교과목 : 정보보호

PyCryptodome
4 RSA & ECDSA

2023학년도 2학기
Suk-Hwan Lee

- PyCryptodome – Cryptographic library for Python

참조자료

- PyCryptodome Homepage : https://www.pycryptodome.org/en/latest/
- Download : https://pypi.org/project/pycryptodome/
- GitHub : https://github.com/Legrandin/pycryptodome

코드 참조

- https://www.pycryptodome.org/en/latest/src/examples.html
- 화이트 해커를 위한 암호와 해킹 (2판), White Hat Python, 장삼용 저, 정보문화사, 2019년9월

RSA Python Ex.

1. RSA 공개키 암호 구현하기

RSA 1024비트 개인키와 공개키 생성한 후, 단문 메시지를 공개키로 암호화하고 암호화한 메시지를 개인키로 복호

```
from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

def rsa_enc(msg):
    private_key = RSA.generate(1024)
    public_key = private_key.publickey()

    cipher = PKCS1_OAEP.new(public_key)
    encdata = cipher.encrypt(msg)
    print(encdata)

    cipher = PKCS1_OAEP.new(private_key)
    decdata = cipher.decrypt(encdata)
    print(decdata)

def main():
    msg = 'We test RSA encryption by PyCryptoDome'
    rsa_enc(msg.encode('utf-8'))

main()
```

- PKCS1_OAEP 모듈과 RSA 모듈 Import
  - ✓ PKCS (Public Key Cryptography Standard) : RSA Security에서 정한 공개키 암호 사용 방식에 대한 표준 프로토콜
  - ✓ 기능이나 특성에 따라 PKCS#1 ~ PKCS#15까지 구분
  - ✓ PKCS#1 : 일반적 공개키 암호 표준
  - ✓ OAEP (Optimal asymmetric encryption padding) : 암호하기 전 메시지에 랜덤값을 padding하여 암호 수행
- private_key.publickey() : 개인키에 대응하는 공개키 생성
- 공개키(public_key)의 PKCS1_OEAP 객체 cipher 생성
- cipher.encrypt() : 메시지 암호화

- 실행 결과 : Binary Stream으로 리턴, 코드 실행할 때마다 암호화 값이 변경

```
b'₩x05₩x87₩x93₩x8a|i₩xf9m₩xfa₩xf4&₩xa3₩xf6₩xa3₩₩₩x87₩xcf:
₩x97₩x02₩x1djSd₩xa0cV₩xf4₩xab₩xe6₩xeb₩xc2₩x06Q₩xa2₩xf7c₩xbd₩xa7₩xcb
₩x83₩x89₩xa6g+Cc,₩x1e₩xbf₩xcb%-i₩xa1₩xa1R₩xa6₩x1eD₩xac₩x1f₩xad₩x84
{₩x8eyVg@₩xc3D₩x1f₩xbby₩x04₩xe6₩xfa%9@₩n₩xe2₩x12Y₩xb8₩x07₩xeb
₩xf1;₩xd2tiH₩xc4q|₩x07?M₩x89^(X₩x18.₩x92₩x88>₩xfc₩xab₩xdf^5;₩xaa₩x9bCJ
₩xdd₩xe1,b9d₩x7f₩x07₩xe9'
b'We test RSA encryption by PyCryptoDome'
```

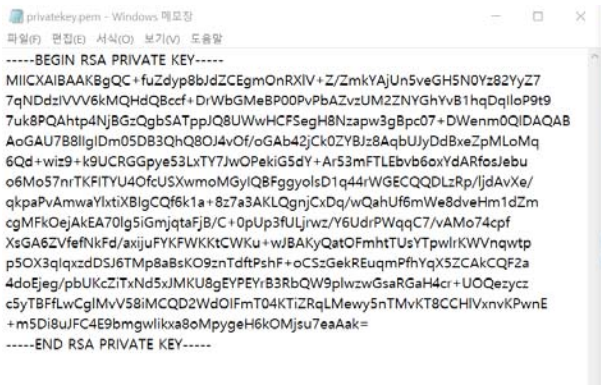
2. RSA 개인키, 공개키 파일 만들기

공개키는 잃어버려도 개인키만 있으면 공개키를 복원할 수 있음
개인키를 파일로 저장하기

```
from Crypto.PublicKey import RSA

def createPEM():
    private_key = RSA.generate(1024)
    h = open('privatekey.pem', 'wb+')
    h.write(private_key.exportKey('PEM'))
    h.close()

    public_key = private_key.publickey()
    h = open('publickey.pem', 'wb+')
    h.write(public_key.exportKey('PEM'))
    h.close()

createPEM()
```

- RSA.generate(1024)로 1024비트 개인키 (private_key) 생성하고, 'privatekey.pem'에 저장
- 생성된 개인키에 대응되는 공개키 (public_key) 생성하여 'publickey.pem' 파일에 저장

- privatekey.pem



- publickey.pem

PKCS1_OAEP.new

https://pycryptodome.readthedocs.io/en/latest/src/cipher/oaep.html?highlight=PKCS1_OAEP

PKCS#1 OAEP (RSA)

PKCS#1 OAEP is an asymmetric cipher based on RSA and the OAEP padding. It is described in RFC8017 where it is called `RSAES-OAEP`.

It can only encrypt messages slightly shorter than the RSA modulus (a few hundred bytes).

The following example shows how you encrypt data by means of the recipient's **public key** (here assumed to be available locally in a file called `public.pem`):

```
>>> from Crypto.Cipher import PKCS1_OAEP
>>> from Crypto.PublicKey import RSA
>>>
>>> message = b'You can attack now!'
>>> key = RSA.importKey(open('public.pem').read())
>>> cipher = PKCS1_OAEP.new(key)
>>> ciphertext = cipher.encrypt(message)
```

The recipient uses its own **private key** to decrypt the message. We assume the key is stored in a file called `private.pem`:

```
>>> key = RSA.importKey(open('private.pem').read())
>>> cipher = PKCS1_OAEP.new(key)
>>> message = cipher.decrypt(ciphertext)
```

Crypto.Cipher.PKCS1_OAEP.new(*key, hashAlgo=None, mgfunc=None, label='', randfunc=None*)

Return a cipher object `PKCS1OAEP_Cipher` that can be used to perform PKCS#1 OAEP encryption or decryption.

Parameters:

- **key** (*RSA key object*) – The key object to use to encrypt or decrypt the message. Decryption is only possible with a private RSA key.
- **hashAlgo** (*hash object*) – The hash function to use. This can be a module under *Crypto.Hash* or an existing hash object created from any of such modules. If not specified, *Crypto.Hash.SHA1* is used.
- **mgfunc** (*callable*) – A mask generation function that accepts two parameters: a string to use as seed, and the lenth of the mask to generate, in bytes. If not specified, the standard MGF1 consistent with `hashAlgo` is used (a safe choice).
- **label** (*bytes/bytearray/memoryview*) – A label to apply to this particular encryption. If not specified, an empty string is used. Specifying a label does not improve security.
- **randfunc** (*callable*) – A function that returns random bytes. The default is *Random.get_random_bytes*.

Pycryptodome –RSA

private_key=RSA.generate(1024)

private_key.exportKey('PEM'), private_key.publickey()

https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html

```
>>> from Crypto.PublicKey import RSA
>>>
>>> key = RSA.generate(2048)
>>> f = open('mykey.pem','wb')
>>> f.write(key.export_key('PEM'))
>>> f.close()
...
>>> f = open('mykey.pem','r')
>>> key = RSA.import_key(f.read())
```

Crypto.PublicKey.RSA.generate(*bits, randfunc=None, e=65537*)

Create a new RSA key pair.

The algorithm closely follows NIST FIPS 186-4 in its sections B.3.1 and B.3.3. The modulus is the product of two non-strong probable primes. Each prime passes a suitable number of Miller-Rabin tests with random bases and a single Lucas test.

Parameters:

- **bits** (*integer*) – Key length, or size (in bits) of the RSA modulus. It must be at least 1024, but **2048 is recommended.** The FIPS standard only defines 1024, 2048 and 3072.
- **randfunc** (*callable*) – Function that returns random bytes. The default is `Crypto.Random.get_random_bytes()`.
- **e** (*integer*) – Public RSA exponent. It must be an odd positive integer. It is typically a small number with very few ones in its binary representation. The FIPS standard requires the public exponent to be at least 65537 (the default).

Returns: an RSA key object (`RsaKey` , with private key).

class **Crypto.PublicKey.RSA.RsaKey**(***kwargs*)

Class defining an actual RSA key. Do not instantiate directly. Use `generate()` , `construct()` or `import_key()` instead.

Variables:

- **n** (*integer*) – RSA modulus
- **e** (*integer*) – RSA public exponent
- **d** (*integer*) – RSA private exponent
- **p** (*integer*) – First factor of the RSA modulus
- **q** (*integer*) – Second factor of the RSA modulus
- **u** – Chinese remainder component ($p^{-1} \bmod q$)

exportKey(*format='PEM', passphrase=None, pkcs=1, protection=None, randfunc=None*)

Export this RSA key.

publickey()

A matching RSA public key.

Returns: a new `RsaKey` object

3. RSA 개인키, 공개키 파일을 이용한 RSA 공개키 암호

```python
__author__ = 'samsjang@naver.com'

from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

def createPEM():
    private_key = RSA.generate(1024)
    with open('privatekey.pem', 'wb+') as f:
        f.write(private_key.exportKey('PEM'))

    public_key = private_key.publickey()
    with open('publickey.pem', 'wb+') as f:
        f.write(public_key.exportKey('PEM'))

def readPEM(pemfile):
    h = open(pemfile, 'r')
    key = RSA.importKey(h.read())
    h.close()
    return key
```

- RSA 개인키와 RSA 공개키 생성하여 pem 파일로 저장
- PEM 파일에 저장된 RSA 개인키 또는 RSA 공개키를 읽어 리턴

```python
def rsa_enc(msg):
    public_key = readPEM('publickey.pem')
    cipher = PKCS1_OAEP.new(public_key)
    encdata = cipher.encrypt(msg)
    return encdata

def rsa_dec(msg):
    private_key = readPEM('privatekey.pem')
    cipher = PKCS1_OAEP.new(private_key)
    decdata = cipher.decrypt(msg)
    return decdata

def main():
    createPEM()
    msg = 'We test RSA encryption by PyCryptoDome'
    ciphered = rsa_enc(msg.encode('utf-8'))
    print(ciphered)
    deciphered = rsa_dec(ciphered)
    print(deciphered)

main()
```

- RSA 공개키와 메시지 msg 암호
- RSA 개인키와 암호화된 msg 복호

4. RSA 공개키 서명 구현

공개키는 서명은 사용자의 개인키로 서명하고, 암호화한 정보를 사용자의 공개키로 확인

해당 정보를 보낸 사람이 당사자인지 확인하는 방법

```
from Crypto.Signature import pkcs1_15
from Crypto.PublicKey import RSA
from Crypto.Hash import SHA256 as SHA

def readPEM(pemfile):
    with open(pemfile, 'r') as h:
        key = RSA.importKey(h.read())
    return key

def rsa_sign(msg):
    private_key = readPEM('privatekey.pem')
    public_key = private_key.publickey()
    h = SHA.new(msg)
    signature = pkcs1_15.new(private_key).sign(h)
    return public_key, signature
```

- 사용자의 개인키 서명은 Pycryptodome의 Crypto.Signature 모듈의 pkcs1_15의 sign() 함수 이용
- msg에 개인키로 서명한 후 상대방에게 보내는 프로세스임

- 개인키를 private_key 저장하고, private_key에 대응하는 공개키 public_key 생성

- msg의 SHA256 해시 (h) 구하고, 이 값에 개인키 private_key를 이용해 pkcs1_15 객체의 sign() 함수로 해시 (h)를 서명 – signature
- 결과값 signature을 상대방에게 전달

4. RSA 공개키 서명 구현

공개키는 서명은 사용자의 개인키로 서명하고, 암호화한 정보를 사용자의 공개키로 확인
해당 정보를 보낸 사람이 당사자인지 확인하는 방법

```
def rsa_verify(msg, public_key, signature):
    h = SHA.new(msg)
    public_key = readPEM('publickey.pem')

    try:
        pkcs1_15.new(public_key).verify(h, signature)
        print ('VERIFIED')
    except Exception as e:
        print(e)
        print ('DENIED')

def main():
    msg = 'My name is PyCryptoDome'
    public_key, signature = rsa_sign(msg.encode('utf-8'))
    rsa_verify(msg.encode('utf-8'), public_key, signature)

main()
```

서명 확인 : (msg, 공개키, 서명)

- 개인키로 서명한 정보를 네트워크를 통해 전달받음
- msg의 SHA256 해시 구하고, 공개키 (public_key) 읽음

- 공개키에 의하여 pkcs1_15 객체 verify() 함수의 인자로 해시값 (h)과 서명(signature)를 입력받아 서명(signature)와 해시(h)가 일치하는 확인
- 확인이 불가하면 예외 발생

PKCS#1 v1.5 (RSA)

https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html?highlight=PKCS1_15#Crypto.Signature.pkcs1_15.PKCS115_SigScheme

PKCS#1 v1.5 (RSA)

An old but still solid digital signature scheme based on RSA.

It is more formally called `RSASSA-PKCS1-v1_5` in Section 8.2 of RFC8017.

The following example shows how a *private* RSA key (loaded from a file) can be used to compute the signature of a message:

```
>>> from Crypto.Signature import pkcs1_15
>>> from Crypto.Hash import SHA256
>>> from Crypto.PublicKey import RSA
>>>
>>> message = 'To be signed'
>>> key = RSA.import_key(open('private_key.der').read())
>>> h = SHA256.new(message)
>>> signature = pkcs1_15.new(key).sign(h)
```

At the other end, the receiver can verify the signature (and therefore the authenticity of the message) using the matching *public* RSA key:

```
>>> key = RSA.import_key(open('public_key.der').read())
>>> h = SHA256.new(message)
>>> try:
>>>     pkcs1_15.new(key).verify(h, signature)
>>>     print "The signature is valid."
>>> except (ValueError, TypeError):
>>>    print "The signature is not valid."
```

`Crypto.Signature.pkcs1_15.new(rsa_key)`

Create a signature object for creating or verifying PKCS#1 v1.5 signatures.

Parameters: **rsa_key** (*RSA object*) – The RSA key to use for signing or verifying the message. This is a `Crypto.PublicKey.RSA` object. Signing is only possible when `rsa_key` is a **private** RSA key.

Returns: a `PKCS115_SigScheme` signature object

h = SHA.new(msg)
signature = pkcs1_15.new(private_key).sign(h)

pkcs1_15.new(public_key).verify(h, signature)

Pycryptodome –RSA

PKCS#1 v1.5 (RSA)

https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html?highlight=PKCS1_15#Crypto.Signature.pkcs1_15.PKCS115_SigScheme

class `Crypto.Signature.pkcs1_15.PKCS115_SigScheme`(*rsa_key*)

A signature object for `RSASSA-PKCS1-v1_5`. Do not instantiate directly. Use `Crypto.Signature.pkcs1_15.new()`.

can_sign()

Return `True` if this object can be used to sign messages.

sign(*msg_hash*)

Create the PKCS#1 v1.5 signature of a message.

This function is also called `RSASSA-PKCS1-V1_5-SIGN` and it is specified in section 8.2.1 of RFC8017.

Parameters: **msg_hash** (*hash object*) – This is an object from the `Crypto.Hash` package. It has been used to digest the message to sign.

Returns: the signature encoded as a *byte string*.

Raises:
- **ValueError** – if the RSA key is not long enough for the given hash algorithm.
- **TypeError** – if the RSA key has no private half.

verify(*msg_hash*, *signature*)

Check if the PKCS#1 v1.5 signature over a message is valid.

This function is also called `RSASSA-PKCS1-V1_5-VERIFY` and it is specified in section 8.2.2 of RFC8037.

Parameters:
- **msg_hash** – The hash that was carried out over the message. This is an object belonging to the `Crypto.Hash` module.
- **signature** (*byte string*) – The signature that needs to be validated.

Raises: **ValueError** – if the signature is not valid.

```
h = SHA.new(msg)
signature = pkcs1_15.new(private_key).sign(h)

pkcs1_15.new(public_key).verify(h, signature)
```

ECDSA Python Ex.

Elliptic Curve Digital Signature Algorithm, **ECDSA**

1. ECDSA 전자서명

- 타원 곡선 기반의 전자서명 알고리즘
- 2008년 TLS v1.2의 기술명세서에 소개된 공개키 암호 체계임
- 동일한 보안 수준으로 비교할 때, ECDSA는 RSA보다 훨씬 작은 키 크기를 가짐
  - ✓ 예) 3072비트 키 크기 RSA는 768 Byte 데이터 크기를 가짐
  - ✓ 동일한 보안 수준의 ECDSA는 RSA의 1/24 크기인 32 Byte 데이터 크기를 가짐
- 서명 속도는 RSA가 빠르나, 서명 검증 속도는 ECDSA가 빠름

2. ECDSA 개인키, 공개키 파일 만들기

```
from Crypto.PublicKey import ECC

def createPEM_ECDSA():
    key = ECC.generate(curve='P-256')
    with open('privkey_ecdsa.pem', 'w') as h:
        h.write(key.export_key(format='PEM'))

    key = key.public_key()
    with open('pubkey_ecdsa.pem', 'w') as h:
        h.write(key.export_key(format='PEM'))

createPEM_ECDSA()
```

- ECDSA 위해 ECC 모듈 import
- ECC.generate() : 타원 곡선 암호를 이용해 개인키 생성
- 타원 곡선 암호에서 곡선을 만들기 위한 상수로 'P-256" 지정 (NIST 에서 권장)

Pycryptodome – ECSDA 전자서명

ECC 모듈

https://pycryptodome.readthedocs.io/en/latest/src/public_key/ecc.html?highlight=ECC#ecc

ECC

ECC (Elliptic Curve Cryptography) is a modern and efficient type of public key cryptography. Its security is based on the difficulty to solve discrete logarithms on the field defined by specific equations computed over a curve.

ECC can be used to create digital signatures or to perform a key exchange.

Compared to traditional algorithms like RSA, an ECC key is significantly smaller at the same security level. For instance, a 3072-bit RSA key takes 768 bytes whereas the equally strong NIST P-256 private key only takes 32 bytes (that is, 256 bits).

This module provides mechanisms for generating new ECC keys, exporting and importing them using widely supported formats like PEM or DER.

Curve	Possible identifiers
NIST P-256	`'NIST P-256'`, `'p256'`, `'P-256'`, `'prime256v1'`, `'secp256r1'`
NIST P-384	`'NIST P-384'`, `'p384'`, `'P-384'`, `'prime384v1'`, `'secp384r1'`
NIST P-521	`'NIST P-521'`, `'p521'`, `'P-521'`, `'prime521v1'`, `'secp521r1'`

For more information about each NIST curve see FIPS 186-4, Section D.1.2.

The following example demonstrates how to generate a new ECC key, export it, and subsequently reload it back into the application:

```
>>> from Crypto.PublicKey import ECC
>>>
>>> key = ECC.generate(curve='P-256')
>>>
>>> f = open('myprivatekey.pem','wt')
>>> f.write(key.export_key(format='PEM'))
>>> f.close()
...
>>> f = open('myprivatekey.pem','rt')
>>> key = ECC.import_key(f.read())
```

The ECC key can be used to perform or verify ECDSA signatures, using the module `Crypto.Signature.DSS`.

key = ECC.generate(curve='P-256')

`Crypto.PublicKey.ECC.generate(**kwargs)`

Generate a new private key on the given curve.

Parameters:

- **curve** (*string*) – Mandatory. It must be a curve name defined in Table 1.
- **randfunc** (*callable*) – Optional. The RNG to read randomness from. If `None`, `Crypto.Random.get_random_bytes()` is used.

Pycryptodome – ECSDA 전자서명

ECC 모듈, ECC.generate, ECC.import_key

https://pycryptodome.readthedocs.io/en/latest/src/public_key/ecc.html?highlight=ECC#ecc

`Crypto.PublicKey.ECC.import_key(encoded, passphrase=None)`

Import an **ECC** key (public or private).

Parameters:

- **encoded** (*bytes or multi-line string*) –
  The **ECC** key to import.

  An **ECC** **public** key can be:

  - An X.509 certificate, binary (DER) or ASCII (PEM)
  - An X.509 `subjectPublicKeyInfo`, binary (DER) or ASCII (PEM)
  - An OpenSSH line (e.g. the content of `~/.ssh/id_ecdsa`, ASCII)

  An **ECC** **private** key can be:

  - In binary format (DER, see section 3 of RFC5915 or PKCS#8)
  - In ASCII format (PEM or OpenSSH 6.5+)

  Private keys can be in the clear or password-protected.

  For details about the PEM encoding, see RFC1421/RFC1423.

- **passphrase** (*byte string*) – The passphrase to use for decrypting a private key.
  Encryption may be applied protected at the PEM level or at the PKCS#8 level.
  This parameter is ignored if the key in input is not encrypted.

Returns: a new **ECC** key object

3. ECDSA 전자서명

```
from Crypto.PublicKey import ECC
from Crypto.Signature import DSS
from Crypto.Hash import SHA256 as SHA

def readPEM_ECC(pemfile):
    with open(pemfile, 'r') as h:
        key = ECC.import_key(h.read())

    return key

def ecdsa_sign(msg):
    privateKey = readPEM_ECC('privkey_ecdsa.pem')
    sha = SHA.new(msg)
    signer = DSS.new(privateKey, 'fips-186-3')
    signature = signer.sign(sha)

    return signature
```

- privkey_ecdsa.pem, pubkey_ecdsa.pem 파일에 저장된 ECC 개인키와 공개키 이용하여 ECDSA 전자서명 구현
- DSS(Digital Signature Standard) 모듈 import
- DSS 이용하여 전자서명 위한 객체 생성
- UTF-8로 인코딩된 msg를 privkey_ecdsa.pem의 ECC 개인키로 서명 signature 리턴
- msg의 SHA-256 해시 생성 (sha)
- DSS.new(privateKey, 'fips-186-3') : FIPS-186-3 규정된 전자서명 규격 사용, 개인키로 전자서명 위한 객체 생성하고 signer에 할당
  - ✓ FIPS(Federal Information Processing Standards)는 NITS에서 제안한 연방 정보 처리 표준임
- DSS 객체의 sign()은 msg의 해시 sha에 개인키 서명한 signature을 리턴

3. ECDSA 전자서명

```
def ecdsa_verify(msg, signature):
    sha = SHA.new(msg)
    publicKey = readPEM_ECC('pubkey_ecdsa.pem')
    verifier = DSS.new(publicKey, 'fips-186-3')
    try:
        verifier.verify(sha, signature)
        print('+++ The message is authentic')
    except ValueError:
        print('--- The message is not authentic')

def main():
    msg = 'My name is PyCryptoDome'
    signature = ecdsa_sign(msg.encode('utf-8'))
    ecdsa_verify(msg.encode('utf-8'), signature)

main()
```

- ECC 개인키로 서명된 signature에 대해 pubkey_ecdsa.pem에 저장된 ECC 공개키로 검증
- 검증할 메시지의 SHA-256 해시값 생성 (sha)
- pubkey_ecdsa.pem에 저장된 공개키로 DSS 객체 생성 (verifier)
- DSS 객체 verifier의 verify() : 메시지 해쉬 sha와 전자서명 결과 signature 이용하여 검증 수행
- 검증하려는 msg나 공개키가 일치하지 않으면 ValueError 발생

Pycryptodome – ECSDA 전자서명

Digital Signature Algorithm (DSA and ECDSA)

https://pycryptodome.readthedocs.io/en/latest/src/signature/dsa.html?highlight=DSS

A variant of the ElGamal signature, specified in FIPS PUB 186-4.

It is based on the discrete logarithm problem in a prime finite field (DSA) or in an elliptic curve field (ECDSA).

A sender can use a *private* key (loaded from a file) to sign a message:

```
>>> from Crypto.Hash import SHA256
>>> from Crypto.PublicKey import ECC
>>> from Crypto.Signature import DSS
>>>
>>> message = b'I give my permission to order #4355'
>>> key = ECC.import_key(open('privkey.der').read())
>>> h = SHA256.new(message)
>>> signer = DSS.new(key, 'fips-186-3')
>>> signature = signer.sign(h)
```

The receiver can use the matching *public* key to verify authenticity of the received message:

```
>>> from Crypto.Hash import SHA256
>>> from Crypto.PublicKey import ECC
>>> from Crypto.Signature import DSS
>>>
>>> key = ECC.import_key(open('pubkey.der').read())
>>> h = SHA256.new(received_message)
>>> verifier = DSS.new(key, 'fips-186-3')
>>> try:
>>>     verifier.verify(h, signature)
>>>     print "The message is authentic."
>>> except ValueError:
>>>     print "The message is not authentic."
```

`Crypto.Signature.DSS.new(key, mode, encoding='binary', randfunc=None)`

Create a signature object `DSS_SigScheme` that can perform (EC)DSA signature or verification.

Note

Refer to NIST SP 800 Part 1 Rev 4 (or newer release) for an overview of the recommended key lengths.

Parameters:

- **key** (*a key object*) –
  The key to use for computing the signature (*private* keys only) or verifying one: it must be either `Crypto.PublicKey.DSA` or `Crypto.PublicKey.ECC`.

  For DSA keys, let `L` and `N` be the bit lengths of the modulus `p` and of `q`: the pair `(L,N)` must appear in the following list, in compliance to section 4.2 of FIPS 186-4:

  - (1024, 160) *legacy only; do not create new signatures with this*
  - (2048, 224) *deprecated; do not create new signatures with this*
  - (2048, 256)
  - (3072, 256)

  For ECC, only keys over P-256, P384, and P-521 are accepted.

Pycryptodome – ECSDA 전자서명

Digital Signature Algorithm (DSA and ECDSA)

https://pycryptodome.readthedocs.io/en/latest/src/signature/dsa.html?highlight=DSS

`Crypto.Signature.DSS.new(key, mode, encoding='binary', randfunc=None)`

Create a signature object `DSS._SigScheme` that can perform (EC)DSA signature or verification.

Note

Refer to NIST SP 800 Part 1 Rev 4 (or newer release) for an overview of the recommended key lengths.

Parameters:

- **key** (*a key object*) –
- **mode** (*string*) –
  The parameter can take these values:
  - *'fips-186-3'*. The signature generation is randomized and carried out according to FIPS 186-3: the nonce `k` is taken from the RNG.
  - *'deterministic-rfc6979'*. The signature generation is not randomized. See RFC6979.
- **encoding** (*string*) –
  How the signature is encoded. This value determines the output of `sign()` and the input to `verify()`.
  The following values are accepted:
  - *'binary'* (default), the signature is the raw concatenation of `r` and `s`. It is defined in the IEEE P.1363 standard.
    For DSA, the size in bytes of the signature is `N/4` bytes (e.g. 64 for `N=256`).
    For ECDSA, the signature is always twice the length of a point coordinate (e.g. 64 bytes for P-256).
  - *'der'*, the signature is a ASN.1 DER SEQUENCE with two INTEGERs (`r` and `s`). It is defined in RFC3279. The size of the signature is variable.
- **randfunc** (*callable*) – A function that returns random *byte strings*, of a given length. If omitted, the internal RNG is used. Only applicable for the *'fips-186-3'* mode.

교과목 : 정보보호

PyCryptodome
5 Hash & Blockchain

2023학년도 2학기
Suk-Hwan Lee

- PyCryptodome – Cryptographic library for Python

참조자료

- PyCryptodome Homepage : https://www.pycryptodome.org/en/latest/
- Download : https://pypi.org/project/pycryptodome/
- GitHub : https://github.com/Legrandin/pycryptodome

코드 참조

- https://www.pycryptodome.org/en/latest/src/examples.html
- 화이트 해커를 위한 암호와 해킹 (2판), White Hat Python, 장삼용 저, 정보문화사, 2019년9월

Hash Function

1. MD5

- MD5(Message-Digest Algorithm 5) : 1991년 만들어진 128비트 길이의 해시값 출력하는 해시함수
- 패스워드, 스위치, 라우터 등 장비간 상호 인증 등에 활용
- 128비트 작은 크기의 해시값과 알고리즘 자체 결함으로 최근 네트워크 전송 파일의 무결성 검증 등에서만 활용

- 파이썬 제공하는 hashlib 모듈의 md5나 Pycryptodome 모듈에서 제공하는 MD5를 import하면 됨

```
from hashlib import md5
or
from Crypto.Hash import MD5
```

```
from hashlib import md5

msg = 'I love Python'
m = md5()
m.update(msg.encode('utf-8'))
ret = m.hexdigest()
print(ret)
```

- hashlib 모듈의 md5 이용하여 MD5 해시값 16진수로 출력
- 출력 결과

```
27eb2f69c24aa5f3503a6ae610f23a83
```

2. SHA

- SHA(Secure Hash Algorithm) : 1993년 미국 NSA 만들고, 미국 국립표준기술연구소에 표준으로 제정한 해시함수
- SHA-0, SHA-1, SHA-2, SHA-3로 발전
  - ✓ SHA-0 : 비트토렌트 (P2P 파일 공유 시스템 원조격)에서 파일의 무결성이나 인덱싱으로 활용
  - ✓ SHA-2 중 SHA-256 (256비트 해시 출력) : 패스워드 암호화나 블록체인 등에서 활용
  - ✓ SHA-2 중 SHA-512 (512비트 해시 출력) : 유닉스나 리눅스 OS에서 패스워드 암호화 방법으로 활용
  - ✓ 현재 SHA-2 시리즈의 SHA-256, SHA-512 광범위하게 활용
  - ✓ SHA-3 : SHA-1, SHA-2와 전혀 다른 알고리즘으로 아직까지 결함이 알려져 있지 않음
- hashlib 모듈에서 SHA-2, SHA-3 시리즈의 SHA-256, SHA-512 함수 제공

```
from hashlib import sha256, sha512      # SHA-2 시리즈의 SHA-256, SHA-512
from hashlib import sha3_256, sha3_512  # SHA-3 시리즈의 SHA-256, SHA-512
```

```
from hashlib import sha256

msg = 'I love Python'
sha = sha256()
sha.update(msg.encode('utf-8'))
ret = sha.hexdigest()
print('SHA-2 SHA-256 : ', ret)
```

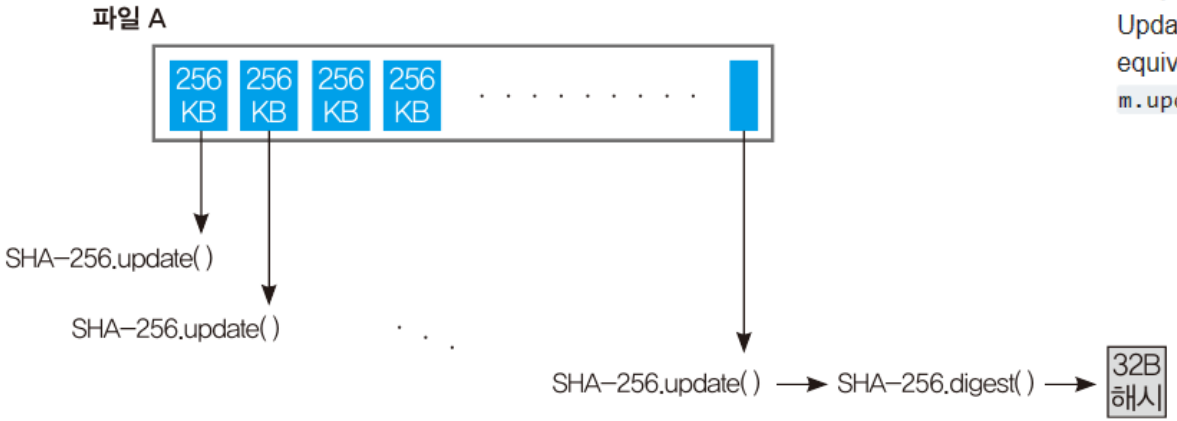
- SHA-3 시리즈의 SHA-256 활용하려면 sha3_256 import하고 sha3_256() 함수 사용

- 출력 결과

```
SHA-2 SHA-256 :  24e19c4fdadbd5e4670ae6ed98e2e581afe9ecf81e859da25c065404364ace52
```

3. 두 파일의 무결성 검증

- **파일에 대한 데이터 무결성 검증**
  - ✓ 파일 내용을 한꺼번에 읽어 SHA256 해시를 구하기에는 현실적으로 비효율적임
  - ✓ 파일에서 256KB 크기로 읽어 해시를 업데이트



`hash.update(data)`

Update the hash object with the bytes-like object. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

[그림 4.1] 파일에 대한 해시값 구하기

3. 두 파일의 무결성 검증

- SHA-256.update() – hash.update(data)

  참조 : https://docs.python.org/3/library/hashlib.html

hash.**update**(*data*)

Update the hash object with the bytes-like object. Repeated calls are equivalent to a single call with the concatenation of all the arguments: `m.update(a); m.update(b)` is equivalent to `m.update(a+b)`.

```
encPlusText = 4783e784b4fa2fba9e4d6502dbc64f8f
encText = 4783e784b4fa2fba9e4d6502dbc64f8f
OnlyText1 = bb747b3df3130fe1ca4afa93fb7d97c9
```

해쉬 동일

해쉬 다름

```python
import hashlib

text1 = 'ABCDEFG'
text2 = 'H'
enc1 = hashlib.md5()
enc1.update(text1.encode('utf-8'))
enc1.update(text2.encode('utf-8'))
encPlusText = enc1.hexdigest()
print ('encPlusText = %s' %encPlusText)

enc2 = hashlib.md5()
enc2.update((text1+text2).encode('utf-8'))
encText = enc2.hexdigest()
print ('encText = %s' %encText)

enc3 = hashlib.md5()
enc3.update(text1.encode('utf-8'))
encText = enc3.hexdigest()
print ('OnlyText1 = %s' %encText)
```

3. 두 파일의 무결성 검증

```python
from hashlib import sha256 as SHA
import codecs

SIZE = 1024*256

def getFileHash(filename):
    sha = SHA()

    h = open(filename, 'rb')
    content = h.read(SIZE)
    while content:
        sha.update(content)
        content = h.read(SIZE)

    h.close()

    hashval = sha.digest()

    return hashval
```

- 파일에서 256KB 만큼 읽음
- 읽은 256KB 정보만큼 해시할 데이터 갱신
- 파일에서 다음 256KB 읽음
- 최종 해시값 계산

```python
def hashCheck(file1, file2):
    hashval1 = getFileHash(file1)
    hashval2 = getFileHash(file2)

    if hashval1 == hashval2:
        print('Two Files are Same')
    else:
        print('Two Files are Different')

def main():
    file1 = 'plain.txt'
    file2 = 'plain.txt.enc.dec'

    hashval = getFileHash(file1)
    print(codecs.encode(hashval, 'hex_codec'))


    hashCheck(file1, file2)


main()
```

- file1, file2에 대한 해시값 계산
- 두 해시값 동일성 검증

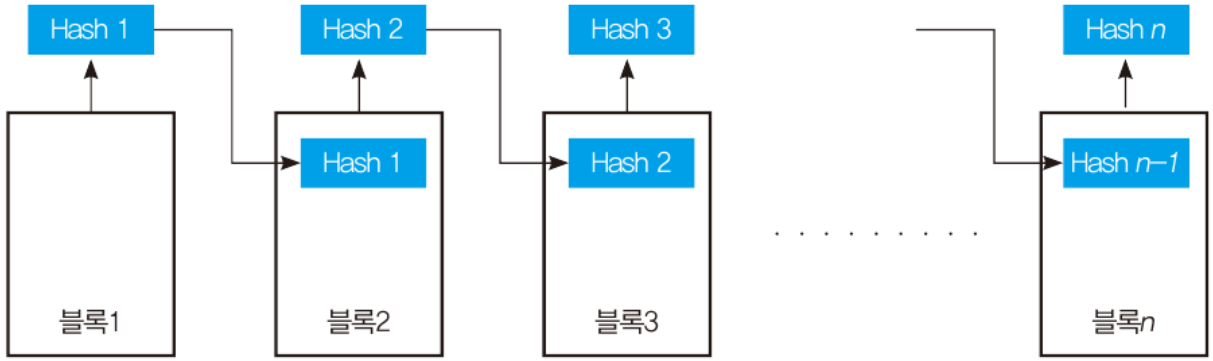
Blockchain에서 Hash 활용

- 해시가 가장 광범위하게 활용되고 있는 기술이 블록체인임

1. 블록체인 개념과 구조

- ‘블록’ 이라고 하는 거래 기록을 담고 있는 데이터들이
- 거래 참여자들의 합의로 생성된 체인 형태의 연결고리를 가진 형태로 구성되어
- 모든 거래 참여자들에 배포됨
- 누구나 기록된 거래 내용의 결과를 열람 가능하고, 어느 누구도 임의로 수정할 수 없도록 만든
- 분산 컴퓨팅 기반의 데이터 위변조 방지 기술임

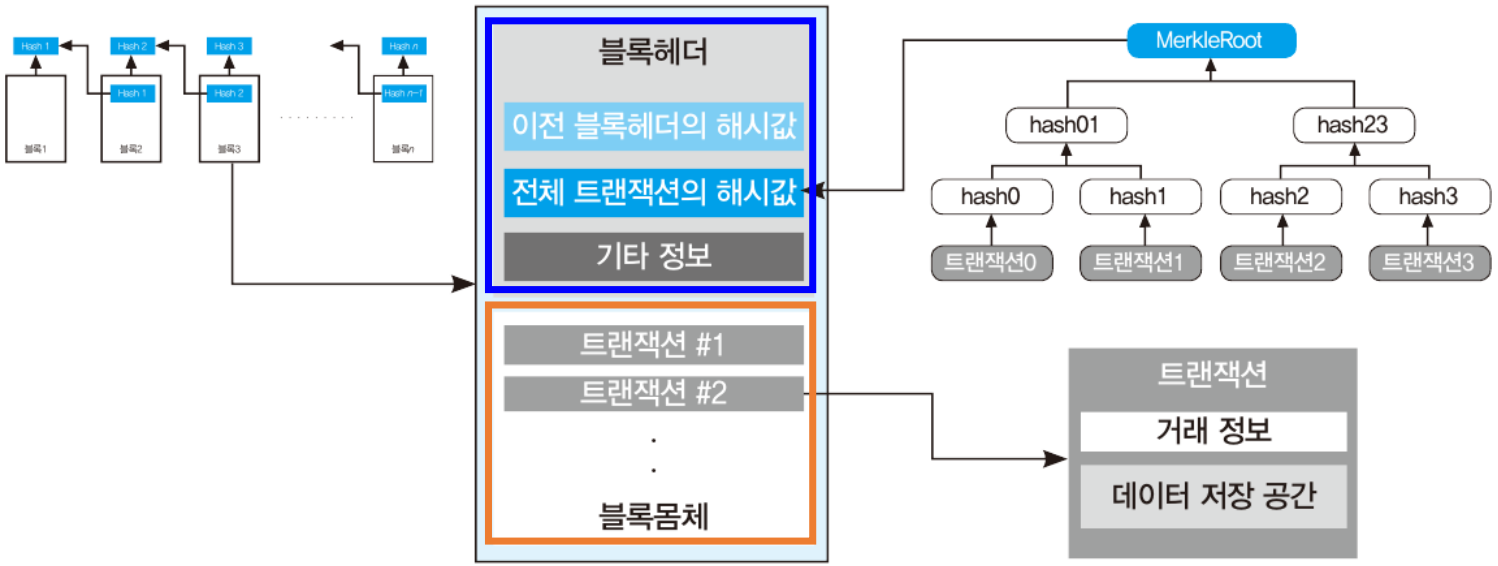
1. 블록체인 개념과 구조



[그림 4.2] 블록체인의 일반적인 구조

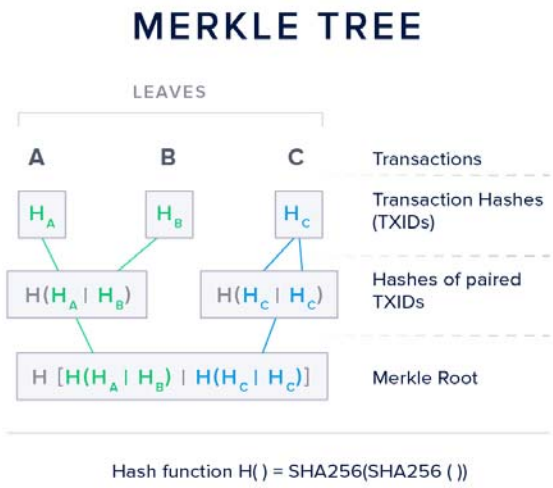
- 블록 1에서부터 거래 기록 – 특정한 조건인 합의 알고리즘을 통해 블록 1에 더 이상 거래 기록할 수 없도록 블록을 동결하고 완성
  - ✓ (첫번째 블록) 블록 1을 제네시스 블록이라 함
  - ✓ 이 블록의 정보가 거짓이면, 블록체인 전체 정보가 거짓이 되므로, 반드시 참값을 기록해야 함
- 블록 1에 기록된 정보의 해시값 Hash1 생성하고, Hash1은 다음 블록 2에 기록
- 블록 2도 특정한 조건을 만족하면 블록 1과 같은 과정을 수행 – 다음 블록인 3에 거래 기록
- 블록체인은 거래에 참여하는 모든 당사자들에게 배포됨

1. 블록체인 개념과 구조



[그림 4.3] 블록의 구조

- 해시값은 SHA-256 이용
- Merkle Tree 방법



- 블록 : Block header (이전 block header의 해시값, 전체 거래에 대한 해시값, 기타) + Block body (거래 기록)
- 블록이 만들어지는 조건인 합의 알고리즘이나 블록 크기 등은 블록체인 종류에 따라 달라짐

2. 블록체인과 작업증명

- 블록은 거래 참여자들의 합의에 의해 생성, 배포
- 합의 알고리즘
  - ✓ 분산 네트워크 상에 존재하는 서로 신뢰할 수 없는 정보들에 (수학적) 계산된 값을 특정 절차에 따라 상호 검증하여 신뢰 가능하도록 보장하는 알고리즘

➢ 블록체인 연결구조와 합의 알고리즘 통해 정보의 위변조가 매우 어려워짐

- 블록체인에 활용되는 합의 알고리즘
  - ✓ Proof of Work (PoW, 작업증명)
    - 목표값 이하 해시를 찾는 과정을 무수히 반복하여 해당 작업에 참여했음을 증명
    - 채굴과정(mining) 통해 작업증명을 함
    - 비트코인, 이더리움, 라이트코인, 비트코인캐시, 비트코인골드 등 암호화폐에서 작업증명 방식 사용
  - ✓ Proof of Stake (PoS, 지분증명)
    - 암호화폐 보유 지분율에 비례하여 의사결정 권한을 줌. 채굴 과정이 필요없음
    - 카르다노, 큐텀, 피어코인 등 암호화폐 지분증명 방식에 사용
    - *이더리움도 현재 작업증명 방식에서 지분증명 방식으로 변경할 예정*
  - ✓ PBFT (Practical Byzantine Fault Tolerance), Raft

2. 블록체인과 작업증명

- **작업증명**
  - ✓ 주어진 해시값보다 작아지도록 SHA-256에 입력되는 값을 찾아내는 것
  - ✓ 작업증명의 난이도는 주어진 해시값에 따라 달라짐

문제 1. 아래 조건을 만족하는 value를 찾아내는 것이 작업증명 메카니즘

SHA-256(value) ≤ 000c007…3c240b36c312

✓ "Attack at 9PM!" 문장에 대한 SHA-256 해시값은 다음과 같을 때

SHA-256("Attack at 9PM!") = c53ae0b1db6f94ce4177112…9f5a19f47d473a785dc97afc

✓ "Attack at 9PM!" 문장 뒤에 숫자 0을 더해서 대한 SHA-256 해시값을 구하면

SHA-256("Attack at 9PM!0") = 8eb6977a5765fda7cf7d6…20ede2b3c371ab6c9adc41a0

2. 블록체인과 작업증명

문제 2. hashcash 유형

"Attack at 9PM!" 에 어떤 숫자를 추가하여 구한 SHA-256 해시값이 00으로 시작하도록 하는 어떤 숫자를 구하라

- ✓ 숫자 0부터 1씩 증가하여 SHA-256 해시값 구하여 00으로 시작하는 숫자를 찾으면 됨
- ✓ 문제 2의 정답은 여러 개가 있지만, 처음 나타나는 숫자는 240임

SHA-256("Attack at 9PM!240") = 00676d333dd739b59baa…7afabff7bdb29e7a422468a16

- hashcash
  - ✓ 단순하게 0, 1, 2, …. 대입하여 해시값 확인
  - ✓ 0, 1, 2, …. 와 같이 값을 변화하면서 원래 메시지에 추가하는 값을 nonce라고 부름
  - ✓ hashcash 문제는 주어진 해시값과 같아지는 nonce 값을 찾는 것임

- 00 → 000으로 시작하는 해시값이라면 hashcash 문제의 난이도는 증가됨

✓ 작업증명은 hashcash 문제를 푸는 원리와 동일

2. 블록체인과 작업증명

"Attack at 9PM!" 에 대한 hashcash 구하는 python 코드

```python
from hashlib import sha256 as sha

def hashcash(msg, difficulty):
    nonce = 0
    print('+++ Start')
    while True:
        target = '%s%d' %(msg, nonce)
        ret = sha(target.encode()).hexdigest()

        if ret[:difficulty] == '0'*difficulty:
            print('+++ Bingo')
            print('--->', ret)
            print('---> NONCE=%d' %nonce)
            break

        nonce += 1

def main():
    msg = 'Attack at 9PM!'
    difficulty = 2
    hashcash(msg, difficulty)

main()
```

- 변수 difficulty는 hashcash의 난이도 설정
  - ✓ difficulty가 2로 되어 있으면 '00' 으로 시작하는 hashcash 구하는 것임
  - ✓ difficulty 커지면 계산 시간이 기하급수적으로 증가됨

- 결과

```
+++ Start
+++ Bingo
---> 00676d333dd739b59baa37ec1f393c294c774317afabff7bdb29e7a422468a16
---> NONCE=240
```

- difficulty=2, nonce=240
- difficulty=3, nonce=6541
- difficulty=4, nonce=103032
- difficulty=5, nonce=1016604
- difficulty=6, nonce=4503371

3. 작업증명의 효과

- 비트코인이나 이더리움 같은 블록체인 네트워크 : 네트워크에 참여하는 모든 노드의 컴퓨팅 파워를 총 동원하여 일정한 시간 동안 계산해야 nonce를 구할 수 있을 정도로 hashcash 난이도를 조정

- 비트코인 경우 참여하는 모든 노드의 컴퓨팅 파워을 동원하여 10분 정도 걸리는 hashcash 난이도로 설정

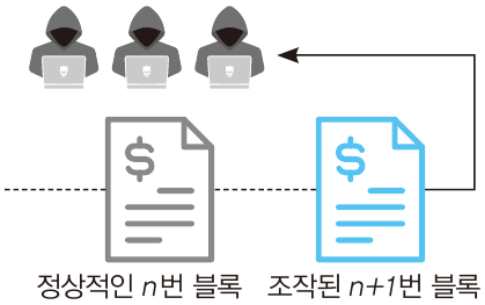
- 실제 작업증명 mechanism
  ① 거래 정보가 담긴 1번 블록 정보를 이용해 hashcash 문제 제시
  ② 블록체인에 참여한 모든 노드의 컴퓨팅 파워로 10분 정도 걸리는 난이도가 설정
  ③ 특정 노드에서 hashcash 문제 답을 구했다면, 1번 블록에 정답을 기록하고 (더 이상 거래 기록을 담지 못하게 동결) 이 블록을 모든 참여 노드에게 전달
  ④ 블록을 전달받은 모든 블록에 포함된 hashcash 정담을 검증하고 정답이면 블록 수용 (아니면 폐기)
  ⑤ 새로운 2번 블록에 거래를 기록
  ⑥ 2번 이후 블록에 대해 1~5 과정을 반복

3. 작업증명의 효과

가정

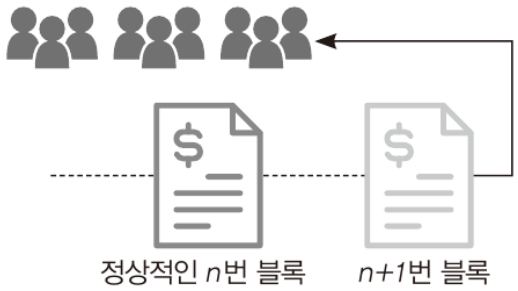
- 100개의 노드 (각 노드는 동일한 컴퓨팅 파워 가지고 있음을 가정) 참여, 현재 n번째 블록까지 생성한 상태
- 100개의 노드 중 10개 노드가 n+1번째 블록의 거래 내용을 조작한다고 할 때,

조작된 *n+1*번 블록 내용을 기반으로 100명이 10분간 풀어야 할 문제를 10명이 풀어서 답을 구한 후 조작된 블록에 첨부하고 배포해야 함



VS

*n+1*번 블록 내용을 기반으로 100명이 10분간 풀어야 할 문제를 90명이 풀어서 답을 구하고 있음



- ✓ 조작된 n+1번째 블록 내용을 기반으로 100개 노드가 10분간 풀어야 할 hashcash 문제를 10개의 노드가 해결해야 함
- ✓ 그동안 정상적인 n+1번째 블록 내용을 기반으로 100개 노드가 10분간 풀어야 할 hashcash 문제를 90개 노드가 해결하고 있는 중임
- ✓ 확률적으로 정상적인 n+1번째 블록에 대한 hashcash 문제가 훨씬 발리 해결될 것이고, n+1번째 블록을 생성한 후 n+2번째 블록에 대한 hashcash 문제를 90개 노드가 해결하고 있는 상황이 됨
- ✓ 10개의 노드는 n+1번째 블록뿐만 아니라 n+2번째 블록에 대한 hashcash 문제를 해결해야 되는 지경에 이름

✓ 10개의 노드는 블록 내용을 위변조하는 것이 거의 불가능에 가까움

3. 작업증명의 효과

가정

- 이럼에도 불구하고, 10개 노드가 조작된 블록에 대한 hashcash 문제를 해결하고, 모든 노드에 배포하면
- 조작된 블록이 포함된 블록체인의 길이가 정상적인 블록체인 길이보다 짧을 것임

- 블록체인 시스템은 2개 이상의 블록체인이 배포될 때, 길이가 긴 블록체인의 채택을 원칙으로 함

4. 비트코인의 블록 구조

- 비트코인 : 블록체인 기술을 활용한 최초 분산원장 기술 (가장 잘 알려진 암호화폐 시스템)
- 비트코인의 블록 구조
  - ✓ 블록체인의 일반 구조와 동일

✓ Time 필드 : 현재 시간 기록하는 타임스탬프, 블록이 완성될 때까지 지속적으로 변한

✓ Bits 필드 : 작업증명을 위한 target 저장 (Bits에 저장된 값을 특정한 공식에 의해 target 산출하고, target보다 블록헤더의 해시값이 작아지도록 nonce 구하면 됨)

✓ Transaction counter : 거래 정보가 블록몸체에 추가될 때마다 1 증가함

✓ Coinbase transaction (생성 거래) : 초기에 무의미한 값들로 구성, 작업증명이 성공하면, 새로 발행되는 비트코인을 성공한 노드의 비트코인 주소로 입금하는 거래 정보로 변경됨

4. 비트코인의 블록 구조

- 비트코인 블록헤더를 이용하여 작업증명 mechanism 이해
- Bits는 비트코인 참여 노드의 수에 따라 값이 변함
- 실제 비트코인의 125,552번째 블록헤더의 내용
- 125,552번째 블록헤더에 대한 SHA-256 해시값은 125,553번째 블록헤더의 HashPrevBlock 필드 값으로 입력
- 필드값은 little-endian 방식으로 메모리에 저장 (하위 주소의 메모리에 낮은 자리의 숫자를 기록)

- little-endian 방식으로 메모리에 저장
- 16진수 : 0x1b0404cb

하위 주소	0xcb	0x04	0x04	0x1b	상위 주소

Version	01000000
HashPrevBlock	81cd02ab7e569e8bcd9317e2fe99fede44d49ab2b8851ba4a3080000000000000
HashMerkleRoot	e320b6c2fffc8d750423db8b1eb942ae710e951ed797f7affc8892b0f1fc122b
Time	C7f5d74d
Bits	f2b9441a
Nonce	42a14695
Transcation counter	4
Coinbase	
Transacrion	
...	

실제
Bits 값 0x1a44b9f2
Nonce 값 0x9546a142

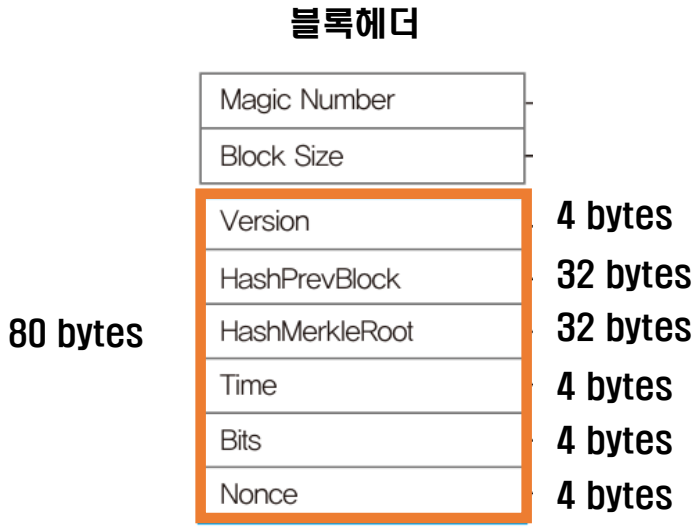
실제 비트코인 125,553번째 블록헤더의 HashPrevBlock의 값으로 입력될 해시

1dbd981fe6985776b644b173a4d0385ddc1aa2a829688d1e0000000000000000

5. 비트코인의 작업증명

- 비트코인 작업증명은 블록헤더 6개 필드에 대한 hashcash 구하는 문제임
- hashcash 난이도는 Bits 필드에 저장된 Target 값으로 결정

비트코인 작업증명
SHA-256(블록헤더 6개 필드) ≤ Target



- Target 값은 Bits에 저장된 값을 변환 공식을 적용하여 얻어짐
- 비트코인 난이도 설정값 4바이트 크기의 Bits인 경우

Bits의 값이 0x1a44b9f2 인 경우,
1. Bits의 앞 2자리와 뒤 6자리를 다음과 같이 분리함
2. 1a 44b9f2
3. Target = 0x44b9f2 * 2**(8*(0x1a-3))

- Bits : f2b9441a (little-endian) → 0x1a44b9f2

1a → 지수
44b9f2 → 계수

- 참고 : https://en.bitcoin.it/wiki/Difficulty

참고 PoW in Bitcoin

P. Mueller et al. "The Bitcoin Universe: An Architectural Overview of the Bitcoin Blockchain," LNI, DFN-Forum, 2018

- Paper 상 PoW 내용만 발췌
- To run PoW
  - ✓ Miner – hash the block header, including the block hash of the previous block, and compare it with a **predefined target** which is stored in the **nBits** parameter of the header.
  - ✓ The target is formally defined as

$$target = 2^{224}/d, \text{ where } d : \text{difficulty}$$

  - ✓ The target can be calculated from a compact scientific notation

$$target = c \times 2^{8(e-3)}$$

where the first byte of nBits denotes the exponent e and the next 3 bytes the coefficient c.

참고 PoW in Bitcoin

P. Mueller et al. "The Bitcoin Universe: An Architectural Overview of the Bitcoin Blockchain," LNI, DFN-Forum, 2018

- ✓ The proof of work is formally defined as

$$PoW = F_d(c|x) \rightarrow SHA256\,(SHA256(h|x)) \leq 2^{224/d}$$

where h can be seen as a *challenge*, x is the *nonce*, d is the actual *difficulty*

- ✓ If $F_d(c|x)$ is not smaller than the current target, the miner will modify the nonce (1씩 증가) and calculate the block hash again.
- ✓ At the current level of *difficulty* in the Bitcoin network, miners have to try **quadrillions (1000조) of times** before finding a nonce that results in a low enough block header hash.
- ✓ The length of time it takes to mine a block can be controlled with the *difficulty* d. In order to keep the block generation time to **10 minutes** on average regardless of the technology (increasing compute power of miners) used, the difficulty of mining must be adjusted.
- ✓ In fact, *difficulty* is a dynamic parameter that is periodically (*every 2016 blocks or every 14 days*) adjusted to meet a 10-minute block target. Adjustment of the difficulty occurs automatically and independently on every full node after every 2016 blocks, and all nodes retarget the proof of work difficulty. The equation for retargeting difficulty compares the time to find the last **2,016 blocks** to **20,160 minutes**, i.e., the time needed based on the 10-minute average block generation time. This can be put in simple terms as follows: if the network is finding blocks faster than every 10 minutes, the difficulty increases; if block creation time is slower than expected, the difficulty decreases.

5. 비트코인의 작업증명

```
from hashlib import sha256 as sha
import codecs

# 인자 bits를 실제 16진수 값으로 리턴
def decodeBitcoinVal(bits):
    decode_hex = codecs.getdecoder('hex_codec')
    print(decode_hex(bits))
    binn = decode_hex(bits)[0]
    ret = codecs.encode(binn[::-1], 'hex_codec')
    return ret

def getTarget(bits):
    bits = decodeBitcoinVal(bits)
    bits = int(bits, 16)

    print('Bits = %x' %bits)
    bit1 = bits >> 4*6
    base = bits & 0x00ffffff

    sft = (bit1 - 0x3)*8
    target = base << sft
    print('Target = %x' %target)

Bits = 'f2b9441a'
getTarget(Bits)
```

- little-Endian으로 저장된 Bits를 Target으로 변하는 Python code

✓ 비트코인 블록헤더의 필드에 저장된 형식(little-endian 16진수)의 bits 값에 해당하는 실제 16진수를 나타내는 바이트 객체 리턴

✓ 16진수값 (little-endian) 표현된 문자열 Bits를 입력받고, 실제 16진수로 변환한 후 (decode_hex(bits), 해당되는 정수값 리턴 (ret)

✓ binn은 바이트 객체로 표현 – 바이트 순서를 거꾸로 하면 원래의 16진수가 됨

✓ bits → target로 변환

결과

```
(b'\xf2\xb9D\x1a', 8)
Bits = 1a44b9f2
Target = 44b9f2000000000000000000000000000000000000000000000000
```

✓ 실제 비트코인 125,552번째 Bits 값임

5. 비트코인의 작업증명

```
from hashlib import sha256 as sha
import codecs

# 인자 bits를 실제 16진수 값으로 리턴
def decodeBitcoinVal(bits):
    decode_hex = codecs.getdecoder('hex_codec')
    print(decode_hex(bits))
    binn = decode_hex(bits)[0]
    ret = codecs.encode(binn[::-1], 'hex_codec')
    return ret

def getTarget(bits):
    bits = decodeBitcoinVal(bits)
    bits = int(bits, 16)

    print('Bits = %x' %bits)
    bit1 = bits >> 4*6
    base = bits & 0x00ffffff

    sft = (bit1 - 0x3)*8
    target = base << sft
    print('Target = %x' %target)

Bits = 'f2b9441a'
getTarget(Bits)
```

결과

```
(b'\xf2\xb9D\x1a', 8)
Bits = 1a44b9f2
Target = 44b9f200000000000000000000000000000000000000000000000000
```

- ✓ 바이트 객체가 표현하는 16진수를 정수값으로 변환
- ✓ Bits = 1a44b9f2
- ✓ bits의 첫 1바이트 1a를 얻어 bit1에 할당
- ✓ 나머지 3바이트 44b9f2를 base에 할당

3. Target = 0x44b9f2 * 2**(8*(0x1a-3))

5. 비트코인의 작업증명

Target = 0x44b9f2 * 2**(8*(0x1a-3)) → Target = 44b9f2000000000000000000000000000000000000000000000000

- 비트코인 블록헤더에 기록된 6개 필드에 대한 SHA-256 해시값이 Target 보다 작게되는 nonce값 찾음
- SHA-256 해시는 32 Byte로 표현되므로, Target을 32 Byte로 표현

```
Target = 00000000000044b9f2000000000000000000000000000000000000000000000000
```

- 다음 식을 만족하는 Nonce 값을 계산 – 비트코인의 125,552번째 블록이 완성

```
SHA-256(Version+HashPrevBlock+HashMerkleRoot+Time+Bits+Nonce)
≤ 00000000000044b9f2000000000000000000000000000000000000000000000000
```

5. 비트코인의 작업증명

- 비트코인의 125,552번째 블록에 대한 Nonce값 검증하는 코드
  - ✓ 125,552번째 블록은 이미 오래전에 만들어진 블록이므로, 기록된 Nonce 값이 이전 hashcash 문제를 만족하는 값일 것임

```python
from hashlib import sha256 as sha
import codecs

# 인자 bits를 실제 16진수 값으로 리턴
def decodeBitcoinVal(bits):
    decode_hex = codecs.getdecoder('hex_codec')
    binn = decode_hex(bits)[0]
    ret = codecs.encode(binn[::-1], 'hex_codec')
    return ret

def getTarget(bits):
    bits = decodeBitcoinVal(bits)
    bits = int(bits, 16)
    bit1 = bits >> 4*6
    base = bits & 0x00ffffff

    sft = (bit1 - 0x3)*8
    target = base << sft

    return target
```

```python
def validatePoW(header):
    block_version = header[0]
    hashPrevBlock = header[1]
    hashMerkleRoot = header[2]
    Time = header[3]
    Bits = header[4]
    nonce = header[5]

    # 블록 헤더의 모든 값을 더하고 이를 바이트 객체로 변경
    decode_hex = codecs.getdecoder('hex_codec')
    header_hex = block_version + hashPrevBlock + hashMerkleRoot + Time + Bits + nonce
    header_bin = decode_hex(header_hex)[0]
```

블록 header 정보 읽기

Blockchain에서 Hash 활용


```
def validatePoW(header):

    ### 앞 장 내용 중략

    # 실제 비트코인에서는 블록 헤더의 SHA256 해시에 대한 SHA256해시를 이용해서 작업증명을 함
    hash = sha(header_bin).digest()                    (1)
    hash = sha(hash).digest()                          (2)
    PoW = codecs.encode(hash[::-1], 'hex_codec')       (3)

    # 헤더의 Bits 값을 이용해 실제 target 값 추출
    target = getTarget(Bits)                           (1)
    target = str(hex(target))                          (2)
    target = '0'*(66-len(target)) + target[2:]         (3)

    print('target\t=', target)
    print('PoW\t=', PoW.decode())

    # 작업 증명이 성공한 건지 체크
    if int(PoW, 16) <= int(target, 16):
        print('+++ Accept this Block')
    else:
        print('--- Reject this Block')
```

실행 결과

```
target  = 00000000000044b9f2000000000000000000000000000000000000000000000000
PoW = 0000000000000001e8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d
+++ Accept this Block
```

(1) hash = b'\xb9\xd7QS5\x93\xac\x10\xcd\xfb{\x8e\x03\xca\xd8\xba\xbcg\xd8\xea\xea\xc0\xa3i\x9b\x82\x85}\xac\xac\x93\x90'

(2) hash = b'\x1d\xbd\x98\x1f\xe6\x98Wv\xb6D\xb1s\xa4\xd08]\xdc\x1a\xa2\xa8)h\x8d\x1e\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'

(3) PoW = b'0000000000000001e8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d'

(4) PoW.decode() = 0000000000000001e8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d

(1) target = 110438984653392107399822606842181601255647711092336854093004800

(2) target = 0x44b9f2000000000000000000000000000000000000000000000000

(3) target = 00000000000044b9f2000000000000000000000000000000000000000000000000

Blockchain에서 Hash 활용

```
def main():
    block_version = '01000000'
    hashPrevBlock = \
        '81cd02ab7e569e8bcd9317e2fe99f2de44d49ab2b8851ba4a308000000000000'
    hashMerkleRoot = \
        'e320b6c2fffc8d750423db8b1eb942ae710e951ed797f7affc8892b0f1fc122b'
    Time = 'c7f5d74d'
    Bits = 'f2b9441a'
    nonce = '42a14695'
    header = [block_version, hashPrevBlock, hashMerkleRoot, Time, Bits, nonce]
    validatePoW(header)

main()
```

- **실행 결과**

```
target  = 00000000000044b9f2000000000000000000000000000000000000000000000000
PoW = 0000000000000001e8d6829a8a21adc5d38d0a473b144b6765798e61f98bd1d
+++ Accept this Block
```

6. 비트코인의 주소 생성

- 비트코인 주소 (Bitcoin address)
  - ✓ 은행계좌 번호와 비슷한 역할을 하는 블록체인 주소
  - ✓ A가 B에게 비트코인 전송시, A의 비트코인 주소에 담겨있는 비트코인을 B의 비트코인 주소로 보냄
  - ✓ 일반적인 경우 1, 다중 서명이 가능한 경우 3으로 시작하는 34자 길이로 되어 있음

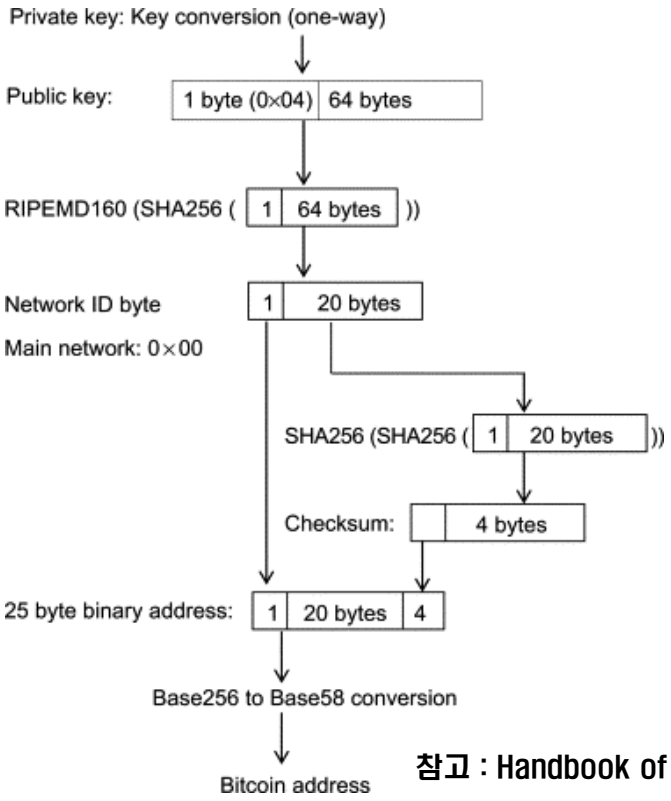
일반적인 비트코인 주소 예: 1BvBMSEYstWetqTFn5Au4m4GFg7xJaNVN2
다중 서명 가능 비트코인 주소 예: 3J98t1WpEZ73CNmQviecrnyiWrnqRhWNLy

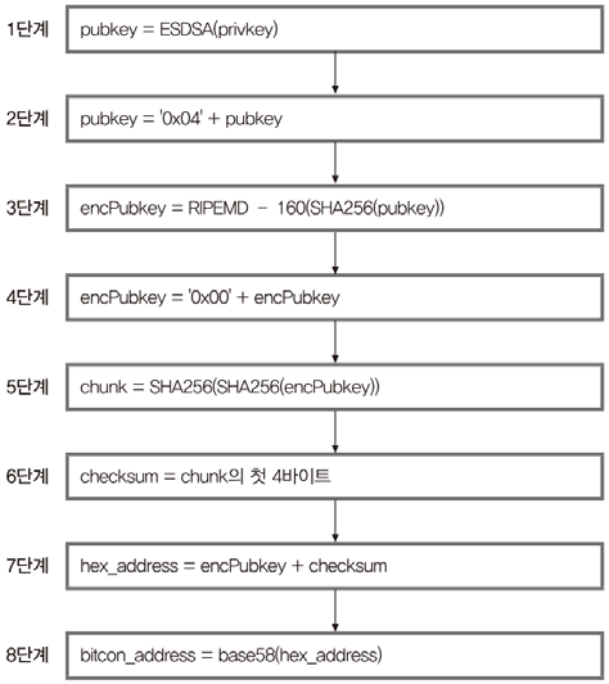
❖ 블록체인 시스템 상 거래 생성시 전자서명 알고리즘 활용
- 비트코인 거래시 비트코인 전송자(A)는 자신의 개인키로 서명하고, 거래 정보와 자신의 공개키를 상대방(B)의 비트코인 주소로 전달
- B는 동봉된 공개키의 서명을 확인하고, 검증되면 거래가 성사

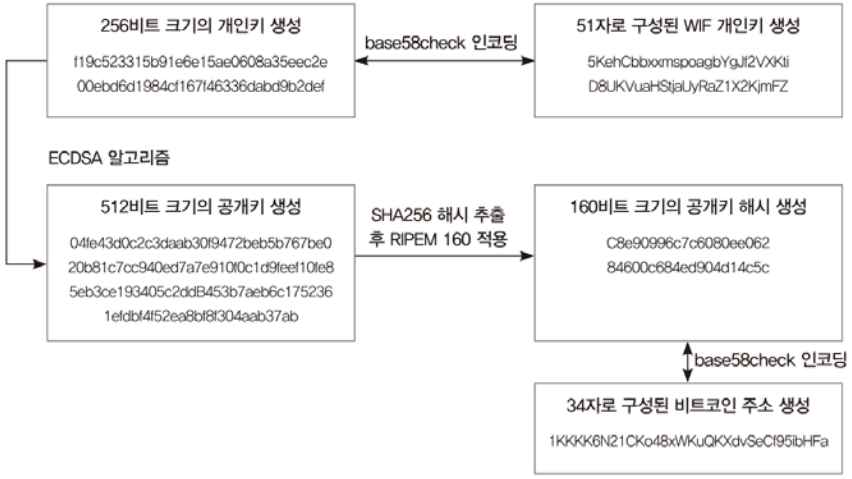
- 비트코인 주소도 공개키 시스템 기반으로 생성됨 (ECDSA 알고리즘 사용)

6. 비트코인의 주소 생성

- 비트코인 주소 생성 mechanism



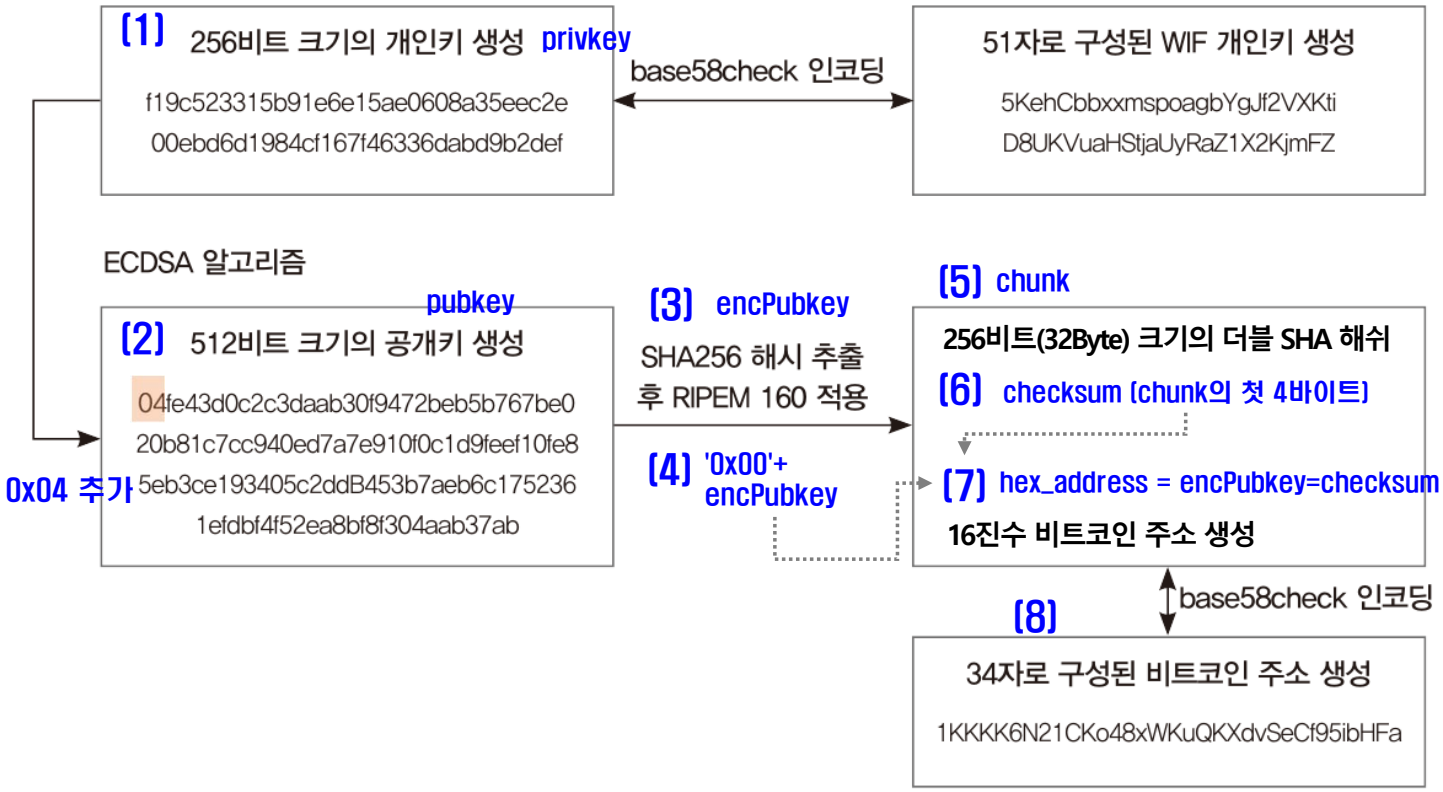


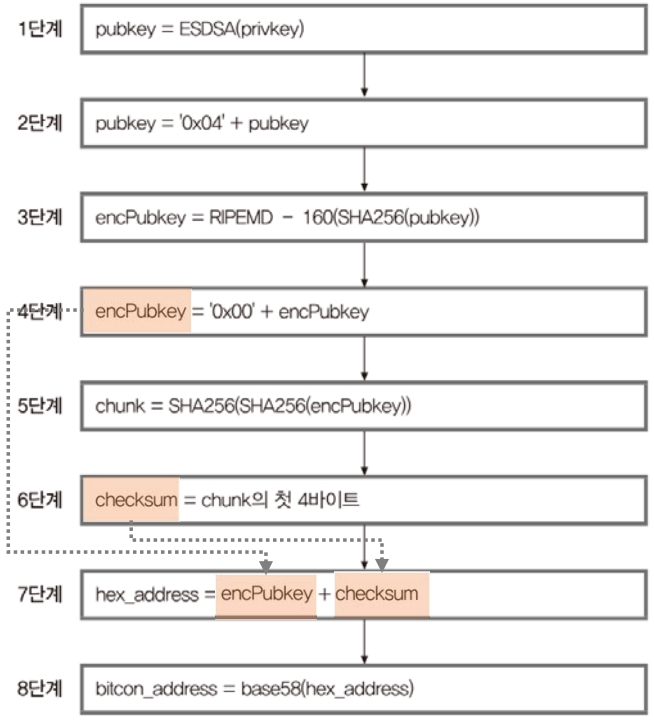


참고 : Handbook of Digital Currency, Chapter 3 Bitcoin Mining Technology, 2015

6. 비트코인의 주소 생성

- 비트코인 주소 생성 mechanism





- ❖ RIPEMD-160 : (RIPE Message Digest) 160 비트 메시지 요약 생성, 32비트 연산에 최적화. 다른 버전과 다르게 특허 제약이 없음
- ❖ RIPEMD-128, RIPEMD-160, RIPEMD-256, RIPEMD-320

6. 비트코인의 주소 생성

- 비트코인 주소 생성 Python
  - ✓ ecdsa, base58check 모듈 설치

```
(venv) C:\Users\skylee\Documents\PyCrypto>pip install base58check
Collecting base58check
  Downloading base58check-1.0.2-py2.py3-none-any.whl (6.0 kB)
Installing collected packages: base58check
Successfully installed base58check-1.0.2

(venv) C:\Users\skylee\Documents\PyCrypto>pip install ecdsa
Collecting ecdsa
  Downloading ecdsa-0.15-py2.py3-none-any.whl (100 kB)
     |████████████████████████████████| 100 kB 441 kB/s
Collecting six>=1.9.0
  Downloading six-1.15.0-py2.py3-none-any.whl (10 kB)
Installing collected packages: six, ecdsa
Successfully installed ecdsa-0.15 six-1.15.0

(venv) C:\Users\skylee\Documents\PyCrypto>
```

```python
import os
import hashlib
from hashlib import sha256 as sha
from base58check import b58encode
import ecdsa

def ripemd160(x):
    ret = hashlib.new('ripemd160')
    ret.update(x)
    return ret
```

- X에 대한 RIPEMD-160 해시값 리턴

- base58check : A modified Base 58 binary-to-text encoding known as Base58Check is used for encoding Bitcoin addresses.

6. 비트코인의 주소 생성

```
def generateBitcoinAddress():
    # 개인키 생성
    privkey = os.urandom(32)                    (1) BYTE

    # WIF (Wallet Import Format) -> 비트코인 거래위한 약식 개인키 생성
    fullkey = '80' + privkey.hex()              (2),(3) HEX

    a = bytes.fromhex(fullkey)                  (4) BYTE
    sha_a = sha(a).digest()                     (5) BYTE
    sha_b = sha(sha_a).hexdigest()              (6) HEX
    c = bytes.fromhex(fullkey+sha_b[:8])        (7) BYTE

    WIF = b58encode(c)                          (8) BYTE
    # WIF
```

- 비트코인은 비트코인 거래를 위해 WIF라는 개인키를 약식으로 구성한 키 이용 (51자)

- os.urandom() : 원하는 길이(byte)의 랜덤한 unsigned 값 생성, 암호화 알고리즘의 키 생성에 유용
- privkey : 32byte 개인키 (os.urandom(32)) 생성
- privkey.hex() : 바이트 객체인 privkey를 16진수로 표현되는 문자열로 변경

(1) privkey = b'p\xcck\xa0u\xaf\xf3\xa5;0\xbc\xe8</VBa\xa2\x85q\xd8\x92.\xc1\x9b\xd6\x88\x9b\xc7\x97\xdd\t'

(2) privkey.hex() = 70cc6ba075aff3a53b4fbce83c2f564261a28571d8922ec19bd6889bc797dd09

(3) fullkey = 8070cc6ba075aff3a53b4fbce83c2f564261a28571d8922ec19bd6889bc797dd09

- bytes.fromhex() : 16진수로 표현된 문자열을 바이트 객체로 변환

(4) a = b'\x80p\xcck\xa0u\xaf\xf3\xa5;0\xbc\xe8</VBa\xa2\x85q\xd8\x92.\xc1\x9b\xd6\x88\x9b\xc7\x97\xdd\t'

(5) sha_a = b'\xfe\t\xf356\xba-\x9d\x04L\x91J\x19K\x19\xcaUe|9q\xe46R\x81\xa8\x0b\xf9:\x17\x13'

(6) sha_b = ff44d279779e44bdd6b0f8515ccb8eea1200cb0f6f61b65f14e19dc70841264e

(7) c = b'\x80p\xcck\xa0u\xaf\xf3\xa5;0\xbc\xe8</VBa\xa2\x85q\xd8\x92.\xc1\x9b\xd6\x88\x9b\xc7\x97\xdd\t\xffD\xd2y'

(8) WIF = b'5JfxsK1sbzZeu3Jgfaemv5kHQHDkKE9aGrYkF7UPVMemjchA9zg'

len(WIF) = 51

6. 비트코인의 주소 생성

```
# 1단계 ECDSA 공개키 획득
signing_key = ecdsa.SigningKey.from_string(privkey, curve=ecdsa.SECP256k1)
verifying_key = signing_key.get_verifying_key()   [1] BYTE
pubkey = (verifying_key.to_string()).hex()

# 2단계
pubkey = '04' + pubkey                            [2] HEX

# 3단계
pub_sha = sha(bytes.fromhex(pubkey)).digest()     [3] BYTE
encPubkey = ripemd160(pub_sha).digest()

# 4단계
encPubkey = b'\x00' + encPubkey                   [4] BYTE
```

- 32바이트 privkey로 ECDSA 서명키 (signing_key) 생성
  - ✓ curve=ecdsa.SPECP256k1 : ECC 타원곡선 상수 (ECC.generate(curve='P-256')의 'P-256' 타원곡선 상수와 유사함
- ECDSA 서명키에 해당되는 인증키 (verifying_key)을 이용하여 16진수 문자열의 공개키 pubkey 생성

(1) verifying_key = VerifyingKey.from_string(b'\x02\xcf\x1bT\x0f\xb9\x95\x17\xb3\xe4Unz\x1e\xca\xf3hd~\x8f\xe4J\xb8\xf1\xf6\xa8\x1b\x89\x1e9vov', SECP256k1, sha1)

(2) pubkey = 04cf1b540fb99517b3e4556e7a1ecaf368647e8fe44ab8f1f6a81b891e39766f768c82b830206f5b737c307e821d7c086938c8989d8bd9855c870999c57bba238c

(3) pub_sha = b'\xce\xa9\xda\xf9\x06\x14\xf2\x8b\xe8\xf0[\xac&\xe4\xa40\xd2\x8d\x96?\x828\xdbj\x86\xf6b\x8f\x05\xbd\xadA'

(4) encPubkey= b'\x00}^\xba\xc2\x87!\xeb@d\x9c\xd0\x14]\xc0L\xe3\x0b\xa5\xa7v'

- ecdsa 코드 참조 : https://github.com/warner/python-ecdsa

6. 비트코인의 주소 생성

```
    # 5단계
    chunk = sha(sha(encPubkey).digest()).digest()      [1] BYTE

    # 6단계
    checksum = chunk[:4]                               [2] BYTE

    # 7단계
    hex_address = encPubkey + checksum                 [3] BYTE

    # 8단계
    bitcoinAddress = b58encode(hex_address)            [4] BYTE

    # WIF와 생성된 비트코인 주소 출력
    print('+++WIF = ', WIF.decode())
    print('+++Bitcoin Address = ', bitcoinAddress.decode())

generateBitcoinAddress()
```

[1] chunk = b'L8\xd1\xfcu\xea\x1b]\xf8!b\x8e\xad\x1c\xef\xa2\t\xf7\xf3\xa8\xcd\xe8\xcdD\x0c\xa3igz\xfe\xf2f'

[2] checksum = b'L8\xd1\xfc'

[3] hex_address = b'\x00}^\xba\xc2\x87!\xeb@d\x9c\xd0\x14]\xc0L\xe3\x0b\xa5\xa7vL8\xd1\xfc'

[4] bitcoinAddress = b'1CRtwuKHnD9d1dbu62vArrZ6phBMuUfEcT'

실행 결과

+++WIF = 5JiE8Q5NahB3bSqQyMd9oNX66zZ1TVUUNaQ4cX357QLeLAzmDuT

+++Bitcoin Address = 1CRtwuKHnD9d1dbu62vArrZ6phBMuUfEcT