**_Benjamin Fenster_**
CS-266 (Ling)
19 December 2005

# Notes – Hash Functions

I. Introduction
   a. Many things done with secret key cryptography can also be done with hash functions
   b. These are one-way functions. h(m) = d, but there is no h'(d) such that h'(d) = m
   c. Example: Take n, look at the middle digit.
   d. Given the computational power we have available, cannot find ($m_1$, $m_2$) such that $d_1 = d_2$
   e. The size of d is fixed; the size of m is not.
   f. With a hash of m bits, would need to search $2^{m/2}$ messages at random to find two with the same hash (on average)
   g. Could potentially generate two such messages, but it's a lot of work to compute all those hash values
   h. Convention: Use a 128-bit hash
   i. A good hash function should look at every bit of the input, so each will be different
      i. MD2 did this by splitting into bytes, since that's what the hardware supported
      ii. MD4, MD5, SHS are all 32-bit for the same reason.

II. Randomness
   a. We want each bit of the output to be 1 half the time, so no information can be inferred
   b. Sending two very similar inputs through should yield two completely different results.
   c. Monte Carlo Simulations
      i. Want to simulate random variables
      ii. Usually picked in a sequence (though a lengthy one)
      iii. Average = 0.5, uniform distribution
      iv. Completely predictable, but it's not a secret
   d. For picking keys:
      i. Want numbers that can't be predicted
      ii. Don't want to rely on time – generating two numbers simultaneously can yield two completely different numbers
      iii. Usually use audio/video (or quantum computing) to get random data)

III. Authentication
   a. Generate some "random" number $r_A$; send plaintext to the other party (B)
   b. B computes the hash of $K_{AB}|r_A$.
   c. Don't just want $h(K_{AB})$ because then somebody could observe the hash and use it themselves
   d. A can then compute the same thing and compare the results
   e. There's no need to decrypt anything, but it still proves B has the key

IV. MIC
   a. Message Authentication / Integrity Code
   b. If you compute MD(m) and send that to verify integrity because anybody could calculate MD(m') and use that to spoof a message
   c. Could use $MD(K_{AB} \mid m)$ but the way the hash works (from beginning to end, one chunk at a time), Trudy could add more text to the end without knowing the key (just continue the computation
   d. Use $h(k_{AB} \mid m \mid K_{AB})$

V. One-Time Pad
   a. $b_1 = MD(K_{AB} \mid IV)$
   b. $b_2 = MD(K_{AB} \mid b_1)$ et cetera
   c. Could even use $b_2 = MD(K_{AB} \mid c_1)$ for $c_1 = b_1 \oplus m_1$
   d. So it's possible to do encryption using just a hash function
   e. Why does this matter? Because of export controls, one cannot export encryption algorithms, but could use a hash function instead.

VI. UNIX Passwords
   a. Exactly eight bytes (if you enter more than that it's ignored)
   b. Create a salt = 12 bit number (random)
   c. Store salt | MD(salt | password)

d. Break the password into $m_1|m_2|\ldots|m_n$
e. Encrypt 0 with $m_1$ as the key, encrypt that with $m_2$ as the key, et cetera
f. It is one-way, but it's not a very good hash function

VII. MD2
a. One of the earliest hash functions
b. Gives 128 bit results and does computations as bytes
c. Algorithm
    i. Add bytes to the message until it's a multiple of 16 butes
        1. Add 16 – (r mod 16) butes of (r mod 16) where r is the number of bytes in the original message
        2. Always add bytes, even if it's already a multiple of 16.
    ii. Append an insecure MD2 checksum to the end
        1. Initialized to zero, 16 bytes long.
        2. For each byte in the message, $c_n = \pi(m_{nk} \oplus c_{n-1}) \oplus c_n$
        3. Where $\pi$ is a permutation supposedly based on the actual number $\pi$
        4. Append the checksum to the end of the message
    iii. Process the whole message
        1. Operate on 16-byte chunks
        2. Have a 48-byte quantity (digest | chunk | digest $\oplus$ chunk)
        3. Do 18 passes over q, each time updating one byte
            a. $c_{-1} = 0$ initially, then $c_{-1} = c_{47}$ + pass number mod 256
            b. $c_n = c_n \oplus \pi(c_{n-1})$
            c. Concept: Update each byte based on the formulae above
            d. After pass 17, the working digest becomes the digest for the beginning of the next chunk.
        4. A problem: Could end up doing two passes, but don't need to! Compute the checksum as you go, then feed it into the last chunk in the final pass