Summary from the last class

- Security is multifaceted phenomenon
  - Confidentiality, availability, integrity
- Various security threats and defense challenges
- Common defense flavors
  - Prevent attacks
  - Detect attacks and respond to them or sustain them
  - Learn from mistakes
- Cryptography
  - Secret key, public protocol
  - Symmetric, asymmetric, hash functions

Types Of Cryptographic Functions

- Symmetric key crypto: one key
  - We will call this secret key or shared key
  - Both Alice and Bob know the same key
- Asymmetric key crypto: two keys
  - Alice has public key and private key
  - Everyone knows Alice’s public key but only Alice knows her private key
  - One can encrypt with public key and decrypt with private key or vice versa
- Hash functions: no key
  - Output depends on input in non-linear fashion

Uses Of Symmetric Key Crypto

- Symmetric key crypto: one key
- Transmitting over an insecure channel
  - Classic use: Alice and Bob encrypt messages they exchange
  - They must first securely exchange the key
- Secure storage on insecure media
  - Encrypt stored data so someone who breaks in cannot read it

Uses Of Symmetric Key Crypto

- Authentication – prove the identity
  - Pass phrase – what if Mallory asks for the pass phrase
  - Strong authentication without revealing the secret
    - Alice
      - $R_a$
      - $k_a(R_a)$
      - $k_b(R_b)$
    - Bob
      - $R_b$
      - $k_a(R_b)$
      - $k_b(R_b)$
- Integrity check
  - Calculate the checksum and encrypt it – MIC: message integrity code
    - $M$
    - $MIC = E_{k_a}(check(M))$
Asymmetric key crypto can do everything symmetric key crypto can but much (about 1,500 times) slower. However, it can do some things better!

Transmitting over an insecure channel:
- Secure key exchange is difficult for symmetric crypto – chicken and egg problem
- With asymmetric keys, Alice can publicly broadcast her public key
- How about scale? How many keys does Alice need to talk to 10 different recipients?

Secure storage on insecure media: Same as with symmetric key crypto

Authentication:
- Alice wants to verify Bob’s identity
- She sends to Bob $E_{PubBob}(R_A)$
- Bob decrypts and sends back $R_A$
- This can be done with symmetric keys too but if Bob wanted to authenticate himself to Carol he would need to remember a new key. Not so with asymmetric keys.
- Alice doesn’t need to store any secret info which is good if she is a computer

Digital signatures:
- Alice orders books online from Bob
- She signs every order using her private key
- If she claims she didn’t place the order Bob can prove she did – non-repudiation
- Can symmetric key crypto do this?

Known also as one-way functions or message digests

Take an arbitrary-length message M and transform it into fixed-length hash $h(M)$

Properties:
- Knowing M is easy to calculate $h(M)$, but it is very hard to calculate M knowing $h(M)$
- It is very hard to find $M1 \neq M$ so that $h(M1) = h(M)$, this is collision-free property
- E.g., take the message M as a number, add a large constant to it, square it, and take middle $n$ digits as the hash
Uses Of Hash Algorithms

- Storing hashed password info
- Message integrity
  - Use message M and a shared secret S, run this through hash function and produce MIC = secure hash
  - Why do we need a shared secret?
- Message fingerprint
  - Hash the files to detect tampering
  - Works for download security too
- Signing message hash instead of the whole message is faster

Let's Formalize A Little ...

<table>
<thead>
<tr>
<th>Alice</th>
<th>E_{K1}(M)</th>
<th>C</th>
<th>D_{K2}(C)</th>
<th>Bob</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>M</td>
<td></td>
<td></td>
<td>M</td>
</tr>
</tbody>
</table>

- M – message
- K1 – encryption key
- E_{K1}(M) – message M is encrypted using key K1
- C – ciphertext
- K2 – decryption key
- D_{K2}(C) – ciphertext C is decrypted using key K2

Cyphertext-only attack: Eve can gather and analyze C’s to learn K2

How does Eve know she got the right key?
Eve has to have enough ciphertext – having XYZ with monoalphabetic cipher would not be enough
What if K2 depends on a password in a known way?

Known-plaintext attack: Eve can attempt to learn K2 by observing many ciphertexts C for known messages M

How does Eve obtain the plaintext?
Chosen-plaintext attack: Mallory can feed chosen messages M into encryption algorithm and look at resulting ciphertexts C. Learn either K2 or messages M that produce C. Assumption is that extremely few messages M can produce same C. For a monoalphabetic cipher she could feed a message containing all the letters of the alphabet.

Man-in-the-middle attack:
- Mallory can substitute messages
- Mallory can modify messages
- So that they have different meaning
- So that they are scrambled
- Mallory can drop messages
- Mallory can replay messages to Alice, Bob or the third party

Brute-force attack: Eve has caught a ciphertext and will try every possible key to try to decrypt it. This can be made infinitely hard by choosing a large keyspace.

Substitution
- Goal: obscure relationship between plaintext and ciphertext
- Substitute parts of plaintext with parts of ciphertext

Transposition (shuffling)
- Goal: dissipate redundancy of the plaintext by spreading it over ciphertext
- This way changing one bit of plaintext affects many bits of the ciphertext (if we have rounds of encryption)
Substitution

- **Monoalphabetic** – each character is replaced with another character
  - Caesar’s cipher – each letter is shifted by 3, a becomes d, b becomes e, etc.
  - Keep a mapping of symbols into other symbols
  - Drawback: frequency of symbols stays the same and can be used to break the cipher

- **Homophonic** – each character is replaced with a character chosen randomly from a subset
  - Ciphertext alphabet must be larger than plaintext alphabet – we could replace letters by two-digit numbers
  - Number of symbols in the subset depend on frequency of the given letter in the plaintext
  - The resulting ciphertext has all alphabet symbols appearing with the same frequency

- **Polygram** – each sequence of characters of length $n$ is replaced with another sequence of characters of length $n$
  - Like monoalphabetic cipher but works on $n$-grams

- **Polyalphabetic** – many monoalphabetic ciphers are used sequentially
  - First mapping is used for the first letter, second mapping for the second letter and so on
  - XOR is a polyalphabetic cipher in binary domain
One-Time Pad

- Polyalphabetic cipher with infinite key
- Combine letters from the message with the letters from an infinite key, randomly generated
- Never reuse the key
- Key needs to be generated using a very good RNG (to avoid any patterns)
- This cipher cannot be broken
- Sender and receiver must be perfectly synchronized

Symmetric Crypto Algorithms

- Stream ciphers: polyalphabetic
  - Work on message a bit or a byte at a time
  - Same bit/byte will encrypt differently, depending on the position of the key
- Block ciphers: polygram
  - Work on message block by block
  - Block size is usually the same as key size
  - Same plaintext block may encrypt into the same ciphertext block, depending on the cipher mode
- Assume XOR with the key

Stream Cipher Example

<table>
<thead>
<tr>
<th>Plaintext</th>
<th>SuperCalifragilistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>SANTA CLAUS SANTA CLAUS</td>
<td>S U P E R C A L I F R A G I L I S T I C</td>
</tr>
<tr>
<td>Key</td>
<td>Cipher Text</td>
</tr>
<tr>
<td>LVDYSFMMDYKBUCMLEUDV</td>
<td></td>
</tr>
</tbody>
</table>

Bonus question: What was the encryption algorithm I used here?

Stream Ciphers

- If Eve can get hold of plaintext/ciphertext pair she can retrieve the key
- Keystream is generated continuously and is the function of the secret stored inside the RNG
- Key should be pseudorandom - hard to break but easily reproduced for decryption
- Security depends entirely on RNG generating the key – keystream generator (KSG)
**Synchronous KSG**

- Keystream is generated from the key K
- Sender and receiver must be synchronized
- One–bit error in ciphertext produces one–bit error in plaintext
- Any deletions and insertions will cause loss of synchronization
  - Both sides start afresh with a new key
- Malory cannot replay messages but she can change bits and predict how this changes plaintext

**Internal State**

- Output Function

- Next State Function

- Keystream

**Self-Synchronizing KSG**

- Internal state is the function only of the previous $n$ ciphertext bits and depends on the key K
- Decryption keystream generator will completely synchronize with encryption generator after receiving $n$ bits
- Advantage:
  - Recovery from loss of bits after $n$ bits
- Drawback:
  - Error extension - one–bit error in ciphertext produces $n$ errors in plaintext
  - Malory can replay messages, change bits

**Synchronous Stream Cipher**

- Keystream is generated from the key K
- Sender and receiver must be synchronized
- One–bit error in ciphertext produces one–bit error in plaintext
- Any deletions and insertions will cause loss of synchronization
  - Both sides start afresh with a new key
- Malory cannot replay messages but she can change bits and predict how this changes plaintext
Generating Random Numbers

- We need to generate a sequence that looks random but is reproducible (next state function in synchronous KSG)
- There shouldn’t be any obvious regularities, otherwise Eve can learn the pattern after seeing several numbers, and guess the next ones
- We would like to cover the whole range of numbers (e.g. $2^n$ if the number has $n$ bits)

Linear Congruential Generators

- Generators of the form $X_n = (aX_{n-1} + b) \mod m$
  - A period of a generator is number of steps before it repeats the sequence
  - If $a$, $b$ and $m$ are properly chosen, this generator will be maximal period generator and have period of $m$
  - It has been proven that any polynomial congruential (modulo) generator can be broken

Linear Feedback Shift Registers

- Used for cryptography today
- A shift register is transformed in every step through feedback function
  - Contents are shifted one bit to the right, the bit that “falls out” is the output
  - New leftmost bit is XOR of some bits in the shift register – tap sequence
  - If we choose a proper tap sequence period will be $2^n-1$
Linear Feedback Shift Registers

- Proper tap sequences are those where a polynomial from a tap sequence + 1 is a primitive polynomial in GF(2).
- There are tables of primitive polynomials.
- LFSR is fast in hardware but slow in software.
- LFSR are not themselves secure but they are used as building blocks in encryption algorithms.

Block Cipher Example

plaintext: SANTA CLAUS SANTA CLAUS
key: SUPER

plaintext: SANTA CLAUS SANTA CLAUS
key: SUPER

ciphertext: LVDS

ciphertext: LVDS VGUZK
Block Cipher Example

plaintext: SANTA CLAUS SANTA CLAUS
key: SUPER

cipher: LVDYS VGQZK LVDYS

Block Cipher Example

plaintext: SANTA CLAUS SANTA CLAUS
key: SUPER

cipher: LVDYS VGQZK LVDYS

Block Encryption In Rounds

Encrypting A Large Message

- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- k-bit Cipher Feedback Mode (CFB)
- k-bit Output Feedback Mode (OFB)

Things to consider:
- Can we encrypt/decrypt efficiently (as soon as bits arrive)
- How hard it is to break encryption
- What if a bit is flipped on the channel
- What if we lose a bit on the channel
Electronic Code Book (ECB)

- Store mapping for every possible block
  - Fast encryption/decryption – just a table lookup
  - Ability to process text in any order and in parallel
  - Table size could be enormous so we need to make the mapping depend on the key
- Eve can detect which blocks map to other blocks, by seeing several plaintext and corresponding ciphertext messages
- Due to language redundancy even partial decryption might provide enough information
- Bit error invalidates one block
- Bit loss/addition is not recoverable

Block Replay

12B7 7783 38AC CDC7

Bank A ➔ Mallory ➔ Bank B

Transfer $100 to my account in Bank B

- Mallory does this couple of times, looks for similar block sequences
- She can generate messages to transfer different sum to diff. acct.
- She can replay 12B7 7783 38AC CDC7 at will

Bank A ➔ Mallory ➔ Bank B

Block Replay

3231 12B7 7783 38AC CDC7

Bank A ➔ Mallory ➔ Bank B

Transfer $100 to my account in Bank B

- Bank adds timestamps
- Mallory picks specific blocks of message carrying her name and account number and replaces those in other messages between Bank A and Bank B

Cipher Block Chaining (CBC)

- Problem with ECB is that Mallory can replace, add or drop blocks at will
- Chaining prevents this by adding feedback
  - Each ciphertext block depends on all previous blocks
- With CBC, same plaintext blocks will encrypt to different ciphertext blocks thus obscuring patterns in plaintext
Initialization vector (IV) is just a block of random numbers, to ensure that no messages have the same beginning. Both the sender and the receiver must use the same IV. Can be transmitted with the message but must be unpredictable.

An error in ciphertext affects one block and several bits of plaintext.

Mallory can:
- Add blocks
- Drop blocks
- Introduce bit errors

Bit loss/addition is not recoverable.
Cipher-Feedback Mode (CFB)

- IV must be unpredictable
- If a $k$-bit unit is lost or added, next $n/k-1$ units will be garbled but then the algorithm will recover from error.
  - Not so for $m$-bit loss/add, $m \neq k$
- One-bit error in ciphertext produces one-bit error in plaintext and $n/k-1$ subsequent plaintext units are garbled ($n$ is the block size, $k$ is the unit size)

Self-synchronizing KSG!

Output-Feedback Mode (OFB)

- Similar to CFB but unit is taken from the output queue, not from the ciphertext
- IV must be unique (counter)
- Output block generation can be done offline, plaintext is then just XOR-ed when it arrives
- One-bit error in ciphertext produces one-bit error in plaintext
- Bit loss/addition is not recoverable
Which Cipher Is The Best?
- Stream ciphers can be analysed mathematically and can be efficiently implemented in hardware
- Block ciphers are more general and can be efficiently implemented in software
- ECB is easiest and fastest but also weakest. Can be used for encrypting random data, such as other keys.
- CBC is good for encrypting files, no danger of lack of synchronization
- CFB is good for encrypting streams of characters
- OFB is good if error propagation cannot be tolerated, and it can be made fast by precomputing

Public Key Cryptography
- Everyone has two keys:
  - Public key K1 that everyone knows
  - Private key K2 that only he knows
- Encryption algorithm and key properties ensure that
  \[ D_{K2}(E_{K1}(M)) = M \]

Modular Arithmetic
- Galois Field GF(n)
  - All operations are on numbers 0,...,n-1
  - Observe all operations in Galois Field GF(n)
    - \( a = b \mod n \) if \( \exists k, a = k \cdot n + b \)
      - e.g. 26 mod 16 = 10 so
      - 26 = 10 mod 16
    - Modulo operation (modular reduction) can be performed at any point, e.g.
      - \((a + b) \mod n = ((a \mod n) + (b \mod n)) \mod n\)

Modular Exponentiation
- Key step in asymmetric crypto
- How many operations are needed to calculate \( a^x \)?
- Exponentiation can be performed very efficiently (addition chaining):
  - We want to calculate \( a^x \mod n \)
  - Write x as a binary number, \( result = 1 \)
  - Traverse x from left to right
    - If digit is 1, result = result \( \times a \)
    - If digit is 0, result = result \( ^2 \)
    - Perform modular reduction often to keep result small. This is cheap if \( n = 2^m \)
Example

19 mod 23
19 = 10011
19 mod 23 =

step1 is 1: result = 1^2 * 4 = 4
step2 bit is 0: result = 4^2 mod 23 = 16
step3 bit is 0: result = 16^2 mod 23 = 256 mod 23 = 3
step4 bit is 1: result = 3^2 * 4 mod 23 = 36 mod 23 = 13
step5 bit is 1: result = 13^2 * 4 mod 23 = 676 mod 23 = 9

Prime Numbers

- A number \( n \) is prime if it is only divisible by 1 and itself
- Numbers \( x \) and \( y \) are relatively prime if they share no factors greater than 1
  - E.g. 7 and 15 are relatively prime, 9 and 15 are not because they have 3 as common factor

Inverses Modulo a Number

- Multiplicative inverse \( y \) for \( x \) is a number that satisfies:
  \[ x \cdot y \equiv 1 \pmod{n} \]
- In GF(\( n \)) inverse \( y \) for \( x \) modulo \( n \) is a number that satisfies:
  \[ x \cdot y \equiv 1 \pmod{n} \] \( x \) and \( y \) will be our public/private key pair
- Inverse \( y \) in GF(\( n \)) can be found uniquely if \( x \) and \( n \) are relatively prime, otherwise it cannot be found
- If \( n \) is prime then it is relatively prime to all numbers \( \{0, n-1\} \) and each number has its inverse in GF(\( n \))

Extended Euclidean Algorithm

- How to find an inverse \( y \) for \( x \) mod \( n \)
  \[ x \cdot y \equiv 1 \pmod{n} \]
  \[ x \cdot y + k \cdot n = 1 \]
- Extended Euclidean algorithm will find \( y \) and \( k \) given \( x \) and \( n \)

Generate one key starting from the other (e.g., have public key, generate private key)
Extended Euclidean Algorithm

- $x=13, n=225, y=?$

\[
\begin{array}{c|c|c|c}
 a & q & x & y \\
 225 & - & 1 & 0 \\
 13 & 0 & 1 & \\
\end{array}
\]

- $225/13 = 17$ remainder 4

\[
\begin{array}{c|c|c|c}
 a & q & x & y \\
 225 & - & 1 & 0 \\
 13 & 17 & 0 & 1 \\
 4 & & & \\
\end{array}
\]

Extended Euclidean Algorithm

- $a_{r-1}/a_{r-2} = q_{r-1}$ and the remainder $a_r$

\[
\begin{array}{c|c|c|c}
 a & q & x & y \\
 225 & - & 1 & 0 \\
 13 & 0 & 1 & \\
\end{array}
\]

- $13/4 = 3$ remainder 1

\[
\begin{array}{c|c|c|c}
 a & q & x & y \\
 225 & - & 1 & 0 \\
 13 & 17 & 0 & 1 \\
 4 & 3 & & \\
 1 & & & \\
\end{array}
\]
### Extended Euclidean Algorithm

- **4/1 = 4 remainder 0**

\[
\begin{array}{c|cc}
 a & q & x & y \\
225 & - & 1 & 0 \\
13 & 17 & 0 & 1 \\
4 & 3 & 1 & 0 \\
1 & 4 & & \\
\end{array}
\]

- **1 - 17*0 = 1**

\[
\begin{array}{c|cc}
 a & q & x & y \\
225 & - & 1 & 0 \\
13 & 17 & 0 & 1 \\
4 & 3 & 1 & 0 \\
1 & 4 & & \\
\end{array}
\]

- **0 - 3*1 = -3 = 222**

\[
\begin{array}{c|cc}
 a & q & x & y \\
225 & - & 1 & 0 \\
13 & 17 & 0 & 1 \\
4 & 3 & 1 & 0 \\
1 & 4 & & 222 \\
\end{array}
\]

- **\(x_{r+1} = x_{r-1} - q_r x_r\)**
Extended Euclidean Algorithm

\[ y_{i+1} = y_{i-1} - q_i \cdot y_i \]

<table>
<thead>
<tr>
<th>a</th>
<th>q</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>225</td>
<td>-</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>222</td>
<td>52</td>
</tr>
</tbody>
</table>

Extended Euclidean Algorithm

\[ 0 - 17 \cdot 1 = 17 = 208 \]

<table>
<thead>
<tr>
<th>a</th>
<th>q</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>225</td>
<td>-</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>208</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>222</td>
<td>52</td>
</tr>
</tbody>
</table>

Extended Euclidean Algorithm

\[ 1 - (3 \cdot 208 \mod 225) = -173 = 52 \]

<table>
<thead>
<tr>
<th>a</th>
<th>q</th>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>225</td>
<td>-</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>208</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>222</td>
<td>52</td>
</tr>
</tbody>
</table>

Factorization of Large Numbers

- We have seen that exponentiation in \( \text{GF}(n) \) can be performed efficiently.
- On the other hand, it is hard to factor large numbers (find possible \( x \) and \( y \), given \( x \cdot y \))
  - Computationally intensive, must use brute-force search.
- Generally factoring time of a large number \( n \) increases exponentially with each binary digit added to \( n \).
Public Key Cryptography (RSA)

- Created by Ron Rivest, Adi Shamir, and Leonard Adleman
- Choose two prime numbers \( p \) and \( q \) of equal length
- Compute \( n = p \times q \), and
- **Euler Totient function** \( \phi(n) = (p-1)(q-1) \)
  - We will work in \( \text{GF}(\phi(n)) \)
- Choose public key \( e \) relatively prime to \( \phi(n) \)

Public Key Cryptography (RSA)

- Using extended Euclidean algorithm calculate \( d \) which is inverse of \( e \) mod \( \phi(n) \)
  \[ d \times e \equiv 1 \mod \phi(n) \]
- Publish \( e \) and \( n \), remember \( d \)
- Encryption:
  \[ E(M) = M^e \mod n \]
- Decryption:
  \[ D(C) = C^d \mod n = M^e \mod n \]
  \[ M^{(d \times (p-1) \times (q-1)) \mod pq} = M \]
- There are proofs that the last equation holds

Digital Signatures

- Provide data integrity
  - Can it be done with symmetric systems?
    - Verification requires shared key
    - Doesn’t provide non-repudiation
- Need proof of provenance
  - Hash the data, encrypt with private key
  - Verification uses public key to decrypt hash
  - Provides non-repudiation
    - Why?
RSA can be used

DSA: Digital Signature Algorithm

- Variant of ElGamal signature
- Adopted as part of DSS by NIST in 1994
- Key size ranges from 512 to 1024 bits
- Royalty-free

One-Way Hash Functions

- Take a variable-length input \( M \) and produce fixed-length output (hash value or message digest or message integrity code)
  \[ h = H(M) \]

- The idea is to fingerprint \( M \)
  - Given \( M \) easy to compute \( h \)
  - Given \( h \) very hard to compute \( M \)
  - One-bit change in \( M \) changes on the average half of the bits in \( h \)
  - Good one-way hash function is collision-free: given \( M \) it is very hard to find \( M' \) such that \( H(M) = H(M') \)
  - One-way hash function is public

Sometimes what we also need is collision resistance: it is hard to find two random messages \( M_1 \) and \( M_2 \) such that \( H(M_1) = H(M_2) \)

For a hash function that produces \( m \) bit hash, it takes \( 2^{m/2} \) trials to find any two messages that hash to the same value

We need large \( m \), currently 128–160
**MD5**

- Divide M into 512-bit blocks
  - Pad M with string of 1 and many zeros so that it is 64 bit short of multiple of 512
  - Concatenate original length in bits as 64-bit number
- Blocks are processed sequentially
- Last result is hash value for the whole message and is 128-bit long

**MD5**

- For each message block (512 bits):
  - Variables A, B, C, D are initialized (for first block they are constant values)
    - Each is 32 bits
  - Go through 4 rounds; each round repeats the following operation 16 times:
    - Performs non-linear function on three variables
    - Sums the result, the fourth variable, message subblock (32-bits) and a constant $K_i$ ($i\in[1,16]$)
    - Rotates the result to the left, adds it to one of the variables and replaces this variable
- Output is concatenation of A, B, C, D

**MD5 One Block Processing**

**MD5 One Round = 16 operations**

Non-linear functions (in each round one is chosen for the green square):
- $F(X,Y,Z) = XY \lor \neg(X) \land Z$
- $G(X,Y,Z) = XZ \lor Y \land \neg(Z)$
- $H(X,Y,Z) = X \oplus Y \oplus Z$
- $I(X,Y,Z) = Y \oplus (X \lor \neg(Z))$

Addition mod $2^{32}$

Message subblock (32 bits)

Constant, varies for each operation

Shift left by 5 bits, 5 varies for each operation

Security of MD5 is severely compromised
- Multiple known attacks
- Secure Hash Algorithm (SHA)
- Similar to MD5 but:
  - Has 20 operations per round
  - Operations are different than in MD5 but along similar lines
  - Each message block (16×32 = 512 bits) is expanded into (80×32 bits)
  - Produces 160-bit hash value
- SHA-1 is an improved version of SHA

Data Encryption Standard (used to be)
- Block cipher, symmetric
- Works with 64-bit blocks of plaintext and 56-bit key (weak)
  - Superseded by AES

DES key is too short (64 bits but 8 used for parity)
- Can be broken by brute force
- Fixes the key length problem by applying DES three times
  - With two or three different keys
  - E(K1)D(K2)E(K3)
- Designed to be backward compatible with DES
- Slow, since encryption/decryption is now performed three times
DES-X
- Fixes the key length problem by using a longer key whose portions are XOR-ed with plaintext
  - 64-bits XOR-ed prior to DES
  - 64-bits XOR-ed with DES output
  - Key 56+64+64 bits long
- Same speed as DES but higher security

AES
- Advanced Encryption Standard
- Block cipher, symmetric
- Works with 128-bit blocks of plaintext and 128, 192 or 256-bit key
- Operates on 4x4 matrix of bytes called state, all operations over GF(2^8)
- Algorithm
  - Key expansion – derive round keys (4x4)
  - AddRoundKey – XOR state with round key
  - Rounds – SubBytes (substitution), ShiftRows (transposition), MixColumns, AddRoundKey
  - Final round – SubBytes, ShiftRows, AddRoundKey

NOT ON TESTS
- Substitution box
- Multiplication by a known matrix