Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

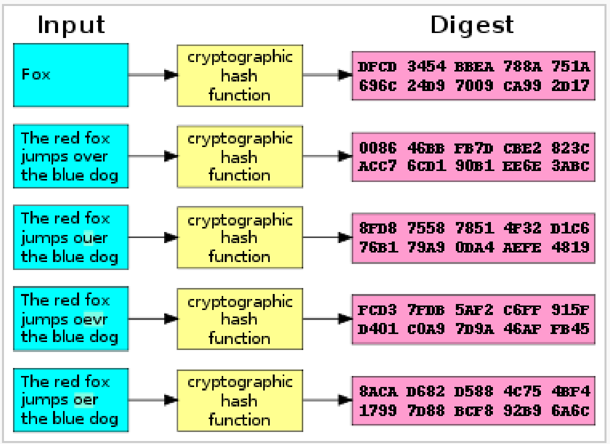
Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
■■■■■
Čeština
Deutsch
Ελληνικά
Español
فارسی
Français
한국어
Italiano
עברית
Lietuvių
Magyar
■■■■■■
日本語
Polski
Português
Русский
Simple English
Slovenčina
Српски / srpski
Suomi
Türkçe
Українська
Tiếng Việt
粵語
中文
Edit links

Cryptographic hash function

From Wikipedia, the free encyclopedia

A **cryptographic hash function** is a hash function which is considered practically impossible to invert, that is, to recreate the input data from its hash value alone. These one-way hash functions have been called "the workhorses of modern cryptography".[1] The input data is often called the *message*, and the hash value is often called the *message digest* or simply the *digest*.


Input
Digest
Fox
The red fox jumps over the blue dog
The red fox jumps ouer the blue dog
The red fox jumps oevr the blue dog
The red fox jumps oer the blue dog
cryptographic hash function
DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17
0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC
8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEFE 4819
FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45
8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C
A cryptographic hash function (specifically, SHA-1) at work. Note that even small changes in the source input (here in the word "over") drastically change the resulting output, by the so-called avalanche effect.

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 generate a message from its hash
- it is infeasible to modify a message without changing the hash
- it is infeasible to find two different messages with the same hash.

Cryptographic hash functions have many information security applications, notably in digital signatures, message authentication codes (MACs), and other forms of authentication. They can also be used as ordinary hash functions, to index data in hash tables, for fingerprinting, to detect duplicate data or uniquely identify files, and as checksums to detect accidental data corruption. Indeed, in information security contexts, cryptographic hash values are sometimes called (*digital*) *fingerprints*, *checksums*, or just *hash values*, even though all these terms stand for more general functions with rather different properties and purposes.

Contents [hide]

Properties [edit]

Most cryptographic hash functions are designed to take a string of any length as input and produce a fixed-length hash value.

A cryptographic hash function must be able to withstand all known types of cryptanalytic attack. At a minimum, it must have the following properties:

- *Pre-image resistance*

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

- *Second pre-image resistance*

  Given an input m_1 it should be difficult to find different input m_2 such that $hash(m_1) = hash(m_2)$. Functions that lack this property are vulnerable to second-preimage attacks.

- *Collision resistance*

  It should be difficult 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. This property is sometimes referred to as *strong collision resistance.* It requires a hash value at least twice as long as that required for preimage-resistance; otherwise collisions may be found by a birthday attack.

These properties imply that a malicious adversary cannot 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.

A function meeting these criteria may still have undesirable properties. Currently popular cryptographic hash functions are vulnerable to *length-extension* attacks: given $hash(m)$ and $len(m)$ but not m, by choosing a suitable m' an attacker can calculate $hash(m \,||\, m')$ where || denotes concatenation.[2] This property can be used to break naive authentication schemes based on hash functions. The HMAC construction works around these problems.

Ideally, one may wish for even stronger conditions. It should be impossible for an adversary to find two messages with substantially similar digests; or to infer any useful information about the data, given only its digest. Therefore, a cryptographic hash function should behave as much as possible like a random function while still being deterministic and efficiently computable.

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.

Degree of difficulty [edit]

In cryptographic practice, “difficult” generally means “almost certainly beyond the reach of any adversary who must be prevented from breaking the system for as long as the security of the system is deemed important”. The meaning of the term is therefore somewhat dependent on the application, since the effort that a malicious agent may put into the task is usually proportional to his expected gain. However, since the needed effort usually grows very quickly with the digest length, even a thousand-fold advantage in processing power can be neutralized by adding a few dozen bits to the latter.

In some theoretical analyses “difficult” has a specific mathematical meaning, such as "not solvable in asymptotic polynomial time". Such interpretations of *difficulty* are important in the study of provably secure cryptographic hash functions but do not usually have a strong connection to practical security. For example, an exponential time algorithm can sometimes still be fast enough to make a feasible attack. Conversely, a polynomial time algorithm (e.g., one that requires n^{20} steps for n-digit keys) may be too slow for any practical use.

Illustration [edit]

An illustration of the potential use of a cryptographic hash is as follows: Alice poses a tough math problem to Bob and claims she has solved it. Bob would like to try it himself, but would yet like to be sure that Alice is not bluffing. Therefore, Alice writes down her solution, computes its hash and tells Bob the hash value (whilst keeping the solution secret). Then, when Bob comes up with the solution himself a few days later, Alice can prove that she had the solution earlier by revealing it and having Bob hash it and check that it matches the hash value given to him before. (This is an example of a simple commitment scheme; in actual practice, Alice and Bob will often be computer programs, and the secret would be something less easily spoofed than a claimed puzzle solution).

Applications [edit]

Verifying the integrity of files or messages [edit]

Main article: File verification

An important application of secure hashes is verification of message integrity. Determining whether any changes have been made to a message (or a file), for example, can be accomplished by comparing message digests calculated before, and after, transmission (or any other event).

For this reason, most digital signature algorithms only confirm the authenticity of a hashed digest of the message to be "signed". Verifying the authenticity of a hashed digest of the message is considered proof that the message itself is authentic.

MD5, SHA1, or SHA2 hashes are sometimes posted along with files on websites or forums to allow verification of integrity.[3] This practice establishes a chain of trust so long as the hashes are posted on a site authenticated by HTTPS.

Password verification [edit]

A related application is password verification (first invented by Roger Needham). Storing all user passwords as cleartext can result in a massive security breach if the password file is compromised. One way to reduce this danger is to only store the hash digest of each password. To authenticate a user, the password presented by the user is hashed and compared with the stored hash. (Note that this approach prevents the original passwords from being retrieved if forgotten or lost, and they have to be replaced with new ones.) The password is often concatenated with a random, non-secret salt value before the hash function is applied. The salt is stored with the password hash. Because users have different salts, it is not feasible to store tables of precomputed hash values for common passwords. Key stretching functions, such as PBKDF2, Bcrypt or Scrypt, typically use repeated invocations of a cryptographic hash to increase the time required to perform brute force attacks on stored password digests.

Main article: Password cracking

In 2013 a long-term Password Hashing Competition was announced to choose a new, standard algorithm for password hashing.[4]

Proof-of-work [edit]

Main article: Proof-of-work system

A proof-of-work system (or protocol, or function) is an economic measure to deter denial of service attacks and other service abuses such as spam on a network by requiring some work from the service requester, usually meaning processing time by a computer. A key feature of these schemes is their asymmetry: the work must be moderately hard (but feasible) on the requester side but easy to check for the service provider. One popular system — used in Bitcoin mining and Hashcash — uses partial hash inversions to prove that work was done, as a good-will token to send an e-mail. The sender is required to find a message whose hash value begins with a number of zero bits. The average work that sender needs to perform in order to find a valid message is exponential in the number of zero bits required in the hash value, while the recipient can verify the validity of the message by executing a single hash function. For instance, in Hashcash, a sender is asked to generate a header whose 160 bit SHA-1 hash value has the first 20 bits as zeros. The sender will on average have to try 2^{20} times to find a valid header.

File or data identifier [edit]

A message digest can also serve as a means of reliably identifying a file; several source code management systems, including Git, Mercurial and Monotone, use the sha1sum of various types of content (file content, directory trees, ancestry information, etc.) to uniquely identify them. Hashes are used to identify files on peer-to-peer filesharing networks. For example, in an ed2k link, an MD4-variant hash is combined with the file size, providing sufficient information for locating file sources, downloading the file and verifying its contents. Magnet links are another example. Such file hashes are often the top hash of a hash list or a hash tree which allows for additional benefits.

One of the main applications of a hash function is to allow the fast look-up of a data in a hash table. Being hash functions of a particular kind, cryptographic hash functions lend themselves well to this application too.

However, compared with standard hash functions, cryptographic hash functions tend to be much more expensive computationally. For this reason, they tend to be used in contexts where it is necessary for users to protect themselves against the possibility of forgery (the creation of data with the same digest as the expected data) by potentially malicious participants.

Pseudorandom generation and key derivation [edit]

Hash functions can also be used in the generation of pseudorandom bits, or to derive new keys or passwords from a single, secure key or password.

Hash functions based on block ciphers [edit]

There are several methods to use a block cipher to build a cryptographic hash function, specifically a one-way compression function.

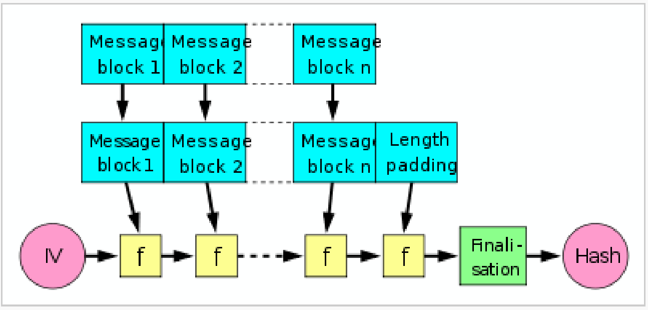
The methods resemble the block cipher modes of operation usually used for encryption. Many well-known hash functions, including MD4, MD5, SHA-1 and SHA-2 are built from block-cipher-like components designed for the purpose, with feedback to ensure that the resulting function is not invertible. SHA-3 finalists included functions with block-cipher-like components (e.g., Skein, BLAKE) though the function finally selected, Keccak, was built on a cryptographic sponge instead.

A standard block cipher such as AES can be used in place of these custom block ciphers; that might be useful when an embedded system needs to implement both encryption and hashing with minimal code size or hardware area. However, that approach can have costs in efficiency and security. The ciphers in hash functions are built for hashing: they use large keys and blocks, can efficiently change keys every block, and have been designed and vetted for resistance to related-key attacks. General-purpose ciphers tend to have different design goals. In particular, AES has key and block sizes that make it nontrivial to use to generate long hash values; AES encryption becomes less efficient when the key changes each block; and related-key attacks make it potentially less secure for use in a hash function than for encryption.

Merkle–Damgård construction [edit]

Main article: Merkle–Damgård construction

A hash function must be able to process an arbitrary-length message into a fixed-length output. This can be achieved by breaking the input up into a series of equal-sized blocks, and operating on them in sequence using a one-way compression function. The compression function can either be specially designed for hashing or be built from a block cipher. A hash function built with the Merkle–Damgård construction is as resistant to collisions as is its compression function; any collision for the full hash function can be traced back to a collision in the compression function.



The Merkle–Damgård hash construction.

The last block processed should also be unambiguously length padded; this is crucial to the security of this construction. This construction is called the Merkle–Damgård construction. Most widely used hash functions, including SHA-1 and MD5, take this form.

The construction has certain inherent flaws, including length-extension and generate-and-paste attacks, and cannot be parallelized. As a result, many entrants in the current NIST hash function competition are built on different, sometimes novel, constructions.

Use in building other cryptographic primitives [edit]

Hash functions can be used to build other cryptographic primitives. For these other primitives to be cryptographically secure, care must be taken to build them correctly.

Message authentication codes (MACs) (also called keyed hash functions) are often built from hash functions. HMAC is such a MAC.

Just as block ciphers can be used to build hash functions, hash functions can be used to build block ciphers. Luby-Rackoff constructions using hash functions can be provably secure if the underlying hash function is secure. Also, many hash functions (including SHA-1 and SHA-2) are built by using a special-purpose block cipher in a Davies-Meyer or other construction. That cipher can also be used in a conventional mode of operation, without the same security guarantees. See SHACAL, BEAR and LION.

Pseudorandom number generators (PRNGs) can be built using hash functions. This is done by combining a (secret) random seed with a counter and hashing it.

Some hash functions, such as Skein, Keccak, and RadioGatún output an arbitrarily long stream and can be used as a stream cipher, and stream ciphers can also be built from fixed-length digest hash functions. Often this is done by first building a cryptographically secure pseudorandom number generator and then using its stream of random bytes as keystream. SEAL is a stream cipher that uses SHA-1 to generate internal tables, which are then used in a keystream generator more or less unrelated to the hash algorithm. SEAL is not guaranteed to be as strong (or weak) as SHA-1. Similarly, the key expansion of the HC-128 and HC-256 stream ciphers makes heavy use of the SHA256 hash function.

Concatenation of cryptographic hash functions [edit]

Concatenating outputs from multiple hash functions provides collision resistance as good as the strongest of the algorithms included in the concatenated result. For example, older versions of TLS/SSL use concatenated MD5 and SHA-1 sums—this ensures that a method to find collisions in one of the functions doesn't allow forging traffic protected with both functions.

For Merkle-Damgård hash functions, the concatenated function is as collision-resistant as its strongest component,[5] but not more collision-resistant.[6] Joux[7] noted that 2-collisions lead to n-collisions: if it is feasible to find two messages with the same MD5 hash, it is effectively no more difficult to find as many messages as the attacker desires with identical MD5 hashes. Among the n messages with the same MD5 hash, there is likely to be a collision in SHA-1. The additional work needed to find the SHA-1 collision (beyond the exponential birthday search) is polynomial. This argument is summarized by Finney. A more current paper and full proof of the security of such a combined construction gives a clearer and more complete explanation of the above.[8]

Cryptographic hash algorithms [edit]

There is a long list of cryptographic hash functions, although many have been found to be vulnerable and should not be used. Even if a hash function has never been broken, a successful attack against a weakened variant thereof may undermine the experts' confidence and lead to its abandonment. For instance, in August 2004 weaknesses were found in a number of hash functions that were popular at the time, including SHA-0, RIPEMD, and MD5. This has called into question the long-term security of later algorithms which are derived from these hash functions — in particular, SHA-1 (a strengthened version of SHA-0), RIPEMD-128, and RIPEMD-160 (both strengthened versions of RIPEMD). Neither SHA-0 nor RIPEMD are widely used since they were replaced by their strengthened versions.

As of 2009, the two most commonly used cryptographic hash functions are MD5 and SHA-1. However, MD5 has been broken; an attack against it was used to break SSL in 2008.[9]

The SHA-0 and SHA-1 hash functions were developed by the NSA.

On 12 August 2004, a collision for the full SHA-0 algorithm was announced by Joux, Carribault, Lemuet, and Jalby. This was done by using a generalization of the Chabaud and Joux attack. Finding the collision had complexity 2^{51} and took about 80,000 CPU hours on a supercomputer with 256 Itanium 2 processors. (Equivalent to 13 days of full-time use of the computer.)

In February 2005, an attack on SHA-1 was reported that would find collision in about 2^{69} hashing operations, rather than the 2^{80} expected for a 160-bit hash function. In August 2005, another attack on SHA-1 was reported that would find collisions in 2^{63} operations. Though theoretical weaknesses of SHA-1 exist,[10][11] no collision (or near-collision) has yet been found. Nonetheless, it is often suggested that it may be practical to break within years, and that new applications can avoid these problems by using later members of the SHA family, such as SHA-2, or using techniques such as randomized hashing[12][13] that do not require collision resistance.

However, to ensure the long-term robustness of applications that use hash functions, there was a competition to design a replacement for SHA-2. On October 2, 2012, Keccak was selected as the winner of the NIST hash function competition. A version of this algorithm became a FIPS standard on August 5, 2015 under the name SHA-3.[14]

See also [edit]

- Avalanche effect
- Comparison of cryptographic hash functions
- Security of cryptographic hash functions
- CRYPTREC and NESSIE - Projects which recommend hash functions
- Keyed-hash message authentication code
- MD5CRK
- Message authentication code
- PGP word list
- Provably secure cryptographic hash function
- SHA-3
- UOWHF - Universal One Way Hash Functions
- Hash chain

Cryptography portal

References [edit]

1. ^ Schneier, Bruce. "Cryptanalysis of MD5 and SHA: Time for a New Standard". *Computerworld*. Retrieved 15 October 2014.
2. ^ "Flickr's API Signature Forgery Vulnerability". Thai Duong and Juliano Rizzo.
3. ^ Perrin, Chad (December 5, 2007). "Use MD5 hashes to verify software downloads". TechRepublic. Retrieved March 2, 2013.
4. ^ "Password Hashing Competition". Retrieved March 3, 2013.
5. ^ Note that any two messages that collide the concatenated function also collide each component function, by the nature of concatenation. For example, if concat(sha1(message1), md5(message1)) == concat(sha1(message2), md5(message2)) then sha1(message1) == sha1(message2) and md5(message1)==md5(message2). The concatenated function could have other problems that the strongest hash lacks – for example, it might leak information about the message when the strongest component does not, or it might be detectably nonrandom when the strongest component is not – but it can't be less collision-resistant.
6. ^ More generally, if an attack can produce a collision in one hash function's *internal state,* attacking the combined construction is only as difficult as a birthday attack against the other function(s). For the detailed argument, see the Joux and Finney references that follow.
7. ^ Antoine Joux. *Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions*. LNCS 3152/2004, pages 306-316 Full text.
8. ^ Hoch, Jonathan J.; Shamir, Adi (2008-02-20). "On the Strength of the Concatenated Hash Combiner when all the Hash Functions are Weak" (PDF).
9. ^ Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, Benne de Weger, MD5 considered harmful today: Creating a rogue CA certificate, accessed March 29, 2009
10. ^ Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu, Finding Collisions in the Full SHA-1
11. ^ Bruce Schneier, Cryptanalysis of SHA-1 (summarizes Wang et al. results and their implications)
12. ^ Shai Halevi, Hugo Krawczyk, Update on Randomized Hashing
13. ^ Shai Halevi and Hugo Krawczyk, Randomized Hashing and Digital Signatures
14. ^ NIST.gov - Computer Security Division - Computer Security Resource Center

External links [edit]

- Paar, Christof; Pelzl, Jan (2009). "11: Hash Functions". *Understanding Cryptography, A Textbook for Students and Practitioners*. Springer. (companion web site contains online cryptography course that covers hash functions)
- "The ECRYPT Hash Function Website".
- Buldas, A. (2011). "Series of mini-lectures about cryptographic hash functions".
- Rogaway, P.; Shrimpton, T. (2004). "Cryptographic Hash-Function Basics: Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance". CiteSeerX: 10.1.1.3.6200.

v · t · e	**Hash functions & message authentication codes**
	Security summary
Common functions	MD5 · SHA-1 · SHA-2 · SHA-3/Keccak
SHA-3 finalists	BLAKE · Grøstl · JH · Skein · Keccak (winner)
Other functions	FSB · ECOH · GOST · HAS-160 · HAVAL · LM hash · MDC-2 · MD2 · MD4 · MD6 · N-Hash · RadioGatún · RIPEMD · SipHash · Snefru · Streebog · SWIFFT · Tiger · VSH · WHIRLPOOL · crypt(3) (DES)
MAC algorithms	DAA · CBC-MAC · HMAC · OMAC/CMAC · PMAC · VMAC · UMAC · Poly1305-AES
Authenticated encryption modes	CCM · CWC · EAX · GCM · IAPM · OCB
Attacks	Collision attack · Preimage attack · Birthday attack · Brute force attack · Rainbow table · Distinguishing attack · Side-channel attack · Length extension attack
Design	Avalanche effect · Hash collision · Merkle–Damgård construction
Standardization	CRYPTREC · NESSIE · NIST hash function competition
Utilization	Salt · Key stretching · Message authentication

v · t · e	**Cryptography**
	History of cryptography · Cryptanalysis · Cryptography portal · Outline of cryptography
	Symmetric-key algorithm · Block cipher · Stream cipher · Public-key cryptography · **Cryptographic hash function** · Message authentication code · Random numbers · Steganography

Categories: Cryptography | Cryptographic primitives | Cryptographic hash functions | Hashing