Hashes
MAA507: Part of Lecture 6

Lars Hellström

February 6, 2014

Hash functions

Last time, it was said that a hash function is a function

- from a "large set", such as the set of all finite strings (length not bounded),
- to a "small set", such as the set of integers in some range from 0 to $2^k - 1$ (i.e., k-bit integers).

The name hash was chosen to suggest *chop up and mix*, since many hash functions do this.

Hash function example: Ousterhout hash (weak)

This simplistic hash function multiplies by 9 and adds each new character.

```
static unsigned
HashStringKey(
    Tcl_HashTable *tablePtr,    /* Hash table. */
    void *keyPtr)               /* Key from which to compute hash v
{
    register const char *string = keyPtr;
    register unsigned int result;
    register char c;

    if ((result = UCHAR(*string)) != 0) {
        while ((c = *++string) != 0) {
            result += (result << 3) + UCHAR(c);
        }
    }
    return result;
}
```

Hash table example

The "common" set of HTML attributes have the keys:

```
xml:space class id title dir xml:lang
onclick ondblclick onmousedown onmouseup
onmouseover onmousemove onmouseout
onkeypress onkeydown onkeyup style
```

Adding these to a hash table using the Ousterhout hash results in the following statistics:

17 entries in table, 16 buckets
number of buckets with 0 entries: 6
number of buckets with 1 entries: 4
number of buckets with 2 entries: 5
number of buckets with 3 entries: 1
number of buckets with 4 entries: 0
number of buckets with 5 or more entries: 0
average search distance for entry: 1.5

CRC-32 hash function (not strong)

This is used for checksums in many old communication protocols.

```
unsigned int
digital_update_crc32(unsigned int crc,
                     const unsigned char *data,
                     size_t len)
{
    while (len > 0)
    {
      crc = table[*data ^ ((crc >> 24) & 0xff)] ^ (crc << 8);
      data++;
      len--;
    }
    return crc;
}
```

CRC32 lookup table I

```
static const unsigned int table[256] = {
0x00000000U,0x04C11DB7U,0x09823B6EU,0x0D4326D9U,
0x130476DCU,0x17C56B6BU,0x1A864DB2U,0x1E475005U,
0x2608EDB8U,0x22C9F00FU,0x2F8AD6D6U,0x2B4BCB61U,
0x350C9B64U,0x31CD86D3U,0x3C8EA00AU,0x384FBDBDU,
0x4C11DB70U,0x48D0C6C7U,0x4593E01EU,0x4152FDA9U,
0x5F15ADACU,0x5BD4B01BU,0x569796C2U,0x52568B75U,
0x6A1936C8U,0x6ED82B7FU,0x639B0DA6U,0x675A1011U,
0x791D4014U,0x7DDC5DA3U,0x709F7B7AU,0x745E66CDU,
0x9823B6E0U,0x9CE2AB57U,0x91A18D8EU,0x95609039U,
0x8B27C03CU,0x8FE6DD8BU,0x82A5FB52U,0x8664E6E5U,
0xBE2B5B58U,0xBAEA46EFU,0xB7A96036U,0xB3687D81U,
0xAD2F2D84U,0xA9EE3033U,0xA4AD16EAU,0xA06C0B5DU,
0xD4326D90U,0xD0F37027U,0xDDB056FEU,0xD9714B49U,
0xC7361B4CU,0xC3F706FBU,0xCEB42022U,0xCA753D95U,
0xF23A8028U,0xF6FB9D9FU,0xFBB8BB46U,0xFF79A6F1U,
0xE13EF6F4U,0xE5FFEB43U,0xE8BCCD9AU,0xEC7DD02DU,
0x34867077U,0x30476DC0U,0x3D044B19U,0x39C556AEU,
0x278206ABU,0x23431B1CU,0x2E003DC5U,0x2AC12072U,
0x128E9DCFU,0x164F8078U,0x1B0CA6A1U,0x1FCDBB16U,
0x018AEB13U,0x054BF6A4U,0x0808D07DU,0x0CC9CDCAU,
0x7897AB07U,0x7C56B6B0U,0x71159069U,0x75D48DDEU,
0x6B93DDDBU,0x6F52C06CU,0x6211E6B5U,0x66D0FB02U,
0x5E9F46BFU,0x5A5E5B08U,0x571D7DD1U,0x53DC6066U,
0x4D9B3063U,0x495A2DD4U,0x44190B0DU,0x40D816BAU,
0xACA5C697U,0xA864DB20U,0xA527FDF9U,0xA1E6E04EU,
0xBFA1B04BU,0xBB60ADFCU,0xB6238B25U,0xB2E29692U,
0x8AAD2B2FU,0x8E6C3698U,0x832F1041U,0x87EE0DF6U,
0x99A95DF3U,0x9D684044U,0x902B669DU,0x94EA7B2AU,
0xE0B41DE7U,0xE4750050U,0xE9362689U,0xEDF73B3EU,
0xF3B06B3BU,0xF771768CU,0xFA325055U,0xFEF34DE2U,
0xC6BCF05FU,0xC27DEDE8U,0xCF3ECB31U,0xCBFFD686U,
0xD5B88683U,0xD1799B34U,0xDC3ABDEDU,0xD8FBA05AU,
```

CRC32 lookup table II

```
0x690CE0EEU,0x6DCDFD59U,0x608EDB80U,0x644FC637U,
0x7A089632U,0x7EC98B85U,0x738AAD5CU,0x774BB0EBU,
0x4F040D56U,0x4BC510E1U,0x46863638U,0x42472B8FU,
0x5C007B8AU,0x58C1663DU,0x558240E4U,0x51435D53U,
0x251D3B9EU,0x21DC2629U,0x2C9F00F0U,0x285E1D47U,
0x36194D42U,0x32D850F5U,0x3F9B762CU,0x3B5A6B9BU,
0x0315D626U,0x07D4CB91U,0x0A97ED48U,0x0E56F0FFU,
0x1011A0FAU,0x14D0BD4DU,0x19939B94U,0x1D528623U,
0xF12F560EU,0xF5EE4BB9U,0xF8AD6D60U,0xFC6C70D7U,
0xE22B20D2U,0xE6EA3D65U,0xEBA91BBCU,0xEF68060BU,
0xD727BBB6U,0xD3E6A601U,0xDEA580D8U,0xDA649D6FU,
0xC423CD6AU,0xC0E2D0DDU,0xCDA1F604U,0xC960EBB3U,
0xBD3E8D7EU,0xB9FF90C9U,0xB4BCB610U,0xB07DABA7U,
0xAE3AFBA2U,0xAAFBE615U,0xA7B8C0CCU,0xA379DD7BU,
0x9B3660C6U,0x9FF77D71U,0x92B45BA8U,0x9675461FU,
0x8832161AU,0x8CF30BADU,0x81B02D74U,0x857130C3U,
0x5D8A9099U,0x594B8D2EU,0x5408ABF7U,0x50C9B640U,
0x4E8EE645U,0x4A4FFBF2U,0x470CDD2BU,0x43CDC09CU,
0x7B827D21U,0x7F436096U,0x7200464FU,0x76C15BF8U,
0x68860BFDU,0x6C47164AU,0x61043093U,0x65C52D24U,
0x119B4BE9U,0x155A565EU,0x18197087U,0x1CD86D30U,
0x029F3D35U,0x065E2082U,0x0B1D065BU,0x0FDC1BECU,
0x3793A651U,0x3352BBE6U,0x3E119D3FU,0x3AD08088U,
0x2497D08DU,0x2056CD3AU,0x2D15EBE3U,0x29D4F654U,
0xC5A92679U,0xC1683BCEU,0xCC2B1D17U,0xC8EA00A0U,
0xD6AD50A5U,0xD26C4D12U,0xDF2F6BCBU,0xDBEE767CU,
0xE3A1CBC1U,0xE760D676U,0xEA23F0AFU,0xEEE2ED18U,
0xF0A5BD1DU,0xF464A0AAU,0xF9278673U,0xFDE69BC4U,
0x89B8FD09U,0x8D79E0BEU,0x803AC667U,0x84FBDBD0U,
0x9ABC8BD5U,0x9E7D9662U,0x933EB0BBU,0x97FFAD0CU,
0xAFB010B1U,0xAB710D06U,0xA6322BDFU,0xA2F33668U,
0xBCB4666DU,0xB8757BDAU,0xB5365D03U,0xB1F740B4U,
};
```

Cryptography grade hashes (very strong)

Typical properties of a cryptography grade hash function would be:

- Output appears entirely random.
- If you pick at random any single bit of the input and change it, then any given bit of the output has a probability of 50% to change as well. Different bits of the output appear independent.
- If any single bit of the input is changed, on average half the bits out the output are changed.

However, the quality of hash functions tends to be more a matter of experience and lack of evidence to the contrary, than a matter of rigorous mathematical proofs.

SHA-1 hash function (cryptography grade) I

```
break message into 512-bit chunks
for each chunk
    break chunk into sixteen 32-bit big-endian words w[i]
       (0 ≤ i ≤ 15)

    Extend the sixteen 32-bit words into eighty 32-bit words:
    for i from 16 to 79
        w[i] = (w[i-3] xor w[i-8] xor w[i-14] xor w[i-16])
               leftrotate 1

    Initialize hash value for this chunk:
    a = h0
    b = h1
    c = h2
    d = h3
    e = h4

    Main loop:
```

SHA-1 hash function (cryptography grade) II

```
for i from 0 to 79
    if 0 ≤ i ≤ 19 then
        f = (b and c) or ((not b) and d)
        k = 0x5A827999
    else if 20 ≤ i ≤ 39
        f = b xor c xor d
        k = 0x6ED9EBA1
    else if 40 ≤ i ≤ 59
        f = (b and c) or (b and d) or (c and d)
        k = 0x8F1BBCDC
    else if 60 ≤ i ≤ 79
        f = b xor c xor d
        k = 0xCA62C1D6

    temp = (a leftrotate 5) + f + e + k + w[i]
    e = d ;  d = c
    c = b leftrotate 30
    b = a
    a = temp
```

SHA-1 hash function (cryptography grade) III

```
Add this chunk's hash to result so far:
h0 = h0 + a
h1 = h1 + b
h2 = h2 + c
h3 = h3 + d
h4 = h4 + e
```

SHA-1 examples

$$\begin{aligned}
\mathrm{SHA1}(0) &= \texttt{b6589fc6ab0dc82cf12099d1c2d40ab994e8410c}_{16}\\
\mathrm{SHA1}(1) &= \texttt{356a192b7913b04c54574d18c28d46e6395428ab}_{16}\\
\mathrm{SHA1}(2) &= \texttt{da4b9237bacccdf19c0760cab7aec4a8359010b0}_{16}\\
\mathrm{SHA1}(3) &= \texttt{77de68daecd823babbb58edb1c8e14d7106e83bb}_{16}\\
\mathrm{SHA1}(4) &= \texttt{1b6453892473a467d07372d45eb05abc2031647a}_{16}\\
\mathrm{SHA1}(5) &= \texttt{ac3478d69a3c81fa62e60f5c3696165a4e5e6ac4}_{16}\\
\mathrm{SHA1}(6) &= \texttt{c1dfd96eea8cc2b62785275bca38ac261256e278}_{16}\\
\mathrm{SHA1}(7) &= \texttt{902ba3cda1883801594b6e1b452790cc53948fda}_{16}\\
\mathrm{SHA1}(8) &= \texttt{fe5dbbcea5ce7e2988b8c69bcfdfde8904aabc1f}_{16}\\
\mathrm{SHA1}(9) &= \texttt{0ade7c2cf97f75d009975f4d720d1fa6c19f4897}_{16}\\
\mathrm{SHA1}(10) &= \texttt{b1d5781111d84f7b3fe45a0852e59758cd7a87e5}_{16}\\
\mathrm{SHA1}(11) &= \texttt{17ba0791499db908433b80f37c5fbc89b870084b}_{16}\\
\mathrm{SHA1}(12) &= \texttt{7b52009b64fd0a2a49e6d8a939753077792b0554}_{16}\\
\mathrm{SHA1}(13) &= \texttt{bd307a3ec329e10a2cff8fb87480823da114f8f4}_{16}\\
\mathrm{SHA1}(14) &= \texttt{fa35e192121eabf3dabf9f5ea6abdbcbc107ac3b}_{16}
\end{aligned}$$

A sense of injectivity

The SHA-1 hash function maps a countably infinite set (the set of all strings) into a finite set (the set of all 160-bit words, which has 2^{160} elements). Hence it is provably not injective.

On the other hand, there are *no* known counterexamples to the claim 'the SHA-1 hash function is injective'—nobody knows any pair of strings s_1 and s_2 such that $\mathrm{SHA1}(s_1) = \mathrm{SHA1}(s_2)$!

However, it has been estimated that it could within the foreseeable future be practical to seek such a counterexample (even if it would cost several million USD to discover just one), so SHA-1 is being phased out from cryptographical protocols.

Hash function applications

- Hash tables (mentioned last time and above)
- Secondary key generation
- Random number generation
- Authentication and digital signatures

Secondary key generation

Confidential communication between two parties is based on them knowing some shared secret—the *master key*.
Using the master key as encryption key for the data communicated is problematic, because if an attacker gathers enough material encrypted using the *same* key (especially if parts of the plaintext are easy to guess), then it becomes feasible for the attacker to figure out the key. Therefore bulk material is encrypted using secondary keys which are switched periodically (so-called rekeying).
A convenient way to generate such secondary keys is to form the string concatenation

$$\langle \textit{master key} \rangle \ \langle \textit{class} \rangle \ \langle \textit{running counter} \rangle$$

and then use the hash of that as the new secondary key.
Here, $\langle \textit{class} \rangle$ is some identifier for the *class* of key that is being generated (so that one doesn't use the same key for two different things) and $\langle \textit{running counter} \rangle$ is simply 1, 2, 3, etc., incremented each time one rekeys.

Random number generation

If H is some hash function, then one way of generating a sequence $(x_n)_{n\in\mathbb{N}}$ of pseudorandom numbers could be to use the formula

$$x_n = H(s.n)$$

where s is some *seed value* and '.' denotes concatenation. For strong hashes, this generates high quality randomness, but it is computationally more expensive than "ordinary" pseudo-random number generators.

Hashes may also be used to improve the quality of randomness gathered from other sources. Natural randomness (e.g. the user hitting keys or moving the mouse at random) tends to have a skew distribution, but as long as one gathers enough low-grade random data and feeds it through a strong hash, the output will be high grade random data (even if there are fewer bits of it).

Authentication

Recall that an asymmetric encryption scheme (also known as *public-key* encryption scheme) consists of two bijective functions:

- D, which is made public (essentially the *public key*),
- $E = D^{-1}$, which is kept secret (the *private key*).

(The most famous system for creating such pairs of functions is RSA, but there are others.)

This can be used for authenticating a party as follows:

1. You known the D function of the party P you want to communicate with.
2. When someone claims to be P, you pick a random number x and ask what $E(x)$ is.
3. You get the answer y, and check whether $D(y) = x$; only someone who knows the secret E can compute the true $y = E(x)$, and P wouldn't tell this to anyone else.

Note that *you* don't need to know E, and thus cannot impersonate P.

Digital signatures

Now suppose P wishes to sign a document m. P's signature is by definition something that only P can produce, but others can recognise.
To sign m, P can use a hash H and the encryption scheme (D, E) as follows:

1. Compute the hash value $h = H(m)$.
2. Append $s = E(h)$ to the document. This is the signature.

To verify the signature, one would check whether $H(m) = D(s)$.

HTTPS website certificates are based on this idea.

Note that s would also seem to be the signature of any document m' satisfying $H(m') = H(m)$, so this requires a cryptography-grade hash.