CPSC 467: Cryptography and Security

Michael J. Fischer

Lecture 16
October 27, 2020

Common Hash Functions

- SHA-2
- SHA-3
- MD5

Hashed Data Structures

- Motivation: Peer-to-peer file sharing networks
- Hash lists
- Hash Trees

Appendix: Birthday Attack Revisited

Common Hash Functions

Popular hash functions

Many cryptographic hash functions are currently in use.

For example, the openssl library includes implementations of MD2, MD4, MD5, MDC2, RIPEMD, SHA, SHA-1, SHA-256, SHA-384, and SHA-512.

The SHA-xxx methods (otherwise known as SHA-2) are recommended for new applications, but these other functions are also in widespread use.

SHA-2

SHA-2 is a family of hash algorithms designed by NSA known as SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224, SHA-512/256.

They produce message digests of lengths 224, 256, 384, or 512 bits.

They comprise the current Secure Hash Standard (SHS) and are described in FIPS 180–4. It states,

> *"Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits)."*

SHA-1 broken

SHA-1 was first described in 1995. It produces a 160-bit message digest.

It was broken in 2005 by Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu: "Finding Collisions in the Full SHA-1". *CRYPTO 2005*: 17-36.

Wang and Yu did their work at Shandong University; Yin is listed on the paper as an independent security consultant in Greenwich, CT.

A new secure hash algorithm

On Nov. 2, 2007, NIST announced a public competition for a replacement algorithm to be known as SHA-3.

The winner, an algorithm named Keccak, was announced on October 2, 2012 and standardized in August 2015. See http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf.

From the SHA-3 standard

Now that the standards document is out, it seems that SHA-3 is considered to be a supplement to the previous standard, not a replacement for it. The quote below is from the abstract of FIPS PUB 202.

> *"Hash functions are components for many important information security applications, including 1) the generation and verification of digital signatures, 2) key derivation, and 3) pseudorandom bit generation. The hash functions specified in this Standard supplement the SHA-1 hash function and the SHA-2 family of hash functions that are specified in FIPS 180-4, the Secure Hash Standard."*

MD5

MD5 is an older algorithm (1992) devised by Rivest.

Weaknesses were found as early as 1996. It was shown not to be collision resistant in 2004.[1]

Subsequent papers show that MD5 has more serious weaknesses that make it no longer suitable for most cryptographic uses.

We present an overview of MD5 here because it is relatively simple and it illustrates the principles used in many hash algorithms.

[1] How to Break MD5 and Other Hash Functions by Xiaoyun Wang and Hongbo Yu.

MD5 algorithm overview

MD5 generates a 128-bit message digest from an input message of any length. It is built from a basic block function

$$g : \text{128-bit} \times \text{512-bit} \to \text{128-bit}.$$

The MD5 hash function h is obtained as follows:

- The original message is padded to length a multiple of 512.
- The result m is split into a sequence of 512-bit blocks $m_1, m_2, \ldots, m_k$.
- h is computed by chaining g on the first argument.

We next look at these steps in greater detail.

MD5 padding

As with block encryption, it is important that the padding function be one-to-one, but for a different reason.

For encryption, the one-to-one property is what allows unique decryption.

For a hash function, it prevents there from being trivial colliding pairs.

For example, if the last partial block is simply padded with 0's, then all prefixes of the last message block will become the same after padding and will therefore collide with each other.

MD5 chaining

The function h can be regarded as a state machine, where the states are 128-bit strings and the inputs to the machine are 512-bit blocks.

The machine starts in state s_0, specified by an initialization vector IV.

Each input block m_i takes the machine from state s_{i-1} to new state $s_i = g(s_{i-1}, m_i)$.

The last state s_k is the output of h, that is,

$$h(m_1 m_2 \ldots m_{k-1} m_k) = g(g(\ldots g(g(IV, m_1), m_2) \ldots, m_{k-1}), m_k).$$

MD5 block function

The block function $g(s, b)$ is built from a scrambling function $g'(s, b)$ that regards s and b as sequences of 32-bit words and returns four 32-bit words as its result.

Suppose $s = s_1 s_2 s_3 s_4$ and $g'(s, b) = s'_1 s'_2 s'_3 s'_4$.

We define

$$g(s, b) = (s_1 + s'_1) \cdot (s_2 + s'_2) \cdot (s_3 + s'_3) \cdot (s_4 + s'_4),$$

where "+" means addition modulo 2^{32} and "$\cdot$" is concatenation of the representations of integers as 32-bit binary strings.

MD5 scrambling function

The computation of the scrambling function $g'(s, b)$ consists of 4 stages, each consisting of 16 substages.

We divide the 512-bit block b into 32-bit words $b_1 b_2 \ldots b_{16}$.

Each of the 16 substages of stage i uses one of the 32-bit words of b, but the order they are used is defined by a permutation π_i that depends on i.

In particular, substage j of stage i uses word b_ℓ, where $\ell = \pi_i(j)$ to update the state vector s.

The new state is $f_{i,j}(s, b_\ell)$, where $f_{i,j}$ is a bit-scrambling function that depends on i and j.

Further remarks on MD5

We omit further details of the bit-scrambling functions $f_{i,j}$,

However, note that the state s can be represented by four 32-bit words, so the arguments to $f_{i,j}$ occupy only 5 machine words. These easily fit into the high-speed registers of modern processors.

The definitive specification for MD5 is RFC1321 and errata. A general discussion of MD5 along with links to recent work and security issues can be found on Wikipedia.

Hashed Data Structures

Peer-to-peer networks

One real-world application of hash functions is to *peer-to-peer* file-sharing networks.

The goal of a P2P network is to improve throughput when sending large files to large numbers of clients.

It operates by splitting the file into blocks and sending each block separately through the network along possibly different paths to the client.

Rather than fetching each block from the master source, a block can be received from any node (peer) that happens to have the needed block.

The problem is to validate blocks received from untrusted peers.

Integrity checking

An obvious approach is for a trusted master node to send each client a hash of the entire file.

When all blocks have been received, the client reassembles the file, computes its hash, and checks that it matches the hash received from the master.

The problem with this approach is that if the hashes don't match, the client has no idea which block is bad.

What is needed is a way to send a "proof" with each block that the client can use to verify the integrity of the block when it is received.

Block hashes

One idea is to compute a hash h_k of each data block b_k using a cryptographic hash function H.

The client *validates* b_k by checking that $H(b_k) = h_k$.

This allows a large *untrusted* data block to be validated against a much shorter *trusted* block hashes.

Problem: How does the client obtain and validate the hashes?

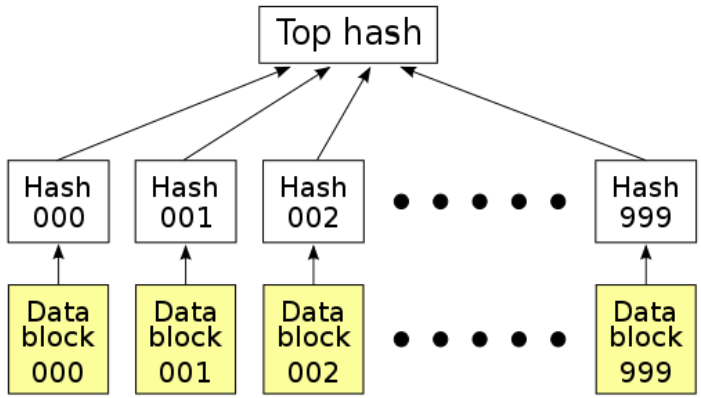
We desire a scheme in which a small amount of trusted data can be leveraged to validate the block hashes as well as the data blocks themselves.

Hash lists

A *hash list* is a two-level tree of hash values.

The leaves of the tree are the block hashes Hash_k.

The concatenation of the Hash_k are hashed together to produce a *top hash*.



From Wikipedia, "Hash List"

Using a hash list

The client receives top hash from the trusted source.

The client receives the list of Hash_k from any untrusted source.

The Hash_k are validated by hashing their concatenation and comparing the result with the stored top hash.

Each data block b_k is validated using the corresponding Hash_k.

Weakness of hash list approach

The main drawback of hash lists is that the entire hash list must be downloaded and verified before data blocks can be checked.

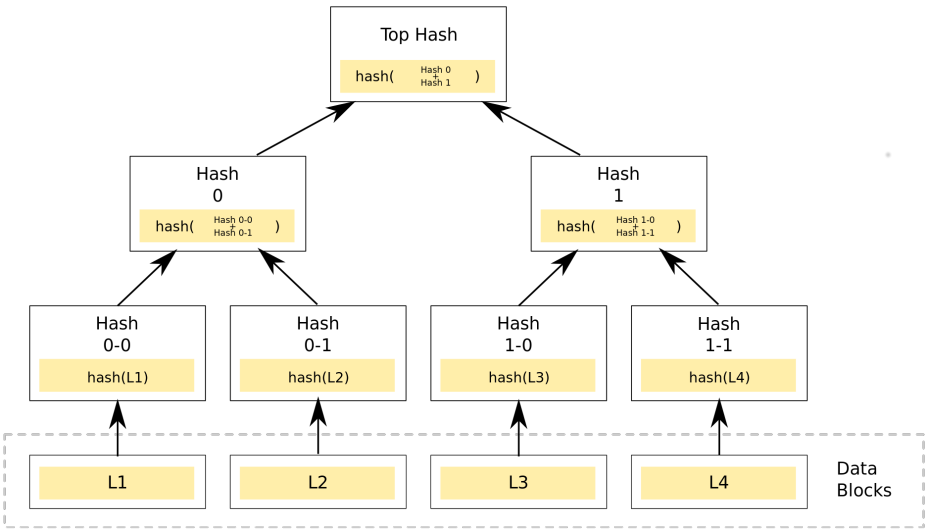
A bad Hash_k would cause a good b_k to be repeatedly rejected and refetched.

BitTorrent places all of the hashes in a single file which is initially downloaded from a trusted source.

Hash trees (Merkle trees)

Hash trees provide a way to attach a small untrusted *validation block* to each data block that allows the data block to be validated directly against a single trusted hash value *top hash*.

Neither the validation block nor the data block need be trusted; errors in either will be detected.



From Wikipedia, "Merkle tree"

Example: To validate L2, use Hash 0-0 and Hash 1 to compute Top Hash. Compare with trusted Top Hash. If they agree, can trust L2, Hash 0-0, and Hash 1.

Tree notation

A hash tree is a *complete binary tree* with $N = 2^n$ leaves.

Label the nodes by strings $\sigma \in \{0,1\}^*$, where σ describes the path from the root to the node.

The *root* is denoted by v_ε, where ε is the *null string*. Its two sons are v_0 and v_1.

The two children of any internal node v_j are denoted by v_{j0} and v_{j1}.

Let σ_k be the path from the root to the k^{th} leaf. Then v_{σ_k} denotes the leaf node corresponding to data block b_k.

Node values in a hash tree

Let v_τ be a node in a hash tree.

Define Hash_τ, the hash value at node v_τ:

- If v_τ is a leaf, then $\tau = \sigma_k$ for some k, and

$$\text{Hash}_\tau = H(b_k).$$

- If v_τ is an internal node, then

$$\text{Hash}_\tau = H(\text{Hash}_{\tau 0} \cdot \text{Hash}_{\tau 1}).$$

Companion nodes

Let v_τ be an internal node, and let $v_{\tau'}$ be its sibling in the tree. We say that τ' is the *companion* of τ. τ' is obtained from τ by flipping the last bit.

Example: The companion of 1011 is 1010 since v_{1011} and v_{1010} are the two children of v_{101}.

Validation block

The *validation block* B_k for data block k consists of the sequence of hash values $\text{Hash}_{\tau'}$ for each companion τ' of each non-null prefix τ of σ_k.

Example: Let $\sigma_k = 1011$.

- The non-null prefixes of σ_k are 1011, 101, 10, 1.
- The corresponding companions are 1010, 100, 11, 0.
- The validation block is

$$B_k = (\text{Hash}_{1010}, \text{Hash}_{100}, \text{Hash}_{11}, \text{Hash}_{0}).$$

Block validation using hash trees

Validating data block b_k requires top hash and validation block B_k.

One proceeds by computing Hash_τ for each τ that is a prefix of σ_k, working in order from longer to shorter prefixes.

- If $\tau = \sigma_k$, then $\text{Hash}_\tau = H(b_k)$.
- Let τ be a proper prefix of σ_k. Assume w.l.o.g. that $\tau 0$ is a prefix of σ_k and $\tau 1$ is its companion. Then $\text{Hash}_{\tau 0}$ has just been computed, and $\text{Hash}_{\tau 1}$ is available in the validation block. We compute

$$\text{Hash}_\tau = H(\text{Hash}_{\tau 0} \cdot \text{Hash}_{\tau 1}).$$

Validation succeeds if $\text{Hash}_\varepsilon = \text{top hash}$.

Appendix: Birthday Attack Revisited

Bits of security for hash functions

MD5 hash function produces 128-bit values, whereas the SHA-xxx family produces values of 160-bits or more.

How many bits do we need for security?

Both 128 and 160 are more than large enough to thwart a brute force attack that simply searches randomly for colliding pairs.

However, the *Birthday Attack* reduces the size of the search space to roughly the square root of the original size.

MD5's effective security is at most 64 bits. ($\sqrt{2^{128}} = 2^{64}$.)

SHA-1's effective security is at most 80-bits. ($\sqrt{2^{160}} = 2^{80}$.)

Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu describe an attack that reduces this number to only 69-bits (Crypto 2005).

Birthday Paradox

We described a *birthday attack* in lecture 9, based on the *birthday paradox*.

The problem is to find the probability that two people in a set of randomly chosen people have the same birthday.

This probability is greater than 50% in any set of at least 23 randomly chosen people.[2].

23 is far less than the 253 people that are needed for the probability to exceed 50% that at least one of them was born on a specific day, say January 1.

[2] See Wikipedia, "Birthday paradox".

Birthday Paradox (cont.)

Here's why it works.

The probability of *not* having two people with the same birthday is is

$$q = \frac{365}{365} \cdot \frac{364}{365} \cdots \frac{343}{365} = 0.492703$$

Hence, the probability that (at least) two people have the same birthday is $1 - q = 0.507297$.

This probability grows quite rapidly with the number of people in the room. For example, with 46 people, the probability that two share a birthday is 0.948253.

Birthday attack on hash functions

The birthday paradox gives a much faster way to find colliding pairs of a hash function than simply choosing pairs at random.

> Method: Choose a random set of k messages and see if any two messages in the set collide.

Thus, with only k evaluations of the hash function, we can test $\binom{k}{2} = k(k-1)/2$ different pairs of messages for collisions.

Birthday attack analysis

Of course, these $\binom{k}{2}$ pairs are not uniformly distributed, so one needs a birthday-paradox style analysis of the probability that a colliding pair will be found.

The general result is that the probability of success is at least $1/2$ when $k \approx \sqrt{n}$, where n is the size of the hash value space.

Practical difficulties of birthday attack

Two problems make this attack difficult to use in practice.

1. One must find duplicates in the list of hash values. This can be done in time $O(k \log k)$ by sorting.
2. The list of hash values must be stored and processed.

For MD5, $k \approx 2^{64}$. To store k 128-bit hash values requires 2^{68} bytes $\approx$ 250 exabytes = 250,000 petabytes of storage.

To sort would require $\log_2(k) = 64$ passes over the table, which would process 16 million petabytes of data.

A back-of-the-envelope calculation

Google was reportedly processing 20 petabytes of data per day in 2008. At this rate, it would take Google more than 800,000 days or nearly 2200 years just to sort the data.

This attack is still infeasible for values of k needed to break hash functions. Nevertheless, it is one of the more subtle ways that cryptographic primitives can be compromised.