CSc 8370

Cryptographic Hash Functions and Digital Signatures

(Trappe & Washington Chapter 8; Pfleiger p 79, Viega p 457-464, Schneier chapters 18,20; Handbook of Applied Cryptography chapters 9-11 - no you don't have to read all of these.)

Digital Signatures are needed to ensure that transactions between computers are legal. It is necessary to ensure that the data are not being corrupted or compromised. This is somewhat different from error correcting codes where we are protecting data against random errors, in that we are also protecting against the deliberate changing of the data by an interested party.

- Authenticity - The signature should prove that the signing party actually signed the document. In essence, it has to be like "one-way" function where only the signing party knows how to generate it, but another party has to be able to validate that that knowledge was used to sign.
- Non-repudiation - It is critical that one party cannot claim to have not signed the document.
- Forging - The signature should be difficult to forge. (It's not that hard to forge pen and paper signatures, just turn it upside down and practice). Since this could be legally binding, you don't want to find a document bearing your signature that you didn't sign.
- Non-alterable - The signature should prevent the document from being changed. This is why in the pen and paper world you initial changes to a contract.
- Non-portable - The signature should only be valid for one document.

This is an active research area with an extensive literature. An instance of the ElGamal signature scheme is used as the DSA (or Digital Signature Algorithm) for the US.

Initialization.

1) generate a large prime, p, and the generator α of the multiplicative group Zp (i.e. α is relatively prime wrt p and $\alpha^n \bmod p$ generates all values from 1 to p-1).
2) select a random integer $1 < a < p-1$.
3) compute $y = \alpha^a \bmod p$
4) The public key is (p,α,y), the private key is a.

To sign a document:

1) select a random secret integer $1< k < p-1$ where $\gcd(k,p-1) = 1$ (i.e. k is relatively prime wrt p-1).
2) calculate $r = \alpha^k \bmod p$.
3) calculate $k^{-1} \bmod (p-1)$

4) calculate $s = k^{-1} \{h(m) - ar\} \bmod (p-1)$ where h(m) is a hash of the message
5) the signature is the pair r,s

To validate the signature:
1) Obtain a ***valid*** copy of the public key (p,α,y).
2) verify $1 < r , p-1$. r outside of this range is invalid.
3) calculate $v1 = y^r r^s \bmod p$
4) calculate h(m) and $v2 = \alpha^{h(m)} \bmod p$
5) if v1 and v2 are the same the signature is valid.

How does this work?

$y = \alpha^a$; $y^r = \alpha^{ar}$
$r = \alpha^k$; $r^s = \alpha^{k \text{ (kinverse) } \{h - ar\} \bmod p-1}$
since $\alpha^{p-1} = 1$ the mod (p-1) is just to make life easier during calculations and
$r^s = \alpha^{h-ar}$
therefore
$y^r r^s \bmod p = \alpha^h$

Note
- The copy has to be a valid copy, there could still be a man-in-the-middle attack.
- The security is equivalent to the discrete logarithm problem so the same caveats about weak bases apply. (i.e. if p-1 has lots of small factors we are in trouble).
- Each message must be signed with a new random number k. h(m) is known and therefore differences between s generated with different messages, but the same integer k can be used to estimate k (just like inverting linear congruential rng's). Once k is known a is trivial to find.

Hash algorithms.

The hash performs a critical function in the signing algorithm. It asserts that the message signed and the message received are the same message. It binds the signature to a specific message.

Hash functions take a lot of data and generate are relatively short digest from it. There are two fundamental security issues with hash functions. 1) the key used in the hash should not be recoverable from the hash (can't invert), and 2) it must be difficult to find two messages that have the same hash (no collisions).

It is possible to generate messages that have collisions. For example, if a 32-bit hash was used (i.e. the hash was an integer between 0 and $2^{32}-1$) then if a message were generated with 32 positions where bits could be changed without altering the message (e.g. swapping space and tab, hidden non-printing characters) then these changes could be

enumerated until a collision was found.

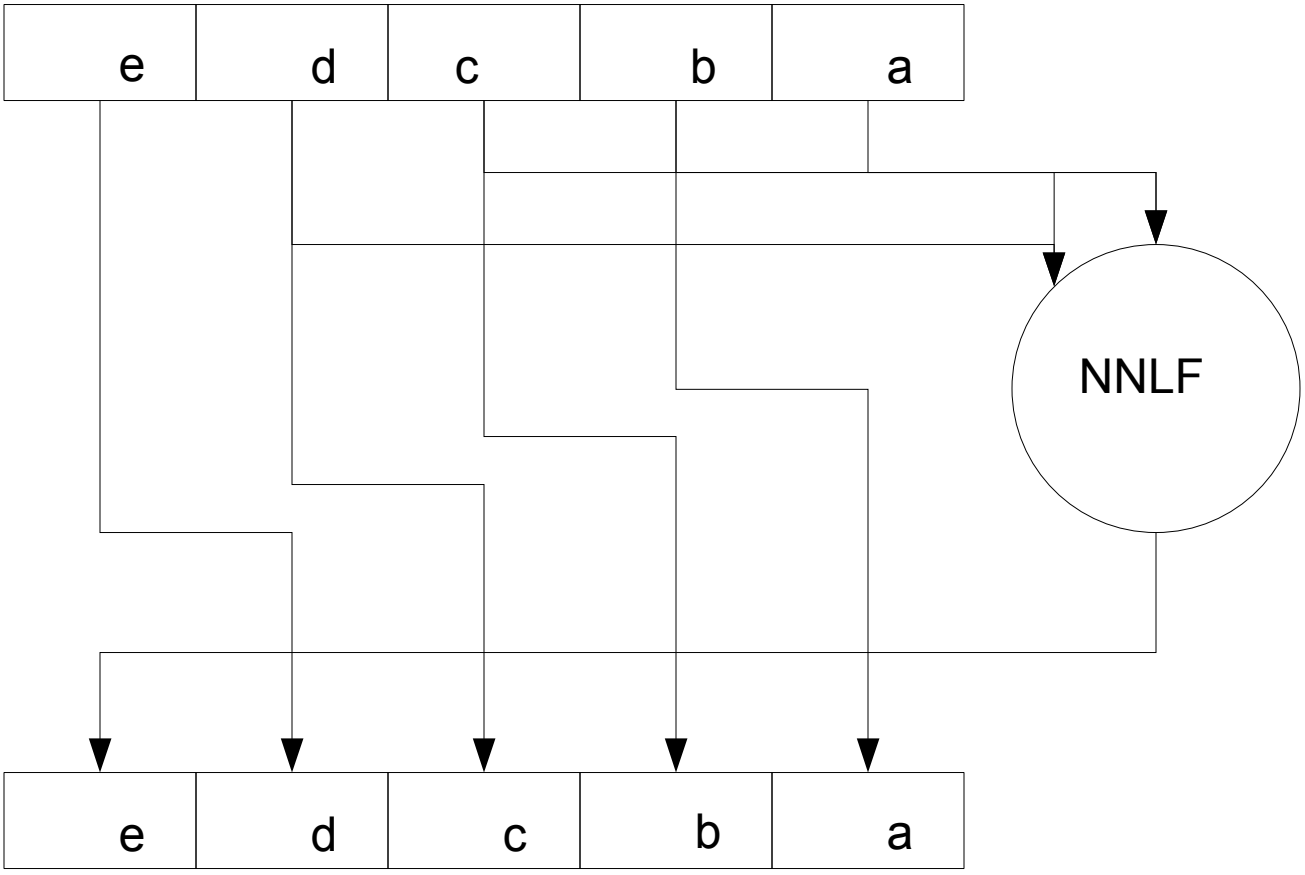
Practical hash functions generate large integers to avoid this (MD5 is 128 bit, SHA is 160 bit), and use sufficiently expensive operations to make the calculation slow.

Some Hash Functions:

1) DES For passwords, use the password as the key and use DES to encrypt a constant several times. The output is then used for authentication by comparing it with a saved value. Doing this over a chained system can be used as a hash.

2) MD5. Similar architecture

3) SHA Fiestel-like algorithm to produce a 160 bit hash. The individual steps are not invertible.



Discussion questions:

1) It is possible to introduce non-printing macro expressions in a certain word processor. These have the effect of changing the printed text. For example:

   if(date < 12/24/08) {"i owe you"} else {"you owe me"} $1,000,000.00

   would print i owe you $1,000,000.00 before 12/24/08 and you owe me afterwards. Since the original document would pass any current digital signature algorithm, the potential for fraud is quite high.

   How would we design a protocol to avoid this?

2) The general ElGamal Algorithm requires that a valid copy of ($p\alpha$,y) be passed to the verifying party. How would you attack this? What approaches could secure it?

```
/*
 * A JavaScript implementation of the RSA Data Security, Inc. MD4 Message
 * Digest Algorithm, as defined in RFC 1320.
 * Version 2.0 Copyright (C) Jerrad Pierce, Paul Johnston 1999 - 2002.
 * Other contributors: Greg Holt, Ydnar
 * Distributed under the BSD License
 * See http://pajhome.org.uk/crypt/md5 for more info.
 */

/*
 * Configurable variables. You may need to tweak these to be compatible with
 * the server-side, but the defaults work in most cases.
 */
var hexcase = 0   /* hex output format. 0 - lowercase; 1 - uppercase        */
var b64pad  = ""  /* base-64 pad character. "=" for strict RFC compliance   */
var chrsz   = 8   /* bits per input character. 8 - ASCII; 16 - Unicode      */

/*
 * These are the functions you'll usually want to call
 */
function hex_md4(s){ return binl2hex(core_md4(str2binl(s), s.length * chrsz)) }
function b64_md4(s){ return binl2b64(core_md4(str2binl(s), s.length * chrsz)) }
function hex_hmac_md4(key, data) { return binl2hex(core_hmac_md4(key, data)) }
function b64_hmac_md4(key, data) { return binl2b64(core_hmac_md4(key, data)) }

/* Backwards compatibility - same as hex_md4() */
function calcMD4(s){ return binl2hex(core_md4(str2binl(s), s.length * chrsz)) }

/*
 * Perform a simple self-test to see if the VM is working
 */
function md4_vm_test()
{
  return hex_md4("abc") == "a448017aaf21d8525fc10ae87aa6729d"
}

/*
 * Calculate the MD4 of an array of little-endian words, and a bit length
 */
function core_md4(x, len)
{
  /* append padding */
  x[len >> 5] |= 0x80 << (len % 32)
  x[(((len + 64) >>> 9) << 4) + 14] = len

  var a =  1732584193
  var b = -271733879
  var c = -1732584194
  var d =  271733878

  for(var i = 0; i < x.length; i += 16)
  {
    var olda = a
    var oldb = b
    var oldc = c
    var oldd = d

    a = ff(a, b, c, d, x[i+ 0], 3 )
    d = ff(d, a, b, c, x[i+ 1], 7 )
    c = ff(c, d, a, b, x[i+ 2], 11)
    b = ff(b, c, d, a, x[i+ 3], 19)
    a = ff(a, b, c, d, x[i+ 4], 3 )
    d = ff(d, a, b, c, x[i+ 5], 7 )
    c = ff(c, d, a, b, x[i+ 6], 11)
    b = ff(b, c, d, a, x[i+ 7], 19)
    a = ff(a, b, c, d, x[i+ 8], 3 )
    d = ff(d, a, b, c, x[i+ 9], 7 )
    c = ff(c, d, a, b, x[i+10], 11)
    b = ff(b, c, d, a, x[i+11], 19)
    a = ff(a, b, c, d, x[i+12], 3 )
    d = ff(d, a, b, c, x[i+13], 7 )
    c = ff(c, d, a, b, x[i+14], 11)
    b = ff(b, c, d, a, x[i+15], 19)

    a = gg(a, b, c, d, x[i+ 0], 3 )
    d = gg(d, a, b, c, x[i+ 4], 5 )
    c = gg(c, d, a, b, x[i+ 8], 9 )
```

```
    b = gg(b, c, d, a, x[i+12], 13)
    a = gg(a, b, c, d, x[i+ 1], 3 )
    d = gg(d, a, b, c, x[i+ 5], 5 )
    c = gg(c, d, a, b, x[i+ 9], 9 )
    b = gg(b, c, d, a, x[i+13], 13)
    a = gg(a, b, c, d, x[i+ 2], 3 )
    d = gg(d, a, b, c, x[i+ 6], 5 )
    c = gg(c, d, a, b, x[i+10], 9 )
    b = gg(b, c, d, a, x[i+14], 13)
    a = gg(a, b, c, d, x[i+ 3], 3 )
    d = gg(d, a, b, c, x[i+ 7], 5 )
    c = gg(c, d, a, b, x[i+11], 9 )
    b = gg(b, c, d, a, x[i+15], 13)

    a = hh(a, b, c, d, x[i+ 0], 3 )
    d = hh(d, a, b, c, x[i+ 8], 9 )
    c = hh(c, d, a, b, x[i+ 4], 11)
    b = hh(b, c, d, a, x[i+12], 15)
    a = hh(a, b, c, d, x[i+ 2], 3 )
    d = hh(d, a, b, c, x[i+10], 9 )
    c = hh(c, d, a, b, x[i+ 6], 11)
    b = hh(b, c, d, a, x[i+14], 15)
    a = hh(a, b, c, d, x[i+ 1], 3 )
    d = hh(d, a, b, c, x[i+ 9], 9 )
    c = hh(c, d, a, b, x[i+ 5], 11)
    b = hh(b, c, d, a, x[i+13], 15)
    a = hh(a, b, c, d, x[i+ 3], 3 )
    d = hh(d, a, b, c, x[i+11], 9 )
    c = hh(c, d, a, b, x[i+ 7], 11)
    b = hh(b, c, d, a, x[i+15], 15)

    a = safe_add(a, olda)
    b = safe_add(b, oldb)
    c = safe_add(c, oldc)
    d = safe_add(d, oldd)

  }
  return Array(a, b, c, d)

  /*
   * These functions implement the basic operation for each round of the
   * algorithm.
   */
  function cmn(q, a, b, x, s, t)
  {
    return safe_add(rol(safe_add(safe_add(a, q), safe_add(x, t)), s), b);
  }
  function ff(a, b, c, d, x, s)
  {
    return cmn((b & c) | ((~b) & d), a, 0, x, s, 0);
  }
  function gg(a, b, c, d, x, s)
  {
    return cmn((b & c) | (b & d) | (c & d), a, 0, x, s, 1518500249);
  }
  function hh(a, b, c, d, x, s)
  {
    return cmn(b ^ c ^ d, a, 0, x, s, 1859775393);
  }
}

/*
 * Calculate the HMAC-MD4, of a key and some data
 */
function core_hmac_md4(key, data)
{
  var bkey = str2binl(key)
  if(bkey.length > 16) bkey = core_md4(bkey, key.length * chrsz)

  var ipad = Array(16), opad = Array(16)
  for(var i = 0; i < 16; i++)
  {
    ipad[i] = bkey[i] ^ 0x36363636
    opad[i] = bkey[i] ^ 0x5C5C5C5C
  }

  var hash = core_md4(ipad.concat(str2binl(data)), 512 + data.length * chrsz)
```

```
  return core_md4(opad.concat(hash), 512 + 128)
}

/*
 * Add integers, wrapping at 2^32. This uses 16-bit operations internally
 * to work around bugs in some JS interpreters.
 */
function safe_add(x, y)
{
  var lsw = (x & 0xFFFF) + (y & 0xFFFF)
  var msw = (x >> 16) + (y >> 16) + (lsw >> 16)
  return (msw << 16) | (lsw & 0xFFFF)
}

/*
 * Bitwise rotate a 32-bit number to the left.
 */
function rol(num, cnt)
{
  return (num << cnt) | (num >>> (32 - cnt))
}

/*
 * Convert a string to an array of little-endian words
 * If chrsz is ASCII, characters >255 have their hi-byte silently ignored.
 */
function str2binl(str)
{
  var bin = Array()
  var mask = (1 << chrsz) - 1
  for(var i = 0; i < str.length * chrsz; i += chrsz)
    bin[i>>5] |= (str.charCodeAt(i / chrsz) & mask) << (i%32)
  return bin
}

/*
 * Convert an array of little-endian words to a hex string.
 */
function binl2hex(binarray)
{
  var hex_tab = hexcase ? "0123456789ABCDEF" : "0123456789abcdef"
  var str = ""
  for(var i = 0; i < binarray.length * 4; i++)
  {
    str += hex_tab.charAt((binarray[i>>2] >> ((i%4)*8+4)) & 0xF) +
           hex_tab.charAt((binarray[i>>2] >> ((i%4)*8  )) & 0xF)
  }
  return str
}

/*
 * Convert an array of little-endian words to a base-64 string
 */
function binl2b64(binarray)
{
  var tab = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
  var str = ""
  for(var i = 0; i < binarray.length * 4; i += 3)
  {
    var triplet = (((binarray[i   >> 2] >> 8 * ( i   %4)) & 0xFF) << 16)
                | (((binarray[i+1 >> 2] >> 8 * ((i+1)%4)) & 0xFF) << 8 )
                |  ((binarray[i+2 >> 2] >> 8 * ((i+2)%4)) & 0xFF)
    for(var j = 0; j < 4; j++)
    {
      if(i * 8 + j * 6 > binarray.length * 32) str += b64pad
      else str += tab.charAt((triplet >> 6*(3-j)) & 0x3F)
    }
  }
  return str;
}
```

Homework/classwork problem set 2. 10/06/09

1) The 5 messages (cipher1,..., cipher5) are on qubit in ~cscrwh/crypto. These were encrypted by rc4 Which of these were encrypted with the same keys? (we covered a test that can be used to answer this, implement it and try it out – the keys are not in the words file that we used before).