Authentication: Integrity Checking

Michael Brockway

March 5, 2018

Overview

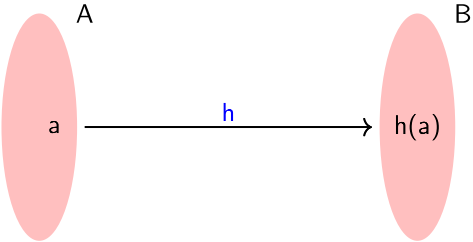
We have met two ways in which we need to be assured or the *authenticity* of a message sent through a network:

- The message really is from the party who purportedly sent it;
- The message data is as sent; it has not been tampered with en route.

These are two different security assurances and we seen one is provided for by a digital signature mechanism (backed up by digital certificates) while the other is provided a *message digest* created by a *hash function*.

The acronym MAC, 'Message authentication code' is confusingly used to refer to either of these. These slides focus of the second and takes a closer look at message digests and hash functions.

Hash Functions



In general, a *function* h maps a set A of objects to a set B of (other) objects, the idea being that for any $a \in A$ there is a (*unique*) $h(a) \in B$. We write $A \xrightarrow{h} B$.

An example: any java Object `ob` has a method:
`public int ob.hashCode()`.
We can think of h mapping `ob` to `ob.hashCode()`. In this case B is the set of `int` values - there are 2^{32} of them.

Hash Functions

Java hash functions are supposed to be contrived so that whenever `(ob1.equals(ob2))` then `(ob1.hashCode() == ob1.hashCode())`. To be useful, we would also like `(!ob1.equals(ob2))` ⇒ `(ob1.hashCode() != ob1.hashCode())`.

This is not guaranteed but a *hash collision*, where `(!ob1.equals(ob2))` but `(ob1.hashCode() == ob1.hashCode())` has very low probability when the hash function is well designed.

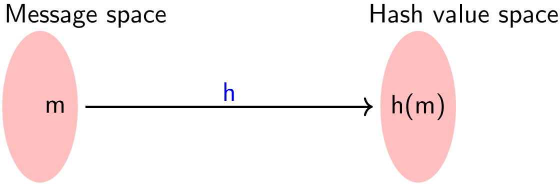
Java programmers overriding Object.hashCode() are supposed to pay attention to this.

Hash Functions

You have met java `HashSets` and `HashMaps` which store objects in *hash tables*.

- A suitably contrived hash function on objects returns a number which indexes into an array.
- The object reference is stored here;
- A collision is resolved by putting the objects in a linked list at the location.
- If the probability of the collision is low than these lists are short.

Cryptographic hash functions



For every message *m*, hash value $h(m)$ is efficiently computable: it is a sequence bits which can be thought of as an integer: $h(m) < 2^s$ where is s is the size of the hash in bits.

- Not only are hash collisions improbable (2^s is 'large'), but a 1-bit change in the message almost always produces a large change in h(m);
- h is *pre-image resistent*: it is infeasible (for an attakcer) to contrive a message m for which $h(m) =$ a desired value - such as the hash of another message;
- It is *strongly collision resistant*: it is infeasible to contrive a pair of messages m, m' such that $h(m) == h(m')$.

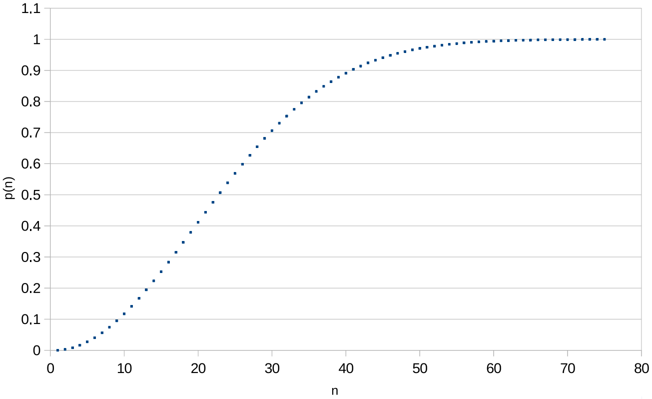
Cryptographic hash functions - use

- The sender of a message computes the hash of the message and appends it before encrypting.
- The recever, after decrypting, computes a hash,
- and compares it with the one that was sent.
- Any mismtch => tampering!

Cryptographic hash functions

The *birthday attack* is an exploit protected against by strongly collision-resistant hashing. The attacker has two versions of, say, a contract, one less favourable than the other, with the same hash value, and can switch them without detection if the hash were *not* strongly collision-resistant.

You have probably heard that in a random sample of n people, the probability two have birthdays on the same day grows with n and passes 0.5-0.5 when $n > 23$...



Cryptographic hash function examples - MD5

MD5 (R Rivest, 1991-2)

- 128-bit hashes: $2^{128} \approx 10^{38}$, 100 million million million million million million values;
- by 2004, not enough! Wang, Feng, Lai and Yu contrived a collision in 1 CPU-hour on an IBM p690
- Updates were issued until 2010
- Now considered insecure, also found to be still used as recently as 2015.
- The Wikipedia article has a neat summary of the algorithm, its security issues and vularabilities.

SHA-1: Secure hash algorithm 1

- 160-bit hashes: $2^{160} \approx 1.4 \times 10^{48}$, a million million million million million million million million values;
- From 2005, collision attacks began to be contrived: Rijmen and Oswald in 2^{80} operations, Wang, Yin and Yu in 2^{69} operations.
- These early attacks were actually prohibitively expensive; but in October 2015 M Stevens and others *demonstrated* a partial attack using a grid of NVIDIA GPUs costing around US$2000 -
- ... and in Feb 2017 the *SHAttered* attack (CWI and Google) ...

`https://www.theregister.co.uk/2017/02/23/google_first_sha1_collision/`

- generated two different PDF files with the same SHA-1 hash in roughly $2^{63.1}$ SHA-1 evaluations.
- 100,000 times faster than brute force birthday attack
- required equivalent of 6,500 years of single-CPU computations or 110 years of single-GPU computations

The Wikipedia article has a neat summary of the algorithm, its security issues and vularabilities.

SHA-2 family

- SHA-224, 256, 384, 512, 512/224, 512/256 (USA NSA)
- SHA-256, for instance outputs a 256-bit number: $2^{256} \approx 10^{77}$ values; currently recommended for TLS although already attacks are being show to be possible.
- A SHA-256 hash is handled as an array of 8 32-bit words (unsigned integers).
- SHA-512 which works with 64-bit words is coming to be recommended for 64 bit machines.

SHA-256 is considered in more detail below and is in a sense typical of this family of hash functions. SHA512 follows similar logic but a ‘state’ consists of 8 x 64- rather than 32-bit words.

SHA-256

The 256-bit hash is handled as an array of 8 x 32-bit integers. These are called *words* in the literature:

- in C they would have type `unsigned int` or `uint32_t`
- in Java, just `int`

The data is organized as 512-bit (64 byte, 16 word) *blocks*. A high-level view of the process is:

- The hash is initialized;
- There is a *round* for each block:
  - an update to the hash is computed (as 8 words) and
  - added, word-wise, to the hash
- Done, once all blocks have been processed. The hash is returned.

SHA-256 helpers

The SHA-256 algorithm employs some constants -

- `word[8] hashInit`, array of 8 x 32-bit constants to initialise the hash;
- `word[64] roundConst`, array of 64x32-bit constants used in each round.

Some bitwise logic functions -

- `word rotr(word wd, int k) {`
  `return (wd >> k) | (wd << (32-k)); }` - rotate wd k bits to right
- `word ch(word x, word y, word x) {`
  `return (x & y) ^ (~x & z); }` - think ‘choice’
- `word maj(word x, word y, word x) {`
  `return (x & y) ^ (x & z) ^ (y & z); }` - think ‘majority’

SHA-256 helpers

Some ‘magic’ functions used in block (round) processing -

- `word` Σ_0`(word x) {`
  `return rotr(x,2) ^ rotr(x,13) ^ rotr(x,22); }`
- `word` Σ_1`(word x) {`
  `return rotr(x,6) ^ rotr(x,11) ^ rotr(x,25); }`
- `word` σ_0`(word x) {`
  `return rotr(x,7) ^ rotr(x,18) ^ (x >> 3); }`
- `word` σ_1`(word x) {`
  `return rotr(x,17) ^ rotr(x,19) ^ (x >> 10); }`

SHA-256 block setup and hash initialisation

The input data has to be a whole number of 16-word blocks. This contrived by add padding in the following form -

- a 1 bit
- some 0 bits
- a 64-bit unsigned integer: the number of bits of data input.

The number of 0-bits in the padding is just what is needed to get the overall bit size a multiple of 512 (ie, 16 words).

The data does not have to be all input at this stage - it can be input on the fly during block processing rounds but the data length needs to be known in advance to set up the padding.

The hash (array of 8 words) it initialised to a copy of `hashInit`.

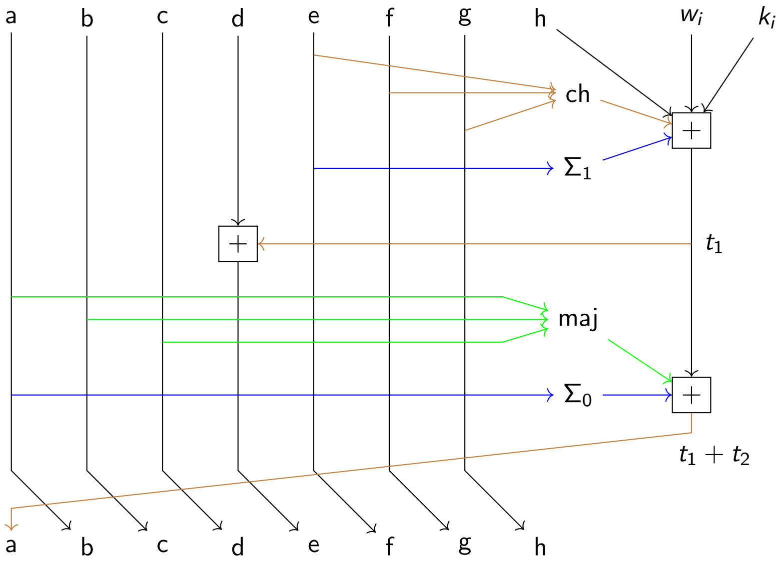
SHA-256 block processing rounds

The data is a whole number of 16-word blocks. For each block,

- a 64-word array `w` is created from the data:
  - `w[0..15]` is copied from the 16 words of the block;
  - for i = 16 to 63 set `w[i] =`
    σ_1`(w[i-2]) + w[i-7] +` σ_0`(w[i-15]) + w[i-16]`
- the hash value from the previous round (in the first round, the initial value) is copied to 8 words excitingly denoted a, b, c, d, e, f, g, h;
- For i = 0 to 63,
  these variables are updated using `w[i]` and `roundConst[i]` as indicated (w_i, k_i) in the diagram below;
- the 64-times updated $a, ..., h$ are added modulo 2^{32} to the hash words:
  `hash[0] += a; hash[1] += b; ...; hash[7] += h;`

NB '+', addition of words, is modulo 2^{32}. Note also we have here a 64x iteration within each block - there are potentially many iterations.

SHA-256 block processing: i^{th} update of $a \dots h$



Further reading

- https://en.wikipedia.org/wiki/MD5
- https://en.wikipedia.org/wiki/SHA-1
- https://www.theregister.co.uk/2017/02/23/google_first_sha1_collision/
- https://en.wikipedia.org/wiki/SHA-2
- http://www.iwar.org.uk/comsec/resources/cipher/sha256-384-512.pdf
- Here is a zip containing java implementations of SHA-1 (mainly of historical interest now!) and SHA-256: http://computing.northumbria.ac.uk/staff/cgmb3/teaching/cryptography/SecureHashAlgs.zip
  `Sha256.java` lines 255-282 cover processing a block; 266-277 correspond to the diagram.