Programming Study
Cryptography

Sungwoo Nam

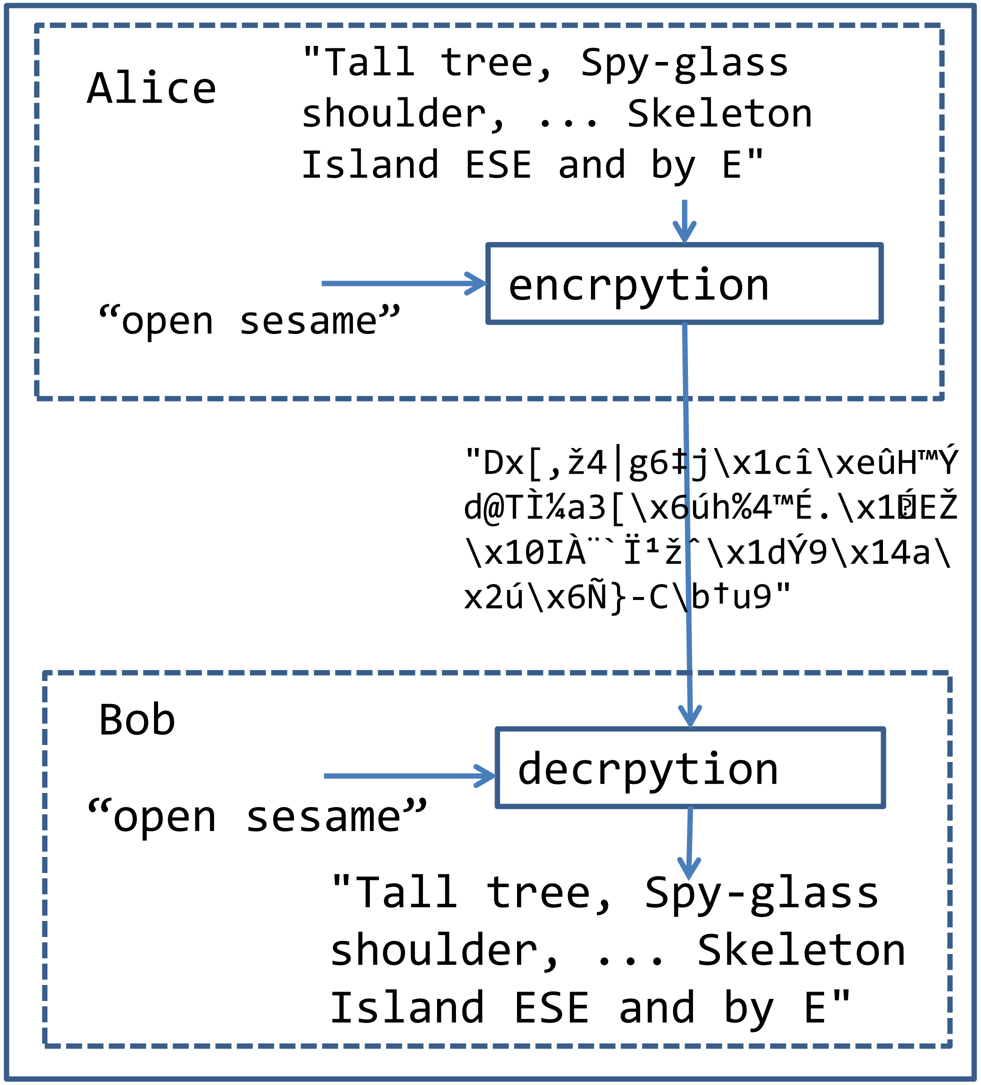
2018.3.9

Poco::Crypto

- PocoCrypto.dll
- Depends on OpenSSL :
  libssl-1_1.dll, libcrypto-1_1.dll

Symmetric Encryption

```
#include "Poco/Crypto/Cipher.h"
#include "Poco/Crypto/CipherKey.h"
#include "Poco/Crypto/CipherFactory.h"

auto& factory = CipherFactory::defaultFactory();
string transmission;

{
  Cipher* alice =
    factory.createCipher( CipherKey(
      "aes-256-ecb",
      "open sesame"));
  transmission = alice->encryptString(
    "Tall tree, Spy-glass shoulder, ...
     Skeleton Island ESE and by E");
}

{
  Cipher* bob =
    factory.createCipher( CipherKey(
      "aes-256-ecb",
      "open sesame"));
  string decrypted =
    bob->decryptString(transmission);
  assert(decrypted ==
    "Tall tree, Spy-glass shoulder, ...
    Skeleton Island ESE and by E");
}
```

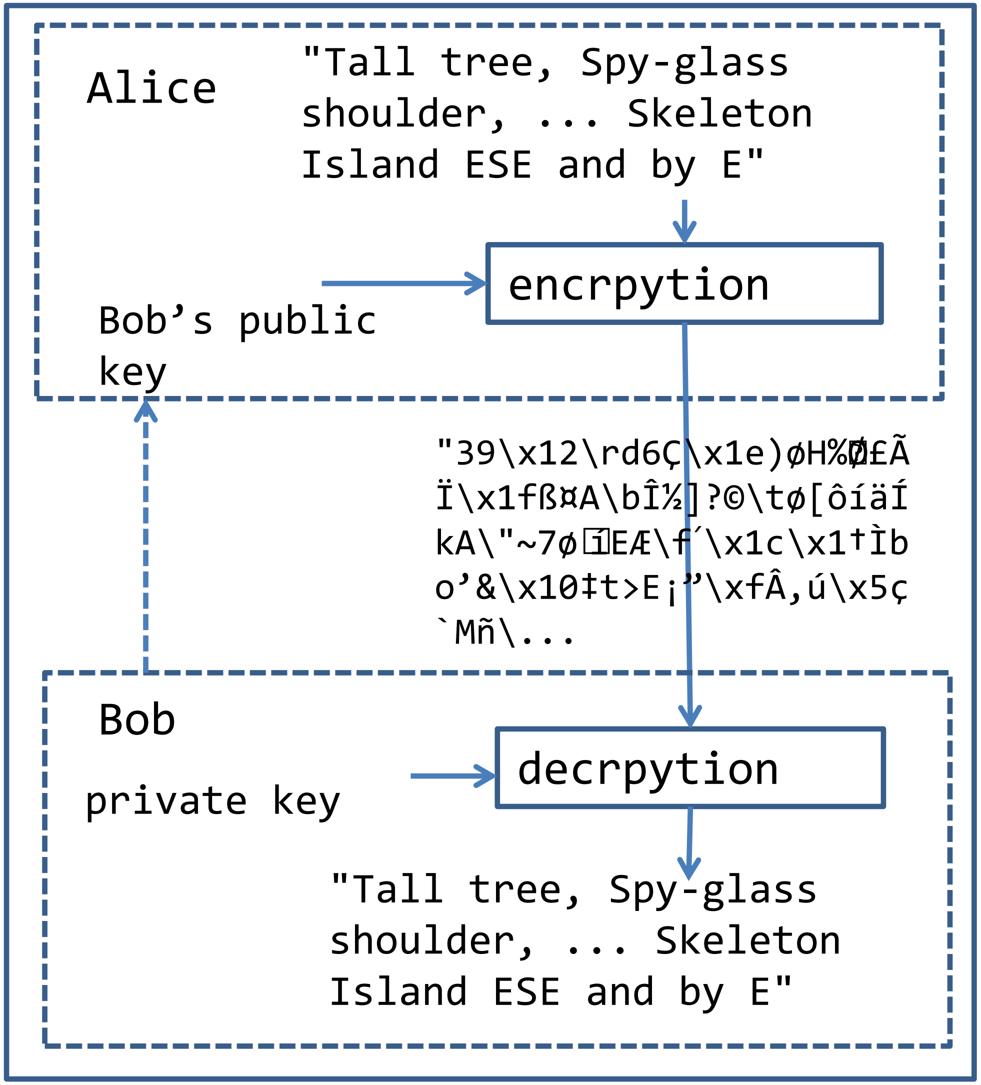


OpenSSL public and private key

```
>openssl.exe genrsa -des3 -out private_bob.pem 2048
Generating RSA private key, 2048 bit long modulus
...................+++
.................................................................+++
unable to write 'random state'
e is 65537 (0x10001)
Enter pass phrase for private_bob.pem:
Verifying - Enter pass phrase for private_bob.pem:


>openssl.exe rsa -in private_bob.pem -outform PEM -pubout -out public_bob.pem
Enter pass phrase for private_bob.pem:
writing RSA key
```

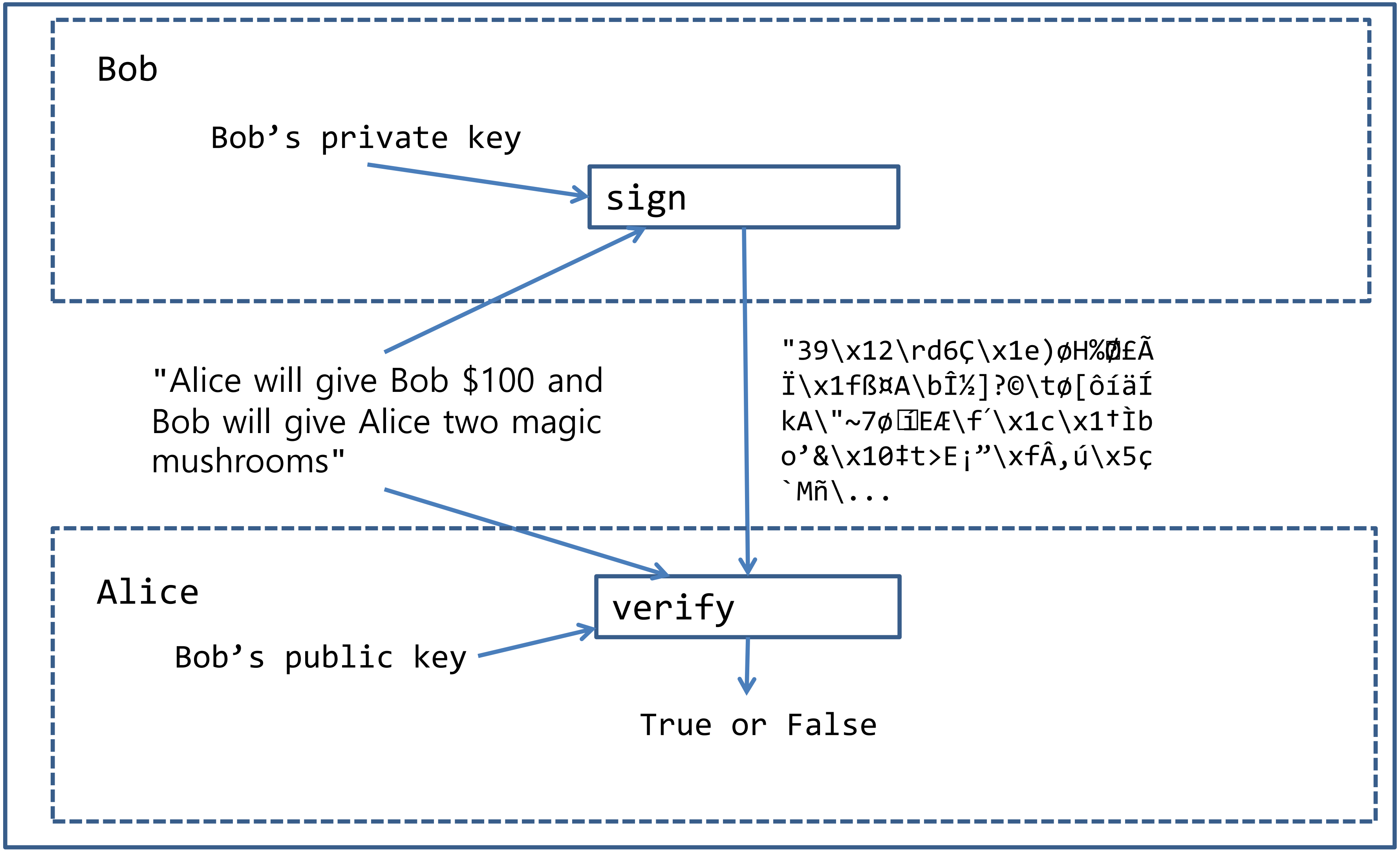
Asymmetric Encryption

```
#include "Poco/Crypto/RSAKey.h"
#include "Poco/Crypto/RSADigestEngine.h“

auto& factory = CipherFactory::defaultFactory();
string transmission;

{
  Cipher* alice = factory.createCipher(RSAKey(
    "public_bob.pem"));
  transmission = alice->encryptString(
    "Tall tree, Spy-glass shoulder, ...
    Skeleton Island ESE and by E");
}

{
  Cipher* bob = factory.createCipher(RSAKey(
    "", "private_bob.pem", "bobbob"));
  string decrypted =
    bob->decryptString(transmission);
  assert(decrypted == "Tall tree, Spy-glass
  shoulder, ... Skeleton Island ESE and by E");
}
```



Alice
"Tall tree, Spy-glass shoulder, ... Skeleton Island ESE and by E"
encrpytion
Bob’s public key
"39\x12\rd6Ç\x1e)øH%□Ø£Ã Ï\x1fß¤A\bÎ½]?©\tø[ôíäÍ kA\"~7ø□íEÆ\f´\x1c\x1†Ìb o’&\x10‡t>E¡”\xfÂ,ú\x5ç `Mñ\...
Bob
decrpytion
private key
"Tall tree, Spy-glass shoulder, ... Skeleton Island ESE and by E"

Digital Signature



Bob

Bob’s private key

sign

"Alice will give Bob $100 and Bob will give Alice two magic mushrooms"

```
"39\x12\rd6Ç\x1e)øH%Ø£Ã
Ï\x1fß¤A\bÎ½]?©\tø[ôíäÍ
kA\"~7ø□EÆ\f´\x1c\x1†Ìb
o’&\x10‡t>E¡”\xfÂ,ú\x5ç
`Mñ\...
```

Alice

Bob’s public key

verify

True or False

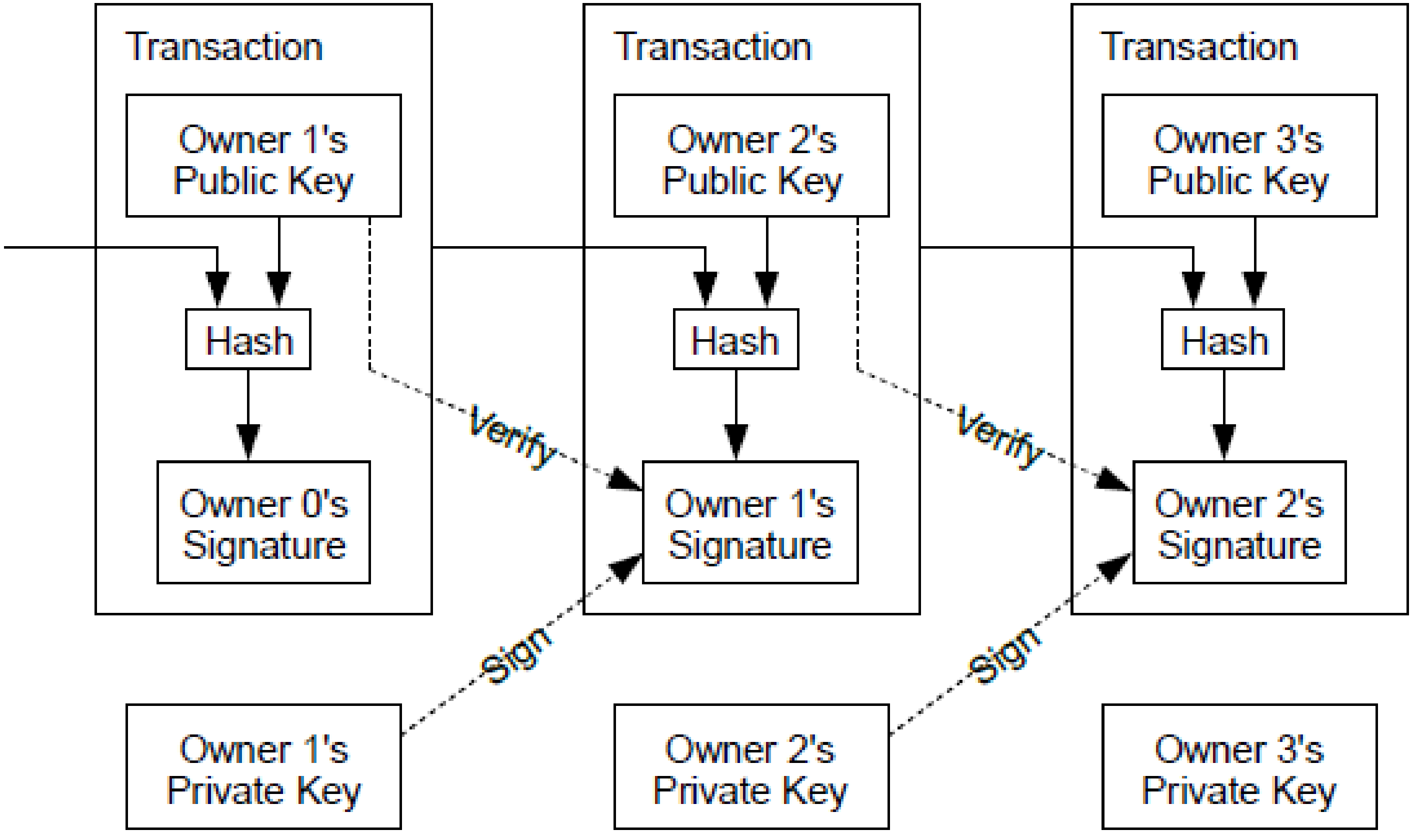
Digital Signature in coding

```
// Poco::DigestEngine::Digest is std::vector<unsigned char>
tuple<string, Poco::DigestEngine::Digest > transmission;

{
  string contract =
    "Alice will give Bob $100 and Bob will give Alice two magic mushrooms";
  RSADigestEngine bob(RSAKey("", "private_bob.pem", "bobbob"));
  bob.update(contract);
  transmission = make_tuple(contract, bob.signature());
}

{
  RSADigestEngine alice(RSAKey("public_bob.pem"));
  alice.update(get<0>(transmission));
  bool isValid = alice.verify(get<1>(transmission));
  assert(isValid == true);
}
```

Digital Signature – cheating

```
// Poco::DigestEngine::Digest is std::vector<unsigned char>
tuple<string, Poco::DigestEngine::Digest > transmission;

{
  string contract =
    "Alice will give Bob $100 and Bob will give Alice two magic mushrooms";
  RSADigestEngine bob(RSAKey("", "private_bob.pem", "bobbob"));
  bob.update(contract);
  transmission = make_tuple(contract, bob.signature());
}

{
  RSADigestEngine alice(RSAKey("public_bob.pem"));
  alice.update(
    "Alice will give Bob $100 and Bob will give Alice three magic mushrooms");
  bool isValid = alice.verify(get<1>(transmission));
  assert(isValid == false);
}
```

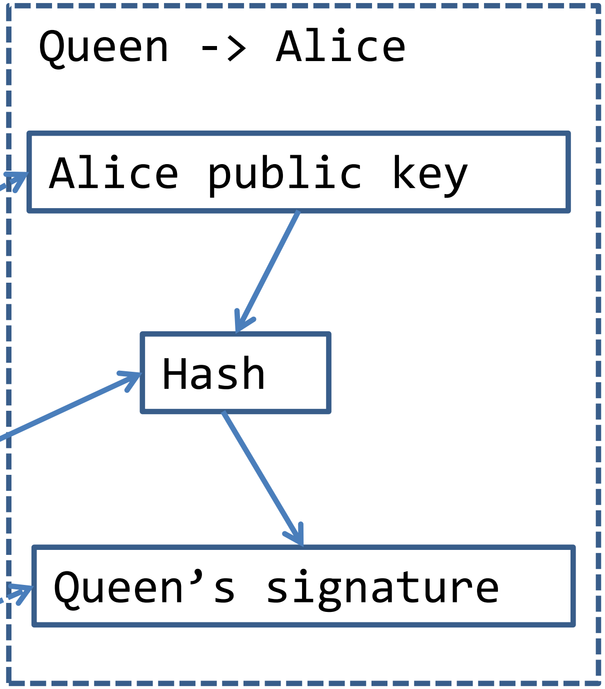
Chained Transaction



Ref : Bitcoin : A Peer-to-Peer Electronic Cash System, Satoshi Nakamoto

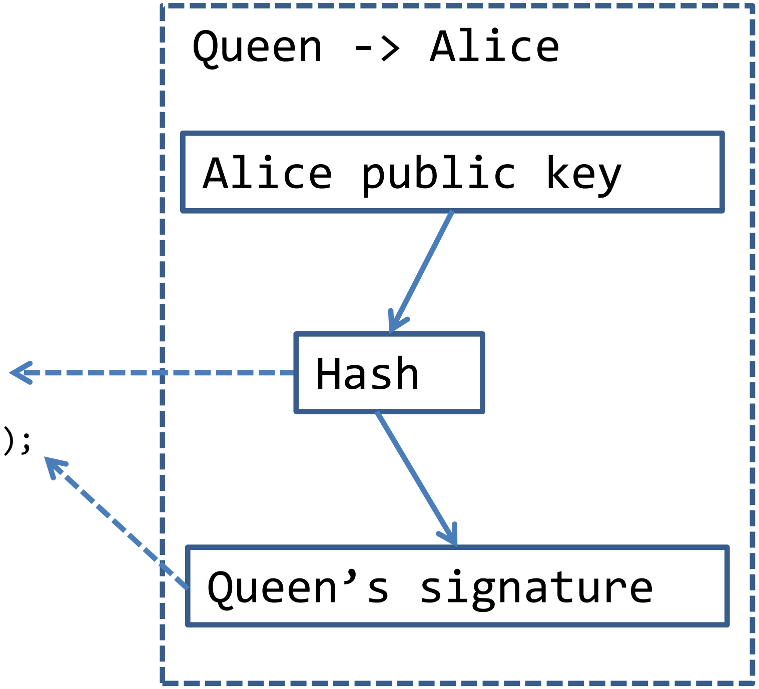
Transaction – Queen to Alice

```
struct Transaction
{
   string Hash;
   vector<unsigned char> Signature;
};

Transaction queenToAlice;
{
   // Queen
   RSAKey alicePublic("public_alice.pem");
   stringstream alicePublickey;
   alicePublic.save(&alicePublickey);

   Crypto::DigestEngine hasher("SHA256");
   hasher.update("Queen's own right to create coin");
   hasher.update(alicePublickey.str());
   string hash = Crypto::DigestEngine::digestToHex(
     hasher.digest());

   RSADigestEngine queen(RSAKey("", "private_queen.pem", "queenqueen"));
   queen.update(hash);

   queenToAlice = Transaction{ hash, queen.signature() };
}
```



Transaction – Alice verification

```
{
   // Alice
   RSADigestEngine alice(RSAKey("public_queen.pem"));
   alice.update( queenToAlice.Hash );
   assert( true == alice.verify(queenToAlice.Signature));
}
```



Queen -> Alice

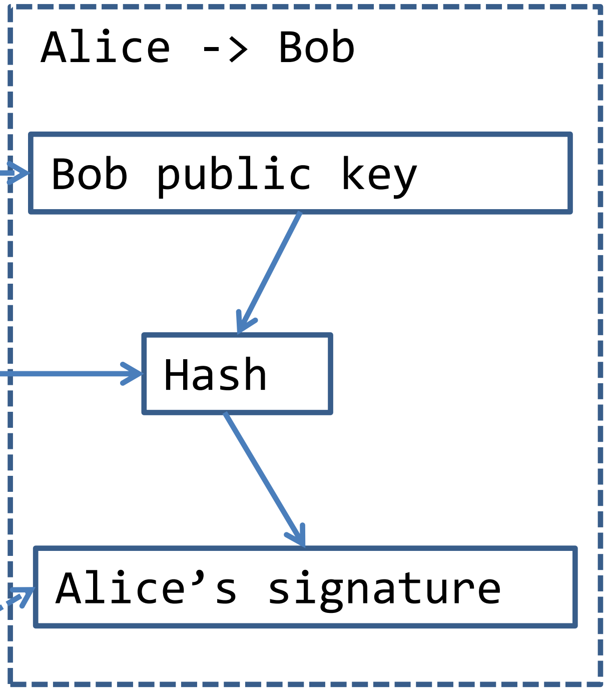
Alice public key

Hash

Queen’s signature

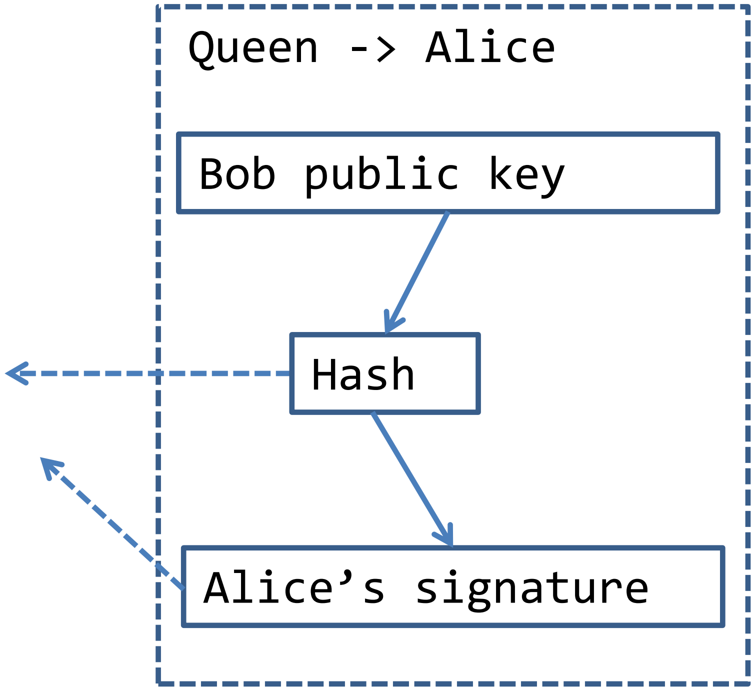
Transaction –Alice to Bob

```
Transaction aliceToBob;
{
  // Alice
  RSAKey bobPublic("public_bob.pem");
  stringstream bobPublickey;
  bobPublic.save(&bobPublickey);

  Crypto::DigestEngine hasher("SHA256");
  hasher.update(queenToAlice.Hash);
  hasher.update(queenToAlice.Signature.data(),
    queenToAlice.Signature.size());
  hasher.update(bobPublickey.str());
  string hash = Crypto::DigestEngine::digestToHex(
    hasher.digest());

  RSADigestEngine alice(RSAKey("",
    "private_alice.pem", "alicealice"));
  alice.update(hash);

  aliceToBob = Transaction{ hash, alice.signature() };
}
```



Transaction – Bob verification

```
{
  // Bob
  RSADigestEngine bob(RSAKey("public_alice.pem"));
  bob.update(aliceToBob.Hash);
  assert(true == bob.verify(aliceToBob.Signature));
}
```



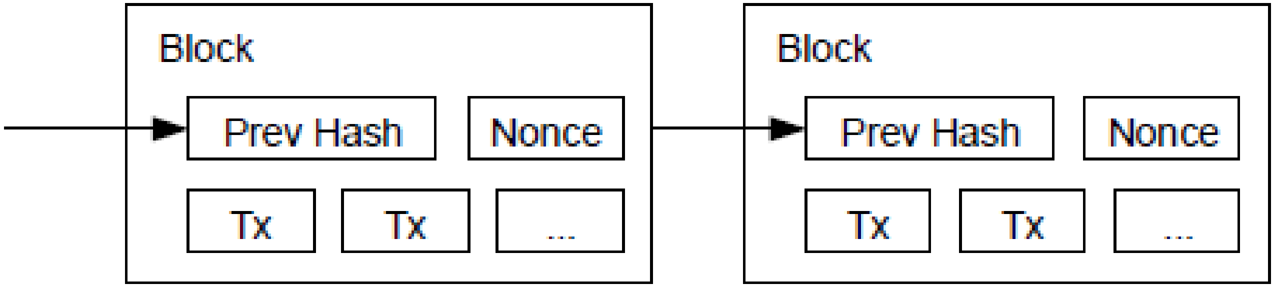
Queen -> Alice

Bob public key

Hash

Alice’s signature

Block Chain



1) New transactions are broadcast to all nodes.
2) Each node collects new transactions into a block.
3) Each node works on finding a difficult proof-of-work for its block.
4) When a node finds a proof-of-work, it broadcasts the block to all nodes.
5) Nodes accept the block only if all transactions in it are valid and not already spent.
6) Nodes express their acceptance of the block by working on creating the next block in the chain, using the hash of the accepted block as the previous hash.

Proof of work

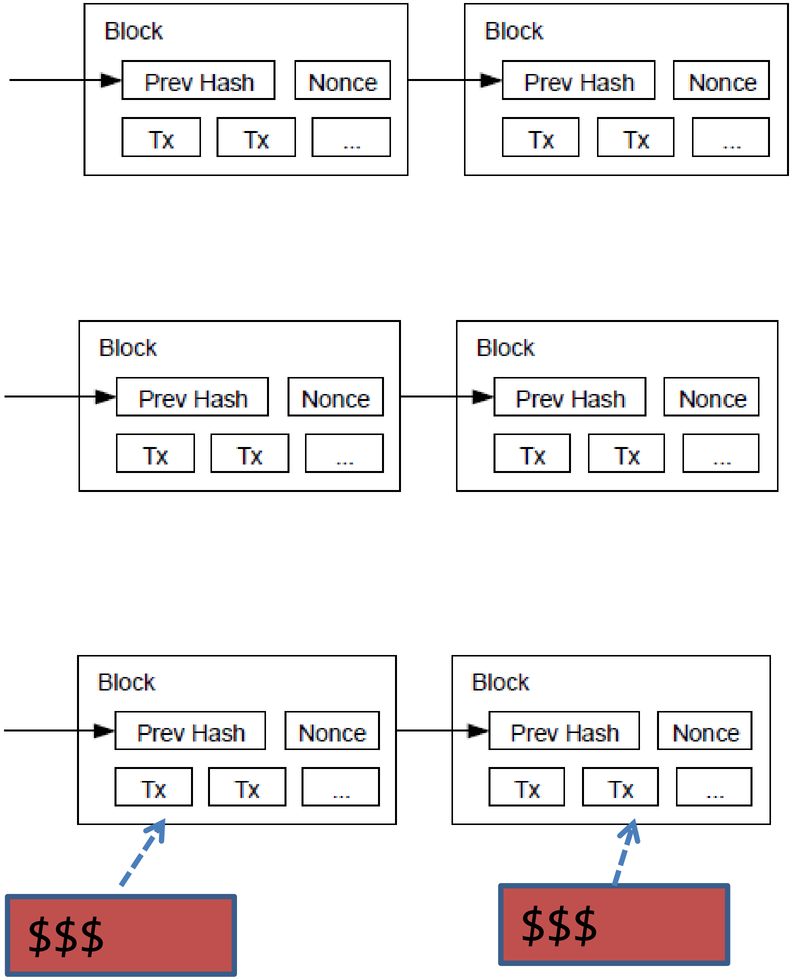
```
struct Transaction {  string Hash, Signature; };

struct Block {
  string PrevHash, Nounce;
  vector<Transaction> Transactions;
};

Block t0 = {
  "Hello, world!", "4250",  { Transaction{ "", "" } }
};
```

```
// verfiy t0 block's transactions and proof-of-work
hasher.update(t0.PrevHash);
for (auto tx : t0.Transactions) {
  hasher.update(tx.Hash); hasher.update(tx.Signature);
}
hasher.update(t0.Nounce);

t1.PrevHash = Crypto::DigestEngine::digestToHex(hasher.digest());
assert(t1.PrevHash.substr(0, 4) == "0000");
```

Proof of work

```
// gather transactions
t1.Transactions = {
  Transaction{ "ab", "cd" },
  Transaction{ "12", "34" },
  Transaction{ "56", "78" } };

// race for the nounce
for (int nounce = 0;; ++nounce)
{
   hasher.reset();
   hasher.update(t1.PrevHash);
   for (auto tx : t1.Transactions) {
     hasher.update(tx.Hash); hasher.update(tx.Signature);
   }
   hasher.update(to_string(nounce));

   string hash = Crypto::DigestEngine::digestToHex(hasher.digest());
   if (hash.substr(0, 4) == "0000") {
      t1.Nounce = to_string(nounce);
      break;
   }
}

// broadcast t1 block to world
```

Honest chain vs Attacker chain



Block
Prev Hash
Nonce
Tx
Tx
...
Block
Prev Hash
Nonce
Tx
Tx
...
Block
Prev Hash
Nonce
Tx
Tx
...
Block
Prev Hash
Nonce
Tx
Tx
...
Block
Prev Hash
Nonce
Tx
Tx
...
Block
Prev Hash
Nonce
Tx
Tx
...
$$$
$$$