MAC, HMAC, Hash functions, DSA, SSL

Vinod Ganapathy

Message Authentication

- message authentication is concerned with:
  - protecting the integrity of a message
  - validating identity of originator
  - non-repudiation of origin (dispute resolution)
- will consider the security requirements
- then three alternative functions used:
  - message encryption
  - message authentication code (MAC)
  - hash function

Message Encryption

- message encryption by itself also provides a measure of authentication
- if symmetric encryption is used then:
  - receiver know sender must have created it
  - since only sender and receiver now key used
  - know content cannot have been altered
  - if message has suitable structure, redundancy or a checksum to detect any changes

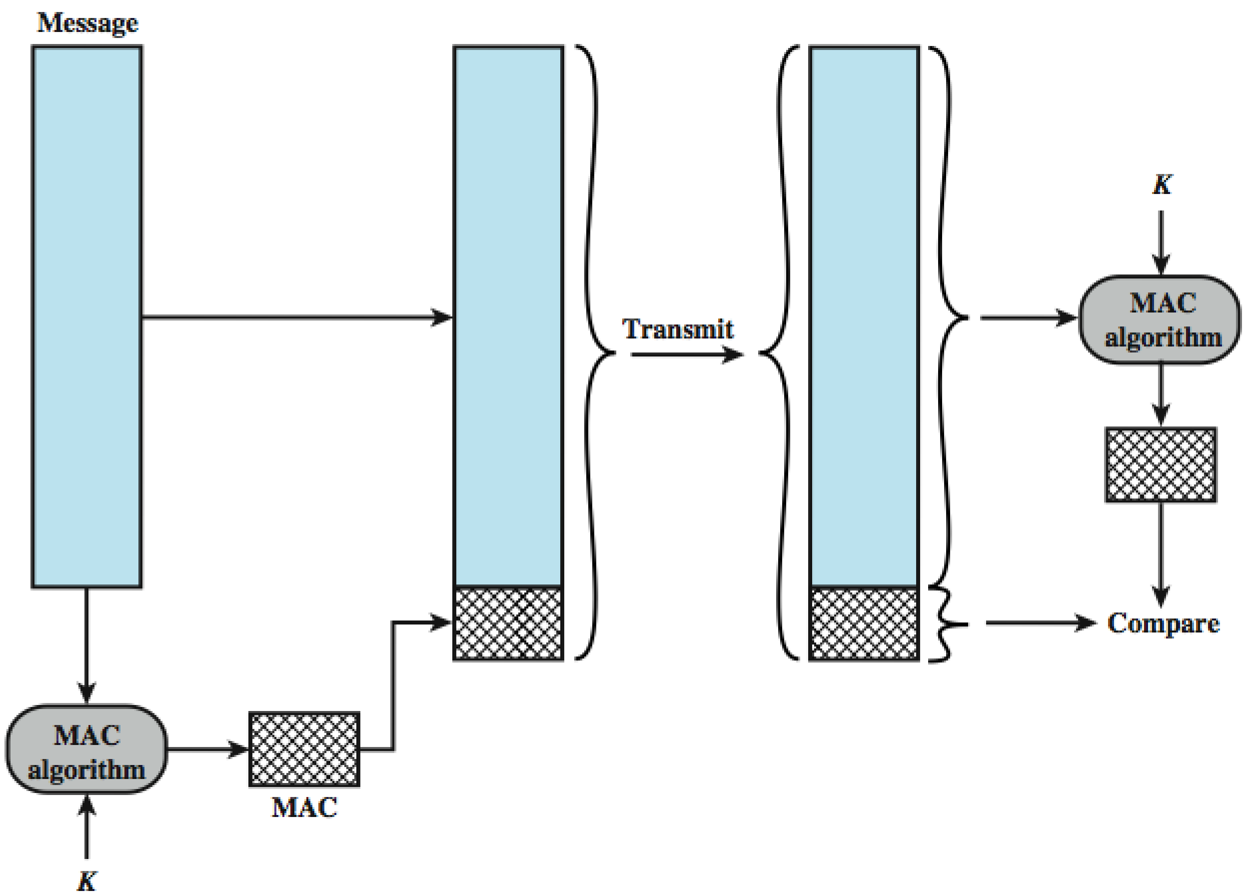
Message Encryption

- if public-key encryption is used:
  - encryption provides no confidence of sender
  - since anyone potentially knows public-key
  - however if
    - sender **signs** message using their private-key
    - then encrypts with recipients public key
    - have both secrecy and authentication
  - again need to recognize corrupted messages
  - but at cost of two public-key uses on message

Message Authentication Code (MAC)

- generated by an algorithm that creates a small fixed-sized block
  - depending on both message and some key
  - like encryption though need not be reversible
- appended to message as a **signature**
- receiver performs same computation on message and checks it matches the MAC
- provides assurance that message is unaltered and comes from sender

Message Authentication Codes



Message Authentication Codes

- as shown the MAC provides authentication
- can also use encryption for secrecy
  - generally use separate keys for each
  - can compute MAC either before or after encryption
  - is generally regarded as better done before
- why use a MAC?
  - sometimes only authentication is needed
  - sometimes need authentication to persist longer than the encryption (eg. archival use)
- note that a MAC is not a digital signature

MAC Properties

- a MAC is a cryptographic checksum

  $MAC = C_K(M)$

  - condenses a variable-length message M
  - using a secret key K
  - to a fixed-sized authenticator
- is a many-to-one function
  - potentially many messages have same MAC
  - but finding these needs to be very difficult

Requirements for MACs

- taking into account the types of attacks
- need the MAC to satisfy the following:
  1. knowing a message and MAC, is infeasible to find another message with same MAC
  2. MACs should be uniformly distributed
  3. MAC should depend equally on all bits of the message

Using Symmetric Ciphers for MACs

- can use any block cipher chaining mode and use final block as a MAC
- **Data Authentication Algorithm (DAA)** is a widely used MAC based on DES-CBC
  - using IV=0 and zero-pad of final block
  - encrypt message using DES in CBC mode
  - and send just the final block as the MAC
    - or the leftmost M bits ($16 \leq M \leq 64$) of final block

Digital Signatures

- have looked at message authentication
  - but does not address issues of lack of trust
- digital signatures provide the ability to:
  - verify author, date & time of signature
  - authenticate message contents
  - be verified by third parties to resolve disputes
- hence include authentication function with additional capabilities

Digital Signature Properties

- must depend on the message signed
- must use information unique to sender
  - to prevent both forgery and denial
- must be relatively easy to produce
- must be relatively easy to recognize & verify
- be computationally infeasible to forge
  - with new message for existing digital signature
  - with fraudulent digital signature for given message
- be practical save digital signature in storage

Direct Digital Signatures

- involve only sender & receiver
- assumed receiver has sender's public-key
- digital signature made by sender signing entire message or hash with private-key
- can encrypt using receivers public-key
- important that sign first then encrypt message & signature
- security depends on sender's private-key

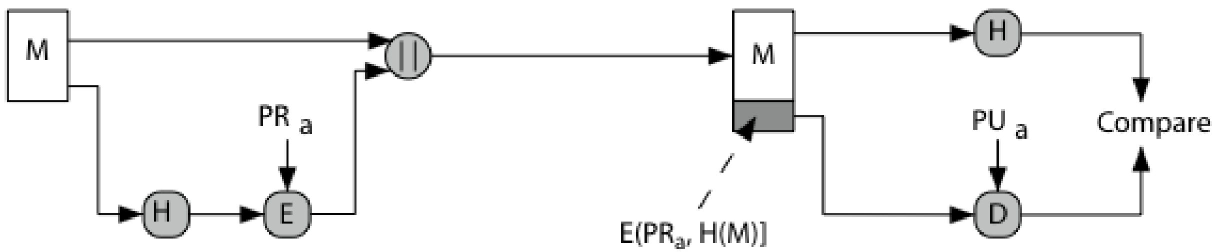
Digital Signature Standard (DSS)

- US Govt approved signature scheme FIPS 186
- uses the SHA hash algorithm
- designed by NIST & NSA in early 90's
- DSS is the standard, DSA is the algorithm
- creates a 320 bit signature, but with 512-1024 bit security
- security depends on difficulty of computing discrete logarithms

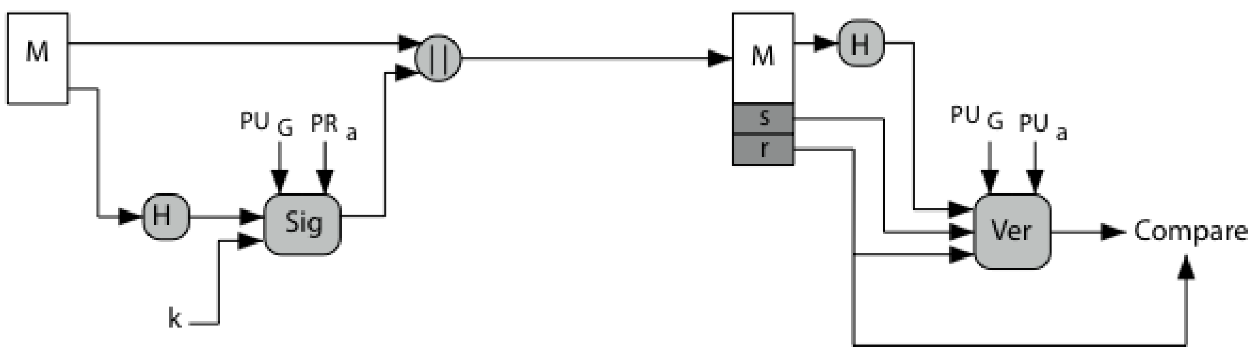
Digital Signature Algorithm (DSA)

- creates a 320 bit signature
- with 512-1024 bit security
- smaller and faster than RSA
- a digital signature scheme only
- security depends on difficulty of computing discrete logarithms
- variant of ElGamal & Schnorr schemes

Digital Signature Algorithm (DSA)



(a) RSA Approach



(b) DSS Approach

Digression - Discrete Logarithms

- the inverse problem to exponentiation is to find the **discrete logarithm** of a number modulo p
- that is to find x such that $y = g^x \pmod{p}$
- this is written as $x = \log_g y \pmod{p}$
- if g is a **primitive root** then it always exists, otherwise it may not, eg.

$x = \log_3 4 \bmod 13$ has no answer

$x = \log_2 3 \bmod 13 = 4$ by trying successive powers

- whilst exponentiation is relatively easy, finding discrete logarithms is generally a **hard** problem

DSA Key Generation

- have shared global public key values (p,q,g):
  - choose q, a 160 bit
  - choose a large prime $p = 2^L$
    - where L= 512 to 1024 bits and is a multiple of 64
    - and q is a prime factor of $(p-1)$
  - choose $g = h^{(p-1)/q}$
    - where $h<p-1$, $h^{(p-1)/q} \pmod{p} > 1$
- users choose private & compute public key:
  - choose $PR_a < q$
  - compute $PU_a = g^{\{PR_a\}} \pmod{p}$

DSA Signature Creation

- to **sign** a message M the sender:
  - generates a random signature key k, $k<q$
  - k must be random, be destroyed after use, and never be reused
- then computes signature pair:

$$r = (g^k \pmod p) \pmod q$$

$$s = k^{-1}.(H(M) + PR_a.r) \pmod q$$

- sends signature (r,s) with message M

DSA Signature Verification

- having received M & signature (r, s)
- to **verify** a signature, recipient computes:

$$w = s^{-1} \pmod{q}$$

$$u1 = (H(M).w) \pmod{q}$$

$$u2 = (r.w) \pmod{q}$$

$$v = (g^{u1}.PU_a{}^{u2} \pmod{p}) \pmod{q}$$

- if $v=r$ then signature is verified
- Why?

Recall $PU_a = g^{\{PR_a\}}$mod p

Substitute u1 and u2 in the value of v

$$v = (g^{\{H(M).w\ mod\ q\}}.\, g^{\{PR_a.r.w\ mod\ q\}} mod\ p)\ mod$$

$$v = (g^{\{(H(M)+PR_a.r).w\ mod\ q} mod\ p)\ mod\ q$$

But $w = s^{-1} (mod\ q)$, i.e., $ws = 1 (mod\ q)$

And $s = k^{-1}.(H(M) + PR_a.r) (mod\ q)$

So, $v = (g^{\{s.k.w.\}modq} \mathrm{modp})\ \mathrm{modq}$

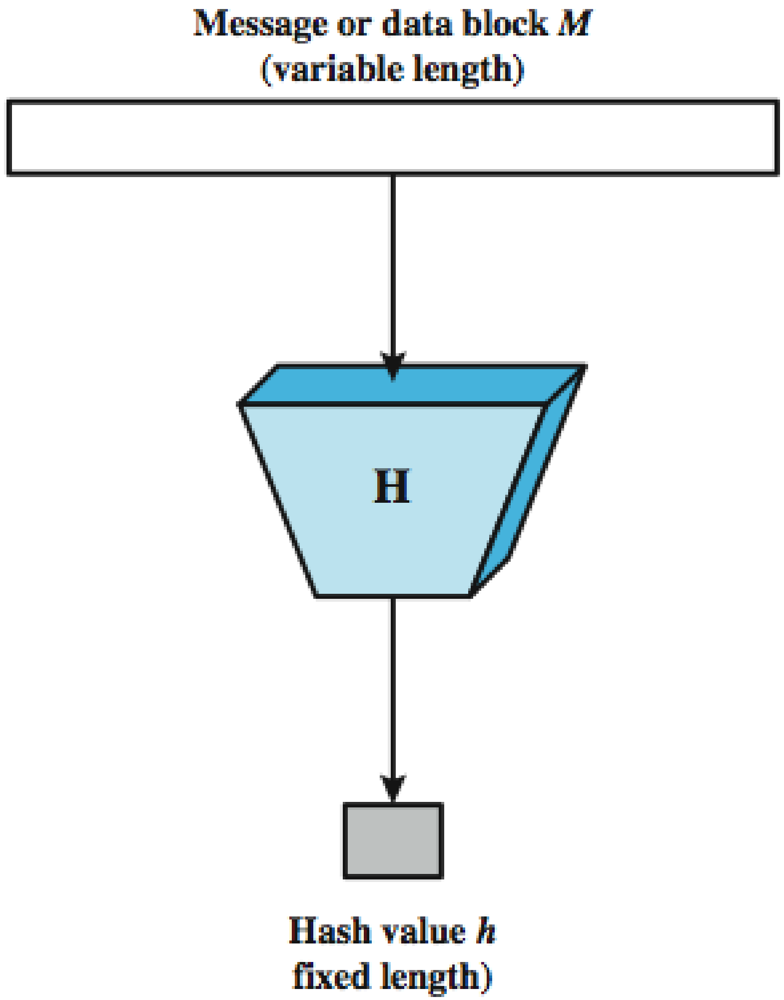
But $s.k.w\ mod\ q = k$

So $v = (g^k mod\ p\ )mod\ q = r$

Hash Algorithms

- Hash Functions
  - condense arbitrary size message to fixed size
  - by processing message in blocks
  - through some compression function
  - either custom or block cipher based
- Examples:
  - MD4, MD5, SHA1

Secure Hash Functions



Hash Function Requirements

- applied to any size data
- H produces a fixed-length output.
- H(x) is relatively easy to compute for any given x
- one-way property
  - computationally infeasible to find x such that H(x) = h
- weak collision resistance
  - computationally infeasible to find $y \neq x$ such that H(y) = H(x)
- strong collision resistance
  - computationally infeasible to find any pair (x, y) such that H(x) = H(y)

Hash Algorithms

- see similarities in the evolution of hash functions & block ciphers
  - increasing power of brute-force attacks
  - leading to evolution in algorithms
  - from DES to AES in block ciphers
  - from MD4 & MD5 to SHA-1 & RIPEMD-160 in hash algorithms
- likewise tend to use common iterative structure as do block ciphers

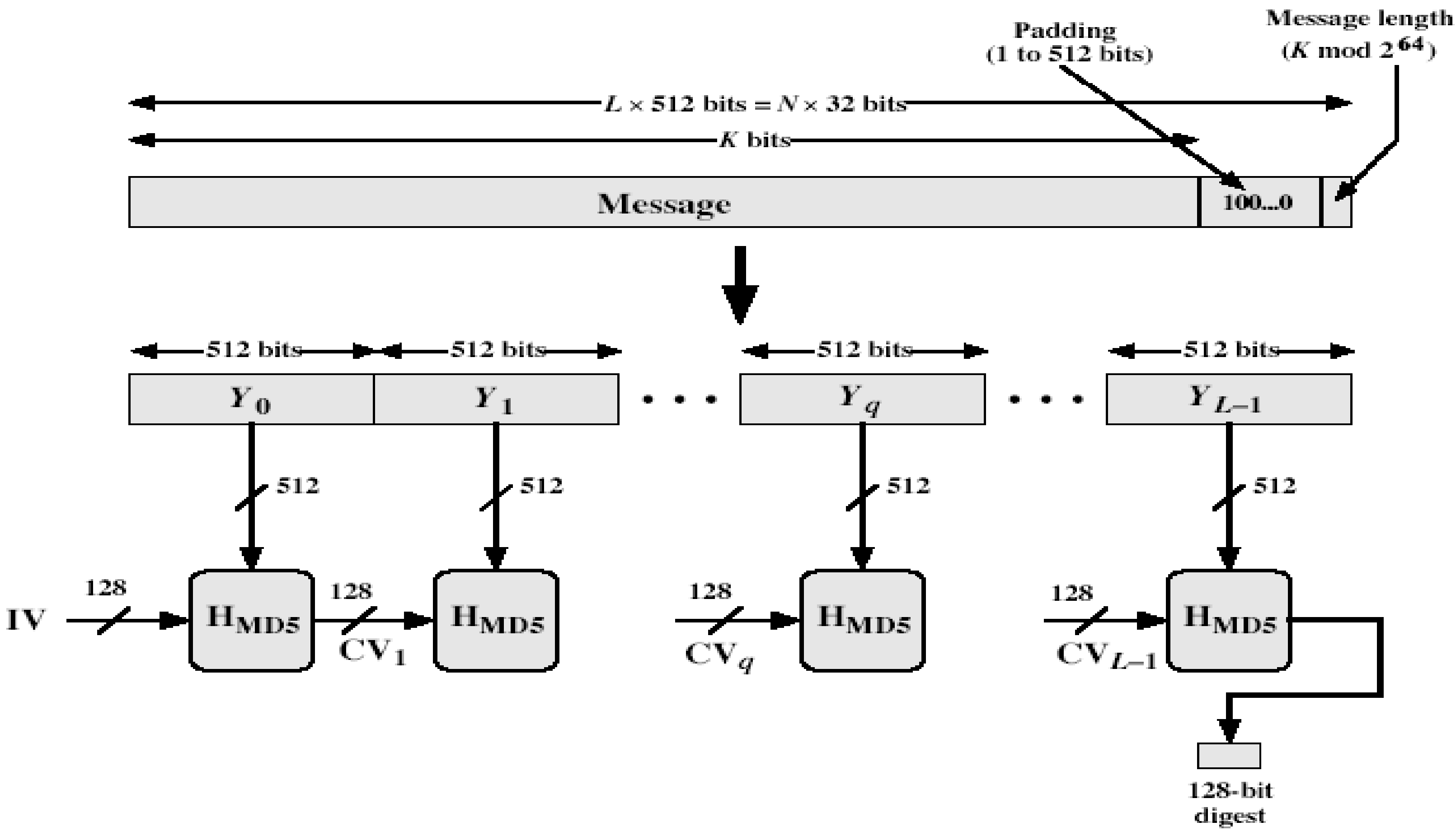
MD5

- designed by Ronald Rivest (the R in RSA)
- latest in a series of MD2, MD4
- produces a 128-bit hash value
- until recently was the most widely used hash algorithm
  - in recent times have both brute-force & cryptanalytic concerns
- specified as Internet standard RFC1321

MD5 Overview

1. pad message so its length is 448 mod 512
2. append a 64-bit length value to message
3. initialize 4-word (128-bit) MD buffer (A,B,C,D)
4. process message in 16-word (512-bit) blocks:
   - using 4 rounds of 16 bit operations on message block & buffer
   - add output to buffer input to form new buffer value
5. output hash value is the final buffer value

MD5 Overview



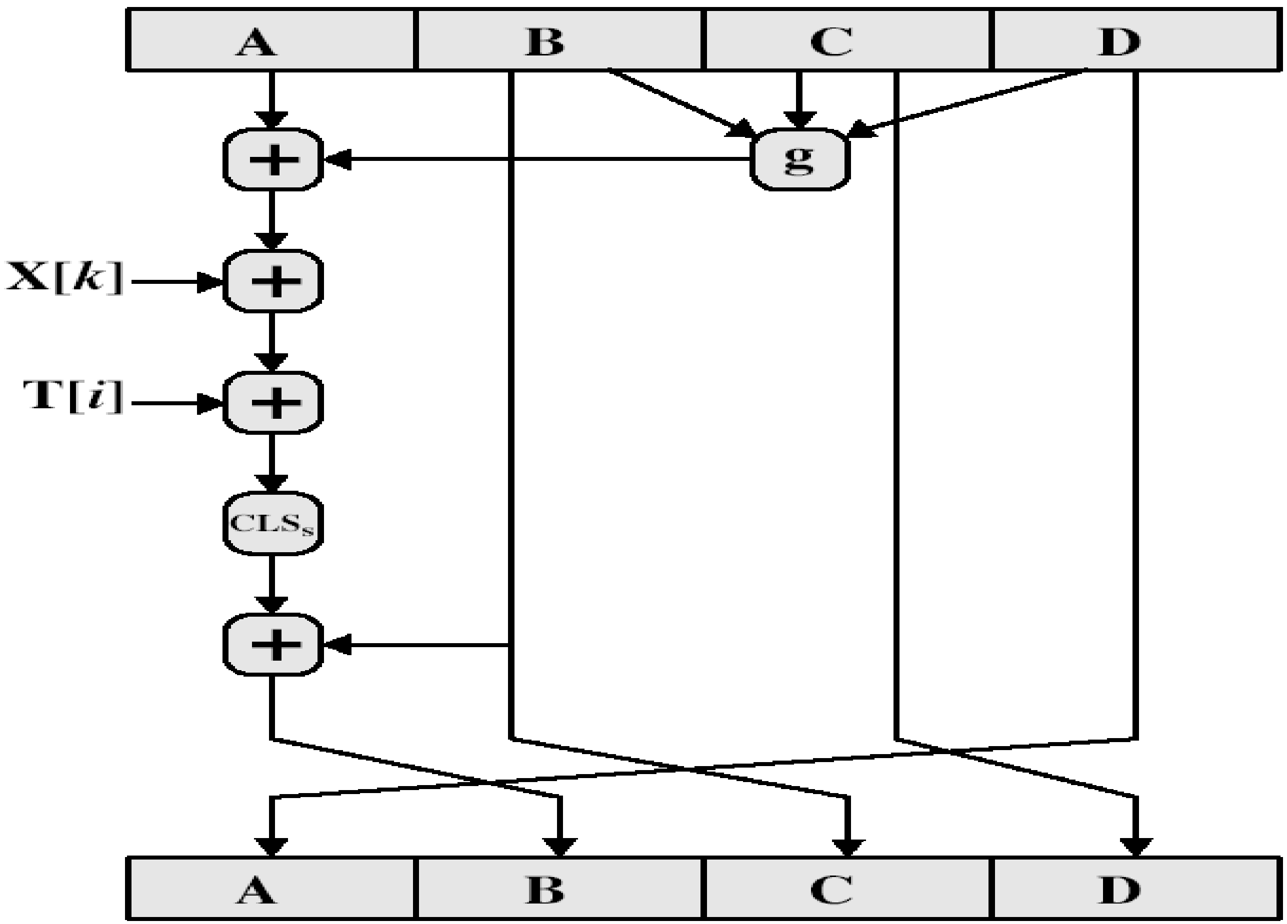
MD5 Compression Function

- each round has 16 steps of the form:

```
a = b+((a+g(b,c,d)+X[k]+T[i])<<<s)
```

- a,b,c,d refer to the 4 words of the buffer, but used in varying permutations
  - note this updates 1 word only of the buffer
  - after 16 steps each word is updated 4 times
- where g(b,c,d) is a different nonlinear function in each round (F,G,H,I)
- T[i] is a constant value derived from sin

MD5 Compression Function



Strength of MD5

- MD5 hash is dependent on all message bits
- Rivest claims security is good as can be
- known attacks are:
  - Berson 92 attacked any 1 round using differential cryptanalysis (but can't extend)
  - Boer & Bosselaers 93 found a pseudo collision (again unable to extend)
  - Dobbertin 96 created collisions on MD compression function (but initial constants prevent exploit)
  - Wang et al. 04 created collisions on entire MD5 in less than one hour using an IBM p960 cluster

Secure Hash Algorithm (SHA-1)

- SHA was designed by NIST & NSA in 1993, revised 1995 as SHA-1
- US standard for use with DSA signature scheme
  - standard is FIPS 180-1 1995, also Internet RFC3174
  - nb. the algorithm is SHA, the standard is SHS
- produces 160-bit hash values
- now the generally preferred hash algorithm
- based on design of MD4 with key differences

SHA Overview

1. pad message so its length is 448 mod 512
2. append a 64-bit length value to message
3. initialize 5-word (160-bit) buffer (A,B,C,D,E) to

(67452301,efcdab89,98badcfe,10325476,c3d2e1f0)

1. process message in 16-word (512-bit) chunks:
   - expand 16 words into 80 words by mixing & shifting
   - use 4 rounds of 20 bit operations on message block & buffer
   - add output to input to form new buffer value
2. output hash value is the final buffer value

You can try both on any Linux machine

```
bash$ cat helloworld.txt
“Hello world!”

bash$ md5sum helloworld.txt
78890504b184be1407cc2880363ddf10

bash$ sha1sum helloworld.txt
398dc9eb139cebe2ba1d8791259440ede011cfba
```

SHA-1 verses MD5

- brute force attack is harder (160 vs 128 bits for MD5)
- not vulnerable to any known attacks (compared to MD4/5)
- a little slower than MD5 (80 vs 64 steps)
- both designed as simple and compact
- optimized for big endian CPU's (vs MD5 which is optimised for little endian CPU’s)

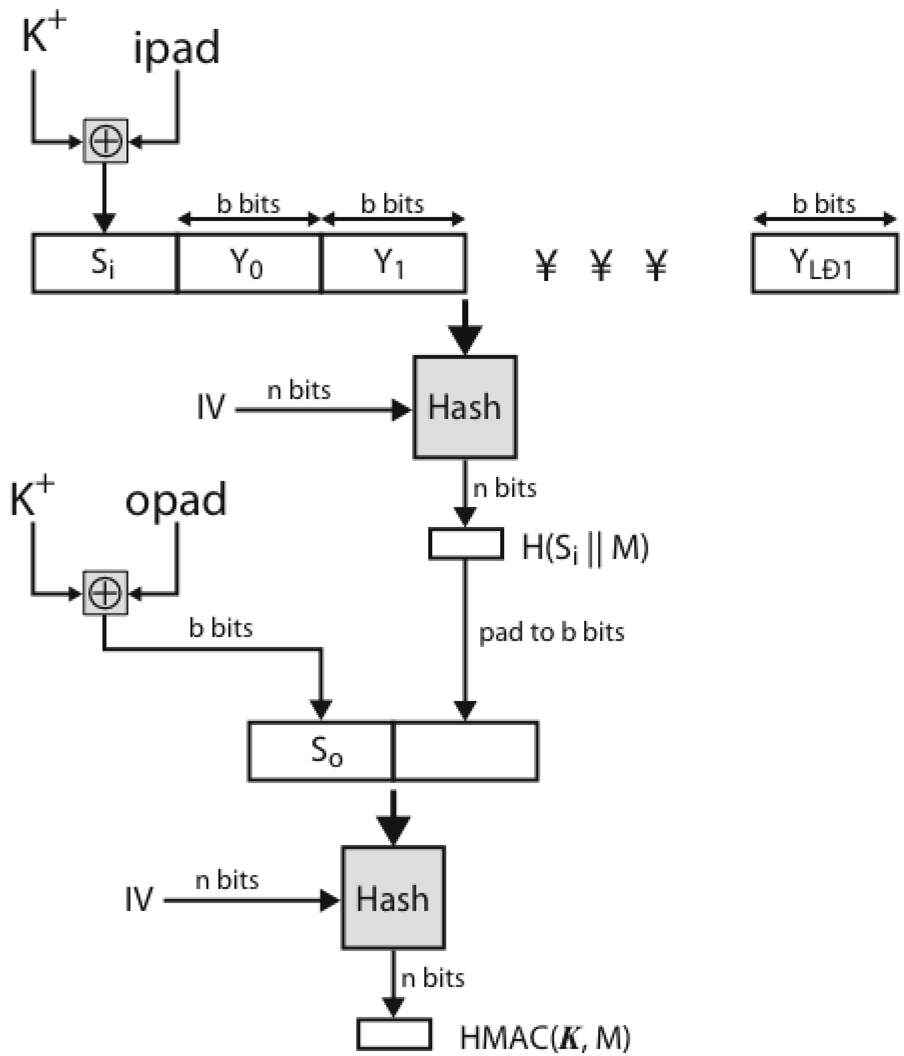
Keyed Hash Functions as MACs

- want a MAC based on a hash function
  - because hash functions are generally faster
  - code for crypto hash functions widely available
- hash includes a key along with message
- original proposal:

```
KeyedHash = Hash(Key|Message)
```

  - some weaknesses were found with this
- eventually led to development of HMAC

HMAC Overview



HMAC

- specified as Internet standard RFC2104
- uses hash function on the message:

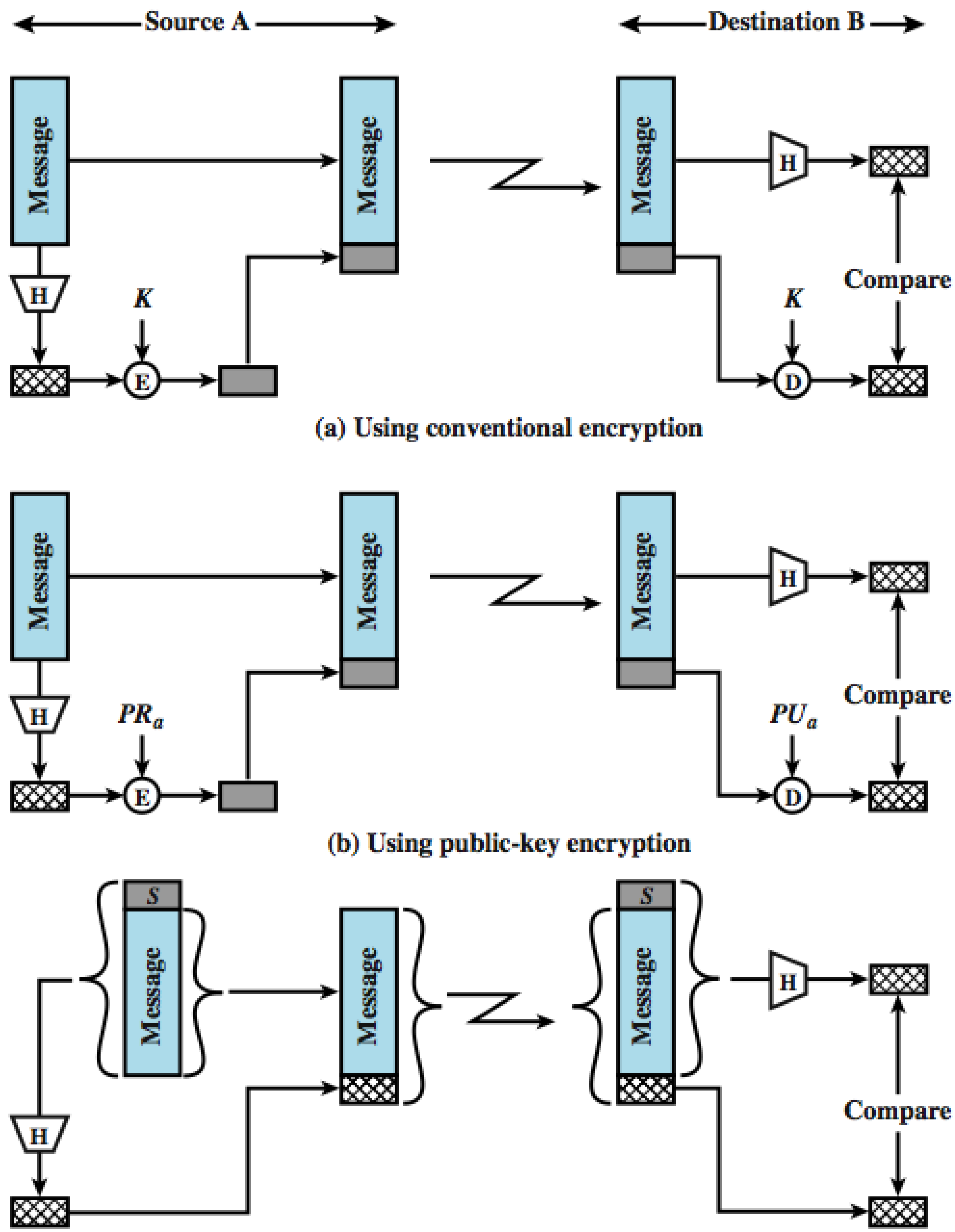
$$HMAC_K = Hash[(K^+ \text{ XOR } opad) \,||\, Hash[(K^+ \text{ XOR } ipad)||M)]]$$

- where K^+ is the key padded out to size
- and opad, ipad are specified padding constants
- overhead is just 3 more hash calculations than the message needs alone
- any hash function can be used
  - eg. MD5, SHA-1, RIPEMD-160, Whirlpool

HMAC Security

- proved security of HMAC relates to that of the underlying hash algorithm
- attacking HMAC requires either:
  - brute force attack on key used
  - birthday attack (but since keyed would need to observe a very large number of messages)
- choose hash function used based on speed verses security constraints

Message Auth



(a) Using conventional encryption

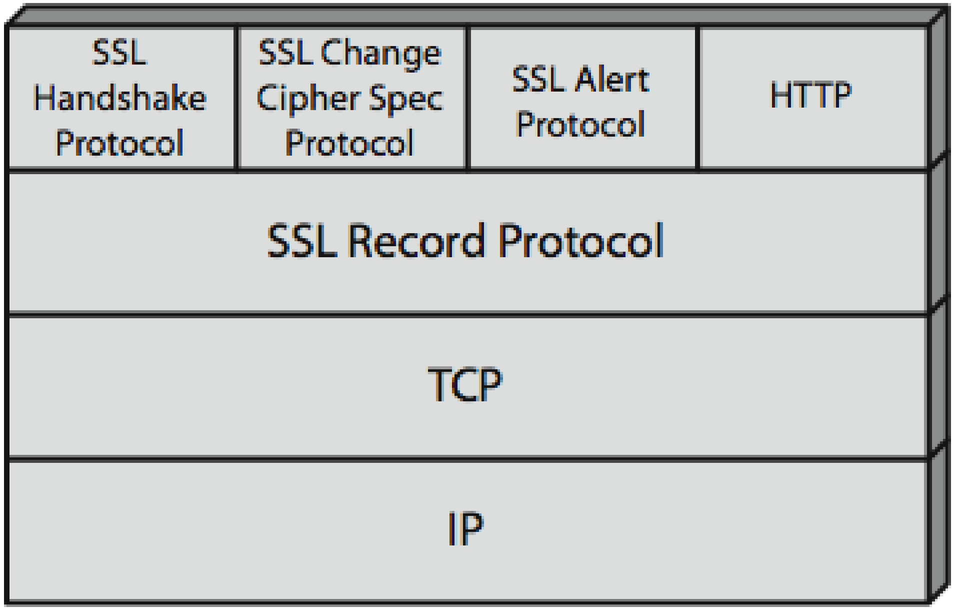
(b) Using public-key encryption

(c) Using secret value

Secure Sockets Layer (SSL)

- transport layer security service
  - originally developed by Netscape
  - version 3 designed with public input
- subsequently became Internet standard RFC2246: Transport Layer Security (TLS)
- use TCP to provide a reliable end-to-end service
- may be provided in underlying protocol suite
- or embedded in specific packages

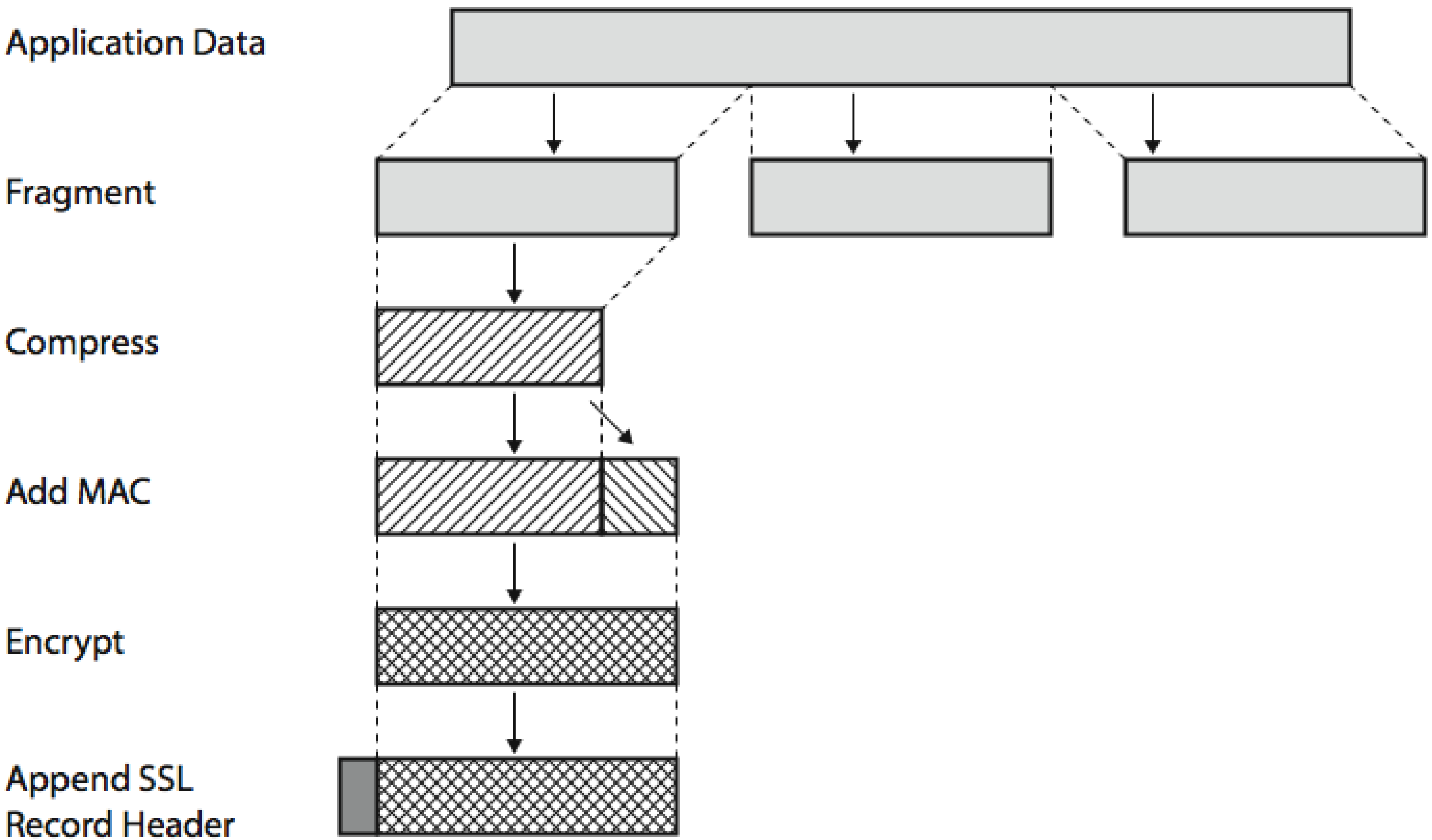
SSL Protocol Stack



SSL Record Protocol Services

- **message integrity**
  - using a MAC with shared secret key
  - similar to HMAC but with different padding
- **confidentiality**
  - using symmetric encryption with a shared secret key defined by Handshake Protocol
  - AES, IDEA, RC2-40, DES-40, DES, 3DES, Fortezza, RC4-40, RC4-128
  - message is compressed before encryption

SSL Record Protocol Operation



SSL Change Cipher Spec Protocol

- one of 3 SSL specific protocols which use the SSL Record protocol
- a single message
- causes pending state to become current
- hence updating the cipher suite in use

SSL Alert Protocol

- conveys SSL-related alerts to peer entity
- severity
  - warning or fatal
- specific alert
  - fatal: unexpected message, bad record mac, decompression failure, handshake failure, illegal parameter
  - warning: close notify, no certificate, bad certificate, unsupported certificate, certificate revoked, certificate expired, certificate unknown
- compressed & encrypted like all SSL data

SSL Handshake Protocol

- allows server & client to:
  - authenticate each other
  - to negotiate encryption & MAC algorithms
  - to negotiate cryptographic keys to be used
- comprises a series of messages in phases
  1. Establish Security Capabilities
  2. Server Authentication and Key Exchange
  3. Client Authentication and Key Exchange
  4. Finish

SSL Handshake Protocol