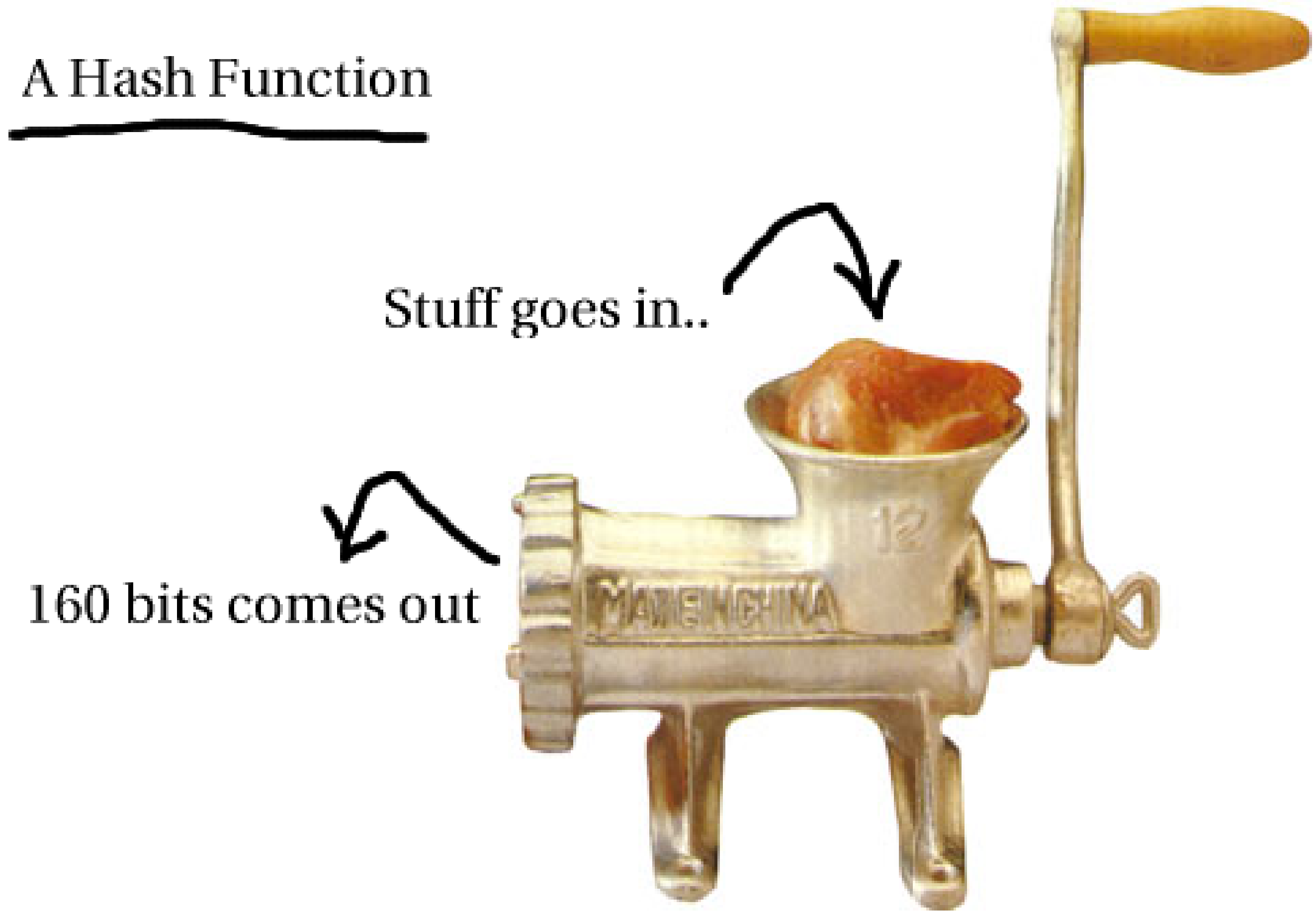
T-79.159 Cryptography and Data Security

Lecture 4: Hashes and Message Digests

Markku-Juhani O. Saarinen
Helsinki University of Technology
mjos@tcs.hut.fi

Cryptographic hash functions

- Maps a message M (a bit string of arbitrary length) as a "message digest" $X = H(M)$ of constant length, e.g. 128, 160, or 256 bits.
- Well-known examples: MD5, SHA-1, RIPEMD-160, SHA-256.
- Security requirement 1:
  One-wayness. Given a message X, it is should be "hard" to find a message M satisfying $X = H(M)$.
- Security requirement 2:
  Collision resistance. It should be "hard" to find two messages $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$.


A Hash Function
Stuff goes in..
160 bits comes out

UNIX Password authentication

1. User enters a password (key):
   `Login:  falken`
   `Password:  ******`

2. System looks up user in `/etc/passwd` file and finds the corresponding hashed key value and other relevant data:
   `falken:`**`cV/h5TT95.pzQ`**`:1085:1085:Prof.  Falken`

3. First 2 chars, cV, is the *salt*. Now the system compares the output of the crypt system call to the encrypted string:
   `char *crypt(const char *key, const char *salt);`

UNIX Password authentication (2)

- No need to store the key itself, just $H(salt \mid\mid key)$

- The password file `/etc/passwd` can be world-readable! (And often is, although this makes systems more vulnerable to dictionary attacks.)

- Salt slows down dictionary attacks. To check whether some user (from a large group) has a given password, the word has to be hashed with each one of the salts.

- UNIX `crypt(3)` is one-way, but not really collision resistant. Based on DES. Developed by Robert Morris (Sr.) ca. 1975 – still in use today.

SHA-1 and MD5 Fingerprints

- How do you know that your system files have not been tampered with (by viruses or trojans installed by intruders) ?

- One way is to maintain a database of file *fingerprints* and compare them to known good values (e.g. www.knowngoods.org).

- Length checking is not sufficient; simple “checksums” won’t be secure enough. One-wayness clearly a requirement.

- Example: Computing a 128-bit MD5 digest of Linux kernel:

```
$ md5sum /boot/vmlinuz
95fb55766efa90bfe10c25cd2e9daaa4 /boot/vmlinuz
```

Collision resistance

- What if the software distributor tries to cheat ? Could he create a “good” file and a “bad” file (say, with a back-door), such that they have the same digest ?

- This is different from one-wayness, since the distributor can create both files (good and bad ones) simultaneously.

- If a n-bit hash is one-way, it takes 2^n effort to find a message M satisfying $H(M) = X$, given just X.

- If a n-bit hash is collision-resistant, it takes no more than $\sqrt{2^n} = 2^{n/2}$ to find two messages $M_1 \neq M_2$ such that $H(M_1) = H(M_2)$. Why ?

Birthday paradox

Question:

“How many persons needs to be in a room before we can expect two of them to have the same birthday?”

Birthday paradox

Question:

"How many persons needs to be in a room before we can expect two of them to have the same birthday?"

Answer:

23.

Why ?

Birthday paradox (2)

n persons make up exactly $\frac{n(n-1)}{2}$ pairs.

Each pair has probability $\frac{364}{365}$ of <u>not</u> having the same birthday. Since these events are very close to being unrelated, the total probability of no-one having the same birthday is roughly $(\frac{364}{365})^{\frac{n(n-1)}{2}}$.

Substituting $n = 23$ we get $(\frac{364}{365})^{253} \approx 0.499523$.

(So this is not a "paradox" at all.)

Birthday paradox (3)

More generally: We wish to find n (“number of persons”) as a function of m (“number of days in year”), so that probability of a match is $\frac{1}{2}$:

$(1 - \frac{1}{m})^{\frac{n(n-1)}{2}} = \frac{1}{2}$, taking logs:

$\frac{n(n-1)}{2} \ln(1 - \frac{1}{m}) = -\ln 2.$

When $x > 2$, there is a bound $-\frac{1}{x} - \frac{1}{x^2} < \ln(1 - \frac{1}{x}) < -\frac{1}{x}$.

We get an approximation $0.7213 * (n^2 - n) \approx m$.

Asymptotically $n = O(\sqrt{m})$.

How to find collisions

The obvious (but very memory-intensive and hence inefficient) algorithm:

- Initialize a table that can hold $\sqrt{n}$ pairs of x values. The table is indexed by first $\frac{1}{2}$ lg $\sqrt{n}$ bits of $H(x)$.

- For $x = 1, 2, 3, \cdots$: Compute $H(x)$ and check if the table at position indexed by $H(x)$ already has a entry. If an entry exists (say y), verify collision $H(x) = H(y)$ and quit. Otherwise just store x in the table position.

This will take about $O(\sqrt{n})$ time and $O(\sqrt{n})$ memory, e.g. if $n = 2^{128}$, roughly 2^{64} iterations and memory slots. The memory factor is the preventive one even if we manage to run the 2^{64} steps.

Floyd's cycle finding algorithm (1)

Consider a sequence where we start from some x_0 and iteratively compute a sequence $x_1, x_2, \cdots$ as the hash of the previous value:

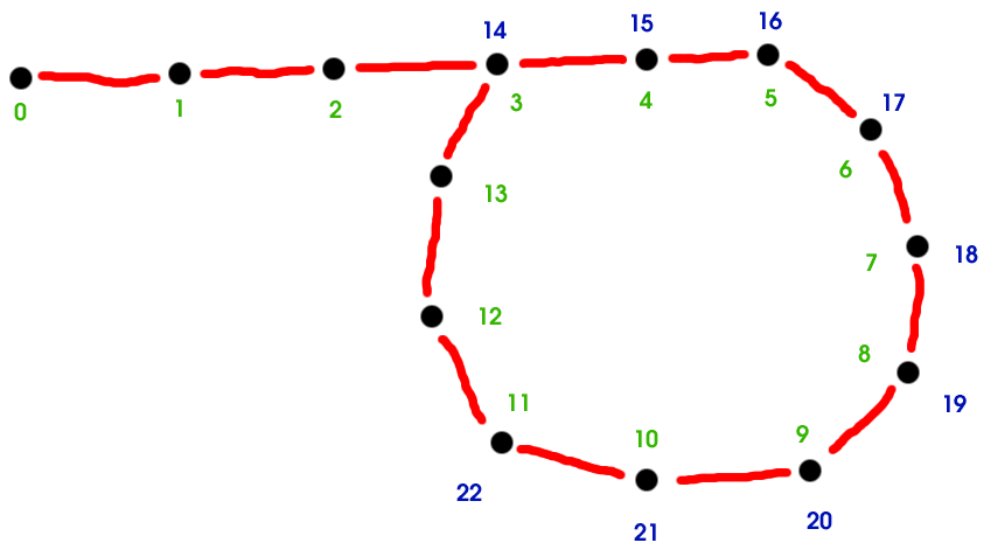
$$x_{i+1} = H(x_i)$$

We have seen that after about $\sqrt{n}$ steps, a collision will probably occur: there will be a pair x_α and x_β so that $x_\alpha = x_\beta$ but $x_{\alpha-1} \neq x_{\beta-1}$.

α is called the *tail* of the cycle.

$\delta = \beta - \alpha$ is the *cycle length*.

Floyd's cycle finding algorithm (2)



Here a collision occurs at $x_3 = x_{14}$.

Hence "tail" $\alpha = 3$, $\beta = 14$ and cycle length $\beta - \alpha = \delta = 11$.

Floyd's cycle finding algorithm (2)

- Clearly $x_i = x_{i+\delta}$ when $i \geq \alpha$.

- Hence $x_i = x_{2i}$ when $2i = i + \delta$; $i = \delta$ (the cycle length).

Thus we can find the cycle length by starting with (x_0, x_0) and compute (x_1, x_2), (x_2, x_4), (x_3, x_6), $\cdots$, (x_i, x_{2i}). (i.e. stop when $x_i = x_{2i}$).

Three hash function invocations needed in each step. Then i will have the cycle length δ.

Finding the collision

From previous step, we have x_δ. Now we compute the sequence

$$(x_0, x_\delta), (x_1, x_{\delta+1}), (x_2, x_{\delta+2}), \cdots, (x_\alpha, x_{\delta+\alpha})$$

.. i.e. stop when $H(x_i) = H(x_{\delta+i})$. Two hash function invocations are needed in each step. At the end $i = \alpha - 1$, and hence we have the collision since $x_i \neq x_{\delta+i}$.

This simple algorithm requires $3\delta + 2\alpha$ invocations of the hash function, and therefore it is asymptotically optimal. However, the memory requirement is very small!

Collision finding, pseudocode:

1. Initialize: $a \leftarrow 0, b \leftarrow 0$.
2. Do: $a \leftarrow H(a)$, $b \leftarrow H(H(b))$ Until $a = b$.
3. Set: $b \leftarrow 0$.
4. Do: Store $(x, y) \leftarrow (a, b)$. $a \leftarrow H(a)$, $b \leftarrow H(b)$ until $a = b$.

When the algorithm terminates: $H(x) = H(y)$, but $x \neq y$, a collision !

Rules of thumb

- As implicated by the birthday paradox, there are algorithms that find a collision (birthday match) with $O(\sqrt{m})$ effort. Neglible memory is required by the algorithms.

- Hence to have collision resistance with n-bit security, the hash should be at least $2n$ bits long; e.g. 128-bit hashes give 64-bit security.

- If only one-wayness is required, then n bits is sufficient for n-bit security.

- Beware that some hash functions (like MD4) have been broken; they do not have the security level implicated by hash size.

How do hash functions actually work?

- Additional design requirement besides one-wayness and collision resistance: it should be possible to hash long messages without storing the whole thing in memory (e.g. signing a backup tape).

- Long message is cut into pieces M_i of equal size and a state variable X_i is maintained.

- The last piece M_n is padded with the length of message and the final value of the state variable X_n is the hash.

- Many other approaches have been proposed, but almost all practical hash functions work like this.

Davies-Meyer (1985)

- Use a block cipher $E(K, P)$. Start with some initial value X_0 and update as $X_{i+1} = E(M_i, X_i) \oplus X_i$. Final value X_n is the hash.

- Provably secure (if the block cipher is secure).

- Since each piece M_i is used to key the block cipher, hashing speed is directly proportional to key size (rather than block size). Resulting hash size is equal to block size.

- Most block ciphers are optimized for fast encryption rather than fast key initialization; hence dedicated hash functions. $E(M_i, X_i) \oplus X_i$ is called "compression function" in the context of these dedicated hash functions.

Message Digest 5 (MD5)

- Very widely used hash function (message digest). Fingerprints, PGP 2.x, PKI x509, etc.

- Designed by Ron Rivest (MIT), 1992. Specified in RFC 1321. MD5 means that this is Rivest's fifth message digest design.

- Produces a 128-bit hash; has no more than 64-bit security. Processes messages in 512-bit blocks.

- Hans Dobbertin (BSI) found a flaw in the compression function of MD5 in 1996; hence its security proofs do not hold. However, collisions have not been computed yet. Do not use in new products.

Secure Hash Algorithm - 1 (SHA-1)

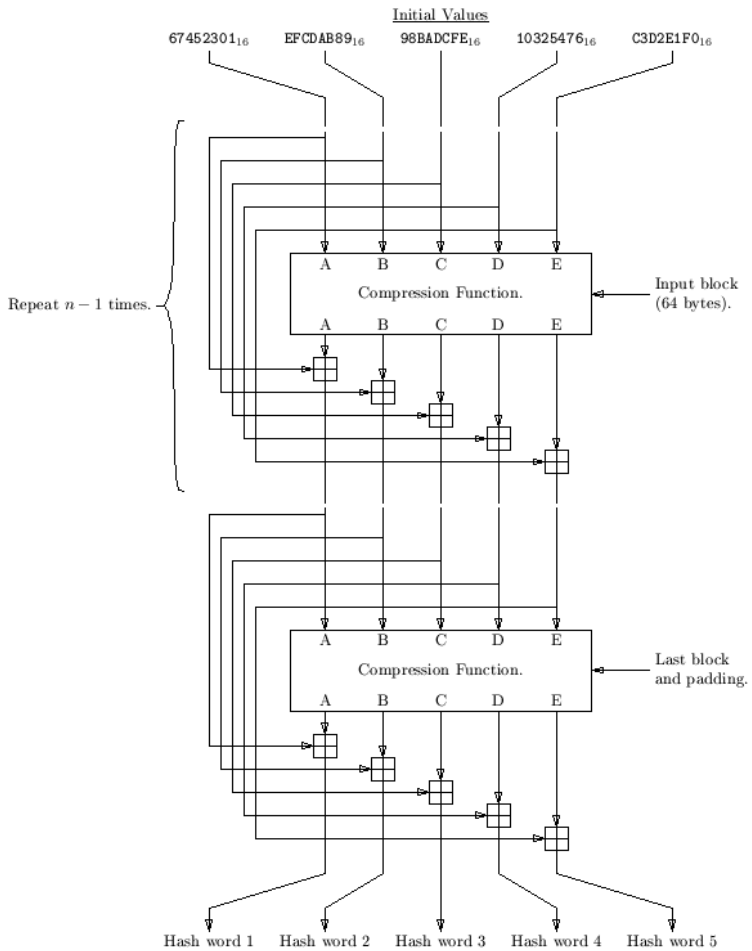
- U.S. / NIST federal standard 180-1/2. Currently the most popular cryptographic hash algorithm.

- Produces a 160-bit hash; 80-bit security. Processes messages in 512-bit blocks. Similar in design to MD4 and MD5.

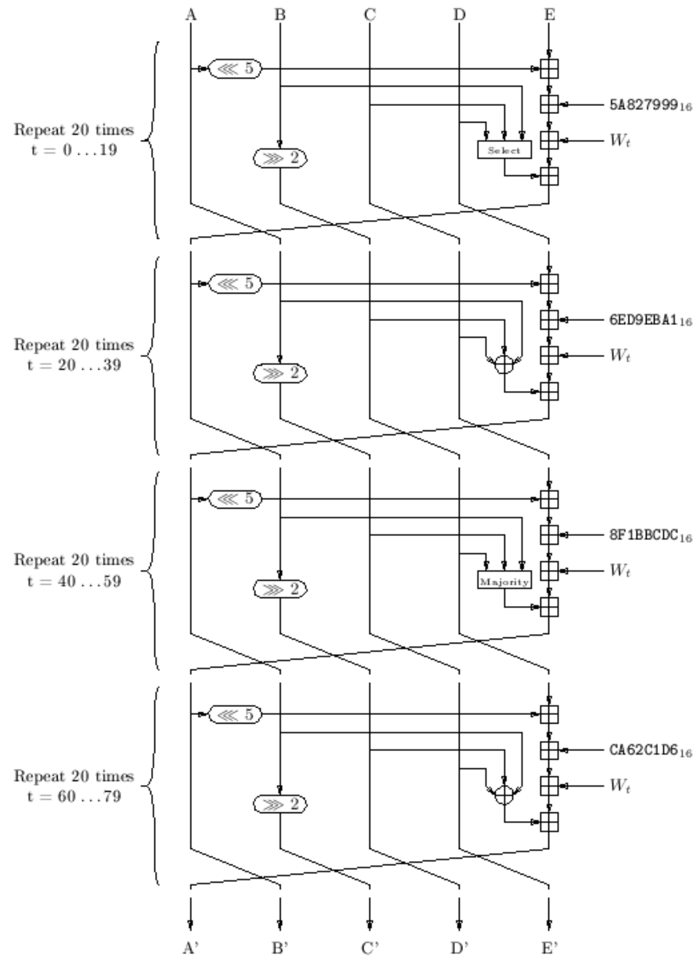
- Designed by unknown persons at NSA in 1993 (original design is known as SHA-0). Slightly modified for (then) unspecified reasons in 1995. New version known as SHA-1.

- Chabaud and Joux (CASSI/SCY/EC) published in 1998 an attack against SHA-0 (collisions with 2^{61} effort rather than 2^{80}) that showed that SHA-1 was indeed more secure than SHA-0.

SHA - 1 (2)



Initial Values
67452301₁₆
EFCDAB89₁₆
98BADCFE₁₆
10325476₁₆
C3D2E1F0₁₆
Repeat $n-1$ times.
A B C D E
Compression Function.
Input block
(64 bytes).
Last block
and padding.
Hash word 1
Hash word 2
Hash word 3
Hash word 4
Hash word 5



A B C D E
Repeat 20 times
$t = 0 \ldots 19$
$\lll 5$
$\ggg 2$
5A827999₁₆
W_t
Select
Repeat 20 times
$t = 20 \ldots 39$
6ED9EBA1₁₆
Repeat 20 times
$t = 40 \ldots 59$
8F1BBCDC₁₆
Majority
Repeat 20 times
$t = 60 \ldots 79$
CA62C1D6₁₆
A' B' C' D' E'

Other dedicated hash algorithms

- RIPE-MD 160 is a robust European hash function. 160-bit hash.
- In 2000, NSA proposed new hash functions that produce 256- and 512 bit hashes. Known as SHA-256 and SHA-512.
- Some speed measurements on a 1.4 GHz AMD Athlon Linux:

```
MD2        5 010 kB/s   MD4       274 556 kB/s
MD5      238 392 kB/s   SHA-1     127 283 kB/s
                  RIPE MD-160      84 896 kB/s
```

Message Authentication Codes (MACs)

- Protects against unauthorized or accidental message manipulation.
- Uses a secret key K to make sure that a message is actually from its assumed sender. MAC is appended to the message. Recipient computes the MAC again from the message and K and verifies it.
- It seems natural to use dedicated hash functions for computation of MACs (fast!), especially if encryption isn't needed.
- Many MACs have been proposed, the most common being HMAC ("hash MAC"), Krawczyk et al (IBM), 1997.

A Stupid MAC

Question:

"Hey! Why not just append the message after the key, hash the whole thing and use that as a MAC ?" (i.e. MAC = $H(K \mid M)$)

A Stupid MAC

Question:

"Hey! Why not just append the message after the key, hash the whole thing and use that as a MAC ?" (i.e. $A = H(K \mid M)$)

Answer:

Eve sees the message M and the MAC A. Because of the way the Davies-Meyer mode works, she has the state of the hash function $X_n = A$ at the end of the current message M. Now she can just add anything after that and compute more iterations $X_{n+1}, X_{n+2}, \cdots$ with the compression function, and finally do a new padding.

MAC must detect changes in the message length as well!

HMAC

- Defined in RFC 2104. Can be used used with many dedicated hash functions: HMAC-MD5, HMAC-SHA1, HMAC-RIPEMD.
- The output can be truncated by simply taking the first n bits of output (e.g. HMAC-SHA1-96 is used in the IPSEC protocol).
- Uses two constants, `ipad` (64 0x36 bytes) and `opad` (64 0x5c bytes).
- Defined as $H(K \oplus \texttt{opad} \mid H(K \oplus \texttt{ipad} \mid M))$
- Only slightly slower than computation of $H(M)$ for long messages.

Key generators

- Where do all of the cryptographic keys come from ?

- Example: AES Needs a 128-bit (16 byte) key, but 16 letters of English contains less than 32 bits of entropy: Directly using a human-understandable key is not a good idea.

- Solution: hash the key first. This way the input key can be of any length! Such long keys are often called passphrases.

- If protocols need random, unpredictable values (nonces), use proper random number generators. These are often based on hash functions.

Pseudorandom Number Generators (PRNGs)

Cautionary tale of the Netscape PRNG in 1995.

- Netscape Navigator 1.1 had the first version of the now-popular SSL protocol. Keys for encryption were generated using a PRNG.
- The PRNG was initialized from `time()` on program startup and the consequent outputs were deterministically based on this seed.
- Guess the 32-bit time value (which is not a secret; everyone has a clock) and you can predict all future outputs of the PRNG!
- Since the eavesdropper knows the outputs of the PRNG, she knows the keys and she can eavesdrop, regardless of encryption strength.

PRNGs (2)

- Most OS's nowadays have built-in cryptographic random number generators for key generation. On UNIX systems:

```
~> hexdump /dev/random
0000000 d938 cb3d e578 7525 292d 68e3 0bd6 16c4
0000010 9cbb d6dc c662 9e5b c326 501b [...]
```

- The randomness is contained in a *random state* (or pool) and it is constantly stirred by events that the operating system gathers: mouse and keyboard inputs, interrupt timings, network events etc. Cryptographic hash functions are used to mix the pool (SHA-1 on Linux).

A Simple PRNG Based on a Hash Function

Stir new input data to state:

$$\text{State} = H(\text{State} \mid \text{counter++} \mid \text{new input data})$$

Extract randomness:

$$\text{Output} = H(\text{State} \mid \text{counter++})$$

.. of course it is good to remember ..

"Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." – John von Neumann (1951)

.. and to use RNGs if available!

Digital Signatures

When signing a message using a public key digital signature algorithm, it is not necessary to sign the message *itself*. It is sufficient to sign a cryptographic hash (message digest) of the message.

Signing:

Signature = Sign(SHA-1(Message), Private Key)

Verifying:

Verify(SHA-1(Message), Signature, Public Key) = OK/FAIL

Note; signature algorithm doesn't even need the message; only its hash is sufficient. More on this in the next lecture..