Hash Functions

SECURE COMMUNICATIONS & CRYPTOGRAPHY

Mark Cummins, Institute of Technology Blanchardstown

Hash Functions

- Hash functions take a message as input and produce an output referred to as a hash-code, hash-result, hash-value or simply hash.
- While related to conventional hash functions, Cryptographic hash functions differ in several important aspects.
- Cryptographic hash functions play a fundamental role in modern cryptography.

Hash Functions

- The basic idea is that a hash value serves as a compact representative image of an input string, sometimes called an imprint, digital fingerprint or message digest.

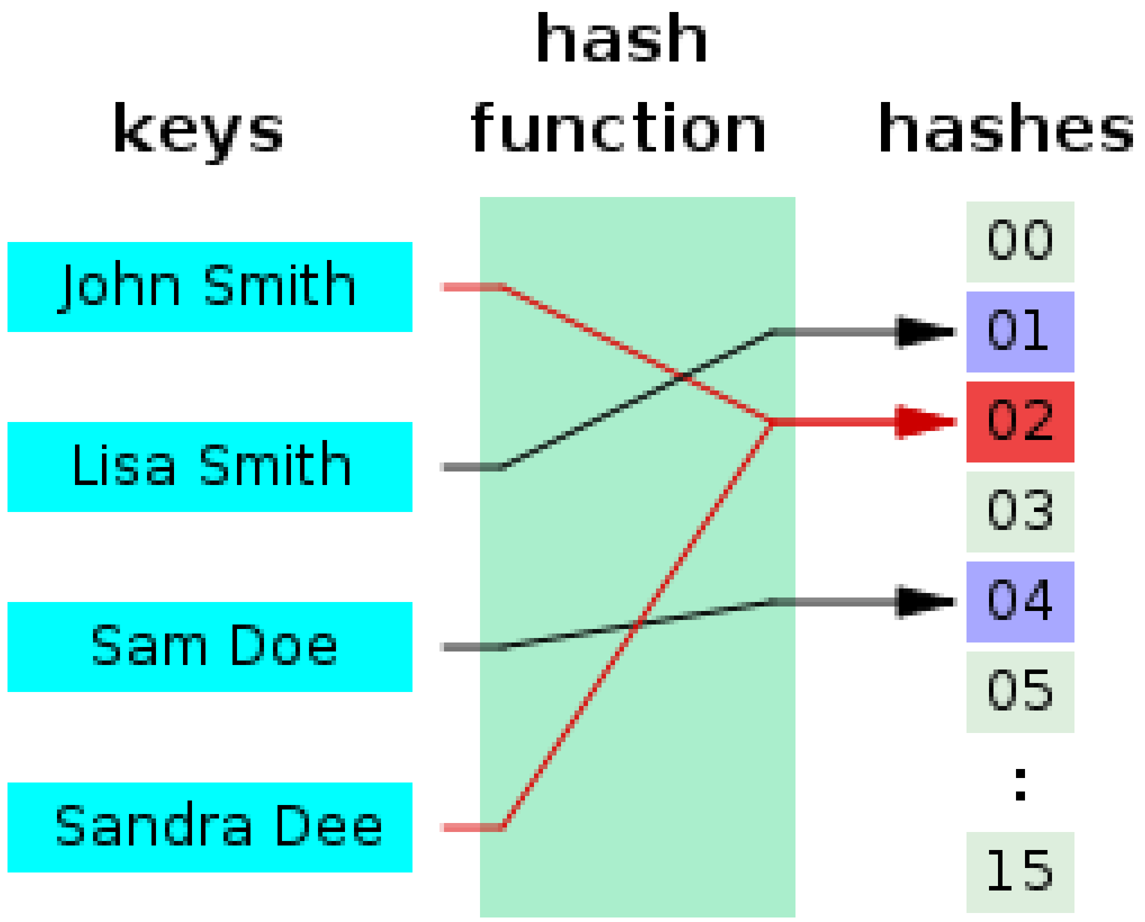
- A **hash function** is any well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into one of a finite number of fixed length hash results.

Hash Functions

- We are mapping a much larger number of possible messages to a hash value from a set of values with a much smaller range.

- This implies that the existence of **collisions** (Pairs of inputs with identical outputs is unavoidable)

Hash Functions



A hash function that maps names to integers from 0 to 15. Note the collision between keys "John Smith" and "Sandra Dee".

Trivial Hash Function

- If the data to be hashed is small enough, one can use the data itself (reinterpreted as an integer in binary notation) as the hashed value. The cost of computing this "trivial" hash function is effectively zero.

- The same technique can be used to map two-letter country codes like 'us' or 'za' to country names (26^2=676 table entries), 5-digit zip codes like 13083 to city names (100000 entries), etc.

Perfect Hashing

- A hash function that is injective — that is, maps each valid input to a different hash value — is said to be **perfect.**
- With such a function one can directly locate the desired entry in a hash table, without any additional searching.
- Unfortunately, perfect hash functions are effective only in situations where the inputs are fixed and entirely known in advance, such as mapping month names to the integers 0 to 11, or words to the entries of a dictionary.

Minimal Perfect Hashing

- A perfect hash function for *n* keys is said to be **minimal** if its range consists of *n consecutive* integers, usually from 0 to $n-1$.

- Besides providing single-step lookup, a minimal perfect hash function also yields a compact hash table, without any vacant slots. Minimal perfect hash functions are much harder to find than perfect ones with a wider range.

Hash Functions

- **Non-cryptographic hash functions**

Pearson hashing	8 bits
Fowler-Noll-Vo hash function	(32, 64, 128, 256,512, or 1024 bits)
MurmurHash	32 or 64 bits
Zobrist hashing	variable
Java hashCode()	32 bits
Bernstein hash	32 bits

Hash Functions

- Hash functions are related to (and often confused with)
  - checksums,
  - check digits,
  - fingerprints,
  - randomization functions,
  - error correcting codes,
  - and cryptographic hash functions.

- Although these concepts overlap to some extent, each has its own uses and requirements and is designed and optimised differently

Hash Functions

- A **cyclic redundancy check** (CRC) is a non-secure function designed to detect accidental changes to raw computer data, and is commonly used in digital networks and storage devices such as hard disk drives.

- A CRC-enabled device calculates a short, fixed-length binary sequence, known as the *CRC code* or just *CRC*, for each block of data and sends or stores them both together. When a block is read or received the device repeats the calculation; if the new CRC does not match the one calculated earlier, then the block contains a *data error* and the device may take corrective action such as rereading or requesting the block be sent again.

Hash Functions

- **Cyclic redundancy checks**

BSD checksum	16 bits
checksum	32 bits
crc16	16 bits
crc32	32 bits
crc32 mpeg2	32 bits
crc64	64 bits
SYSV checksum	16 bits

  - Adler-32 is often classified as a CRC, but it uses a different algorithm.

Hash Functions

- The goal of **checksum** algorithms is to detect *accidental* modification such as corruption to stored data or errors in a communication channel.
- They are not designed to detect intentional corruption by a malicious agent.
- Indeed, many checksum algorithms can be easily inverted, in the sense that one can easily modify the data so as to preserve its checksum.

Hash Functions

- **Checksums**

sum8	8 bits
sum16	16 bits
sum24	24 bits
sum32	32 bits
fletcher-4	4 bits
fletcher-8	8 bits
fletcher-16	16 bits
Adler-32	32 bits
xor8	8 bits
Luhn algorithm	4 bits
Verhoeff algorithm	4 bits

Cryptographic Hash Functions

- The ideal cryptographic hash function has four main properties:
  - it is easy to compute the hash value for any given message,
  - it is infeasible to find a message that has a given hash,
  - it is infeasible to modify a message without changing its hash,
  - it is infeasible to find two different messages with the same hash.

Cryptographic Hash Functions

- These properties imply that a malicious adversary can not replace or modify the input data without changing its digest.

- Thus, if two strings have the same digest, one can be very confident that they are identical.

Cryptographic Hash Functions

- Checksum algorithms, such as CRC32 and other cyclic redundancy checks, are designed to meet much weaker requirements, and are generally unsuitable as cryptographic hash functions.

- For example, a CRC was used for message integrity in the WEP encryption standard, but an attack was readily discovered which exploited the linearity of the checksum.

Avalanche Effect

- The **avalanche effect** refers to a desirable property of cryptographic algorithms, typically block ciphers and cryptographic hash functions.
- The avalanche effect is evident if, when an input is changed slightly (for example, flipping a single bit) the output changes significantly (e.g., half the output bits flip).
- In the case of quality block ciphers, such a small change in either the key or the plaintext should cause a drastic change in the ciphertext.

Avalanche Effect

- The SHA1 hash function exhibits good avalanche effect. When a single bit is changed the hash sum becomes completely different.

- If a block cipher or cryptographic hash function does not exhibit the avalanche effect to a significant degree, then it has poor randomization, and thus a cryptanalyst can make predictions about the input, being given only the output.

- The **strict avalanche criterion** (**SAC**) is a generalization of the avalanche effect. It is satisfied if, whenever a single input bit is complemented, each of the output bits changes with a probability of one half.

Cryptographic Hash Functions

- **Cryptographic hash functions**

elf64	64 bits
HAVAL	128 to 256 bits
MD2	128 bits
MD4	128 bits
MD5	128 bits
Radio Gatún	Arbitrarily long
RIPEMD-64	64 bits
RIPEMD-160	160 bits
RIPEMD-320	320 bits

Cryptographic Hash Functions

- **Cryptographic hash functions**

SHA1	160 bits
SHA256	256 bits
SHA384	384 bits
SHA512	512 bits
Skein	Arbitrarily long
Tiger	192 bits
Whirlpool	512 bits

Cryptographic Hash Functions

- A cryptographic hash function must be able to withstand all known types of cryptanalytic attack.

- As a minimum, it must have the following properties:
  - *Preimage resistance*
  - *Second preimage resistance*
  - *Collision resistance*

Preimage Resistance

- Given a hash *h* it should be hard to find any message *m* such that hash(*m*) = h.
- This concept is related to that of **one way function**.
- Functions that lack this property are vulnerable to **preimage attacks**.

Preimage Resistance

One Way Function

- A **one-way function** is a function that is easy to compute on every input, but hard to invert given the image of a random input.
- The existence of such one-way functions is still an open conjecture.
- In applied contexts, the terms "easy" and "hard" are usually interpreted relative to some specific computing entity; typically "cheap enough for the legitimate users" and "prohibitively expensive for any malicious agents".

Preimage Resistance

One Way Trapdoor Function

- A **trapdoor function** is a function that is easy to compute in one direction, yet believed to be difficult to compute in the opposite direction without special information, called the "trapdoor".
- Trapdoor functions are widely used in cryptography.
- RSA is a well known example of a function believed to belong to this class.

Preimage Resistance

- Given a hash *h* it should be hard to find any message *m* such that hash(*m*) = h.

- This concept is related to that of one way function.

- Functions that lack this property are vulnerable to preimage attacks.

Second Preimage Resistance

- Given an input m_1, it should be hard to find another input, m_2 (not equal to m_1) such that hash(m_1) = hash(m_2).
- This property is sometimes referred to as ***weak collision resistance***.
- Functions that lack this property are vulnerable to second preimage attacks.

Preimage Attack

- A **preimage attack** on a cryptographic hash is an attempt to find a message that has a specific hash value.

- There are two types of preimage attacks:
  - *First preimage attack*: given a hash *h*, find a message *m* such that *hash(m) = h*.
  - *Second preimage attack*: given a fixed message *m1*, find a different message *m2* such that *hash(m2) = hash(m1)*.

Preimage Attack

- For an n-bit ideal hash function, finding a first preimage or a second preimage has complexity 2^n, which is considered too high for a typical output size of n=160 bits.
- If such complexity is the best that can be achieved by an adversary, then the hash function is considered first and second preimage resistant.

Preimage Attack

- Some significant preimaging attacks have already been discovered, but they are not yet practical.

- If a practical preimaging attack is discovered, it would drastically affect many Internet protocols. In this case, "practical" means that it could be executed by an attacker in a meaningful amount of time for a meaningful amount of money.

- All currently known practical or almost-practical attacks on MD5 and SHA-1 are collision attacks.

Collision Resistance

- It should be <u>hard</u> to find two different messages m_1 and m_2 such that hash(m_1) = hash(m_2).
- Such a pair is called a (cryptographic) hash collision, and this property is sometimes referred to as ***strong collision resistance***.
- It requires a hash value at least twice as long as what is required for preimage-resistance, otherwise collisions may be found by a birthday attack.

Collision Attack / Birthday Attack

- A **birthday attack** is a type of cryptographic attack, so named because it exploits the mathematics behind the birthday problem in probability theory.

- Given a function f, the goal of the attack is to find two different inputs x_1, x_2 such that $f(x_1) = f(x_2)$.

- Such a pair x_1, x_2 is called a collision.

Collision Attack / Birthday Attack

- The method used to find a collision is to simply evaluate the function f for different input values that may be chosen randomly or pseudorandomly until the same result is found more than once.
- Because of the aforementioned birthday problem, this method can be rather efficient.

Birthday Problem

- In probability theory, the **birthday problem**, or **birthday paradox** pertains to the probability that in a set of randomly chosen people some pair of them will have the same birthday.

- In a group of at least 23 randomly chosen people, there is more than 50% probability that some pair of them will have the same birthday.

- For 57 or more people, the probability is more than 99%, and it reaches 100% when the number of people reaches 366.

Birthday Problem

10	11.7%
20	41.1%
23	50.7%
30	70.6%
50	97.0%
57	99.0%
100	99.99997%
200	99.999999999999999999999999998%

Rainbow Tables

- A **rainbow table** is a lookup table offering a time-memory trade-off used in recovering the plaintext password from a password hash generated by a hash function, often a cryptographic hash function.
- A common application is to make attacks against hashed passwords feasible.
- A salt is often employed with hashed passwords to make this attack more difficult, often infeasible.

Message-Digest algorithm 5 (MD5)

Designer:	Ron Rivest
First published :	April 1992
Series:	MD, MD2, MD3, MD4, MD5, MD6
Digest sizes:	128 bits
Rounds:	4

MD5 Algorithm

- MD5 processes a variable-length message into a fixed-length output of 128 bits.
- The input message is broken up into chunks of 512-bit blocks (sixteen 32-bit integers);
- The message is padded so that its length is divisible by 512.

MD5 Algorithm

- The padding works as follows:
  - First a single bit, 1, is appended to the end of the message.
  - This is followed by as many zeros as are required to bring the length of the message up to 64 bits fewer than a multiple of 512.
  - The remaining bits are filled up with a 64-bit integer representing the length of the original message, in bits.

MD5 Algorithm

- The main MD5 algorithm operates on a 128-bit state, divided into four 32-bit words, denoted *A*, *B*, *C* and *D*.
- These are initialized to certain fixed constants.
- The main algorithm then operates on each 512-bit message block in turn, each block modifying the state.

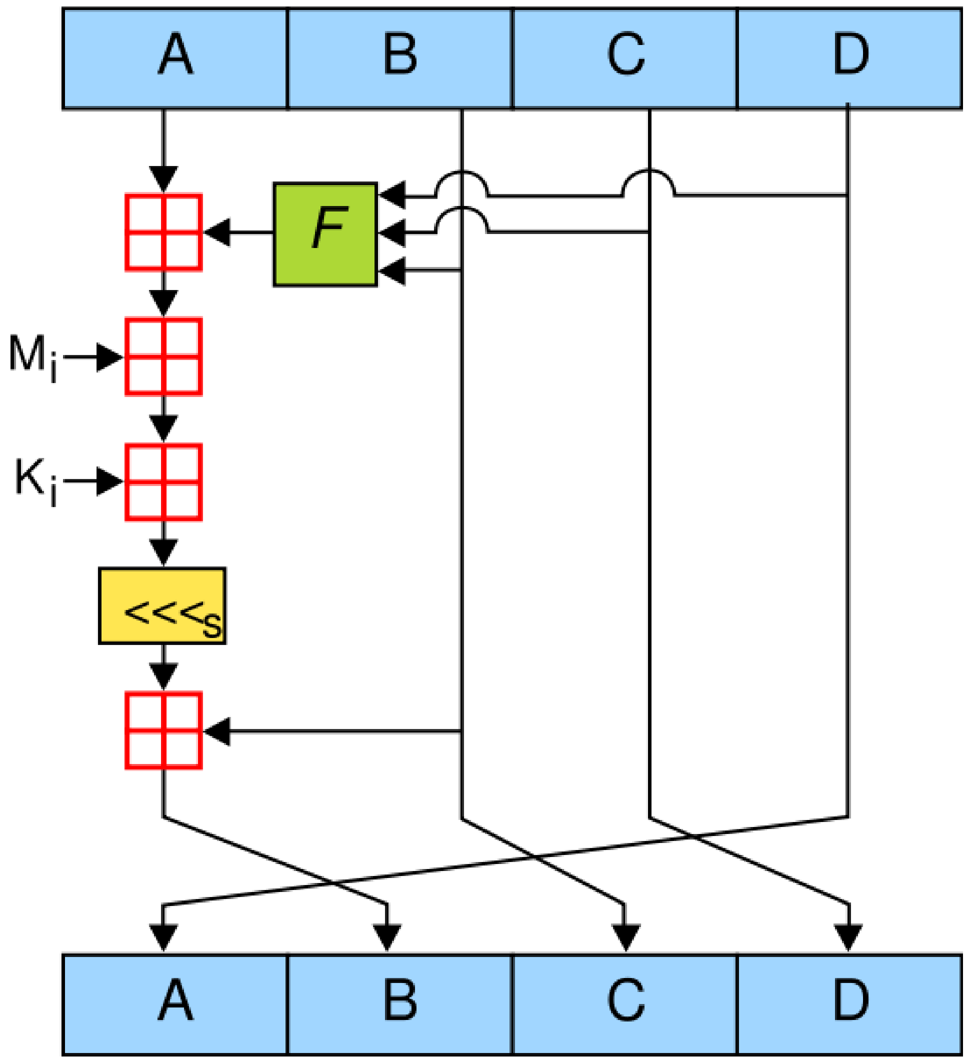
MD5 Algorithm

- The processing of a message block consists of four similar stages, termed *rounds*;

- Each round is composed of 16 similar operations based on a non-linear function *F*.

- There are four possible functions *F*; a different one is used in each round:

MD5 Algorithm



One MD5 operation.

MD5 consists of 64 of these operations, grouped in four rounds of 16 operations.

One function is used in each round.

M_i denotes a 32-bit block of the message input, and K_i denotes a 32-bit constant, different for each operation.

MD5 Algorithm

- The 4 state variables, each of which is a 32 bit integer are initialized as follows:

  - A = 0x67452301
    B = 0xEFCDAB89
    C = 0x98BADCFE
    D = 0x10325476

MD5 Algorithm

- The main part of the algorithm uses four functions to thoroughly mix the above state variables.
- Those functions are as follows:

  F(X,Y,Z) = (X & Y) | (~(X) & Z)
  G(X,Y,Z) = (X & Z) | (Y & ~(Z))
  H(X,Y,Z) = X ^ Y ^ Z
  I(X,Y,Z) = Y ^ (X | ~(Z))

- Where &, |, ^, and ~ are the bit-wise AND, OR, XOR, and NOT operators respectively

MD5 Algorithm

- These functions, using the state variables and the message as input, are used to transform the state variables from their initial state into what will become the message digest.

MD5 Algorithm

- After completion, the message digest is stored in the state variables (A, B, C, and D).

- To get it into the hexadecimal form you are used to seeing, output the hex values of each the state variables, least significant byte first. For example, if after the digest:

  A = 0x01234567;
  B = 0x89ABCDEF;
  C = 0x1337D00D
  D = 0xA5510101

- Then the message digest would be:
  67452301EFCDAB890DD03713010151A5

MD5 pseudo-code

```
// Group the 512 bit message into 16 different 32 bit chunks
For j = 0 to 15 do
  Set X[j] to MessageBits[j*32] through
  MessageBits[j*32 + 31]
end


// Store the digest variables out of harms way
  AA = A
  BB = B
  CC = C
  DD = D
```

MD5 pseudo-code

```
/* Round 1. */
// ---------------------------------------------------------
// Let [abcd k s i] denote the operation:
// a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s).
// ---------------------------------------------------------
// Do the following 16 operations.

   [ABCD 0 7 1] [DABC 1 12 2] [CDAB 2 17 3] [BCDA 3 22 4]
   [ABCD 4 7 5] [DABC 5 12 6] [CDAB 6 17 7] [BCDA 7 22 8]
   [ABCD 8 7 9] [DABC 9 12 10] [CDAB 10 17 11] [BCDA 11 22 12]
   [ABCD 12 7 13] [DABC 13 12 14] [CDAB 14 17 15] [BCDA 15 22 16]
```

MD5 pseudo-code

```
/* Round 2. */
// -------------------------------------------
// Let [abcd k s i] denote the operation:
// a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s).
// -------------------------------------------
// Do the following 16 operations.

    [ABCD 1 5 17] [DABC 6 9 18] [CDAB 11 14 19] [BCDA 0 20 20]
    [ABCD 5 5 21] [DABC 10 9 22] [CDAB 15 14 23] [BCDA 4 20 24]
    [ABCD 9 5 25] [DABC 14 9 26] [CDAB 3 14 27] [BCDA 8 20 28]
    [ABCD 13 5 29] [DABC 2 9 30] [CDAB 7 14 31] [BCDA 12 20 32]
```

MD5 pseudo-code

```
/* Round 3. */
// ------------------------------------------------------
// Let [abcd k s i] denote the operation:
// a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s).
// ------------------------------------------------------
// Do the following 16 operations.

   [ABCD 5 4 33] [DABC 8 11 34] [CDAB 11 16 35] [BCDA 14 23 36]
   [ABCD 1 4 37] [DABC 4 11 38] [CDAB 7 16 39] [BCDA 10 23 40]
   [ABCD 13 4 41] [DABC 0 11 42] [CDAB 3 16 43] [BCDA 6 23 44]
   [ABCD 9 4 45] [DABC 12 11 46] [CDAB 15 16 47] [BCDA 2 23 48]
```

MD5 pseudo-code

```
/* Round 4. */
// ---------------------------------------------------
// Let [abcd k s i] denote the operation:
// a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s).
// ---------------------------------------------------
// Do the following 16 operations.

   [ABCD 0 6 49] [DABC 7 10 50] [CDAB 14 15 51] [BCDA 5 21 52]
   [ABCD 12 6 53] [DABC 3 10 54] [CDAB 10 15 55] [BCDA 1 21 56]
   [ABCD 8 6 57] [DABC 15 10 58] [CDAB 6 15 59] [BCDA 13 21 60]
   [ABCD 4 6 61] [DABC 11 10 62] [CDAB 2 15 63] [BCDA 9 21 64]
```

MD5 pseudo-code

```
/* And finally update the state variables */

  A+=AA
  B+=BB
  C+=CC
  D+=DD
```

If you are confused on what '<<<' does, let it be known that it represents a bit-wise rotation to the left, e.g. 110011 <<< 2 = 001111

MD5 Timeline

- When analytic work indicated that MD4 was likely to be insecure, MD5 was designed in 1991 to be a secure replacement.
- Weaknesses were indeed later found in MD4 by Hans Dobbertin
- In 1993, Den Boer and Bosselaers gave an early, although limited, result of finding a "pseudo-collision" of the MD5 compression function; that is, two different initialization vectors which produce an identical digest.
- i

MD5 Timeline

- In 1996, Dobbertin announced a collision of the compression function of MD5.

- While this was not an attack on the full MD5 hash function, it was close enough for cryptographers to recommend switching to a replacement, such as SHA-1 or RIPEMD-160.

MD5 Timeline

- The size of the hash—128 bits—is small enough to contemplate a birthday attack.
- MD5CRK was a distributed project started in March 2004 with the aim of demonstrating that MD5 is practically insecure by finding a collision using a birthday attack.
- MD5CRK ended shortly after 17 August, 2004, when collisions for the full MD5 were announced by Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu.

MD5 Timeline

- On 1 March 2005, Arjen Lenstra, Xiaoyun Wang, and Benne de Weger demonstrated construction of two X.509 certificates with different public keys and the same MD5 hash, a demonstrably practical collision.

- The construction included private keys for both public keys. A few days later, Vlastimil Klima described an improved algorithm, able to construct MD5 collisions in a few hours on a single notebook computer.

- On 18 March 2006, Klima published an algorithm that can find a collision within one minute on a single notebook computer, using a method he calls tunneling.

MD5 Timeline

- On December 30, 2008, a group of researchers announced at the 25th Chaos Communication Congress how they had used MD5 collisions to create an intermediate certificate authority certificate which appeared to be legitimate when checked via its MD5 hash.

- The researchers used a cluster of Sony Playstation 3s at the EPFL in Lausanne, Switzerland to change a normal SSL certificate issued by RapidSSL into a working CA certificate for that issuer, which could then be used to create other certificates that would appear to be legitimate and issued by RapidSSL.

MD5 Timeline

- VeriSign, the issuers of RapidSSL certificates, said they stopped issuing new certificates using MD5 as their checksum algorithm for RapidSSL once the vulnerability was announced.
- Although Verisign declined to revoke existing certificates signed using MD5
- Bruce Schneier wrote of the attack that "[w]e already knew that MD5 is a broken hash function" and that "no one should be using MD5 anymore."
- The SSL researchers wrote, "Our desired impact is that Certification Authorities will stop using MD5 in issuing new certificates. We also hope that use of MD5 in other applications will be reconsidered as well.“

MD5 Timeline

- US-CERT of the US Department of Homeland Security said in 2008 that MD5 "should be considered cryptographically broken and unsuitable for further use,“ and most U.S. government applications will be required to move to the SHA-2 family of hash functions by 2010.

MD5 Rainbow Tables

- Recently, a number of projects have created MD5 rainbow tables which are easily accessible online, and can be used to reverse many MD5 hashes into strings that collide with the original input, usually for the purposes of password cracking.
- However, if passwords are combined with a salt before the MD5 digest is generated, rainbow tables become much less useful.
- The use of MD5 in some websites' URLs means that Google can also sometimes function as a limited tool for reverse lookup of MD5 hashes.
- This technique is also rendered ineffective by the use of a salt.

Secure Hash Algorithm (SHA)

- The **SHA hash functions** are a set of cryptographic hash functions designed by the National Security Agency (NSA) and published by the NIST as a U.S. Federal Information Processing Standard.
- SHA stands for **Secure Hash Algorithm**. The three SHA algorithms are structured differently and are distinguished as *SHA-0*, *SHA-1*, and *SHA-2*.
- The *SHA-2* family uses an identical algorithm with a variable digest size which is distinguished as *SHA-224*, *SHA-256*, *SHA-384*, and *SHA-512*.

Secure Hash Algorithm (SHA)

- SHA-1 is the best established of the existing SHA hash functions, and is employed in several widely used security applications and protocols.
- In 2005, security flaws were identified in SHA-1, namely that a possible mathematical weakness might exist, indicating that a stronger hash function would be desirable.
- Although no attacks have yet been reported on the SHA-2 variants, they are algorithmically similar to SHA-1 and so efforts are underway to develop improved alternatives.
- A new hash standard, SHA-3, is currently under development, the function will be selected via an open competition running between 2008 and 2012.

SHA-0 and SHA-1

- The original specification of the algorithm was published in 1993 as the *Secure Hash Standard*, FIPS PUB 180.
- This version is now often referred to as *SHA-0*. It was withdrawn by NSA shortly after publication and was superseded by the revised version, published in 1995 in FIPS PUB 180-1 and commonly referred to as *SHA-1*.

SHA-0 and SHA-1

- SHA-1 differs from SHA-0 only by a single bitwise rotation; this was done, according to NSA, to correct a flaw in the original algorithm which reduced its cryptographic security.

- However, NSA did not provide any further explanation or identify the flaw that was corrected.

- Weaknesses have subsequently been reported in both SHA-0 and SHA-1. SHA-1 appears to provide greater resistance to attacks, supporting the NSA's assertion that the change increased the security.

SHA-0 and SHA-1

- SHA-1 (as well as SHA-0) produces a 160-bit digest from a message with a maximum length of $(2^{64} - 1)$ bits.

- SHA-1 is based on principles similar to those used by Ronald L. Rivest of MIT in the design of the MD4 and MD5 message digest algorithms, but has a more conservative design.

SHA-2 Family

- NIST published four additional hash functions in the SHA family, named after their digest lengths (in bits): SHA-224, SHA-256, SHA-384, and SHA-512.
- The algorithms are collectively known as SHA-2.
- The algorithms were first published in 2001as a draft and ater review and comment were released as an official standard in 2002.
- SHA-256 and SHA-512 are novel hash functions computed with 32- and 64-bit words, respectively. They use different shift amounts and additive constants, but their structures are otherwise virtually identical, differing only in the number of rounds.
- SHA-224 and SHA-384 are simply truncated versions of the first two, computed with different initial values.

SHA-2 Family

- Unlike SHA-1, the SHA-2 functions are not widely used, despite their better security.
- Reasons might include lack support for SHA-2 on systems running Windows XP SP2 or older, a lack of perceived urgency, or a desire to wait until SHA-3 is standardized.
- SHA-256 is used to authenticate Debian Linux software
- Currently, the best public attacks on SHA-2 break 24 of the 64 or 80 rounds.

SHA-3 The Search

- The NIST hash function competition refers to an open competition for a new SHA-3 function to replace the older SHA-1 and SHA-2 hash functions which was formally announced on November 2, 2007.
- Submissions were due October 31, 2008, with a list of candidates accepted for the first round published December 9, 2008.

SHA-3 The Search

- NIST held a conference in late February 2009 where submitters gave presentations on their algorithms and NIST officials discussed criteria for narrowing down the field of candidates for Round 2.

- The list of 14 candidates accepted to Round 2 was published on July 24, 2009

- The announcement of the final round candidates is tentatively scheduled for 2010, while the proclamation of a winner and publication of the new standard are scheduled to take place in 2012.

SHA-3 The Search

- The following hash function submissions have been accepted for Round Two.

  - BLAKE
  - Blue Midnight Wish
  - CubeHash (Bernstein)
  - ECHO (France Telecom)
  - Fugue (IBM)
  - Grøstl (Knudsen et al.)

SHA-3 The Search

- The following hash function submissions have been accepted for Round Two.

  - Hamsi
  - JH
  - Keccak (Keccak team, Daemen et al.)
  - Luffa
  - Shabal
  - SHAvite-3
  - SIMD
  - Skein (Schneier et al.)

SHA-3 The Search

- NIST Cryptographic Hash Algorithm Competition
  - http://csrc.nist.gov/groups/ST/hash/sha-3/index.html
- NIST official Round One Candidates web site
  - http://csrc.nist.gov/groups/ST/hash/sha-3/Round1/submissions_rnd1.html
- List of known weakness against SHA candidates:
  - http://ehash.iaik.tugraz.at/wiki/The_SHA-3_Zoo
- The Hash function Lounge
  - http://www.larc.usp.br/~pbarreto/hflounge.html