': [],
 'descendantcount': 1,
 'descendantfees': 200000,
 'descendantsize': 225,
 'fee': Decimal('0.00200000'),
 'height': 521707,
 'modifiedfee': Decimal('0.00200000'),
 'size': 225,
 'time': 1525755649,
 'wtxid': 'f5bbdd4483a2e73a3acd139e8fe4d89072cb9ca0cb223dc4a6fbb4a4e3e02f3b'}

Raw TX :
0100000001acb2e48b0dfabf788af8b870a95132c337d7b5ac1eca65fb11b9d64e425cccac010000
006a473044022027ca0a9bb75dcaa809e051e9498f13b34662e4d684384c9e850730f25015f4d002
207127834c800920d7cf2ff986362fbbd07e817da91c279e4dc83b1a907e73ba7e0121037bf71ea7
d95556a17ce65ae7899c3d4c1f00aa919d30aa9637481401a3e91f86feffffff0240c11322000000
001976a914042439e27819a5b6fb6af9fd776eb182caf9a0be88ac50cec6df030000001976a914b3
86350126ae56f50a4a9b38ced3eced5e1fe21a88acebf50700

TXID : 0b4d9757fc5a6279351b216021a3bcc1982ad970e41c09ded804bb0f99e68a48
{'ancestorcount': 1,
 'ancestorfees': 169500,
 'ancestorsize': 225,
 'depends': [],
```

History log | IPython console

(실습 파일 : 7-7.GetPeerInfo.py)

Bitcoin Core API : Remote Procedure Calls (RPCs) – Python 연습

- 현재 접속된 노드들의 정보를 조회함. IP 주소, 주고 받은 비트코인 P2P 프로토콜 메시지 양 등을 확인할 수 있음.

```python
# Bitcoin Core API : JSON-RPC 기능 시험
#
# 현재 통신 중인 노드들의 정보를 조회한다.
#
# 패키지 : https://github.com/petertodd/python-bitcoinlib (by Peter Todd)
#
# 2018.5.8 아마추어 퀀트 (조성현)
# ------------------------------------------------------------------------
from bitcoin.rpc import RawProxy
import pprint
pp = pprint.PrettyPrinter(indent=1)

# Bitcoin Core에 접속한다.
p = RawProxy()

# 통신 중인 Peer 노드들의 정보를 확인한다.
peer = p.getpeerinfo()
print("\n\nNumber of Nodes :", len(peer))
print("\n")
pp.pprint(peer)
```

```
Number of Nodes : 8


[{'addnode': False,
  'addr': '77.173.109.50:8333',
  'addrbind': '192.168.0.9:7352',
  'addrlocal': '118.32.168.221:7352',
  'banscore': 0,
  'bytesrecv': 2707961,
  'bytesrecv_per_msg': {'addr': 35052,
                        'block': 1123904,
                        'cmpctblock': 72196,
                        'feefilter': 32,
                        'getdata': 5852,
                        'getheaders': 1021,
                        'headers': 212,
                        'inv': 318022,
                        'notfound': 18372,
                        'ping': 1920,
                        'pong': 1920,
                        'sendcmpct': 66,
                        'sendheaders': 24,
                        'tx': 1129218,
                        'verack': 24,
                        'version': 126},
  'bytessent': 525568,
  'bytessent_per_msg': {'addr': 6105,
                        'feefilter': 32,
                        'getaddr': 24,
                        'getdata': 111793,
                        'getheaders': 1021,
```

History log | IPython console

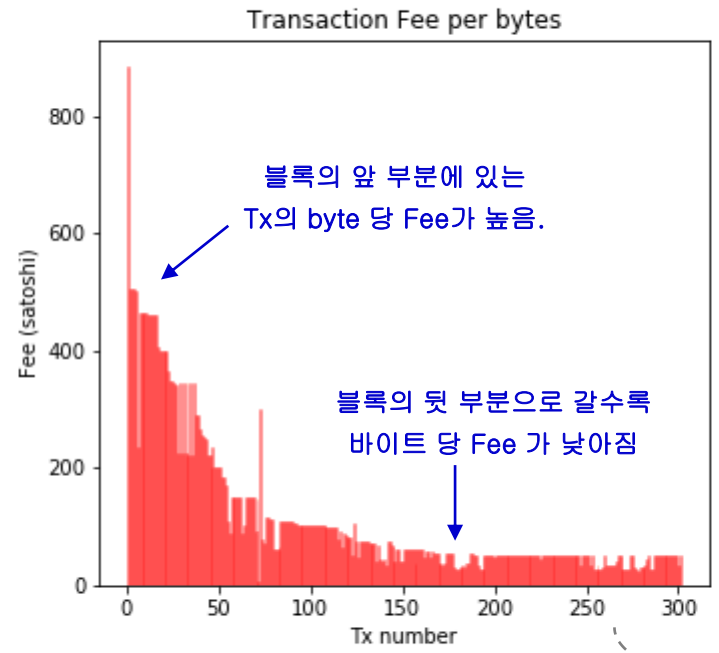
(실습 파일 : 7-8.txFee.py)

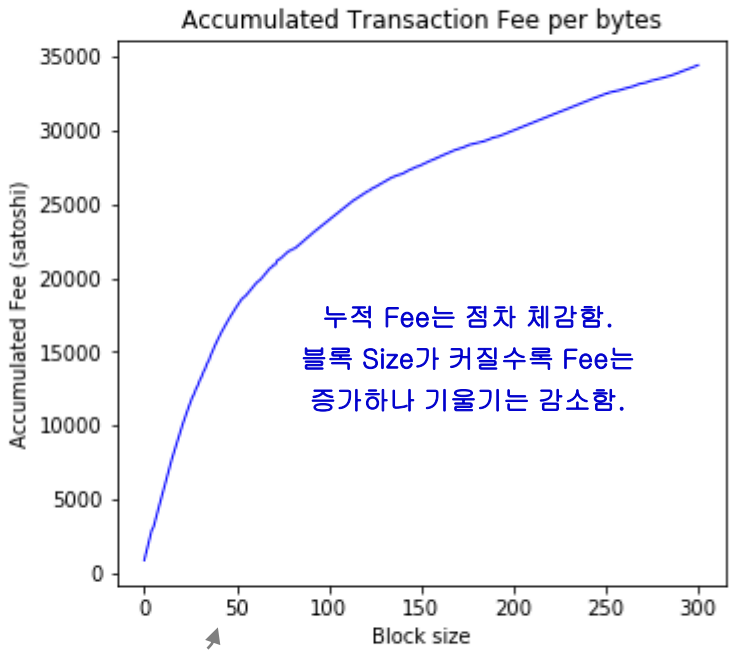
Bitcoin Core API : Remote Procedure Calls (RPCs) – Python 연습

- Miner 들이 MemPool에서 바이트 당 수수료 (Fee per bytes)가 높은 Tx을 우선적으로 선정하여 블록에 포함시키는 것을 확인함.
- 최근 블록의 Tx들을 읽고, Input의 UTXO와 Output의 value를 이용하여 Fee를 계산하고, Tx size로 나누어 바이트 당 수수료와 누적 수수료를 계산함.

```
# Bitcoin Core API : JSON-RPC 기능 시험
#
# Miner가 Tx를 선택하는 기준을 확인한다.
# - Miner는 byte 당 fee가 높은 Tx를 우선적으로 블록에 포
# - 한 블록 안에 있는 Tx를 순차적으로 읽어가면서 fee per
#
# 2018.5.11
# 아마추어 퀀트 (조성현)
# ------------------------------------------------------
from bitcoin.rpc import RawProxy
import matplotlib.pyplot as plt

# Bitcoin Core에 접속한다.
p = RawProxy()

def getUtxo(x, n):
    tt = p.getrawtransaction(x, True)
    return tt['vout'][n]['value']

# 블록체인 tip (끝 부분)의 hash, height를 읽어온다
tip = p.getchaintips()

# 마지막 블록을 읽어온다
height = tip[0]['height']
bHash = tip[0]['hash']
block = p.getblock(bHash)

# 마지막 블록의 Tx를 읽는다.
nTx = len(block['tx'])

# Tx을 차례대로 읽어가면서 Fee를 계산한다. (Coinbase Tx는
```

```
Block Height = 522814
Number of Transactions = 2113
```





누적

```
In [15]:
```

History log | IPython console

- Satoshi Nakamoto, 2008, Bitcoin: A Peer-to-Peer Electronic Cash System
- Andreas Antonopoulos, 2017, Mastering Bitcoin (2nd Edition) – Programming the open blockchain.
- Imran Bashir, 2017, Mastering Blockchain – Distributed ledgers, decentralization and smart contracts explained.
- Christof Paar, 2010, Understanding Cryptography
- secp256k1 : http://www.secg.org/sec2-v2.pdf
- Bitcoin Developer Reference : https://bitcoin.org/en/developer-reference
- Bitcoin Developer Guide : https://bitcoin.org/en/developer-guide
- Bitcoin Improvement Proposal (BIP) : https://github.com/bitcoin/bips/blob/master/README.mediawiki
- Bitcoin Script Language : https://en.bitcoin.it/wiki/Script
- Bitcoin Core : https://bitcoin.org/en/bitcoin-core/
- pybitcointools (by Vitalik Buterin) : https://pypi.python.org/pypi/bitcoin
- Blockchain explorer : https://blockchain.info
- P2P Protocol Analyzer (Wireshark) : https://www.wireshark.org/