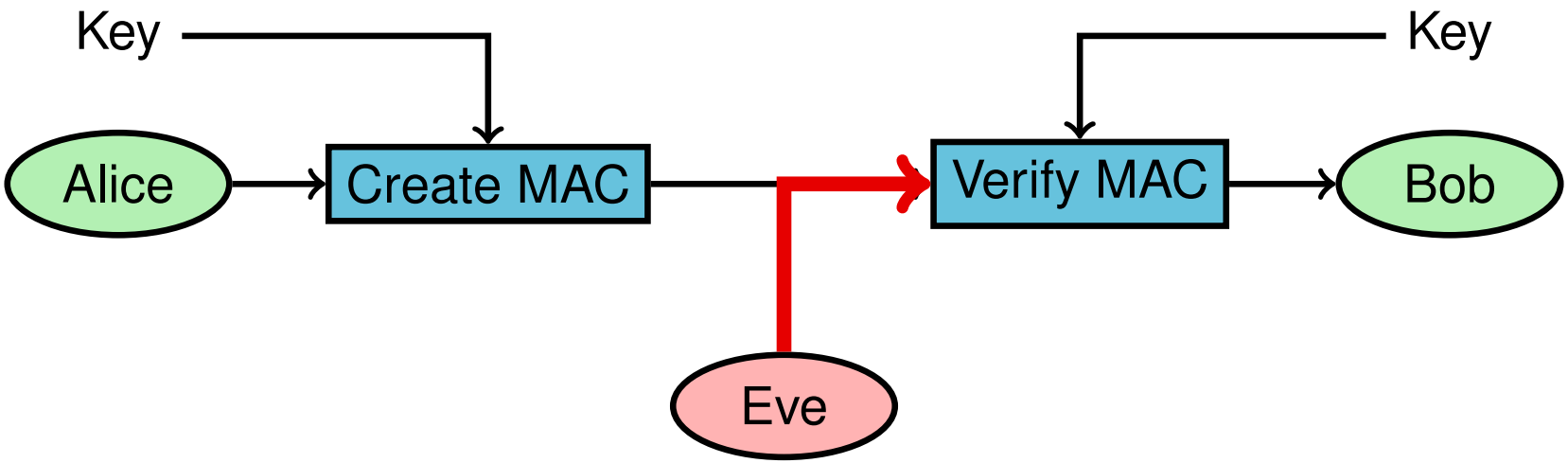
Cryptography Lecture 8

Digital signatures, hash functions

LINKÖPING UNIVERSITY

A Message Authentication Code is what you get from symmetric cryptography

A MAC is used to prevent Eve from creating a new message and inserting it instead of Alice's message



Signature vs MAC

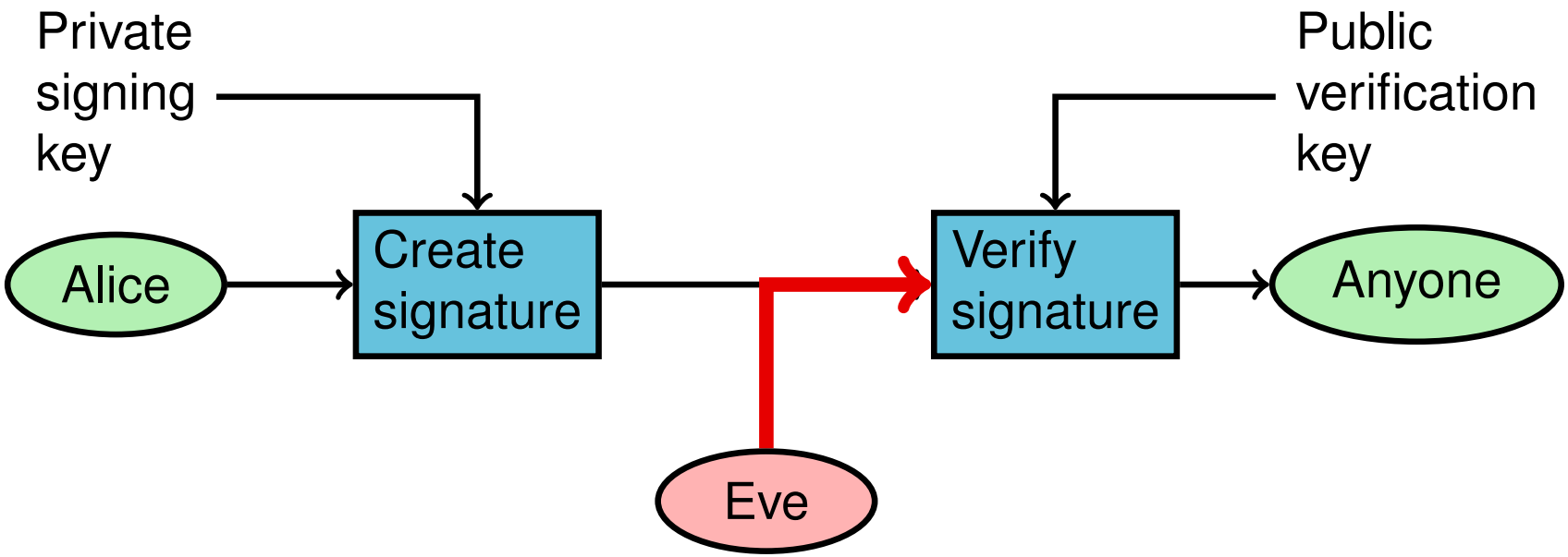
- A MAC, Message Authentication Code, preserves data integrity, i.e., it ensures that creation and any changes of the message have been made by authorised entities
- Only the authorised entities can check a MAC, and all who can check can also change the data
- In most legally interesting cases, you want to be able to verify that one single individual wrote something
- Also, in many situations it is good if everyone is able to check the signature

Digital signatures

- In asymmetric ciphers, one single individual holds the private key, while everyone can get the public key
- So if you encrypt with the private key, and send both cryptogram and message, anyone can check that “decryption” with the public key does indeed create the message
- Note that some public key systems do not allow “encryption” with the private key
- Most systems can be modified to generate and verify signatures

A digital signature can be created using asymmetric cryptography

Used to prevent Eve from creating messages and present them as written by of Alice



Digital signatures

- A digital signature should not only be tied to the signing user, but also to the message
- The example of encrypting with the private key does this: only Alice can create it, and it is valid only if the decryption and the plaintext coincide
- No attempt is made to hide the information (unless it is encrypted using another method)

RSA signatures

- Alice sets up RSA as usual
- In order to sign a message m, Alice uses her private key d (and not Bob's public key) to create the signature

$$s = m^d \bmod n$$

- Alice now gives both m and s to Bob
- He uses Alice's public key to verify the signature by comparing

$$m \quad \text{and} \quad s^e \bmod n$$

Attacks on RSA signatures

- Eve wants to sign another message m_E so that it seems to be from Alice
- Eve cannot generate a signature directly because she does not have the secret key d
- She could try to choose signature s_E first and calculate

$$m_E = s_E^e \bmod n$$

  but it is unlikely that s_E^e is a meaningful message
- Note that two message-signature pairs gives away a third since

$$(m_1 m_2)^d = m_1^d m_2^d \bmod n$$

Variation: Blind RSA signatures

- Bob wants to prove that he has created a document at a certain time, but keep it secret, and Alice agrees to help him. She sets up standard RSA, keeping d for herself.
- Bob chooses a random integer k, and gives Alice the message

$$t = k^e m \bmod n$$

- The number t is random to Alice, but she signs the message and gives the signature to Bob

$$s = t^d = k^{ed} m^d = k m^d \bmod n$$

- Bob can now divide by k (why is this possible?) and retrieve m^d, Alice's signature for m.

Do not use the same key pairs for signing and encryption!

- If Alice allows using the same e for encryption and verification, Eve can intercept $c = m^e \bmod n$. To decrypt c, Eve should create

  $$x = r^e c \bmod n$$

  for some r which yields a reasonable message x, where "reasonable" means that Eve can get Alice to sign x
- If Alice does this, the return value is

  $$s = x^d = (r^e c)^d = rm \bmod n$$

- Eve can now divide by r (why is this possible?) and find

  $$m = sr^{-1} \bmod n$$

ElGamal signatures

- Choose a large prime p, and a primitive root α mod p. Also, take a random integer a and calculate $\beta = \alpha^a \bmod p$
- The public key is the values of p, α, and β, while the secret key is the value a
- Signing uses a random integer k with $\gcd(k, p-1) = 1$, and the signature is the pair (r, s) where

$$\begin{cases} r = \alpha^k \bmod p \\ s = k^{-1}(m - ar) \bmod (p-1) \end{cases}$$

  (encryption: $(\alpha^k, \beta^k m)$)
- Verification is done comparing $\beta^r r^s$ and α^m mod p, since

$$\beta^r r^s = \alpha^{ar} \alpha^{k(m-ar)/k} = \alpha^m \bmod p$$

The need for hashing

- Unfortunately, all known signature algorithms (RSA, ElGamal, . . .) are slow
- Also, all known signature algorithms generate output with the same size as the input
- Therefore, it is much better to shorten (hash) the message first, and sign the short hash:

$$(m, \text{sig}(m)) \text{ becomes } (m, \text{sig}(h(m)))$$

- A typical hash function has 160-512 bit output (giving 80-256 ”bits” of security)

Digital Signature Algorithm ($\sim$ElGamal)

- This is a modification to the ElGamal signature scheme adopted as standard by NIST in 1994
- Some debate followed, comparing DSA and RSA signatures
- The most serious problem was parameter size, which is better in later versions
- The main change from ElGamal is to choose p so that $p-1$ has a 160-bit prime factor q, and exponents are mod q

DSA signatures

- Choose a 160-bit prime q, a large prime p so that q is a factor in $p-1$, and a primitive root g mod p. Set $\alpha = g^{(p-1)/q} \bmod p$ so that $\alpha^q = 1 \bmod p$, take a random integer $a < q-1$ and calculate $\beta = \alpha^a \bmod p$
- The public key is the values of p, q, α, and β, while the secret key is the value a
- Signing uses a random integer $k < q-1$, and the signature is the pair (r, s) where

$$\begin{cases} r = (\alpha^k \bmod p) \bmod q \\ s = k^{-1}(m - ar) \bmod q \end{cases}$$

- To verify, compute $t = s^{-1}m \bmod q$ and $u = s^{-1}r \bmod q$. Verification is done comparing $(\alpha^t \beta^u \bmod p)$ and $r \bmod q$, since

$$(\alpha^t \beta^u \bmod p) = (\alpha^{km/(m+ar)} \alpha^{ar/(m+ar)} \bmod p) = (\alpha^k \bmod p) = r \bmod q$$

ElGamal signatures

- Choose a large prime p, and a primitive root α mod p. Also, take a random integer a and calculate $\beta = \alpha^a \bmod p$
- The public key is the values of p, α, and β, while the secret key is the value a
- Signing uses a random integer k with $\gcd(k, p-1) = 1$, and the signature is the pair (r, s) where

$$\begin{cases} r = \alpha^k \bmod p \\ s = k^{-1}(m - ar) \bmod (p-1) \end{cases}$$

  (encryption: $(\alpha^k, \beta^k m)$)
- Verification is done comparing $\beta^r r^s$ and α^m mod p, since

$$\beta^r r^s = \alpha^{ar} \alpha^{k(m-ar)/k} = \alpha^m \bmod p$$

DSA signatures

- Choose a 160-bit prime q, a large prime p so that q is a factor in $p-1$, and a primitive root g mod p. Set $\alpha = g^{(p-1)/q} \bmod p$ so that $\alpha^q = 1 \bmod p$, take a random integer $a < q-1$ and calculate $\beta = \alpha^a \bmod p$
- The public key is the values of p, q, α, and β, while the secret key is the value a
- Signing uses a random integer $k < q-1$, and the signature is the pair (r, s) where

$$\begin{cases} r = (\alpha^k \bmod p) \bmod q \\ s = k^{-1}(m - ar) \bmod q \end{cases}$$

- To verify, compute $t = s^{-1}m \bmod q$ and $u = s^{-1}r \bmod q$. Verification is done comparing $(\alpha^t \beta^u \bmod p)$ and $r \bmod q$, since

$$(\alpha^t \beta^u \bmod p) = (\alpha^{km/(m+ar)} \alpha^{ar/(m+ar)} \bmod p) = (\alpha^k \bmod p) = r \bmod q$$

DSA (and ElGamal) security requirement

- The parameter k must be chosen completely at random for every signature
- Otherwise, the first half of a DSA signature $r = (\alpha^k \bmod p) \bmod q$ will tell Eve that k is reused
- The second halfs are $s_1 = k^{-1}(m_1 + ar)$ and $s_2 = k^{-1}(m_2 + ar)$, both mod q

$$k = \frac{m_1 + ar}{s_1} = \frac{m_2 + ar}{s_2}$$

$$s_2(m_1 + ar) = s_1(m_2 + ar)$$

$$a = \frac{s_1 m_2 - s_2 m_1}{(s_2 - s_1)r}$$

Why does DSA use arithmetic mod q?

There are advantages when using $\alpha = g^{(p-1)/q}$ mod p (so that $\alpha^q = 1$ mod p) instead of the primitive root directly

- One advantage is that the signatures can be generated mod q instead of mod p, which reduces calculation and increases speed
- Security is still based on the difficulty of discrete log mod p (and discrete log algorithms have problems with large factors in $p - 1$)
- Finally, the verification uses two exponentiations rather than ElGamal's three, this speeds things up

Key length

Table 7.2: Key-size Equivalence.

Security (bits)	RSA	DLOG		EC
		field size	subfield	
48	480	480	96	96
56	640	640	112	112
64	816	816	128	128
80	1248	1248	160	160
112	2432	2432	224	224
128	3248	3248	256	256
160	5312	5312	320	320
192	7936	7936	384	384
256	15424	15424	512	512

Table 7.3: Effective Key-size of Commonly used RSA/DLOG Keys.

RSA/DLOG Key	Security (bits)
512	50
768	62
1024	73
1536	89
2048	103

From “ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012)”

Key length

Table 7.4: Security levels (symmetric equivalent)

Security (bits)	Protection	Comment
32	Real-time, individuals	Only auth. tag size
64	Very short-term, small org	Not for confidentiality in new systems
72	Short-term, medium org Medium-term, small org	
80	Very short-term, agencies Long-term, small org	Smallest general-purpose $<$ 4 years protection (E.g., use of 2-key 3DES, $< 2^{40}$ plaintext/ciphertexts)
96	Legacy standard level	2-key 3DES restricted to 10^6 plaintext/ciphertexts, $\approx$ 10 years protection
112	Medium-term protection	$\approx$ 20 years protection (E.g., 3-key 3DES)
128	Long-term protection	Good, generic application-indep. Recommendation, $\approx$ 30 years
256	”Foreseeable future”	Good protection against quantum computers unless Shor’s algorithm applies.

From “ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012)”

Hash functions are not quite the previously mentioned one-way functions

A one-way function is a function that is easy to compute but computationally hard to reverse

- Easy to calculate $f(x)$ from x
- Hard to invert: to calculate x from $f(x)$

There is no proof that one-way functions exist, or even real evidence that they can be constructed

Even so, there are examples that seem one-way: they are easy to compute but we know of no easy way to reverse them, for example

x^2 is easy to compute mod $n = pq$ but $x^{1/2}$ is not

Hash functions are not quite the previously mentioned one-way functions

A one-way hash function is a function that is easy to compute but computationally hard to find a preimage for (there is no inverse function, there are many preimages)

- Easy to calculate $h(x)$ from x
- Hard to find a preimage: to calculate x' from $h(x)$ so that $h(x') = h(x)$

There is no proof that one-way hash functions exist, or even real evidence that they can be constructed

Even so, there are examples that seem to be one-way hash functions: they are easy to compute but we know of no easy way to find a preimage, and examples will follow

Security of signing a message hash

- Suppose that Eve has seen the pair $(m, \text{sig}(h(m)))$ and wants to sign her own message m_E. This is easy if $h(m_E) = h(m)$
- Therefore, good hash functions should make it difficult to find messages m_E so that $h(m_E) = h(m)$
- This is the reason to use a one-way hash function
- It should be difficult to find m' that under h returns the value $h(m)$; this is sometimes called “preimage resistance”

One-way hash functions

A one-way hash function is a function that is easy to compute but computationally hard to find a preimage for (there is no inverse function, there are many preimages)

- Easy to calculate $h(x)$ from x
- Hard to find a preimage: to calculate x' from $h(x)$ so that $h(x') = h(x)$

There is no proof that one-way hash functions exist, or even real evidence that they can be constructed

Even so, there are examples that seem to be one-way hash functions: they are easy to compute but we know of no easy way to find a preimage

Weakly collision-free hash functions

A weakly collision-free hash function is a function that is easy to compute but computationally hard to find a second preimage for

- Easy to calculate $h(x)$ from x
- ~~Hard to find a preimage: to calculate x' from $h(x)$ so that $h(x') = h(x)$~~
- Hard to find a second preimage: to calculate x' from x so that $h(x') = h(x)$

There is no proof that weakly collision-free hash functions exist, or even real evidence that they can be constructed

Even so, there are examples that seem weakly collision-free: they are easy to compute but we know of no easy way to find a second preimage

Security of signing a message hash

- Suppose that Eve has seen the pair $(m, \mathrm{sig}(h(m)))$ and wants to sign her own message m_E. This is easy if $h(m_E) = h(m)$
- Therefore, good hash functions should make it difficult to, given m, find messages m_E so that $h(m_E) = h(m)$
- This is the reason to use a weakly collision-free hash function
- It should be difficult to, given m, find m' that under h returns the value $h(m)$; this is sometimes called "second preimage resistance"

Weakly collision-free hash functions

A weakly collision-free hash function is a function that is easy to compute but computationally hard to find a second preimage for

- Easy to calculate $h(x)$ from x
- ~~Hard to find a preimage: to calculate x' from $h(x)$ so that $h(x') = h(x)$~~
- Hard to find a second preimage: to calculate x' from x so that $h(x') = h(x)$

There is no proof that weakly collision-free hash functions exist, or even real evidence that they can be constructed

Even so, there are examples that seem weakly collision-free: they are easy to compute but we know of no easy way to find a second preimage

This is of course important when signing hash values

Strongly collision-free hash functions

A strongly collision-free hash function is a function that is easy to compute but computationally hard to find a collision for

- Easy to calculate $h(x)$ from x
- ~~Hard to find a preimage: to calculate x' from $h(x)$ so that $h(x') = h(x)$~~
- ~~Hard to find a second preimage: to calculate x' from x so that $h(x') = h(x)$~~
- Hard to find a collision: to find x and x' ($x' \neq x$) such that $h(x') = h(x)$

There is no proof that strongly collision-free hash functions exist, or even real evidence that they can be constructed

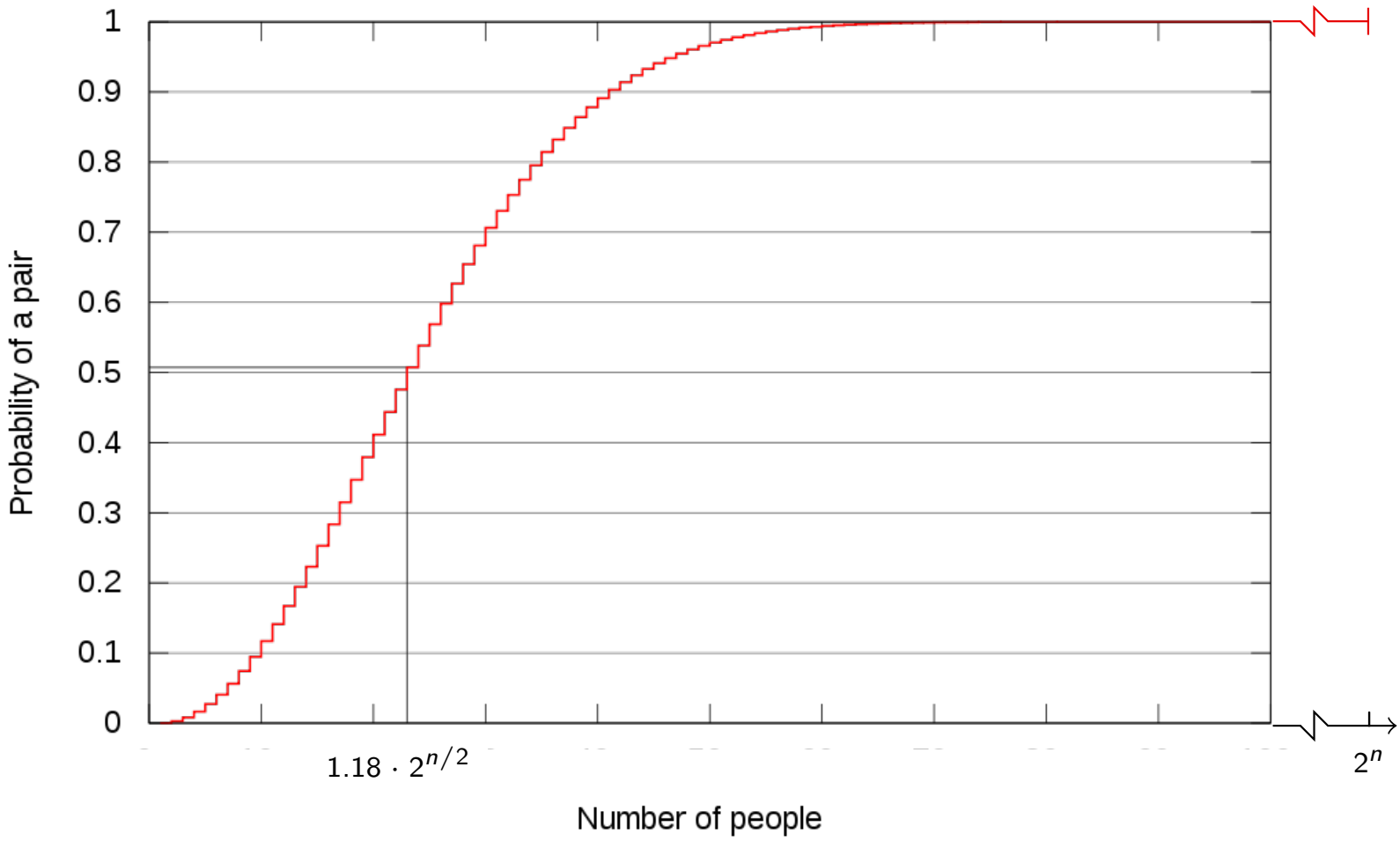
Even so, there are examples that seem strongly collision-free: they are easy to compute but we know of no easy way to find a collision

Birthday attacks on message hash signatures

- Fred (the Fraudster) knows that Alice will sign a contract. His goal is to get a signature from Alice on a different contract.
- Fred takes the original contract and produces small variations in it. He can add spaces at the line ends, change the wording slightly, add nonprinting data, and so on. Thirty changes will give 2^{30} different documents.
- He now does the same with the fraudulent contract, and attempts to find a match for the hash values of the two lists. The same signature will be valid for those two contracts
- If the hash values are shorter than 60 bits, the probability of a match is very high.

The birthday paradox

How many people must there be in a room so that the probability of two of them having the same birthday is larger than 50%?



In practice, weakly collision-free hash functions are used

A weakly collision-free hash function is a function that is easy to compute but computationally hard to find a second preimage for

- Easy to calculate $h(x)$ from x
- ~~Hard to find a preimage: to calculate x' from $h(x)$ so that $h(x') = h(x)$~~
- Hard to find a second preimage: to calculate x' from x so that $h(x') = h(x)$

There is no proof that weakly collision-free hash functions exist, or even real evidence that they can be constructed

Even so, there are examples that seem weakly collision-free: they are easy to compute but we know of no easy way to find a second preimage

The birthday attack is included in the security estimate

Key length

Table 7.4: Security levels (symmetric equivalent)

Security (bits)	Protection	Comment
32	Real-time, individuals	Only auth. tag size
64	Very short-term, small org	Not for confidentiality in new systems
72	Short-term, medium org Medium-term, small org	
80	Very short-term, agencies Long-term, small org	Smallest general-purpose $<$ 4 years protection (E.g., use of 2-key 3DES, $< 2^{40}$ plaintext/ciphertexts)
96	Legacy standard level	2-key 3DES restricted to 10^6 plaintext/ciphertexts, $\approx$ 10 years protection
112	Medium-term protection	$\approx$ 20 years protection (E.g., 3-key 3DES)
128	Long-term protection	Good, generic application-indep. Recommendation, $\approx$ 30 years
256	”Foreseeable future”	Good protection against quantum computers unless Shor’s algorithm applies.

From “ECRYPT II Yearly Report on Algorithms and Keysizes (2011-2012)”

Example: the discrete log hash function

- Choose prime p so that $q = (p-1)/2$ also is prime, and α and β primitive roots mod p
- Messages are numbers m mod q^2, let $x = m$ mod q, and $y = (m/q)$ mod q so that $m = x + qy$ mod q^2. Define

$$h(m) = \alpha^x \beta^y \bmod p$$

- This value is approximately half the size of m

Theorem: If we efficiently can find messages $m \neq m'$ with $h(m) = h(m')$, we can also calculate $L_\alpha(\beta)$ efficiently

Example: the discrete log hash function

Theorem: If we efficiently can find messages $m \neq m'$ with $h(m) = h(m')$, we can also calculate $a = L_\alpha(\beta)$ efficiently

Proof: We have $m = x + qy \neq x' + qy' = m' \bmod q^2$, so $x \neq x'$ or $y \neq y' \bmod q$ (or both). Now

$$\begin{aligned}
\alpha^x \beta^y &= \alpha^{x'} \beta^{y'} \bmod p \\
\alpha^{x+ay} &= \alpha^{x'+ay'} \bmod p \\
x + ay &= x' + ay' \bmod p - 1 \\
x - x' &= a(y' - y) \bmod p - 1 = 2q \\
\begin{cases} x - x' = a(y' - y) \bmod 2 \\ x - x' = a(y' - y) \bmod q \end{cases}
\end{aligned}$$

There are at most two solutions (this occurs when $x - x' = 0 \bmod 2$), but it is easy to check which solution a that gives $\beta = \alpha^a$

Example: the discrete log hash function

- Choose prime p so that $q = (p-1)/2$ also is prime, and α and β primitive roots mod p
- Messages are numbers m mod q^2, let $x = m$ mod q, and $y = (m/q)$ mod q so that $m = x + qy$ mod q^2. Define

$$h(m) = \alpha^x \beta^y \bmod p$$

- This value is approximately half the size of m

Theorem: If we efficiently can find messages $m \neq m'$ with $h(m) = h(m')$, we can also calculate $L_\alpha(\beta)$ efficiently

Example: the discrete log hash function

- Choose prime p so that $q = (p-1)/2$ also is prime, and α and β primitive roots mod p
- Messages are numbers m mod q^2, let $x = m \bmod q$, and $y = (m/q) \bmod q$ so that $m = x + qy \bmod q^2$. Define

$$h(m) = \alpha^x \beta^y \bmod p$$

- This value is approximately half the size of m

Theorem: If the discrete log hash function is not strongly collision resistant, we can calculate $L_\alpha(\beta)$ efficiently

Theoretical security: The Random Oracle Model

- A Random Oracle is a function that gives fixed length output. If it is the first time that it receives a particular input, it gives random output. If it already has received the input, it will repeat the corresponding output
- This is the ideal strongly collision resistant function
- It can be used to prove security of encryption/signing under the assumption that the functions used behave like a random oracle
- This is usually treated as strong evidence rather than a real proof. The evidence falls if weaknesses are found in the functions used

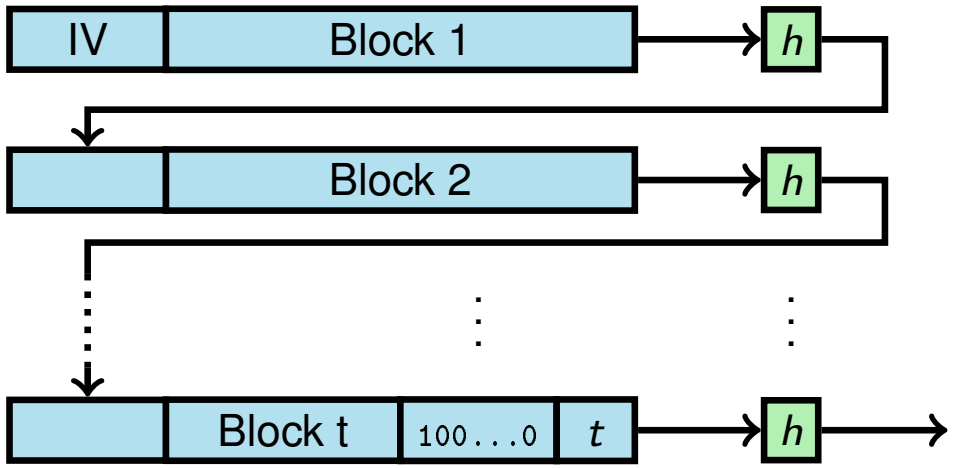
Desirable properties of a practical hash function

- Transformation of messages of any length to a fixed length block
- Dependence on every bit of message
- Everyone should be able to check the validity of a hash of a message

- You could still use a secret parameter, but this means you restrict the group of people that can verify to the group that knows the secret

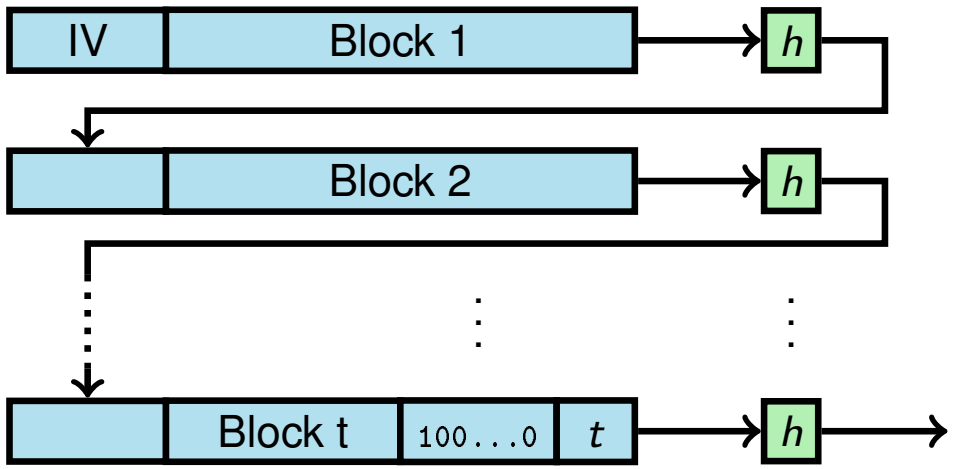
Iterative hash functions

- Mapping any-length input into fixed-length output is complicated
- A simple way is to use an iterative hash function, to divide the message into blocks, and use the following setup:

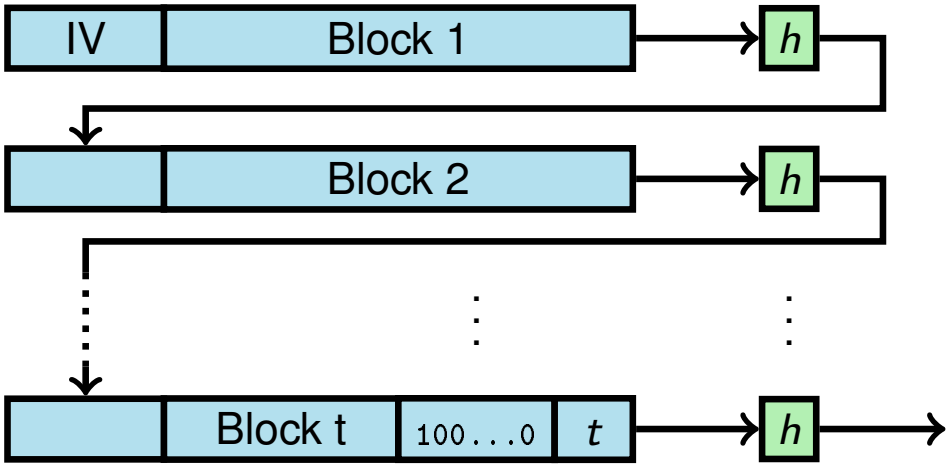


Iterative hash functions with length padding

- Mapping any-length input into fixed-length output is complicated
- A simple way is to use an iterative hash function, to divide the message into blocks, and use the following setup:



Iterative hash functions and multicollisions



- In each iteration, there is a good chance of a collision when we have searched $\approx 2^{n/2}$ messages
- Overall, we need to search $\approx t2^{n/2} = 2^{\log t + n/2}$ messages to generate 2^t collisions
- But for a proper hash function, to get a good chance of a "Multi"-birthday event (k collisions), one should need to search $\approx 2^{n(k-1)/k}$ messages

Constructing cryptographic hash functions

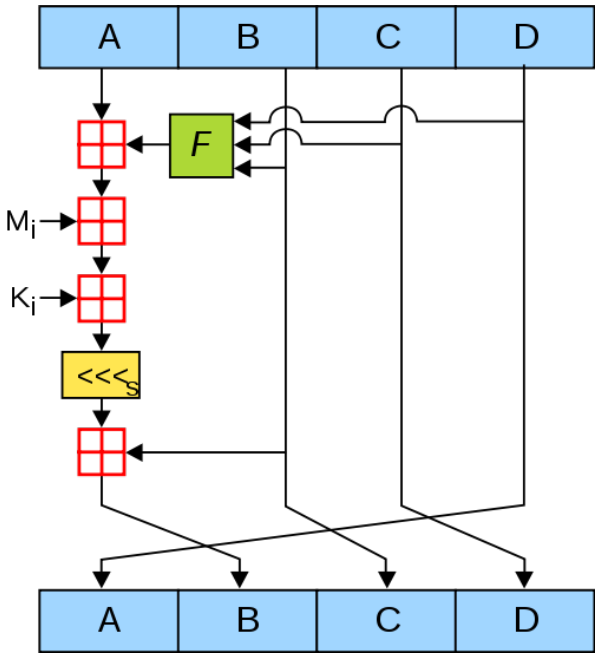
- Be wary of inventing a new hash function
  - double hashing
  - concatenating hashes
  - XORing the results
  - ...
- Adding a few operations to a hash function or combining existing functions is a tempting do-it-yourself fix, but usually adds only an insignificant amount of complexity to the attack, and can easily reduce the complexity of attack
- It was long thought that

$$H(M) = H_1(M)||H_2(M)$$

  would be much stronger than either component—no such luck—it is only slightly better than the best component
- There is a reason for the counters in MD1-5, SHA0-3, ...

MD5

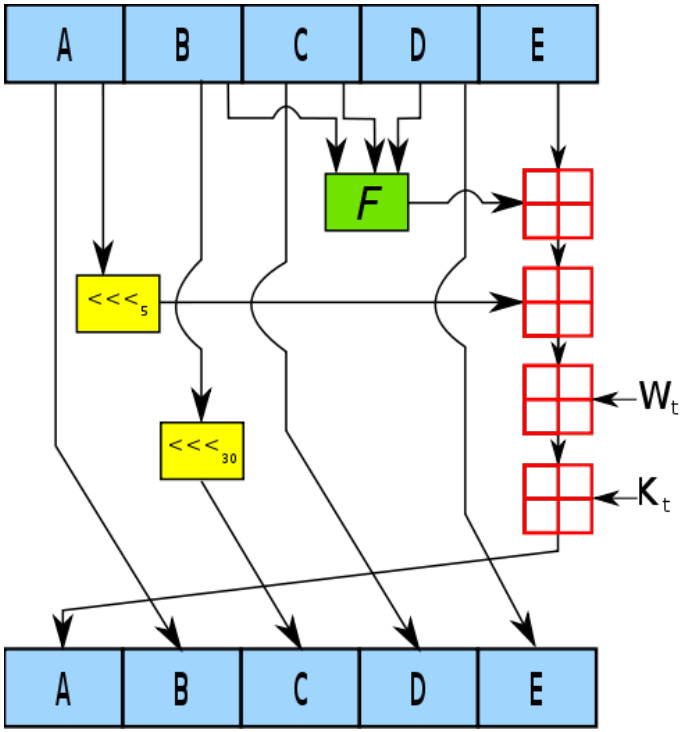
- 128-bit hash
- Very widespread (SSL/TLS, IPsec, . . .) but not secure (X.509 collisions, rouge CA, ...)
- Uses addition mod 2^{32}, message blocks M_i constant blocks K_i,



$$F(B,C,D) = \begin{cases} (B \wedge C) \vee (\neg B \wedge D) & (\text{round } 1-16) \\ (B \wedge D) \vee (C \wedge \neg D) & (\text{round } 17-32) \\ B \oplus C \oplus D & (\text{round } 33-48) \\ C \oplus (B \wedge \neg D) & (\text{round } 49-64) \end{cases}$$

SHA-1

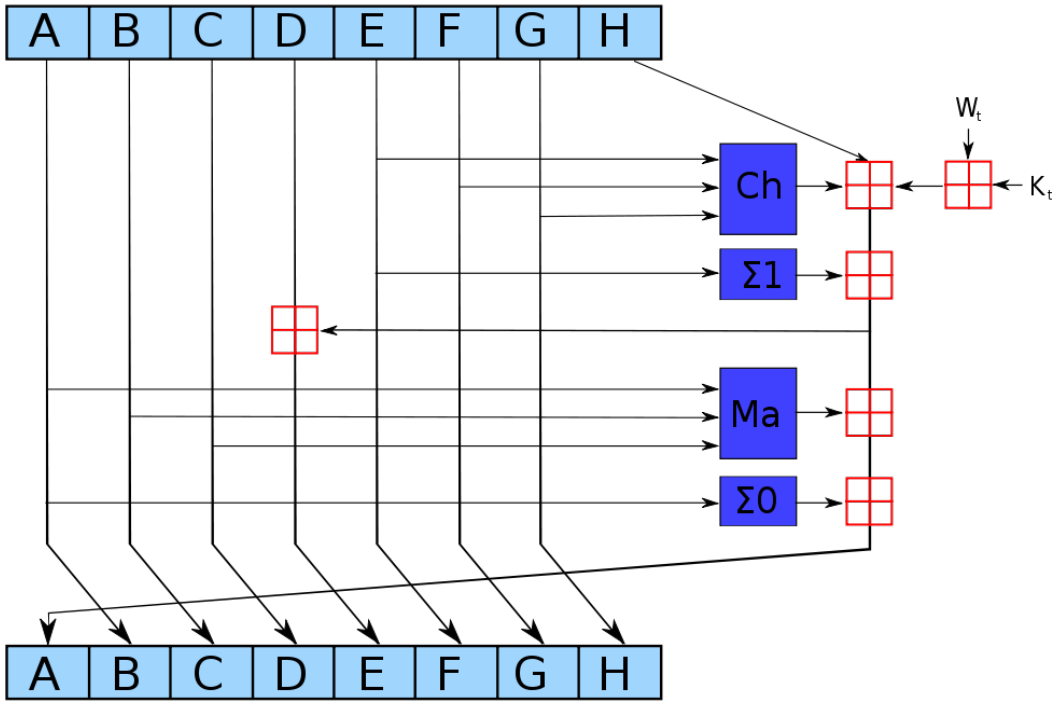
- 160-bit hash
- Very widespread, but being phased out
- Weaknesses found in 2005
- Collision attack in 2017 needs $2^{63.1}$ SHA-1 evaluations
- Uses add mod 2^{32}, 80 rounds, message dependent W_t, constant K_t



$$F(B,C,D) = \begin{cases} (B \wedge C) \vee (\neg B \wedge D) & (\text{round } 1-20) \\ B \oplus C \oplus D & (\text{round } 21-40) \\ (B \wedge C) \vee (B \wedge D) \vee (C \wedge D) & (\text{round } 41-60) \\ B \oplus C \oplus D & (\text{round } 61-80) \end{cases}$$

SHA-2

- 224, 256, 384, 512-bit hash
- Widespread, taking over from SHA-1
- The two longer ones are 10% slower
- Uses add mod 2^{32} or 2^{64}, 64 or 80 rounds, message dependent W_t, constant K_t

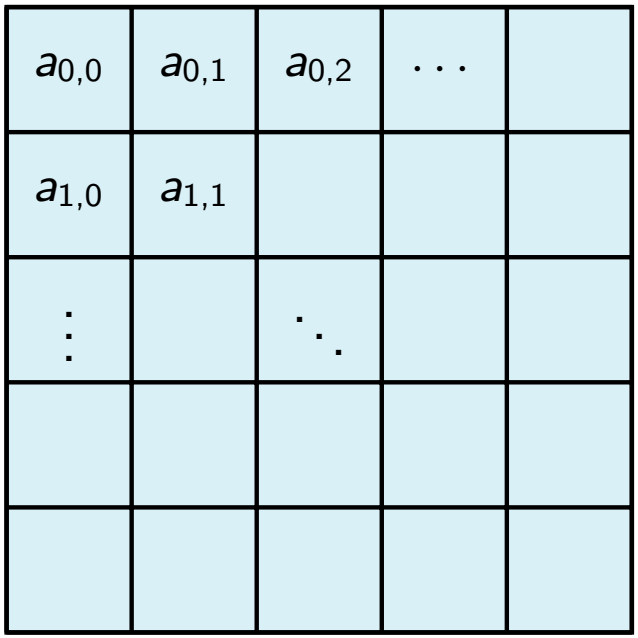


$$Ch(E,F,G) = (E \wedge F) \oplus (\neg E \wedge G)$$
$$Ma(A,B,C) = (A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$$
$$\Sigma 0(A) = (A \gg 2) \oplus (A \gg 13) \oplus (A \gg 22)$$
$$\Sigma 1(E) = (E \gg 6) \oplus (E \gg 11) \oplus (E \gg 25)$$

SHA-3 contest

- NIST started a contest 2007 to find (future) replacements for SHA-1 and SHA-2
- Finalists were chosen for reasons of speed, security, available analysis, and diversity
- NIST selected five SHA-3 candidates for the final evaluation: BLAKE, Grøstl, JH, Keccak, and Skein
- The decision was made Oct 3 2012, and the winner was Keccak
- The standard was published 2015, but there was some controversy as NIST “tuned the hash function for speed”

SHA-3



- 224, 256, 384, 512-bit hash
- 12.5 cycles per byte
- 1600-bit internal state, in 25 registers of $2^l (= 64)$ bits, $12 + 2l$ iterations of:

θ: Add bitwise column parity to adjacent column, $a_{i,j} \oplus = \bigoplus_i (a_{i,j+1} \oplus a_{i,j-1})$

ρ: Rotate registers different triangular-number steps, $a_{i,j} << t_{i,j}$

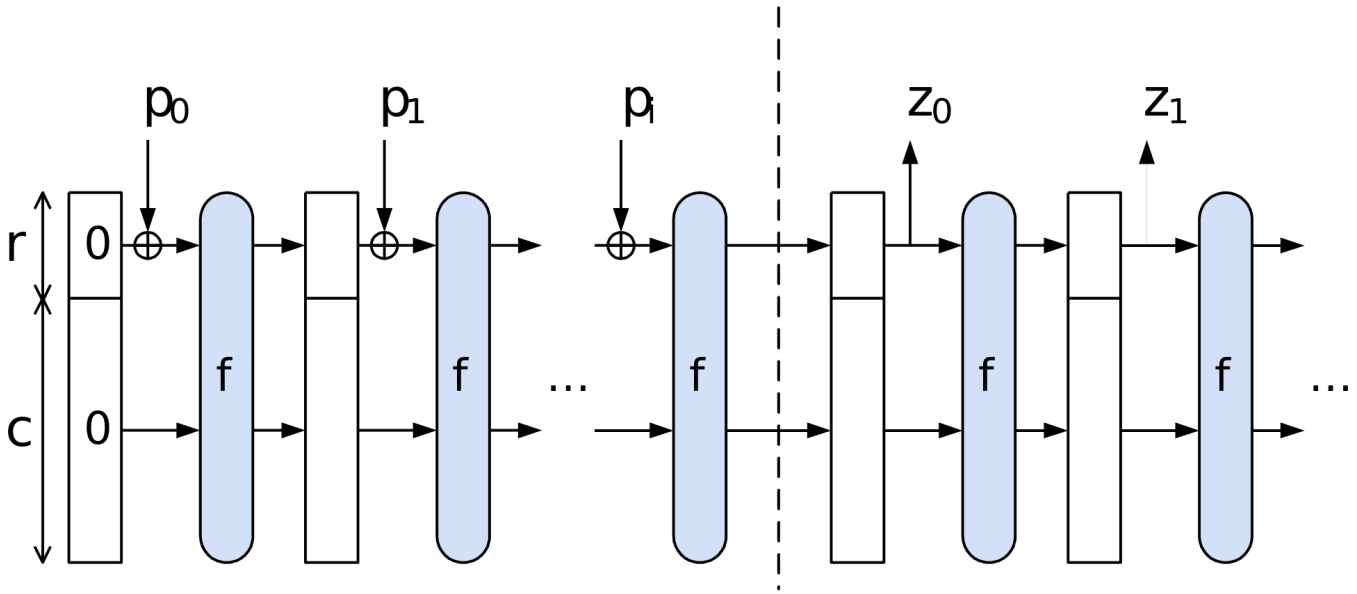
π: Permute words in fixed pattern, $a_{3i+2j,i} = a_{i,j}$

χ: Combine bits nonlinearly in rows, $a_{i,j} \oplus = \neg a_{i,j+1} \wedge a_{i,j+2}$

ι: Break symmetry, in round n ($0 \leq m \leq l$), with b_n from a degree-8 LFSR $a_{0,0}[2^m - 1] \oplus = b_{m+7n}$,

The sponge construction

- Mapping any-length input into fixed-length output is complicated
- The sponge construction is used to absorbe *r* bits in each iteration
- The unused capacity *c* should be twice the desired resistance to collision or preimage attacks
- Still needs secure padding of the last block



Other uses of cryptographic hash functions

- They hide what input to use in order to get a specific output, which is needed in for example password storage
- Cryptographic hash functions can be used to encrypt, basically running it as a block cipher in OFB mode
- Other applications of hash functions in computer science (hash tables, file integrity, . . .) often has weaker requirements than the ones used here
- It is therefore important to remember that hash functions as used in computer science often do not fulfil the requirements of a cryptographic hash function

Example: Unix passwords

- Passwords are assumed to be 8 character 7-bit ASCII
- In total 56 bits
- Encrypt the message 0 using this DES key

Weaknesses

- Dictionary attacks
- DES is fast, and there is dedicated hardware

Example: Unix password weaknesses

- Dictionary attacks
  - Use a salt to make dictionary attacks more difficult
  - the book mentions 12-bit salts
- DES is fast, and there is dedicated hardware
  - Don’t have the salt in the message
  - Instead use it to change the DES algorithm (see the book)
  - This makes hardware attacks more difficult
  - . . . and slows down an attack

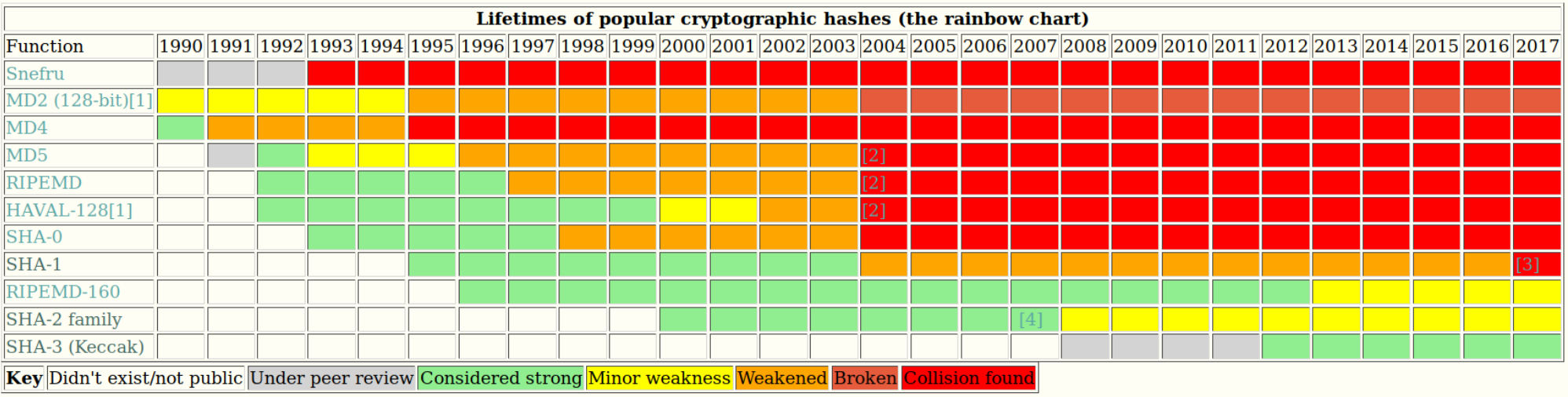
Example: Modern unixes

- Stronger algorithms (proper hash functions)
- Whole password significant
- Longer salts (up to 16 characters)
- Use “key stretching”: many many rounds
- Only root can read the passwd file

mkpasswd -m SHA-512 test
6EUEoZKNThDNKmfdb$3g5AuZFmWHCaDJDJq2GVPdLQ8CAOPdDUG

ID	Method
1	MD5
2a	Blowfish (not in mainline glibc; in some Linuxes, NetBSD)
5	SHA2-256 (since glibc 2.7)
6	SHA2-512 (since glibc 2.7)

Hash function life cycles



Historically, popular cryptographic hash functions have a useful lifetime of around 10 years