Cryptography

Lecture 11

- One-way and trapdoor functions.
- Hash function constructions.
- Weaknesses in hash functions.
- Message Authentication Codes (MACs).
- Elgamal and DSA signature schemes.

One-way functions

A one-way function $f : X \to Y$ has the properties that

- it is computationally "easy" to compute $f(x)$ for any $x \in X$.
- it is computationally "difficult" to invert f, i.e. given $y \in Y$, to find a preimage, i.e. a $z \in X$ such that $f(z) = y$.

This concept has many applications in cryptography.

The above is vague and needs to be more precisely defined. Better, but still not completely precise (we restrict ourselves to $X = Y = \{0,1\}^*$):

- There is a probabilistic, polynomial-time algorithm that computes $f(x)$ for given x.
- For any probabilistic, polynomial-time algorithm and randomly chosen x, the probability that it finds a preimage to $y = f(x)$ is negligible.

Example: Password authentication

Password authentication may use a one-way function f (typically, a cryptographic hash function) as follows:

1. The user chooses a password pw ; only $f(pw)$ is stored in the password file.
2. Later, when the user tries to log in using password pw', the value $f(pw')$ is compared with the stored value.

Advantage: Password file needs less protection; not disastrous if an adversary can read it.

One-way property assures that password pw cannot be computed from $f(pw)$.

. . . but passwords are often chosen poorly, from a too small set (length 6-8 characters, often words with some character(s) changed in a predictable way). This makes it feasible to build inverted dictionaries of plausible passwords.

Trapdoor functions

A trapdoor function is a one-way function f with the further property that if you know some secret extra information, inverting f becomes "easy".

Motivation: For encryption, we use a trapdoor one-way function for which only the receiver knows the secret (the trapdoor).

For the adversary, who does not know the trapdoor, decryption is infeasible.

Refined idea: We need not only one trapdoor function $E : \mathcal{M} \to \mathcal{C}$ but a whole family of such functions, indexed by (public) keys.

Example: RSA

RSA trapdoor function

Let $N = p \cdot q$ and let (e, d) be a RSA key pair (e is public key, d is private key).

Then $Enc(pk, m) = m^e \bmod N$ is (believed to be) a trapdoor function.

Without knowledge of the trapdoor d, decryption is infeasible.

RSA assumption

There is (yet) no proof that $Enc(pk, \cdot)$ is in fact a trapdoor function.

No polynomial-time method to invert this functionis known, but it is not excluded that one will be discovered one day.

The (widely believed) RSA assumption states that $Enc(pk, \cdot)$ is a trapdoor function.

Do one-way functions exist?

A probably difficult question

If we can prove for some function that it is one-way, it follows that $\mathcal{P} \neq \mathcal{NP}$.

Two other candidates

Integer multiplication. $f(p, q) = p \cdot q$. f can be computed in polynomial time (in bit length of the factors), but no polynomial factoring method is known. (If one is found, this also proves the RSA assumption false).

Modular exponentiation. $f(x) = g^x \bmod p$, for some prime p and some generator g for $\mathbb{Z}_p^*$, can be computed in polynomial time by repeated squaring. Inverting f is the discrete log problem, for which no polynomial time algorithm is known.

A collision attack

Bob has two contracts, one that offers Alice a very good deal and one that will ruin her.

He prepares K versions of each by making unimportant changes, hashes all documents and finds a collision between two, one of each type.

Bob has Alice sign the favourable contract and then replaces it with the other, for which he now has a valid signature by Alice.

How large must K be for Bob to have a reasonable chance of success in finding such a collision?

Square-root again!

If the K variants are treated as random messages, a similar analysis as for the birthday attack can be done to show that $K = \sqrt{2^n} = 2^{n/2}$ gives high probability of success, where n is the number of bits in hash values.

Example

For $n = 64$ (too short for good level of security!!), Bob needs $2^{32} \approx 4 \cdot 10^9$ versions of each document.

From a computational point of view, this should be considered feasible.

But from a practical point of view: How to you produce 2^{32} "innocent" variants of a contract?

A fact and a moral

Fact

Producing 2^{32} variations is easy:

- Find 32 places in the document where a change can be made (adding a space, changing punctuation, exchanging a word by a synonym . . .).
- Then produce all possible combinations of these variations. Practice shows that this works (i.e. gives a collision) for common hash functions.

Moral

When given a document to sign, always make a trivial modification before signing.

If Alice follows this practice, how is Bob's effort affected?

An important difference

To find a document m_2 that has the same hash value as a given document m_1 by brute force can be expected to require 2^n trial documents (attack against one-wayness).

It is important in the described attack that both documents can be modified until they "meet in the middle" (attack against collision-resistance).

Remember

Hash functions with 160 bits results have security level 80 bits against birthday and similar attacks.

Hash functions and one-way functions

Recall that a one-way function is one for which it is infeasible to find x, given $f(x)$.

Is a collision-resistant hash function necessarily one-way?

No. Counterexample: $f(x) = x$ has no collisions, but is certainly not one-way.

But if you add the assumption that the function actually compresses its input, the answer becomes positive.

If not one-way, then collision can be found

Let h be a collision-resistant function such that for (almost) all y, there are "many" x with $h(x) = y$. Then h is one-way.

Proof idea

We prove the contrapositive result, so assume that h is not one-way. Then we can find a collision as follows:

Choose x at random and put $y = h(x)$. Since h is not one-way, we can find an x' such that $y = h(x')$. Finally, since y has many corresponding x-es it is unlikely that $x = x'$ and we have a collision.

This needs to be put on proper probabilistic footing ...

Hash functions in practice

If we cannot implement ideal hash functions, we should look for functions that cannot be distinguished from ideal ones, i.e. functions that "look random".

Recent candidates are MD5 (128 bits), SHA-1 and RIPEMD-160 (160 bits), all now broken or suspect. Secure choice is SHA-256 (256 bits), but the whole field needs reconsideration.

All the above are based on a common, iterative structure.

Recently, a new and quite different construction, SHA-3, has been standardized.

Iterative hash functions

We first define a compression function g that takes one n-bit and one k-bit block as arguments and produces an n-bit result.
A typical value for k would be 512.

We also define a fixed n-bit value IV.

Hash of message $m \in \{0,1\}^*$ is computed as follows:

1. Split m into a sequence of k-bit blocks $m_1, m_2, \ldots m_b$. Add some padding at the end so that all blocks have size k bits.
2. Put $H_0 = IV$ and compute $H_i = g(H_{i-1}, m_i)$ for $i = 1, 2 \ldots b$.
3. The hash value $h(m)$ is H_b.

Advantages

The iterative construction has the following advantages:

- Hash values can be computed on the fly as a long message is read; the whole message need not be available before computation can start or stored for hashing.
- The definition is more structured than a direct definition of hash functions for variable length inputs.
- If g can be proven (or is believed) to be collision-resistant, then h is also (since a collision in h must be due to a collision in g).
  Remark: For this property to hold, append one block containing the length of the message before computing hash value. Why? (Solution in the exercises.)

Disadvantage

The main disadvantage of this construction is that given a message m and its hash value d, anyone can produce a hash for an extension of (the padded) m.

In fact, this immediately prevents such hash functions from "looking random", since one can easily test this property, which is very unlikely to hold for a random function.

No commonly used hash function addresses this problem.

Compression functions

We examplify with SHA-1; other constructions are similar. For details see book and link on course web page.

The compression function takes as input the accumulated hash value (five 32-bit words) and a new message block (16 32-bit words).

It updates the hash value in 80 iterations, where each iteration is a combination of several logical and shift operations using the current hash value and some of the words from the new block.

In total, a very complicated and confusing construction.

SHA-1 is broken

In 2005, Chinese cryptographer Xiaoyun Wang found collision-finding attacks that require only 2^{63} operations, rather than the 2^{80} operations of the birthday attack. Such an attack is feasible for a very well-funded adversary.

Consequently, for many applications one needed to look for stronger hash functions. The SHA-2 family, including SHA-224, SHA-256 and SHA-512 already existed, but they are based on similar ideas as SHA-1.

It seemed that new ideas were needed in the design of hash functions, as well as careful analysis of what properties are required for various applications.

NIST conclusions

Policy on hash functions (March 2006)

Federal agencies should stop using SHA-1 for digital signatures, digital time stamping and other applications that require collision resistance as soon as practical, and must use the SHA-2 family of hash functions for these applications after 2010.

Hash competition

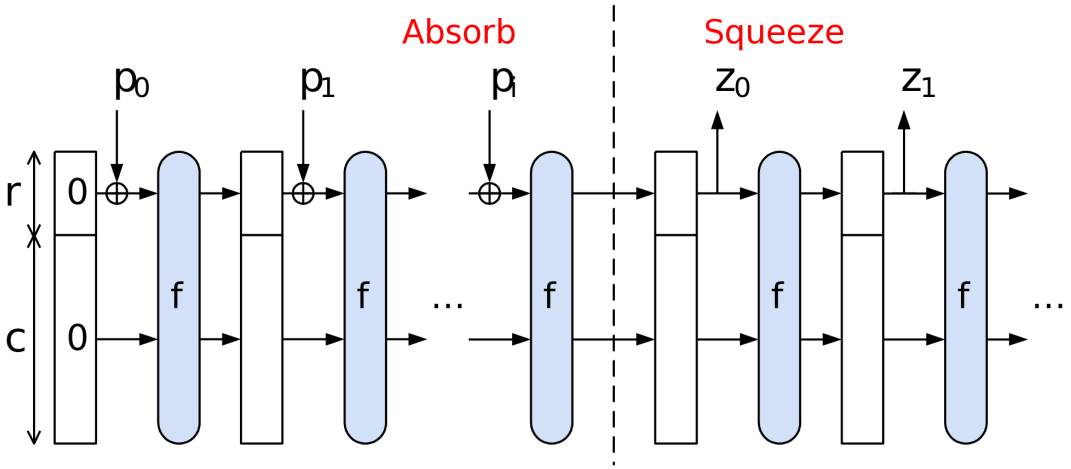
On November 2, 2007, NIST announced a public competition to develop a new cryptographic hash algorithm.
Deadline for proposals was October 31, 2008.
14 candidates selected for 2nd round in 2009, 6 for 3rd round in 2010.
The winner, Keccak, announced on October 2, 2012.
Keccak is now officially SHA-3.

Keccak: basic ideas

The sponge construction

- A general construction to build cryptographic primitives (not only hash functions).
- Two phases: absorb input, then squeeze to produce (arbitrarily long) output.
- Parameterized by a positive width $b = r + c \in \mathbb{N}$ and a permutation f on $\{0,1\}^b$.

The sponge construction



Absorption

- b-block internal state initialized to 0^b (recall $b = r + c$).
- Padded input string split into r-bit blocks $p_0, p_1, \ldots, p_i$
- Input blocks xor-ed with first r bits of state, with permutation f applied between every absorption of a block.

Squeezing

- First r bits of state repeatedly output, with f applied between.
- Arbitrary-size output is $z_0, z_1, \ldots$.

Keccak: the complete definition

Remaining details

- Recommended parameters are $r = 1024$ and $c = 576$ (other choices possible). State is thus 25 64-bit words.
  Same parameters for different output lengths.
- f iterates (24 times) a round function consisting of bitwise logical operations and cyclic shifts.
  This is similar to an (unkeyed) block cipher.

What will happen? Some guesses

- SHA-2 still usable; unlikely (?) to be broken in near future.
  Where already deployed, continued use is likely.
- SHA-3 (Keccak) will be further analyzed and tested and make its way into new systems.
- The sponge construction will be used for other cryptographic purposes.

MD5

A still very widely used hash function.

It is, however, completely broken; collisions can be found in seconds on a notebook PC.

Based on this, one can easily for any two texts T_1 and T_2 produce two files F_1 and F_2, both valid PostScript (for example), such that

- F_1 and F_2 have the same MD5 fingerprint,
- when displayed in a PostScript viewer, F_1 shows T_1 and F_2 shows T_2.
  <Live Demo>

Padding

Typical padding is as follows:

1. add a '1' bit to the message.
2. add the necessary number of '0' bits to make total message length 64 bits less than a multiple of the block size.
3. add a 64 bit representation of the original message length. (Thus the hash function can only hash messages of length $\leq 2^{64}$.)

Example: Swedish e-legitimation

Example services that are presently provided over the web. Many more are expected in the near future.

- access to individual tax account data with the tax authorities (Skatteverket).
- administering parental leave dates with social authorities (Försäkringskassan).
- access to individual pension fund data for forecasting.

These services require identification of individual.

Solution adopted: e-legitimation, a digital signature.

Technology: RSA signatures with 2048 bit key on SHA-256 hash of message.

Protected by ~~12~~ 6 chars password.

Password storage revisited

Using time/memory tradeoff techniques and tera-byte storage, commercial password services can break most passwords that people are able to memorize.
Sophisticated algorithms using rainbow tables and vast computing resources make further security measures necessary.

Improvements

Add salt to the password (some random bytes) before hashing; salt is stored in clear with password hash. Makes dictionary attack more costly.

Increase computing time, e.g. by iterated hashing (1000's of iterations) to make attacks more time-consuming.

Example: MacOS El Capitan

User password record


```
<dict>
<key>SALTED-SHA512-PBKDF2</key>
<dict>
<key>entropy</key>
<data>
d+eFBzz8tEa2bt9/lGM7n8Fl7eysmeVedYFY+KtnjeHDuj2EKzkGkV6BdTZ
d3LbTB/jqoI9VzhD5xXlx5ltFc69mr4TbcvdMl8nK0eUbX2gSaNn2/C4hcp
3HNTq0nrHyF7gw3+xwwkcsfe/9pXolTUz3MXB4i4+8+fVybYyqI=
</data>
<key>iterations</key>
<integer>34843</integer>
<key>salt</key>
<data>
Agij2h+W7f4LRaM59yscvkorhh4IO27Hpkukhefwe7U=
</data>
```

Further uses of hash functions

- Pseudo-random number generation. Standards exist for bit generators, that are though to be cryptographically secure, based on approved hash functions.
- Generating nonces.
- Generating several keys from a master key.
- ...

Message authentication codes (MAC's)

A MAC is similar to a hash function, but it takes two arguments, a message and a secret key.

To achieve authenticated communication, Alice and Bob agree on a MAC algorithm and a secret key K_{AB}.

Then, Alice can send $(M, \mathrm{MAC}(K_{AB}, M))$ to Bob and Bob can recompute the MAC and check that it is correct. If so, Bob believes that sender is Alice and that M is what Alice sent.

A MAC is not an encryption

It is not possible to recover M from $\mathrm{MAC}(K_{AB}, M)$.

MAC's – properties required

A MAC should have the following properties:

- A MAC function takes as input an arbitrary length message and a key and produces a fixed length output (common lengths 128 and 256 bits).
- It must be infeasible for an efficient Adversary to construct a message M and its $\mathrm{MAC}(M, K)$ without knowledge of the key K, even if she has access to a large number of $(M', \mathrm{MAC}(M', K))$ pairs where $M \neq M'$. (This should be made more precise!)

  Note that the Adversary is considered to succeed even if the constructed M is "meaningless".

Encryption and authentication

Encryption does not provide data integrity.

Adding a MAC does not provide confidentiality.

Thus, often the two must be combined, i.e.

$$C = E_{K_e}(M)||\mathrm{MAC}(K_a, E_{K_e}(M)).$$

Note the use of different keys for encryption and authentication!

Which is more important, confidentiality or integrity?

MAC's from hash functions

A good hash function h can be used to define a MAC; most popular is the HMAC construction

$$\mathrm{HMAC}_K(m) = h(K \oplus a \,||\, h(K \oplus b \,||\, m))$$

where a and b are fixed constants of size equal to the block size for h (before xor-ing, K is padded with zeros).

Note that the outer hash computation has a very short input.

The nested application of h takes care of the disadvantage of iterative hash functions.

Is a MAC really necessary?

Reconsider the encrypted, authenticated message:

$$C = E_{K_e}(M)||\mathrm{MAC}(K_a, E_{K_e}(M)).$$

Why not the much simpler

$$C = E_{K_e}(M)$$

where K_e is a key known only by Alice and Bob?

If Bob decrypts and gets a meaningful message M, then the sender must have known K_e and thus be Alice. So, a MAC is not necessary. Or?

An unexpected case

What if M is the following seven-block message (block size is 16 bytes):

`You can safely e`
`nter the lion's `
`cave. The key to`
`  the exit door i`
`s in the last bl`
`ock. Good luck! `
$b_1 b_2 \ldots \quad b_{16}$

DSA signatures

Since RSA private exponents are large, RSA signing is a costly operation (even if only a hash of the message is signed).

A more efficient signature scheme is the DSA algorithm, which is the basis for DSS, the Digital Signature Standard.

Like Elgamal, DSA is based on discrete logarithms.

First we recall Elgamal encryption.

Elgamal encryption

We fix a large prime p for which the discrete logarithm is hard and a generator g of $\mathbb{Z}_p^*$. All computations are in $\mathbb{Z}_p^*$.
p and g are known to everyone.

Alice (the receiver) chooses a random exponent x as secret key and computes $h = g^x$ as public key. So, $(sk, pk) = (x, h)$.

Encryption: The sender Bob wants to send a message $m \in \mathbb{Z}_p^*$ to Alice. He chooses a random exponent r and computes $c_1 = g^r$ and $c_2 = h^r \cdot m$. He sends the pair $(c_1, c_2))$ to Alice.

Decryption: Alice computes $K = c_1^x$, and $m = c_2 \cdot k^{-1}$.

How do you turn this into a signature scheme?

Elgamal in subgroups

Recall that a subgroup of $\mathbb{Z}_p^*$ is a subset which is closed under the group operation.

Recall also that Elgamal encryption can be done by choosing a subgroup and a generator g for that subgroup.

The setting of DSA is a subgroup of $\mathbb{Z}_p^*$ with q elements and generator g.

Here both p and q are primes and q is a divisor of $p - 1$.

Typically, the size of q is 160 bits and that of p is 1024 bits.

DSA: signing

Let q, p and g be as on the previous slide and let H be a hash function that produces 160-bit hash values.

Each user has a private key x and a public key $y = g^x \bmod p$.

To sign message m Alice does the following:

1. Compute the hash value $z = H(m)$.
2. Choose a random k with $0 < k < q$.
3. Compute $r = (g^k \bmod p) \bmod q$.
4. Compute $s = (z + x \cdot r)k^{-1} \bmod q$.
5. The signature is (r, s), a pair of 160-bit numbers.

DSA: verifying

To verify signature (r, s) on message m from a user with public key X we do as follows:

1. Compute the hash value $z = H(m)$.
2. Compute $w = s^{-1} \bmod q$.
3. Compute $u_1 = (z \cdot w) \bmod q$.
4. Compute $u_2 = (r \cdot w) \bmod q$.
5. Compute $v = ((g^{u_1} y^{u_2}) \bmod p) \bmod q$.
6. Accept the signature if $v = r$.

Justification of DSA

We note that for a properly signed message

$$\begin{aligned} v &= ((g^{u1} y^{u2}) \bmod p) \bmod q \\ &= ((g^{z \cdot s^{-1} \bmod q} (g^x \bmod p)^{r \cdot s^{-1} \bmod q}) \bmod p) \bmod q \\ &= (g^{(z \cdot s^{-1} + x \cdot r \cdot s^{-1}) \bmod q} \bmod p) \bmod q \\ &= (g^k \bmod p) \bmod q \\ &= r. \end{aligned}$$

Textbook references

Chapter references for this lecture's material

- Katz-Lindell: sections 4.2, 4.3, 5.3, 5.4.
- Stallings: sections 11.3, 11.5, 12.1-12.4, 13.2.

Next time

Elliptic curve cryptography: the future?