COMP343
Hash Functions and Digests

MACQUARIE
University

Revision : 1.12

Table of Contents

Basic Principles

Early Error Detection and Correction

We have used a range of error detection techniques, some dating back to the early days of computing.

- Parity (used in serial data communications, RAID 4 & 5, etc.)
- Checksums (used in simple serial protocols such as XMODEM/Christensen Protocol)
- Cyclic redundancy checks (used in Ethernet, etc.)
- Hamming Codes (used in error checking and correcting memory)
- LUN-10 check digits (used in credit card numbers)

Classical Hashing

In computer science hashes have been used for fast random access to records, by mapping a string to a unique record number.

- Hash functions defined as

  $$H : \{0,1\}^N \to \{0,1\}^n$$

  where N is the length of the key record and n is the length of indices that point to appropriate buckets, where $N > n$.

- Expected properties
  1. uniform distribution of collisions – buckets contain roughly the same number of records,
  2. efficient computation of indices.

Cryptographic Hashing - Concept

A *hash function* is required to produce a digest of a fixed length for a message of an arbitrary length. Let the hash function be

$$h : \{0,1\}^* \to \{0,1\}^n,$$

where

$$\{0,1\}^* = \bigcup_{i \in \mathcal{N}} \{0,1\}^i.$$

It is said that two different messages m_1, m_2 *collide* if

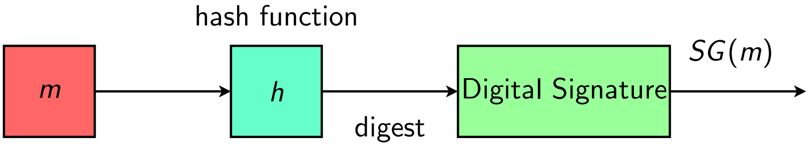
$$h(m_1) = h(m_2).$$

Types of Digests

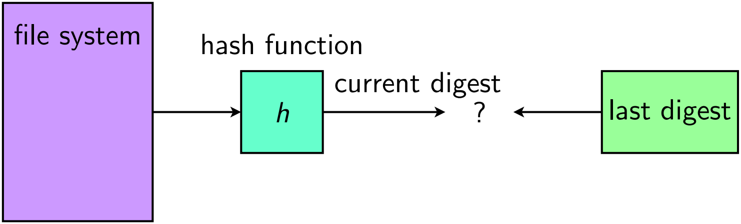
- Keyed Digests and Hash Functions - uses a shared secret key. Also known as Message Authentication Codes
- Unkeyed Hash Hash Functions - also known as Message Integrity Codes, Modification Detection Codes
  - One Way Hash Functions (OWHF's)
  - Collision-Resistant Hash Functions

Cryptographic Hashing - Applications

- Message digests for digital signatures – short and unique “fingerprint” for messages (very short and very long)

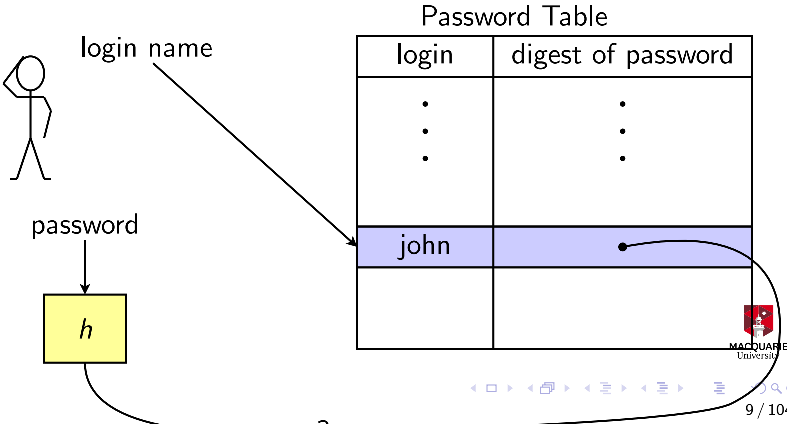


- Integrity checking – detecting changes to files/directories (e.g. Tripwire, git)



Cryptographic Hashing - Applications

- One-way function – storing 'fingerprints' or digests of passwords without revealing them



Password Table

login	digest of password
⋮	⋮
john	•

Properties of Hash Functions

Given a hash function $h : \{0,1\}^* \to \{0,1\}^n$. The function is

- *Preimage resistant* – if for (almost) any digest d, it is computationally intractable to find the preimage (message m) such that

  $$d = h(m).$$

  This means that the function is *one-way*.
- *2nd preimage resistant* – if given the description of the function h and a chosen message m_1, it is computationally intractable to find another message m_2 which collides with m_1, i.e.

  $$h(m_1) = h(m_2).$$
- *Collision resistant* – if given the description of the function h, it is computationally infeasible to find two distinct messages m_1, m_2 which collide, i.e.

  $$h(m_1) = h(m_2).$$

  Collision resistance is equivalent to *strong collision resistance*.

Properties of Hash Functions

Hash functions are also expected to provide:

- Pseudorandomness – keyed compression function is a pseudorandom function (PRF) family.
- Random oracle – hash functions behave like a random function.
- Indistinguishability – the output of hash function is indistinguishable from the output of a random function.

So we sometimes use them as pseudorandom number generators or oracles.

Properties of Hash Functions

There are, however, two major classes of hash function defined as follows:

1. *A one-way hash function* (OWHF) compresses messages of arbitrary length into digests of fixed length. The computation of the digest for a message is easy. The function is
   - preimage and
   - 2nd preimage resistant.

   Equivalently, the function is termed *weak one-way hash function*.
2. *A collision resistant hash function* (CRHF) compresses messages of arbitrary length into digests of fixed length. The computation of the digest for a message is easy. The function is collision resistant. Equivalently, the function is termed *strong one-way hash function*.

Collision Resistance Implies One-Wayness

This statement can be proved by contradiction.

Assume that a hash function h is not one-way, i.e. there is a probabilistic polynomial time algorithm R which for a given digest d returns a message $m = R(d)$ such that $d = h(m)$.

The algorithm R can be used to generate collisions in the following way:

- Select at random m and find its digest $d = h(m)$.
- Call the algorithm R which returns $m' = R(d)$ such that $d = h(m')$.
- If $m \neq m'$, then this is a collision. Otherwise select other random message and repeat the steps.

Collisions must always exist, due to the pigeonhole principle: there does not exist an injective function on finite sets whose codomain is smaller than its domain.

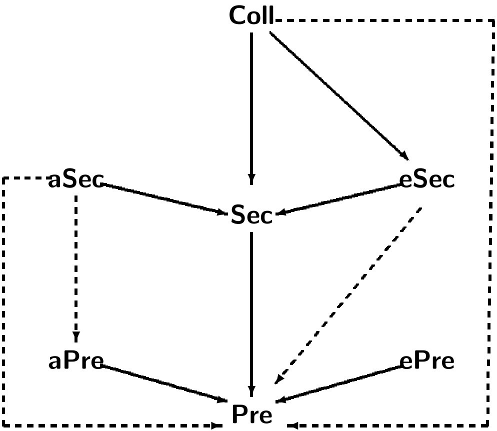
Family of Hash Functions

- Definition

$$H : \mathcal{K} \times \mathcal{M} \rightarrow \{0,1\}^n$$

- Properties (Rogaway and Shrimpton, [RS04])
  - Preimage (Pre) resistant; Second preimage (Sec) resistant; and Collision (Coll) resistant
  - everywhere Preimage resistant (ePre) – random key and fixed challenge message
  - always Preimage resistant (aPre) – fixed key and random challenge message
  - everywhere Second preimage resistant (eSec)
  - always Second preimage resistant (aSec)

Relation Among Properties



Properties of Hash Functions

- Ease of computation
- Compression - $h : D \mapsto R$ with $|D| > |R|$
- For unkeyed hash functions:
  - Preimage resistance: it is computationally infeasible to find any input which hashes to a pre-specified output, i.e. given y to find m s.t. $h(m) = y$
  - 2nd-preimage resistance: it is computationally infeasible to find a second input which produces the sae hash as a given input, i.e. given m_1, to find m_2 s.t. $h(m_2) = h(m_1)$ and $m_1 \neq m_2$.
  - Collision resistance: it is computationally infeasible to find *any* two inputs $m_1, m_2, m_1 \neq m_2$ such that $h(m_2) = h(m_1)$

For MAC's

  - Computation-resistance: given zero or more text-MAC pairs $(m_i, h_k(m_i))$ it is computationally infeasible to compute any text-MAC pair $(x, h_k(x))$ for any new input $x \notin m_i$ (including possibly for $h_k(x) = h_k(x_i)$ for some i). This implies *key non-recovery*; if this does not hold, the algorithm is vulnerable to *key recovery*.

Applications

- Assuring message integrity (MIC's)
- Assuring authenticity of origin (MAC's)
  - Related question - can MAC's provide non-repudiation?
- Confirmation of knowledge
- Key Management
- PRNG's

Key Management

- Key Derivation, e.g. PBKDF2 (Password-Based Key Derivation Function 2)
- Key updating (*backward security*)
  - Principals who share a key hash it periodically to generate a new key: $k_i = h(k_{i-1})$.
  - An attacker who compromises a key will not be able to derive previous keys
  - The one-way nature of the hash function breaks the chain
- Autokeying (*forward security*)
  - Principals who share a key hash it periodically along with messages exchanged since the previous key change: $k_i = h(k_{i-1}, m_1, m_2, \ldots)$
  - If an attacker compromises a key, if the principals exchange a message which the attacker does not observe, he will not be able to generate future keys
  - Used in EFT terminals.

Building Hash Functions

MACQUARIE
University

Structures for Hash Functions

Structures for Hash Functions

How to build $H : \{0,1\}^* \rightarrow \{0,1\}^n$?

- Merkle-Damgård construction – sequential application of compression function

$$h : \{0,1\}^N \rightarrow \{0,1\}^n.$$

- Damgård tree construction – tree of the compression function

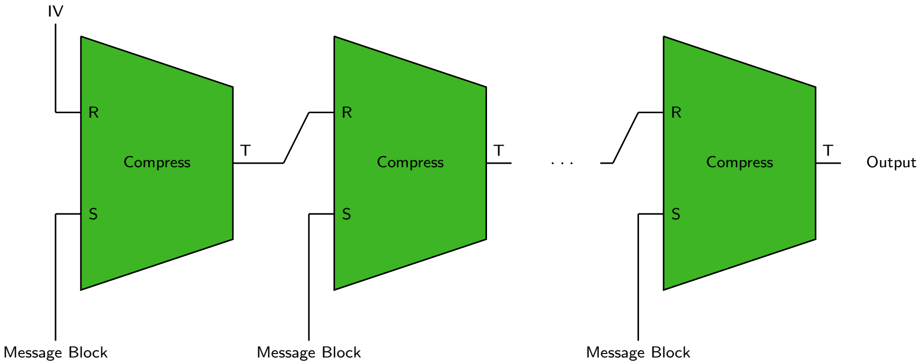
$$h : \{0,1\}^N \rightarrow \{0,1\}^n.$$

- Sponge construction – application of permutation

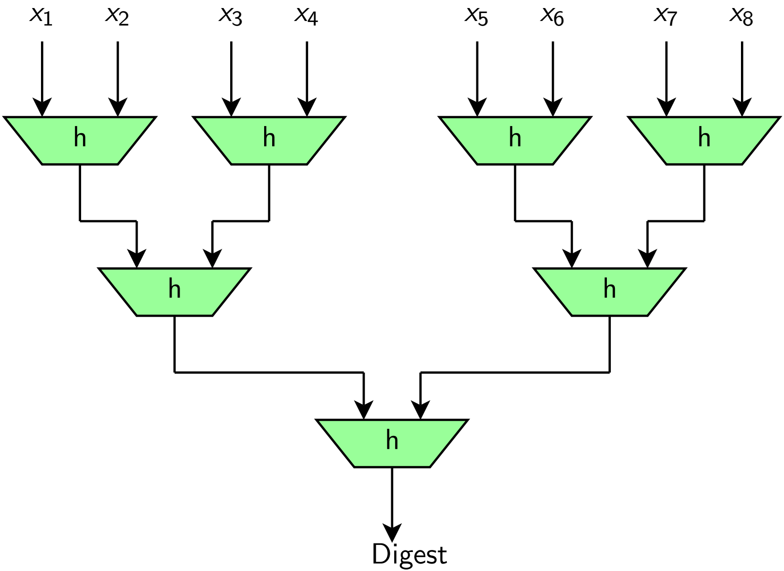
$$f : \{0,1\}^N \rightarrow \{0,1\}^N$$

  and two phases: absorbing messages and squeezing out digest.
- Wide-pipes and double-pipes

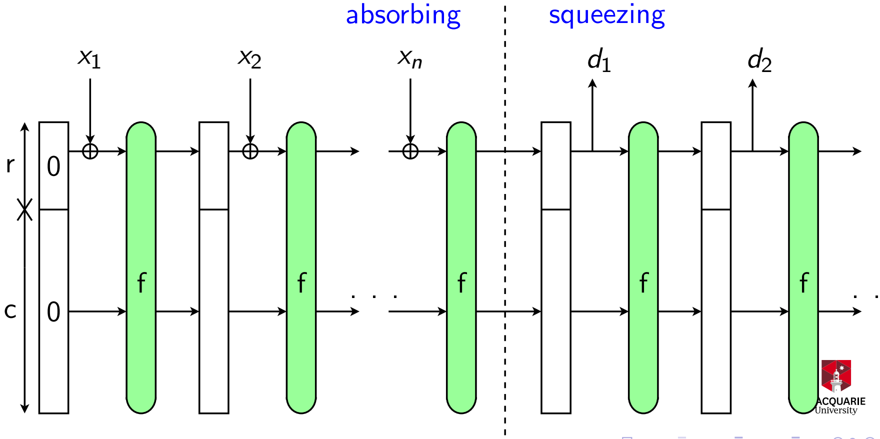
Merkle-Damgård Structure



Damgård Tree Construction

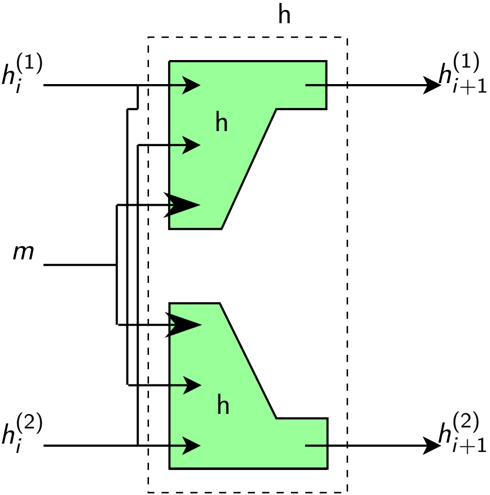


Sponge Construction



Double-pipe

$$H(h_i^{(1)},\ h_i^{(2)},\ m) = (h_{i+1}^{(1)}, h_{i+1}^{(2)}) = \Big(h(h_i^{(1)},\ h_i^{(2)},\ m),\ h(h_i^{(2)},\ h_i^{(1)},\ m)\Big)$$



Structures for Compression Functions

Structures for Compression Functions

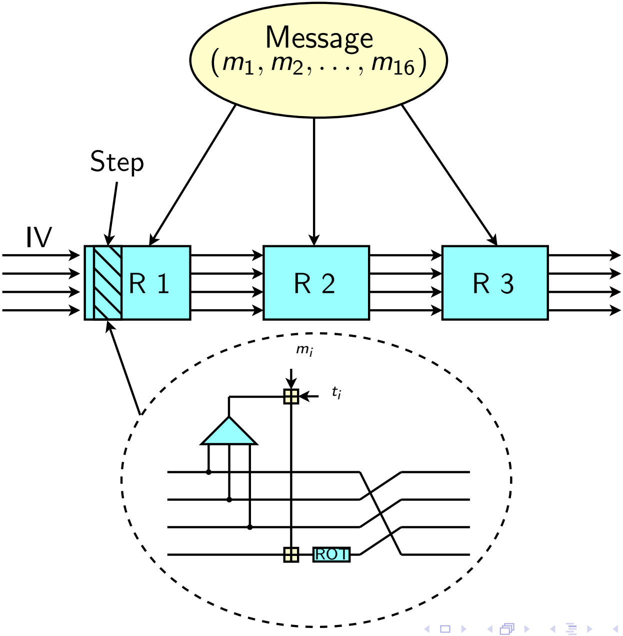
How to build compression function

$$H : \{0,1\}^N \to \{0,1\}^n,$$

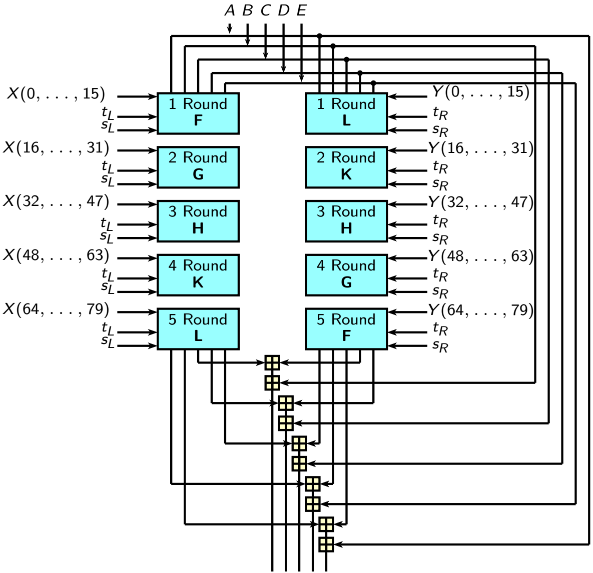
where $N > n$?

- Single chain of encryption/Feistel permutations (block cipher, MD4, MD5)
- Parallel chains of encryption/Feistel permutations
  - Double parallel chains (double block cipher, RIPE-MD)
  - Four parallel chains (FORK)

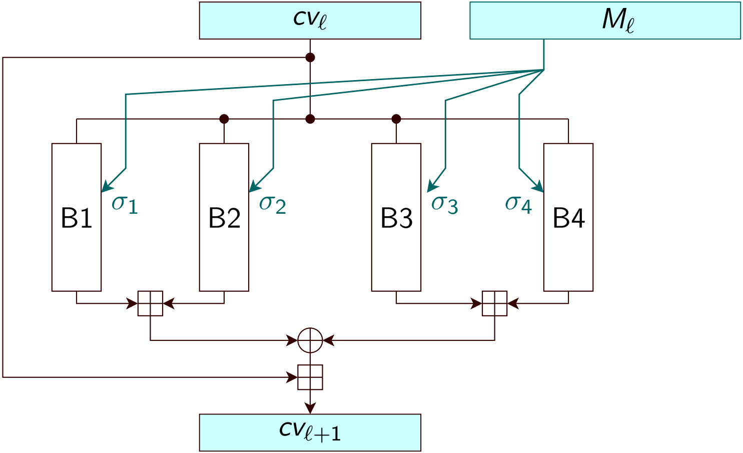
Single Chain of Feistel Permutations - MD4



Double Parallel Chains - RIPE-MD



Four Parallel Chains - FORK



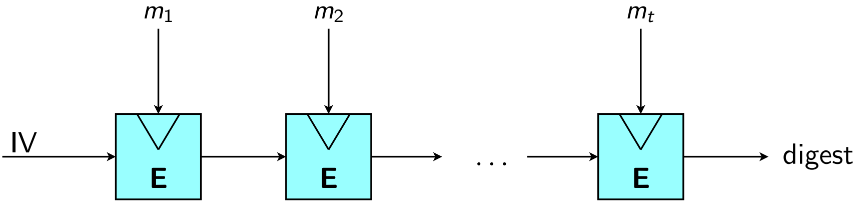
Hash Functions from Block Ciphers

Rabin Scheme

- Any secret-key cryptosystem $E : \Sigma^{2n} \to \Sigma^{n}$ can be used for hashing.
- Suppose that message $m = (m_1, m_2, \ldots, m_\ell)$. The hashing is performed according to

$$\begin{aligned} h_0 &= IV \\ h_i &= E(m_i, h_{i-1}) \text{ for } i = 1, 2, \ldots, \ell \\ d &= h_\ell \end{aligned}$$

where h_i are intermediate results of hashing, and d is the final digest of m.



Davies-Meyer Hashing Scheme

- Winternitz suggested to design a one-way function from a block cryptosystem E.

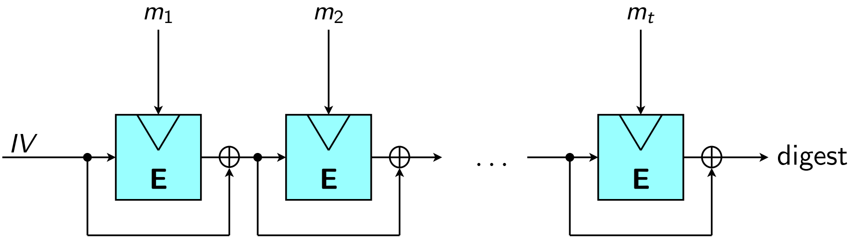
$$E^*(k\|m) = E(k,m) \oplus m.$$

- Davies used the one-way function E^* to design the following hash scheme.

$$\begin{aligned} h_0 &= IV \\ h_i &= E(m_i, h_{i-1}) \oplus h_{i-1} \text{ for } i = 1, 2, \ldots, \ell \\ d &= h_\ell \end{aligned}$$

- This scheme is immune against the meet-in-the-middle attack but may be subject to attacks based on key collision search and weak keys.

Davies-Meyer Hashing Scheme



Hashing Based on Block Ciphers - Pros and Cons

PROS

- Block cipher can be easily adapted for hashing
- Reuse of existing and well developed technology
- Block cipher designs withstood the test of time (DES)

CONS

- mismatch between key scheduling (cipher) and message expansion (hash)
- hashing is slower than custom-designed one

Custom Designed Hash Functions

Requirements for Hashing Algorithms

(1) secure, i.e. collision free – this immediately forces the digest to be at least 256 bits long (as of 2018),

(2) fast and easy to implement both in software and hardware.

A modified Feistel permutation

- Feistel permutations can be used as the basic component in the design of hash algorithm.

$$F_m : \underbrace{\{0,1\}^r \times \ldots \times \{0,1\}^r}_{\ell} \to \underbrace{\{0,1\}^r \times \ldots \times \{0,1\}^r}_{\ell}$$

Modified Feistel Permutation

- Let the input

  $$A = (A_1, \ldots, A_\ell) \in \{0,1\}^n$$

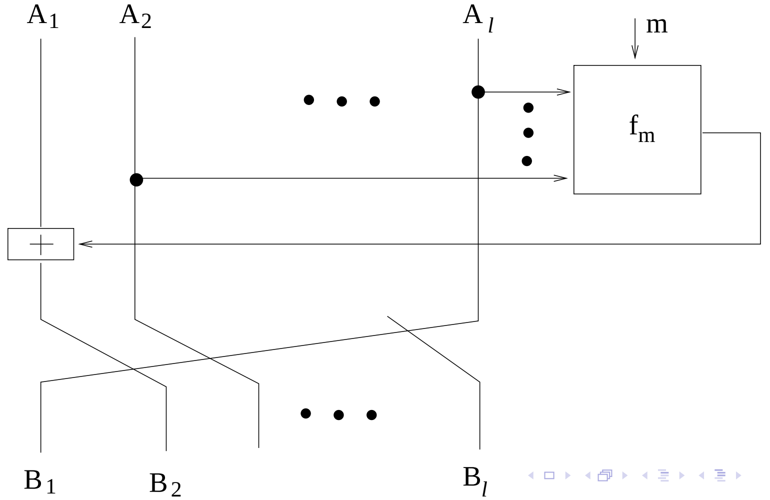
  and the output

  $$B = (B_1, \ldots, B_\ell) \in \{0,1\}^n,$$

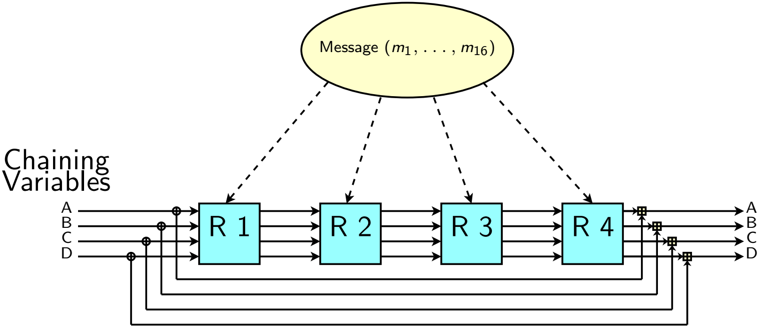
  then

  $$\begin{array}{rcl} B_1 & = & A_\ell + f_m(A_2, \ldots, A_\ell) \\ B_2 & = & A_1 \\ & \vdots & \\ B_\ell & = & A_{\ell-1}. \end{array}$$

Modified Feistel Permutation



MD Family – MD4

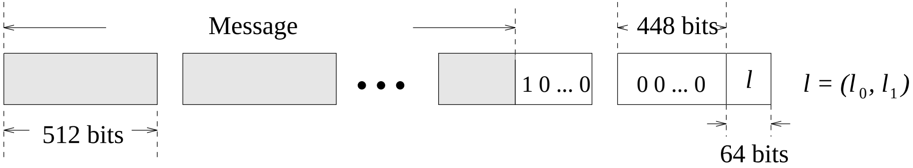


- Designed by Rivest in 1990
- Based on Feistel permutations (3 rounds/48 steps)
- collision resistance:
  - collisions for 2 rounds [Merkle90, denBoerBosselaers91]
  - collisions for full MD4 by hand [Wang04]

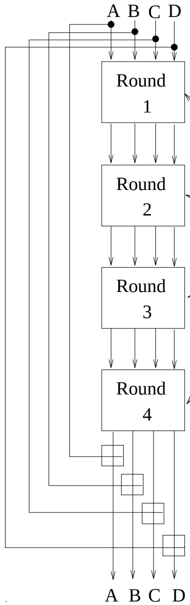
MD Family – MD5

- MD5 is a strengthened version of MD4
- MD5 compresses 512-bit messages into 128-bit digests using the 128-bit chaining input.
- Message of arbitrary length is first appended bit 1 and enough 0's so it is congruent 448 modulo 512.
- 64-bit string which includes the binary representation of the length of the original message is appended to the padded message. Now the message length is a multiple of 512.
- Hashing is done as in the MD method – block by block and each block is 512 bits long.

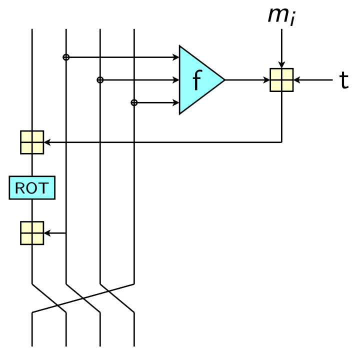
MD5 – Padding



MD5 – Compression Function



MD5 – Single Step



MD5 applies four Boolean functions:

$$\begin{aligned} f(x,y,z) &= xy \vee \bar{x}z, && \text{Round 1} \\ g(x,y,z) &= xz \vee y\bar{z}, && \text{Round 2} \\ h(x,y,z) &= x \oplus y \oplus z, && \text{Round 3} \\ k(x,y,z) &= y \oplus (x \vee \bar{z}), && \text{Round 4} \end{aligned}$$

where $\vee$ is OR, $\oplus$ is XOR and xy stands for x AND y.

MD5

Collision resistance:

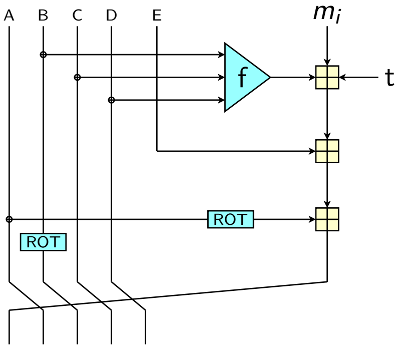
- collisions for compression function [Dobbertin96]
- collisions for full MD5 in 2^{39} [Wang04]

Efficiency

- MD5 was meant to be fast on machines with a little-endian architecture.
- For a 32-bit word $(a_0, \ldots, a_{31})$, a machine with *little-endian* architecture converts the string into integer $a_{31}2^{31} + \ldots + a_1 \cdot 2 + a_0$.
- In *big-endian* architecture, the same integer is $a_0 2^{31} + \ldots + a_{30} \cdot 2 + a_{31}$.
- Speed of MD5 is around 1 Mbytes/sec.

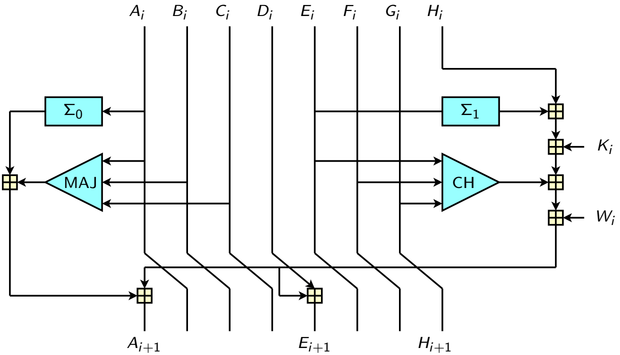
SHA-1

- SHA-0 designed by NIST in 1993 and in 1995 upgraded to SHA-1
- Based on Feistel permutations (5 rounds/80 steps)
- collision resistance:
    - collisions for SHA-0 in 2^{51} [Joux04]
    - collisions for full SHA-1 in 2^{63} [Wang05]



SHA-256

- NIST published SHA-256 in 2001 (FIPS PUB180-2)
- Feistel permutation (64 steps)



where

$$\begin{aligned}
\mathrm{MAJ}(A,B,C) &= (A \wedge B) \vee (A \wedge C) \vee (B \wedge C),\\
\mathrm{IF}(E,F,G) &= (E \wedge F) \vee (\neg E \wedge G)\\
\Sigma_0(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x),\\
\Sigma_1(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)
\end{aligned}$$

SHA-256

Collision resistance:

- collisions for SHA-256_{23} with complexity 2^{18}
- collisions for SHA-256_{24} with complexity $2^{28.5}$ [Indesteege et al, March 2008]

Custom Designed Hash - Pros and Cons

PROS

- Specifically designed for hashing
- Diffusion as the main cryptographic property
- S-box theory comes in handy
- Basic operations can be chosen to maximize speed
- Two basic components: message expansion and processing of chain variables

CONS

- Security is not proven
- Hashing is a relatively new field and lagging behind secret-key cryptography

Message Authentication

Integrity vs Authenticity of Origin

So far, we have examined *unkeyed* hash functions. These provide confirmation of message integrity but protect only against noise in the communication channel or accidental corruption.

They do not defend against a man-in-the-middle attacker (Mallory) who can intercept a message, modify it and simply append a recalculated digest.

Such a digest is known as a:

- *message integrity code* or MIC,
- *integrity check value*, or
- *cryptographic checksum*.

Message Authentication Code

We want to achieve both assurance of message integrity and authentication of origin of a message. A mechanism that can do this is known as a:

- *message authentication code* or MAC, or
- *authentication tag*.

There are two main techniques which achieve this:

- Incorporation of a shared secret (which is not transmitted) in the computation of an unkeyed hash, or
- Use of a *keyed hash* which takes a secret key, known only to the communicating parties.

Shared Secrets

Incorporation of a Shared Secret

Alice and Bob share a secret, s - usually a string, and often thought of as a password. Alice wants to send a message, m, to Bob. Alice calculates:

$$h_a = h(s \parallel m)$$

and transmits

$$m \parallel h_a$$

In order to check the message integrity, Bob also calculates

$$h_b = h(s \parallel m)$$

and compares $h_b = ?h_a$.

Notice that *s is not transmitted*. This technique is the basis of simple entity authentication protocols like CHAP (Challenge Handshake Authentication Protocol) and Cisco's OSPF protocol implementation.

Keyed Hashes

Keyed Hashing

A keyed hash function $H = \{\bar{H}_n \mid n \in \mathcal{N}\}$ is collision free if

1. any instance function h_k can be applied for messages of arbitrary length,
2. function H is a *trapdoor one-way* function, that is
   - given a key k and message m, it is easy (in polynomial time) to compute the digest $d = h_k(m)$,
   - for any polynomial size collection of pairs $(m_i, d_i = h_k(m_i))$; $i = 1, \dots, \ell(n)$, it is intractable to find the key $k \in \Sigma^n$, where $\ell(n)$ is a polynomial in n.
3. without the knowledge of k, it is computationally difficult to find a collision, that is, two distinct messages $m, m' \in \Sigma^*$ with the same digest $d = h_k(m) = h_k(m')$.

An example of keyed hashing in CBC mode

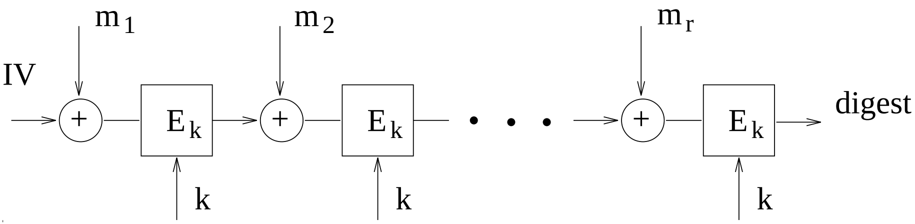
- For a message $m = (m_1, \ldots, m_r)$, hashing involves r iterations and

$$\begin{aligned} h_0 &= IV \\ h_i &= E_k(h_{i-1} \oplus m_i); \quad i = 1, \ldots, r-1 \\ d &= E_k(h_{r-1} \oplus m_r) \end{aligned}$$

  where IV is an initial value, E_k encryption function, and d digest.

- To prevent the meet-in-the-middle attack, the message block size must be at least 128 bits.

An example of keyed hashing in CBC mode



HMAC - Keyed Hashing for Message Authentication

Any hash function - MD5, SHA-1, SHA-2 - can be used in the construction of an HMAC. Let i (inner padding) be the constant value 0x36 repeated b times, where b is the block size, and o (outer padding) be the value 0x5C repeated b times. Then the HMAC is given by:

$$h(K \oplus o || h(K \oplus i || m))$$

HMAC is not subject to collision attacks because the beginning of the hash is based on a secret key which is not known to the attacker. It also avoids key recovery attacks, but is still vulnerable to generic birthday attacks (but these will require $2^{n/2}$ interactions with the system under attack - harder than $2^{n/2}$ local computations).

A Question

You are appearing as an expert witness in a trial before Mr. Justice Trent.

Bob is suing Alice, claiming that she asked him, via an authenticated message, to write some software for her, for which she would pay $50,000. Bob claims she is refusing to pay, and asks the court to order her to make payment. In evidence, Bob produces the message, which has an HMAC message authentication code.

Alice's defence is that she did commission Bob to write the software, but that she only offered to pay $5,000, and she was surprised when he demanded $50,000.

What is your advice to the judge? Should he find in Bob's favour?

Attacks on Hash Functions

Generic Attacks on Hash Functions

Given $H : \{0,1\}^* \rightarrow \{0,1\}^n$.
Ideally, one would wish to have hash function that is

- Preimage resistant – any attack have complexity at least 2^n
- Second Preimage resistant – any attack have complexity at least 2^n
- Collision resistant – any attack should have complexity at least $2^{n/2}$

Birthday Paradox

(Q) What is the minimum number of pupils in a classroom so the probability that at least two pupils have the same birthday, is greater than 0.5 ?

(A) 23

Why ?

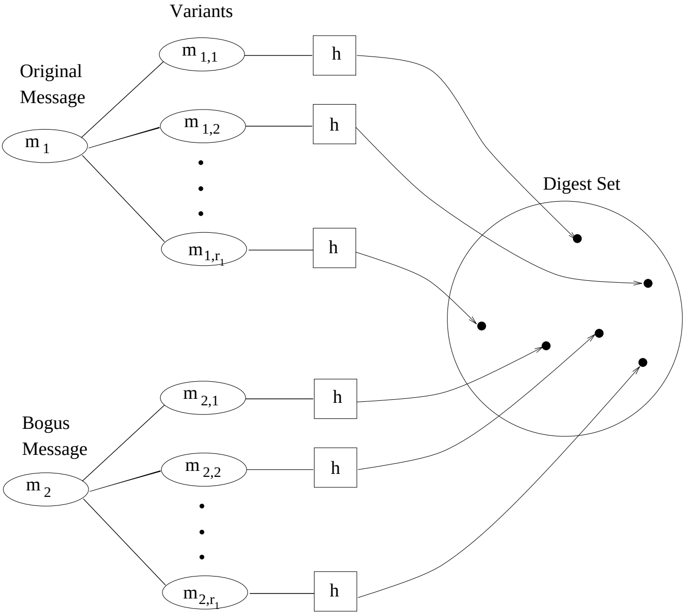
- The probability that the birthday of the first pupil falls on a specific day of the year : $\frac{1}{365}$.
- The probability that t students have different birthdays : $(1 - \frac{1}{365})(1 - \frac{2}{365}) \dots (1 - \frac{t-1}{365})$
- The probability that at least two of them have the same birthday is

$$P = 1 - \left(1 - \frac{1}{365}\right)\left(1 - \frac{2}{365}\right) \dots \left(1 - \frac{t-1}{365}\right)$$

- For $t \geq 23$, P is bigger than 0.5.

See [MVOV97], Section 2.1.5, "Birthday Problems"

Birthday Attack



Birthday Attack

- Assume that the number of bits in the digest is n.
- Assume that an adversary generates r_1 variants of an original message and r_2 variants of a bogus message.
- The probability of finding a pair of variants (one of the genuine and one of the bogus message) which hash to the same digest is

$$P \approx 1 - e^{-\frac{r_1 r_2}{2^n}}$$

  where $r_2 \gg 1$.
- When $r_1 = r_2 = 2^{\frac{n}{2}}$, the probability P is about 0.63.
- Any hashing algorithm which produces digests of the length around 64 bits is insecure as the time complexity function is $\approx 2^{32}$.

Birthday Attack

- Hash value should be longer than 256 bits to achieve a sufficient security against the birthday attack.
- Birthday attack is generic – works for any hash function:
  - for random hash functions (with digests of the length n), it requires

    $$O(2^{\frac{n}{2}})$$

    hash operations.
  - for non-random hash functions it may work even faster!

Attacks on Iterative Hash Functions

Length Extension

Given a message m split into blocks $m_1, m_2, \ldots, m_k$ which hashes to the value K, we can construct a message m' which splits into blocks $m_1, m_2, \ldots, m_k, m_{k+1}$.

Because the first k blocks of m' are identical to the k blocks of message m, the hash value $h(m)$ is merely the intermediate hash value after k blocks in the computation of $h(m')$. Iterative hash functions, such as MD5 and the SHA-1 and SHA-2 family of hashes, make the construction of m' slightly more difficult because of the padding and length field, but this is not a problem as the method of constructing these fields is known.

This problem arises because there is no special processing at the end of the hash function computation, so that $h(m)$ is produced from the intermediate state after k blocks of m'.

Partial Message Collision

Assume that a system authenticates a message m with $h(m||X)$ where X is the authentication key, and the attacker can get the system to do this *once only*.

In general, the best approach for an attacker is to choose an m, get the system to authenticate it as $h(m||X)$ and then do an exhaustive search for X.

But if h is an iterative hash function, the attacker can:

- Find two messages m and m' such that $h(m) = h(m')$
  - By using the birthday attack (below) in $2^{\frac{n}{2}}$ steps
- Get the system to authenticate m
- Replace the message with m'
  - Because hashing is iterative, once there is a collision, if subsequent hash inputs are the same, the hash will be the same

Notice - this attack works independent of the value of X.

Fixing Length Extension

Let h be any iterative hash function. Instead of $m \mapsto h(m)$ we could use $m \mapsto h(h(m)||m)$. This *double hashing* places $h(m)$ in front of the message being hashed, so that the second hash computation depends upon all the bits of the message, and no partial-message or length extension attacks can work.

If h is any of the SHA-2 family of hash functions, this construction has a security level of n bits, where n is the size of the hash output.

Disadvantages: performance, requires entire message to be buffered.

A More Efficient Fix

Instead of using $h(m)$ or $h(h(m)||m)$, we use

$$h(h(0^b)||m))$$

as the hash function, but claim a security level of only $n/2$ bits (i.e. $2^{n/2}$. (Hash functions are designed to be collision-resistent and so are suitably large).

Truncate the Output

If a hash function produces n bits of output, use only $n - s$ bits, for some positive s.

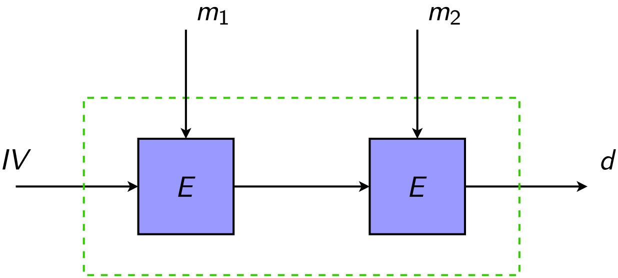
E.g. for 256-bit security, hash with SHA2-512, drop the last 256 bits of the output and return the remaining 256 bits as the hash value.

(In fact, that's what SHA2-224 and SHA2-384 effectively are: SHA2-256 with 32 bits of output dropped, and SHA2-512 with 128 bits of output dropped).

Other Attacks on Hash Functions

Meet-in-the-Middle Attack

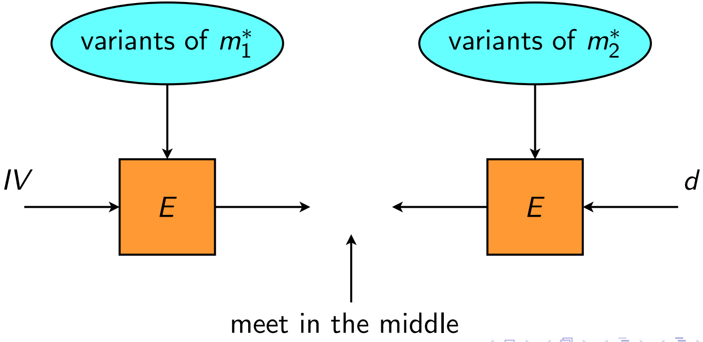
Consider the following hash function



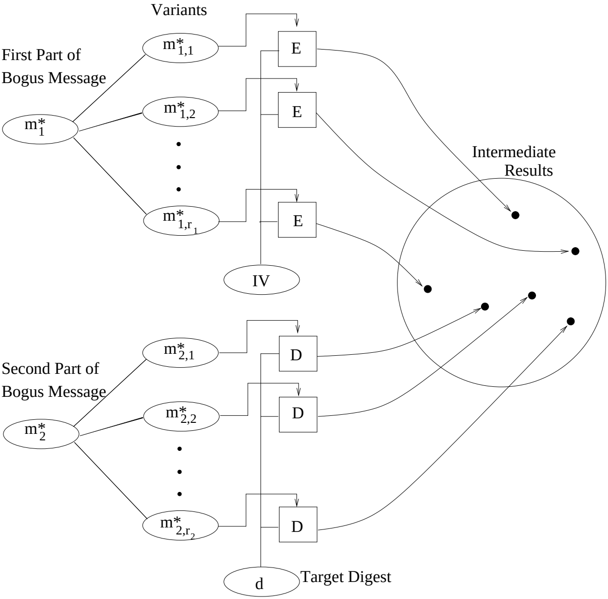
hash function h

Meet-in-the-Middle Attack

We can launch a variant of the birthday attack called meet-in-the-middle attack



Meet-in-the-Middle Attack



Meet-in-the-Middle Attack

- The attack can be launched against schemes which employ some sort of block chaining in their structure.
- In contrast to birthday attack, the meet-in-the-middle attack enables an attacker to construct a bogus message with a digest selected by the attacker.
- The attacker generates r_1 variants of the first part of a bogus message.
- He starts from the initial value and goes forward to the intermediate stage.
- He also generates r_2 variants on the second part of the bogus message.
- He starts from the desired target digest and goes backwards to the intermediate stage.
- The probability of a match in the intermediate stage is the

NIST Call for SHA-3

SHA-3 – Background

- Impact of Xiaoyun Wang attacks - On April 26, 2006, NIST comments:
  "NIST accepts that Prof. Wang has indeed found a practical collision attack on SHA-1."
- January 23, 2007 – NIST announces the Development of New Hash Algorithm(s), Secure Hash Standard
- November 2, 2007 – NIST calls for SHA-3 submissions
- October 31, 2008 – deadline for submissions
- July 24, 2009 – second round – 14 candidates
- December 2010 – third round – 5 finalists: BLAKE, Grostl, JH, KECCAK, Skein
- October 2012 – KECCAK is the winner !!!

details see
http://csrc.nist.gov/groups/ST/hash/index.html

SHA-3 – Requirements

New SHA-3 algorithm should support

- digital signature standard (DSS) – FIPS 186-2
- keyed-hash message authentication code (HMAC) – FIPS 198
- pair-wise key establishement – SP 800-56A
- random number generator – SP 800-90

Terms of reference

- SHA-3 algorithms should offer features that exceed the SHA-2 hash functions
- NIST expects SHA-3 to have a security strength that is at least as good as SHA-2
- NIST *strongly* desires a single hash algorithm family
- However, if more than one suitable candidate is identified, NIST may consider more than one family for inclusion

SHA-3 – Requirements - cont.

- algorithms with tunable security parameters
- NIST encourages constructions that differ from the MD model
- implementable in a wide range of hardware and software platforms
- capable to support digest sizes of 224, 256, 384 and 512 bits
- maximum message length at least $2^{64} - 1$ bits.

IP issues

- algorithm must be available worldwide and be royalty free

SHA-3 – Security

- if used as PRF, it must resist any distinguishing attack that requires fewer than $2^{n/2}$ queries
- collision resistance of $2^{n/2}$
- preimage resistance of 2^n
- second preimage resistance of 2^{n-k} for any message shorter than 2^k bits
- resistance to length-extension attacks
- resistance against other attacks such as multicollision attacks, is encouraged by NIST

SHA-3 – Evaluation Process

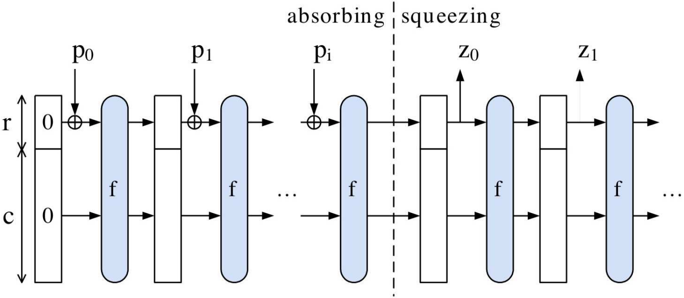
- evaluation process is public
- after the deadline October 31, 2008, NIST will review the submission and organize a public conference
- Round 1 – 12 month review to cull weaker submissions (no modifications). Two workshops at the very begining:
  - First SHA-3 Candidate Conference - Feb 25-28, 2009
- Round 2 – 12-15 months careful evaluation (modification allowed). At the end the third workshop
  - Second SHA-3 Candidate Conference - August 23-24, 2010
- Announcement of five finalists - December 10, 2010
- Announcement of SHA-3 - October 2, 2012

The SHA-3 Five Finalists

- BLAKE (Jean-Phillipe Aumasson, Luca Henzen, Willi Meier, Raphael C-W Phan)
- Grøstl (Praveen Gauravaran, Lars Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, Søren S. Thomsen)
- JH (Hongjun Wu)
- Keccak (Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles van Assche)
- Skein (Bruce Schneier, Stefan Lucks, Niels Ferguson Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, Jesse Walker)

Keccak

Keccak uses *sponge construction*. The *state* is a 5 x 5 x 64-bit array, of which r bits are used to absorb input and the *capacity*, $c = 25w - r$ bits (where $w = 64$ bits), is the number of state bits that are not involved in input/output.



Keccak Block Permutation f

The permutation consists of $12 + log_2 w$ iterations of five subrounds

- θ - Computation of parity of each 5-bit column and XOR'ing into two nearby columns
- ρ - Bitwise rotation of each of the 25 words by a different triangular number $(0, 1, 3, 6, 10, 15, \ldots)$
- π - Permute the 25 words in a fixed pattern
- X - Bitwise combination along rows, using $a = a \oplus (\neg b \wedge c)$.
- I - In round n, for $0 \leq m \leq log_2 w$, XOR $a[0][0][2^m - 1]$ with bit $m + 7n$ of a degree 8 LFSR sequence, to break symmetry.

Entity Authentication

Passphrase Authentication

- Store a hash of the passphrase the user provides
- To authenticate, hash the supplied passphrase and compare with the stored hash
  - If equal, the passphrase is correct and the user is authenticated
  - If not equal, display a generic message
    - Do not reveal whether it is the username or the passphrase that failed to match
- Do not use symmetric crypto to store an encrypted passphrase!
  - This requires a key, which must be stored somewhere and loaded into your code
  - If an attacker gets that key, they instantly have all passphrases!

Passphrases

- Default on many systems
- Conceptually simple, basically understood
- Problems:
  - People choose weak, guessable, passphrases
  - People reuse passphrases across multiple systems and web sites
  - People share passphrases
  - People write them down
    - Security audits should check for post-it notes inside unlocked drawers, passwords taped underneath keyboards, etc.
  - Shoulder surfing

Passphrase Rules

So we make rules ...

The longest password ever

We laugh -- **but** her I. D. is safe.

During a recent password audit by a company, it was found that an employee was using the following password:

"MickeyMinniePlutoHueyLouieDeweyDonaldGoofySacramento"

When asked why she had such a long password, she rolled her eyes and said: "Hello! It has to be at least 8 characters and include at least one capital."

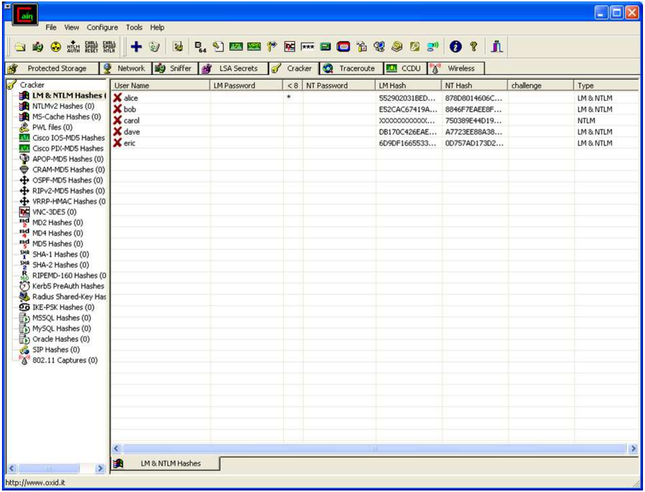
Attacks on Passphrases

- Brute force
  - Online attack - attempt login repeatedly until success
- Password sniffing
  - The password is passed over the network connection, e.g. as part of an HTTP request header
  - So protect it with encryption
- Attacks on the hashes
  - If the attacker can get the hashes out of the database (perhaps by an SQL injection attack)
  - Dictionary attack
    - Take each word (phrase) in turn, calculate its hash and compare against the captured hashes
  - Rainbow tables attacks
    - A precomputed table of hash values which allows instant lookup and return of a corresponding passphrase
    - Use salt to defeat this

Dictionary Attacks

- Hash each word in a dictionary, one by one, comparing against the hashed values
- When you find a match, you know the passphrase
- Dictionary attack programs use
  - Dictionary words, forwards and backwards
  - Custom and domain-specific dictionaries
  - Dictionary words with letter/digit transpositions and substitutions
  - Non-dictionary letter/digit sequences: aaa, aab, aac, etc

Cain+Abel



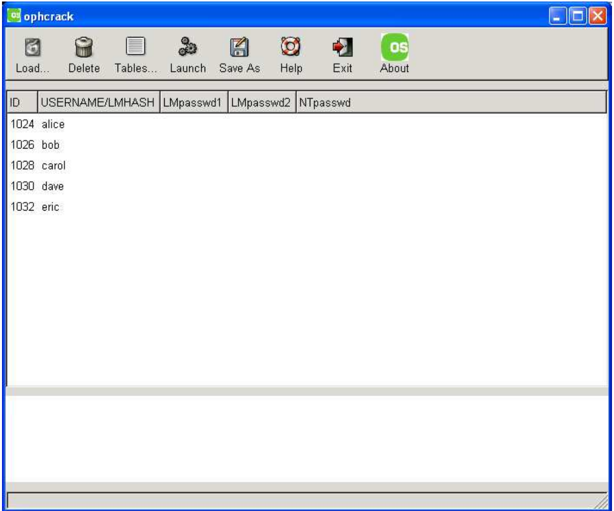
- Cracker
  - LM & NTLM Hashes (
  - NTLMv2 Hashes (0)
  - MS-Cache Hashes (0)
  - PWL files (0)
  - Cisco IOS-MD5 Hashes
  - Cisco PIX-MD5 Hashes
  - APOP-MD5 Hashes (0)
  - CRAM-MD5 Hashes (0)
  - OSPF-MD5 Hashes (0)
  - RIPv2-MD5 Hashes (0)
  - VRRP-HMAC Hashes (0
  - VNC-3DES (0)
  - MD2 Hashes (0)
  - MD4 Hashes (0)
  - MD5 Hashes (0)
  - SHA-1 Hashes (0)
  - SHA-2 Hashes (0)
  - RIPEMD-160 Hashes (0
  - Kerb5 PreAuth Hashes
  - Radius Shared-Key Has
  - IKE-PSK Hashes (0)
  - MSSQL Hashes (0)
  - MySQL Hashes (0)
  - Oracle Hashes (0)
  - SIP Hashes (0)
  - 802.11 Captures (0)

User Name	LM Password	< 8	NT Password	LM Hash	NT Hash	challenge	Type
✗ alice		*		552902031BED...	878D8014606C...		LM & NTLM
✗ bob				E52CAC67419A...	8846F7EAEE8F...		LM & NTLM
✗ carol				XXXXXXXXXXXX...	750389E44D19...		NTLM
✗ dave				DB170C426EAE...	A7723EE88A38...		LM & NTLM
✗ eric				6D9DF1665533...	0D757AD173D2...		LM & NTLM

Rainbow Tables

- Developed by Philippe Oechslin, [Oec03]
- Large tables of partially pre-computed hashes
    - Typical set of tables for upper/lower case, digits, punctuation symbols is 18GB in size
    - More comprehensive tables are much larger
- Use captured hash value as a key into the database
    - Lookup is near-instant
    - Or submit the hash to a web site, matching passphrase is returned by email
- Buy tables on DVDs or hard drives (at conferences) or download in exchange for running a distributed table-generation client

ophcrack



ID	USERNAME/LMHASH	LMpasswd1	LMpasswd2	NTpasswd
1024	alice			
1026	bob			
1028	carol			
1030	dave			
1032	eric			

Passphrase Storage

- Passphrase files should not be world-readable
  - E.g. use shadow password files, not `/etc/passwd`
- Password files should never contain plaintext passwords
- Store a salted hash - SHA2-256 or better
- Salt: some random characters prefixed to the password before hashing
  - Original UNIX salt: 2 6-bit characters
  - Use the user ID, email address, system name or domain name
  - Better: salt with a random string, then store with the hash for use in comparison hash calculation

PHP Password Hashing Functions

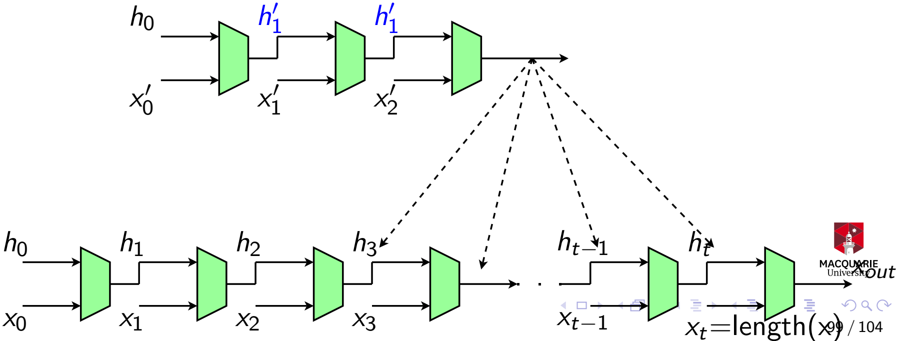
- `string password_hash(string $password, integer $algo [, array $options]`
  - Hashes a password, by default using the bcrypt algorithm and a random salt
- `boolean password_verify(string $password, string $hash)`
  - Verifies a password against a hash created using `password_hash()`

MACQUARIE
University

Advanced Attacks (Optional Material)

Generic Attacks on Merkle-Damgård Hash Functions

On Second Preimage - after hashing 2^t message blocks, the complexity of attack is $t2^{n/2+1} + 2^{n-t+1}$ compression function evaluations



Generic Attacks on Merkle-Damgård Hash Functions

The attack proceeds according to the following steps:

- find fixed points of the compression function such that

  $$H(x_1', h_1') = h_1'$$

  The fixed point is used to expand the message to the correct length x_t. This takes $2^{\frac{n}{2}+1}$ compression function evaluations
- determine a message block that links to one of the intermediate hash functions.

The colliding messages are

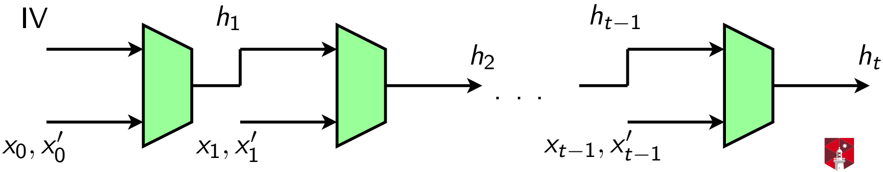
$$H(x_1', x_2', x_2', \cdots, x_3', \cdots, x_t) = H(x_1, x_2, x_3, \cdots, x_t)$$

Note that the message has the correct message length.

Multicollisions

Assume we have t collisions:

$$
\begin{aligned}
H(IV, x_0) &= H(IV, x_0') \\
H(h_1, x_1) &= H(h_1, x_1') \\
&\vdots \\
H(h_{t-1}, x_{t-1}) &= H(h_{t-1}, x_{t-1}')
\end{aligned}
$$



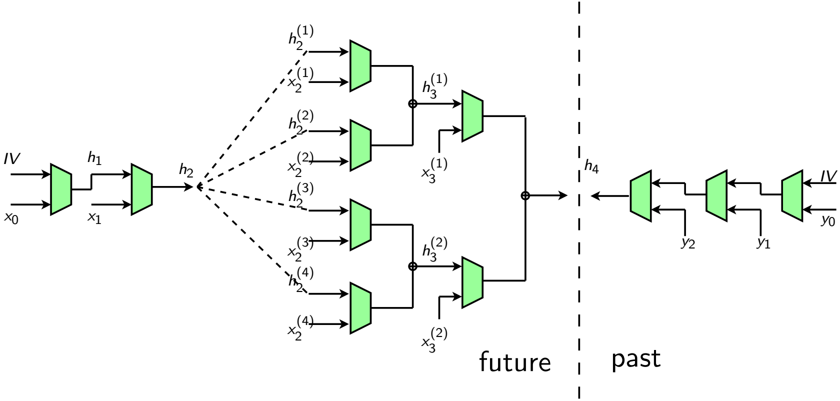
then we can create 2^t multicollisions, i.e.

Herding (Nostradamus) Attack

Given a commitment

$$h = H(IV \parallel y_0 \parallel y_1 \parallel y_2)$$

which predicts closing prices of the stock market on a given date:



Herding Attack

Steps in the attack:

- Create a collision tree of 2^t values costs $\approx 2^{t/2}2^{n/2}$ compression hash evaluations.
- Finding the match for h_2 takes 2^{n-t} compression hash evaluations.

Note that all messages on the left-hand side collide with the message $(IV \parallel y_0 \parallel y_1 \parallel y_2)$ or

$$H(IV \parallel x_0 \parallel x_1 \parallel x_2^{(i)} \parallel x_3^{(j)}) = H(IV \parallel y_0 \parallel y_1 \parallel y_2)$$

for all i, j defined in the figure.

References

A. J Menezes, Paul C Van Oorschot, and Scott A Vanstone.
Handbook of applied cryptography.
CRC Press, Boca Raton, 1997.

Philippe Oechslin.
Making a Faster Cryptanalytic Time-Memory Trade-Off.
In *Advances in Cryptology - CRYPTO 2003*, pages 617–630.
Springer, Berlin, Heidelberg, August 2003.

Phillip Rogaway and Thomas Shrimpton.
Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance.
In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, number 3017 in Lecture Notes in Computer Science, pages 371–388.
Springer Berlin Heidelberg, January 2004.