P2P Systems and Blockchains
Spring 2018, 
instructor: Laura Ricci
laura.ricci@unipi.it

Lesson 11:
CRYPTOGRAPHIC TOOLBOX FOR BLOCKCHAINS
11/4/2018
OUTLINE OF THE NEXT LESSONS

• Tools for the development of block-chains

• Cryptographic tools
  • Cryptographic hash functions
  • Digital signatures

• Data structures
  • Bloom filters
  • Merkle trees
  • Patricia tries
HASH FUNCTIONS

• Arbitrary-length message to fixed-length digest

• generally used in programming to implement the “dictionary” data structure for fast lookups

• hash value is also called digest

This is a clear text that can easily read without using the key. The sentence is longer than the text above.
HASH FUNCTIONS

- Input:
  - input length is counted in bits, you can hash a 123 bits value.
  - zero input length is permitted, normally a maximum input length.
- Output:
  - fixed length output; normally 128/160/256/512-bit output length
  - a family of hash functions will share the similar design with different parameters and output length
• **Pigeonhole Principle:** states that if \( n \) items are put into \( m \) containers, with \( n > m \), then at least one container must contain more than one item
  • seems rather intuitive and naive, but it is used to demonstrate possibly unexpected results
  • since the codomain is smaller of the domain you can have collisions
HASH FUNCTIONS: PROPERTIES

Let $X$ be the domain and $Y$ the codomain of the hash function:

- for any $x \in X$, it is easy to compute $f(x)$
- preimage resistance, also hiding property: for any $y \in Y$, it is hard to find $x \in X$ such that $f(x) = y$
- one-way function
HASH FUNCTIONS: PROPERTIES

Let $X$ be the domain and $Y$ the codomain of the hash function:

- **second preimage resistance**: given $h = H(M)$, it is hard to find $M'$ that $H(M') = h$
SECOND PREIMAGE

• A function which is not collision resistant: 8-bit block parity

\[ m = 110100101000100111100101000101001000100010101 \]

\[ b_1 = 11010010 \\
   b_2 = 10001001 \\
   b_3 = 11100101 \\
   b_4 = 00010100 \\
   b_5 = 10100010 \\
   b_6 = 00010100 \]

\[ \text{digest} = 00011100 \text{ (column-wise @)} \]

• a simple way to find another message with the same hash:
  • invert any even number of bits in m that are in the same column and the parity will not change

\[ m_1 = \begin{array}{c}
   11010010 \\
   10001001 \\
   11100101 \\
   00010100 \\
   10100010 \\
   00010100 \\
\end{array} \quad m_2 = \begin{array}{c}
   11110010 \\
   10001101 \\
   11000101 \\
   00110000 \\
   10100010 \\
   00110100 \\
\end{array} \]

\[ \text{digest}(m_2) = 00011100 \]

• Hash which is not preimage resistant
Let $X$ be the domain and $Y$ the codomain of the hash function:

- **collision resistance**: given any $x_1$, it is hard to find another $x_2$ different from $x_1$ such that $H(x_1) = H(x_2)$
nobody can find x and y such that x != y and H(x)=H(y) collision exists, but it is very hard to find them
Question: what is the maximum number of guesses required to certainly find a collision?

- pick $2^{256} + 1$ distinct values in the domain
- compute the hashes of each of them, and check if any two outputs are equal
- the maximum number of guesses required to certainly find a collision is $2^{256} + 1$

$O(2^n)$ time complexity
$O(1)$ space complexity,
where $n = \text{len}(H)$
Question: what is the maximum number of guesses required to certainly find a collision?

- pick random inputs and compute their hash values
- you will find a collision with high probability long before examining $2^{256} + 1$ values
Question: what is the maximum number of guesses required to certainly find a collision?

- \( \approx 50\% \) probability of a collision after \( \approx 2^{128} \)
- randomly choose just \( 2^{128} + 1 \) inputs, roughly the square root of the number of possible outputs
- there's a chance that at least two of them are going to collide.

\[
\begin{align*}
O(2^{n/2}) & \text{ time complexity} \\
O(2^{n/2}) & \text{ space complexity, } \\
& \text{where } n = \text{len}(H)
\end{align*}
\]
THE BIRTHDAY PARADOX

• What is the minimum value of $k$ such that the probability is greater than 0.5 that at least two people in a group of $k$ people have the same birthday?

• Given a hash function $H$, with $n$ possible outputs and a specific value $H(x)$, if $H$ is applied to $k$ random inputs, what must be the value of $k$ so that the probability that at least one input $y$ satisfies $H(y) = H(x)$ is 0.5?
THE BIRTHDAY PARADOX

- Within a group of k people selected at random, what is the probability that two or more will share a birthday?

- Hypothesis:
  - a year is made of 365 days (no leap years)
  - all days are equally probable

- result: if k = 23, then two people will share a birthday with a probability just above 50%.
THE BIRTHDAY PARADOX

- Another formulation: find the smallest number $n$ of people such that ($E=$ expected value)

\[ E(\text{pairs of common birthday}) \geq 1 \]

ways to pair 4 people:

\[ 3 + 2 + 1 = 6 \]

ways to pair $n$ people:

\[
(n-1) + \ldots + 3 + 2 + 1 = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}
\]

\[
E(\text{pairs of common birthday}) = \frac{n^2 - n}{2 \times 365}
\]
HOW MANY COMMON BIRTHDAYS?

\[ E(\text{pairs}) = \frac{n^2 - n}{2 \times 365} \]

Using \( n = 30 \):

\[ E(\text{pairs}) = \frac{n^2 - n}{2 \times 365} = \frac{900 - 30}{2 \times 365} = 1.192 \]
HOW MANY COMMON BIRTHDAYS?

• in general, if we generalize with respect to the days of the year and select \( n \) items out of \( N \), number of repeats expected

\[
\frac{n^2 - n}{2N}
\]

• we expect first repeat in

\[
n = \Theta(\sqrt{N}) \text{ trials}
\]

• we can find a collision by only examining roughly the square root of the number of possible outputs

  • for output of 256 bits,
  • randomly choose just \( 2^{130} + 1 \) inputs, (square root of \( 2^{256} \))
  • it turns out there’s a 99.8% chance that at least two of them are going to collide.
**CRYPTOGRAPHIC HASH: COLLISION RESISTANCE**

- The problem with previous methods is that they both take a very, very long time to do.

- For a hash function with a 256-bit output
  - Compute the hash function $256 + 1$ times in the worst case
  - About 128 times on average.

- If a computer calculates 10,000 hashes per second,
  - More than $10^{27}$ years to calculate $2^{128}$ hashes

- If every computer ever made by humanity was computing since the beginning of the entire universe, up to now, the probability that they would have found a collision is still infinitesimally small. [narayanan2016bitcoin]
SECURITY OF HASH FUNCTIONS

• If no design flaws exist, the security of a hash function depends on the bit length of the output hash value.

• Given a m-bit hash function, the attacker needs $2^{m/2}$ brute force computation to find a collision.
  • MD5 is $128/2 = 64$ bits security
  • SHA-1 is $160/2 = 80$ bits security
  • SHA-256 is $256/2 = 128$ bits security
  • SHA-512 is $512/2 = 256$ bits security

• At least 80 bits is required, to assure security

• Bitcoin’s blockchain uses SHA-256 (Secure Hash Algorithm).
## REAL LIFE HASH FUNCTIONS

<table>
<thead>
<tr>
<th>Name</th>
<th>Output Length (bits)</th>
<th>Security status</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>128</td>
<td>Collisions found</td>
</tr>
<tr>
<td>SHA1</td>
<td>160</td>
<td>Can be broken in $\sim 2^{61}$ iterations</td>
</tr>
<tr>
<td>SHA2</td>
<td>224-512</td>
<td>No known attacks</td>
</tr>
<tr>
<td>$\rightarrow$ SHA-256</td>
<td>$\rightarrow$ 256</td>
<td>No known attacks</td>
</tr>
<tr>
<td>SHA3</td>
<td>224-512</td>
<td>No known attacks</td>
</tr>
</tbody>
</table>

Bitcoin typically uses SHA-256(SHA-256(transaction))

Play with hash functions

http://www.xorbin.com/tools/sha256-hash-calculator
The Merkle-Damgård transform can be used to convert a fixed-length hash function to a hash function taking inputs of arbitrary length, while preserving collision resistance.

- we can focus our attention on designing collision resistant compression functions operating on short, fixed-length inputs, and automatically convert such compression functions into full-fledged hash functions.

- Adopted by most used hash functions.
Merkle Damgard Transform
CRYPTOGRAPHIC HASH APPLICATIONS

- Digital signatures
- Bitcoin transaction ID
- Deduplication
- Password storage
CRYPTOGRAPHIC HASH APPLICATIONS

- Generate data fingerprinting
- Digest: if we know $H(x) = H(y)$
  - then it’s safe to assume that $x = y$.
  - useful because the hash is small: do not compare entire files
- e-Mule, for instance, exploited MD-5 to verify that two files are the same, even if they are described by different keywords
CRYPTOGRAPHIC HASH APPLICATIONS

- File or message integrity

- Use the hash value as the checksum to check if the data is changed or modified.

- to recognize if a content C is the same of a content C1 that we saw before,
  - just remember the hash of C1, hash is a proxy of C1!
  - compute the hash of C and compare with that of C1
  - if the two hashes are equal, the content has not be tampered
CRYPTOGRAPHIC HASH APPLICATIONS

- A distributed hash table (DHT) is a class of a decentralized distributed system that provides a lookup service similar to a hash table: (key, value).
- Pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key.
CRYPTOGRAPHIC HASH APPLICATIONS

- Bitcoin use block chain (hash chain) to store transaction ledger in a P2P (Peer-to-Peer) network
- tamper freeness property

![Diagram showing hash chain with blocks B1, B2, and B3 connected by hash operations.]
Hash Search Puzzles

- Based on partial pre-image attack

The hash/search puzzle consists of:
- a cryptographic hash function, $H$
- a random value, $r$
- a target set, $S$
- a solution of the puzzle is a value $x$, such that:

$$m = r|x$$

$$H(m) \in S$$

- Bitcoin Proof of Work (PoW) is based on a hash/search puzzle
512 input bits  

$S$  

256 output bits
512 input bits

\[ m \]

\[ H \]

256 output bits

\[ S \]

\[ H(m) \in S \]

- \( m \) is a valid puzzle solution
• m is a no valid puzzle solution
The difficulty may be tuned by defining the size of $S$:

- if $S$ is large, the puzzle is less difficult
- in Bitcoin is defined by the number of leading zeros of SHA-256
Puzzle-friendliness property: a hash function $H$ is said to be puzzle-friendly if

- for every possible $n$-bit output value $y$
- if $k$ is chosen from a distribution with high min-entropy,
- then it is infeasible to find $x$ such that $H(k \ || \ x) = y$ in time significantly less than $2^n$.

- Puzzle-friendly property implies that no solving strategy to solve a search puzzle is much better than trying random values of $x$. 

CRYPTHOGRAPHIC FUNCTIONS: RECAP

Hash function:

- arbitrary size input
- fixed-size output
- efficiently computable

cryptographic hash function must have also some security properties:

- hiding:
- collision resistance

collision freedom and hiding can be violated trivially through brute force

- compute the hash of all possible values for pre-digest until you find one that produces the desired digest
- have to be rendered computationally infeasible by making sure that domain \( X \) is very large
SYMMETRIC KEY ENCRYPTION: REVIEW
Send a confidential message protected with a public key
• Digital signatures are the second cryptographic primitive needed as building blocks for the cryptocurrencies
DIGISTAL SIGNATURES: A REALISTIC SCENARIO

Bob sends digitally signed message:

- large message $m$
- $H$: Hash function
- digital signature (encrypt) $K_B^-(H(m))$
- Bob’s private key $K_B^-$
- encrypted msg digest $K_B^-(H(m))$

Alice verifies signature and integrity of digitally signed message:

- encrypted msg digest $K_B^-(H(m))$
- large message $m$
- $H$: Hash function
- digital signature (decrypt) $K_B^+(H(m))$
- Bob’s public key $K_B^+$

If the calculated hashcode does not match the result of the decrypted signature, either
- the document was changed after being signed, or
- the signature was not created with the private key of the sender
DIGITAL SIGNATURES

- **data integrity**: authentication of content
- **data origin authentication**: authentication of sender
- **non-repudiation**: signer cannot deny signing message
- does not guarantee data **confidentiality**, the message is sent in clear
- to guarantee confidentiality + data integrity
  - the sender signs the document with its private key and encrypts the document with the public key of the receiver
  - the receiver decrypts the document with its private key and it applies to the resulting document the public key of the sender
  - if the result “makes sense” then return “ok”
API FOR DIGITAL SIGNATURES

\[(sk, pk) := \text{generateKeys}(\text{keysize})\]

\[sk: \text{secret signing key} \]
\[pk: \text{public verification key} \]

\[\text{sig} := \text{sign}(sk, \text{message}) \] /*cipher the message through the secret key and obtain the signature.*/

\[\text{isValid} := \text{verify}(pk, \text{message}, \text{sig}) \] /*decipher the signature through the public key and compare the result with the message*/

and the following property must hold:

\[\text{verify}(pk, \text{message}, \text{sign}(sk, \text{message})) = \text{true} \]
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import javax.crypto.Cipher;

public class SignatureTest {

    public static void main(String[] args) throws Exception {
        // generate public and private keys
        KeyPair keyPair = buildKeyPair();
        PublicKey pubKey = keyPair.getPublic();
        PrivateKey privateKey = keyPair.getPrivate();

        // encrypt the message
        byte[] encrypted = encrypt(privateKey, "This is a secret message");
        System.out.println(new String(encrypted)); // <<encrypted message>>

        // decrypt the message
        byte[] secret = decrypt(pubKey, encrypted);
        System.out.println(new String(secret)); // This is a secret message
    }
}
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import javax.crypto.Cipher;
public class SignatureTest {
    public static void main(String [] args) throws Exception {
        // generate public and private keys
        KeyPair keyPair = buildKeyPair();
        PublicKey pubKey = keyPair.getPublic();
        PrivateKey privateKey = keyPair.getPrivate();
        // encrypt the message
        byte [] encrypted = encrypt(privateKey, "This is a secret message");
        System.out.println(new String(encrypted));  // <<encrypted message>>
        // decrypt the message
        byte[] secret = decrypt(pubKey, encrypted);
        System.out.println(new String(secret));  // This is a secret message
PLAYING WITH JAVA AND SIGNATURES

public static KeyPair buildKeyPair() throws NoSuchAlgorithmException {
    final int keySize = 2048;
    KeyPairGenerator keyPairGenerator =
            KeyPairGenerator.getInstance("RSA");
    keyPairGenerator.initialize(keySize);
    return keyPairGenerator.generateKeyPair();
}

public static byte[] encrypt(PrivateKey privateKey, String message) throws Exception {
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.ENCRYPT_MODE, privateKey);
    return cipher.doFinal(message.getBytes());
}

public static byte[] decrypt(PublicKey publicKey, byte[] encrypted) throws Exception {
    Cipher cipher = Cipher.getInstance("RSA");
    cipher.init(Cipher.DECRYPT_MODE, publicKey);
    return cipher.doFinal(encrypted);
}
This is a secret message
• Encryption is two way, and requires a key to encrypt/decrypt

• Hashing is one-way. There is no 'de-hashing'