Lesson 11:
CRYPTOGRAPHIC TOOLBOX FOR BLOCK CHAINS
10/4/2019
OUTLINE OF THE NEXT LESSONS

• Introduction of tools for the development of blockchains and distributed ledgers
  • cryptographic tools
    • cryptographic hash functions
    • digital signatures
  • data structures
    • Bloom filters
    • Merkle trees
    • Patricia tries
    • Patricia Merkle tries
CRYPTOGRAPHIC HASH FUNCTIONS

\[ \mathcal{H} : \{0, 1\}^* \rightarrow \{0, 1\}^\lambda \]

- SHA256
- RIPEMD160

Arbitrarily long data \rightarrow Fixed sized hash/digest

- referred also one-way transformations
- properties:
  - take any byte sequence as input
  - fixed size output
  - efficiently computable
  - some security properties (to be seen in the next slides)
- family of hash functions share similar design with different parameters and output length
- related terms: hash, message digest, fingerprint, checksum
CRYPTOGRAPHIC HASH FUNCTIONS

- input:
  - input length is counted in bits
  - normally a maximum input length, zero input length is permitted,

- output:
  - fixed length output: normally 128/160/256/512-bit
CRYPTOGRAPHIC HASH FUNCTIONS

- property:
  - a small change in the input produces a completely different output
CRYPTOGRAPHIC HASH FUNCTIONS

• classical security properties:
  • Pre-image resistance
  • Second pre-image resistance
  • Collision-resistance

• Also required (useful for cryptocurrencies and blockchains)
  • Hiding
  • Puzzle-friendliness
HASH FUNCTIONS: SECURITY PROPERTIES

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

- **preimage resistance:** for any $y \in Y$, it is hard to find $x \in X$ such that $h(x) = y$
- one-way function
- may be also replaced by the hiding property (to be seen later)
• **Pigeonhole Principle:** 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 than the domain you will have collisions
CRYPTOGRAPHIC HASH: COLLISION RESISTANCE

- nobody can find x and y such that \( x \neq y \) and \( H(x) = H(y) \)
- collisions exist, but it is very hard to find them
HASH FUNCTIONS: SECURITY PROPERTIES

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

- **second preimage resistance**: given $M$ and thus $h = H(M)$, it is hard to find another value $M'$ that $H(M') = h$

  ![Diagram](Diagram of hash function)

  - also called “weak collision resistance”
  - may require exhaustive search looking for $M'$
SECOND PREIMAGE RESISTANCE: AN EXAMPLE

- a function which is not preimage resistant: 8-bit block parity

\[
m=1101000101000100011110010100010100001000101
\]

```
b_1=11010010
b_2=10001001
b_3=11100101
b_4=00010100
b_5=10100010
b_6=00010100
```

digest=00011100 (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

```
\begin{array}{cccc}
m_1 & 11010010 & 10001001 & 11110010 \\
    & 10001001 & 10001001 & 10001001 \\
    & 00010100 & 00010100 & 00010100 \\
    & 00110000 & 00110000 & 00110000 \\
    & 10100010 & 10100010 & 10100010 \\
    & 00110100 & 00110100 & 00110100 \\
\end{array}
```

digest(\( m_2 \)) = 00011100

- hash which is not second preimage resistant
HASH FUNCTIONS: SECURITY PROPERTIES

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

- **collision resistance**: it is hard to find a pair of values \(x_1\) and \(x_2\), \(x_1\) different from \(x_2\), such that \(H(x_1) = H(x_2)\)

  ![Diagram showing collision resistance](image)

- also called strong collision resistance
ATTACKING HASH FUNCTIONS

approaches for attacking an hash function:

• cryptanalysis involves exploiting logical weaknesses in the algorithm

• performing a brute-force attack
  • in cryptography, a brute-force attack, or exhaustive search might be used when it is not possible to take advantage of other weaknesses in the hashing system.
  • it consists of systematically checking all possibilities until the correct one is found.
SECURITY OF HASH FUNCTIONS

• the strength of a hash function against brute-force attacks depends solely on the length of the hash code produced by the algorithm.

• for a hash code of length $n$, the level of effort required is proportional to the following.

<table>
<thead>
<tr>
<th>Property</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preimage resistant</td>
<td>$2^n$</td>
</tr>
<tr>
<td>Second preimage resistant</td>
<td>$2^n$</td>
</tr>
<tr>
<td>Collision resistant</td>
<td>$2^{n/2}$</td>
</tr>
</tbody>
</table>

• 56 bit hash which can be brute-forced using $2^{56}$ operations has a security level of 56 bits.

• $2^{128}$ operations is infeasible by current machine, $2^{80}$ becomes feasible.

• note, $2^{81}$ operations take twice the time of $2^{80}$
FINDING COLLISIONS

- what is the maximum number of guesses required to certainly find a collision by a brute force attack?

512 input bits

H

...  

256 output bits

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

\[
O(2^n) \text{ time complexity} \\
O(1) \text{ space complexity, where } n=\text{len}(H)
\]
FINDING COLLISIONS

- what is the maximum number of guesses required to certainly find a collision by brute force?

512 input bits

- pick random inputs and compute their hash values
  - but it is possible to show that you will find a collision with high probability long before examining $2^{256} + 1$ values
  - why?

256 output bits

$H(m)$
FINDING COLLISIONS

- what is the maximum number of guesses required to certainly find a collision by brute force?

  512 input bits

  \[ H \]

  256 output bits

  \[ H(m) \]

- \( \approx 50\% \) 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 is a high chance that at least two of them are going to collide.

  \( O(2^{n/2}) \) time complexity
  \( O(2^{n/2}) \) space complexity, where \( n = \text{len}(H) \)

- Why just \( 2^{128} \)? See the birthday paradox in the next slides.
THE BIRTHDAY PARADOX

- how many people (k) should be in a room so that the probability that two of them share a birthday becomes larger than 50%?

- 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

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

• how many ways to pair $n$ people:

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

• the probability that each pair of people have birthday in the same day is

  \[
  \frac{1}{365}
  \]

Expected value(pairs of common birthday)
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

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

• to compute the first expect first repeat

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

\( n = \Theta(\sqrt{N}) \) trials

• we can find a collision by only examining roughly the square root of the number of possible outputs
  • for output of 256 bits,
  • try \( 2^{130} + 1 \) randomly chosen 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.
connection with hash functions: given
- a hash function $H$, with $n$ possible outputs
- if $H$ is applied to $k$ random inputs, what must be the value of $k$ so that the probability that at least a pair of input $x, y$ satisfy $H(y) = H(x)$ is 0.5?

for a hash function with a 256-bit output
- about $2^{128}$ times, on average.

this works no matter what $H$ is, but it takes too long to matter
- if a computer calculates 10,000 hashes/sec, it would take $10^{27}$ years to compute $2^{128}$ 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 $2^{61}$ iterations</td>
</tr>
<tr>
<td>SHA2 $\rightarrow$ SHA-256</td>
<td>224-512</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))

Retired:
- SHA1, MD2 (output: 128 bit, Rivest), and MD4 (output: 128 bit, Rivest), vulnerabile
- MD5 (output: 128 bit, Rivest)
  - vulnerabile, but ok for a large set of applications

Current:
- SHA2, SHA3, available for 224,256,384,512 bits fingerprints.
- ripemd160 (output: 160 bit) also used in Bitcoin

Play with hash functions: https://www.pelock.com/products/hash-calculator
HASH FUNCTIONS LIFE CYCLE

1. New function proposed
2. Security evaluated
3. Function standardized
4. Theoretical attacks proposed
5. Attacks improved and are practical

Laura Ricci
Cryptographic toolbox for blockchains
• historically, popular cryptographic hash functions have a useful lifetime of around 10 years
• the last one SHA-3 (Keccak)
hiding is related to pre-image property: hide the input of the function

the degree of hiding offered by the hashing function depends also on the range of values that the input $x$ can take

- consider trying to hide the outcome of a coin toss
- encode of the outcome of the coin toss $HEAD=0$ or $TAIL=1$
- then hash the output: $H_{toss}(outcome) = (1 + outcome) \mod 2$
  - $H_{toss}(HEAD)=1$, $H_{toss}(TAIL)=0$.

if the domain $\{HEAD, TAIL\}$ is known to the attacker

- given a hash value of 1, it might simply try hashing each value in the set
- it may easily find out that the outcome of the toss is $HEAD$

the main problem:

- the input domain is easily enumerable
CRYPTOGRAPHIC HASH: HIDING

- high min entropy: all the values in the distribution are negligibly likely, and no particular value is more likely than others

a solution:
- pick a random integer $R$ for instance of 256 bits, from a distribution with a high min entropy
- append $R$ to the original input
- the input space becomes extremely hard to enumerate ($2^{257}$ possibilities).
- given a hash value, it is extremely hard for an attacker without the knowledge of the random key $R$ to deduce the outcome of the toss.

- Hiding property
  - a hash function $H$ is said to be hiding if when a secret value $R$ is chosen from a probability distribution that has high min-entropy, then, given $H(R || x)$, it is infeasible to find $x$. 
HIDING APPLICATION: COMMITMENT

• want to commit to a value and reveal it later
  • seal a value in an envelope and put that envelope out on the table where everyone can see it.
    • cannot change the value inside it, but it remains secret now
  • open the envelope later and reveal the value

• commitment API
  • com ← commit (value, nonce)
  • match ← verify(com, nonce, value)

• sealing the envelope
  • com ← commit (value, nonce)
  • publish com

• opening the envelope
  • publish (nonce, value); has been com already published
  • anyone can use verify(.... ) to check validity
• want to commit to a value and reveal it later
  • seal a value in an envelope and put that envelope out on the table where everyone can see it.
    • cannot change the value inside it, but it remains secret now
  • open the envelope later and reveal the value

• commitment API
  • \((\text{com}; \text{key}) \leftarrow \text{commit} (\text{value}, \text{nonce})\)
  • \(\text{match} \leftarrow \text{verify}(\text{com}, \text{key}, \text{value})\)

• implementation
  • \(H(\text{msg} \mid \text{nonce}) \leftarrow \text{commit}(\text{msg}, \text{nonce})\)
  • \(H(\text{msg} \mid \text{nonce}) = \text{com} \leftarrow \text{verify}(\text{com}, \text{key}, \text{msg})\)
AN EXAMPLE OF COMMITMENT

• Alice wishes to play paper-scissors-stone over the telephone with Bob

• the game:
  • Alice and Bob both choose simultaneously one of paper, scissors or stone
  • the outcome of the game is determined by the rules
    • paper wraps stone
    • stone blunts scissors
    • scissors cut paper
    • if both Alice and Bob choose the same item then the game is declared a draw.

• problem: when conducted over the telephone we have the problem that whoever goes first is going to lose the game
AN EXAMPLE OF COMMITMENT

• to solve the problem:
  • the party who goes first, ‘commits’ to his/her choice
  • the other party cannot determine what was committed to
  • the other party can then verify that the revealing party has not altered
    its choice between the commitment and the revealing stage

• implementation through hash functions:
  \[ A \rightarrow B : h_{A} = H(RA \mid| \text{paper}) \]
  \[ B \rightarrow A : \text{scissors} \]
  \[ A \rightarrow B : RA, \text{paper} \]

• at the end of the protocol Bob needs to verify that the \( h_{A} \) sent by Alice is
  equal to \( H(RA \mid| \text{paper}) \).
  • if the values agree Bob knows that Alice has not cheated
  • the result is that Alice loses the game since scissors cut paper
AN EXAMPLE OF COMMITMENT

• Bob is not able to determine that Alice has committed to the value “paper”, since:
  • he does not know the random value of RA used
  • he is unable to invert the hash function, by the hiding property

• as soon as Bob sends the value scissors to Alice, she knows she has lost but is unable to cheat
  • she would need to come up with a different value of RA, say R0A, which satisfies
    \[ H(RA || \text{paper}) = H(R0A || \text{stone}). \]
  • but this would mean that Alice could find collisions in the hash function
  • his does not happen if the hash function is second-preimage resistant

• this property of the commitment scheme is called binding
  • Alice cannot change her mind after the commitment procedure,
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:

\[ m = r || x \]
\[ H(m) \in S \]

- based on partial pre-image attack: you have to find a part of the input, such that the output belongs to a set (not a single value like in the pre-image attack)

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

256 output bits

$S$
• $m$ 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 $H(k \parallel x) = y$ 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 exhaustively all the values of $x$. 
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.
MERCKLE DAMGARD TRANSFORM

Diagram of the Merkle-Damgard transform process, showing how blocks of data are processed through a compression function to produce a final hash value.
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
- 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 uses a block chain (hash chain) to store the transaction ledger in a P2P (Peer-to-Peer) network.
- Tamper freeness property.
CRYPTOGRAPHIC FUNCTIONS: RECAP

Hash function:
- arbitrary size input
- fixed-size output
- efficiently computable

cryptographic hash function must have also some security properties:
- hiding:
- collision resistance
- puzzle friendliness

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
• what are the assumptions when you verify them?
• when you produce them?
Digital signatures are the second cryptographic primitive needed as building blocks for blockchains.

Three algorithms (KeyGen, Sign, Verify)
- **KeyGen**: takes as input the security parameter. Returns the signing-key and verification-key.
- **Sign**: takes as input the signing-key and the message to be signed and returns a signature.
- **Verify**: takes as input the verification-key, a message and a signature on the message and returns either True or False.

Major challenge:
- What prevents the adversary from learning how to sign messages by analysing the verification-key?
API FOR DIGITAL SIGNATURES

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

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

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

\[\text{isValid} := \text{verify}(pk, \text{message}, \text{sig}) /*decipher the signature through the public key and compare the result with the message and the following property must hold: \]
\[\text{verify}(pk, \text{message}, \text{sign}(sk, \text{message})) == \text{true}\]
Send a confidential message protected with a public key

\[ D_{S_B}[E_{P_B}(m)] = m \]
DIGITAL SIGNATURES: INTEGRITY

Alice signs a message $m$ with her private key $s_A$, creating a signature $s$. Bob verifies the signature $s$ using Alice's public key $P_A$. The message $m$ is then compared to the signature $s$ to ensure integrity.
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”
DIGITAL SIGNATURES CONSTRUCTION

- Based on the RSA (Rivest Shamir Adleman), one way trapdoor function (with hardness that relates to the factoring problem).

- The RSA algorithm
  - Based on the discrete-logarithm problem.
  - the DSA algorithm

- Bitcoin.
  - uses ECDSA, a DSA variant over elliptic curve groups.
  - typical Bitcoin transaction
    - input: contains a signature and public-key
    - output: contains the code (smart contract) for the verification procedure
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
    }
}
public static KeyPair buildKeyPair() throws NoSuchAlgorithmException {
    final int keySize = 2048;
    KeyPairGenerator keyPairGenerator =
        KeyPairGenerator.getInstance("RSA");
    keyPairGenerator.initialize(keySize);
    return keyPairGenerator.genKeyPair();
}

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
CONCLUSION: HASH VS ENCRYPTION

- Encryption is two way, and requires a key to encrypt/decrypt

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