Overview

- Modern public-key cryptosystems:
  - RSA, proposed 1978
    - Asymmetric cryptosystem
      - Simplifies key distribution and management
      - Facilitates the creation of digitally signed messages
  - The Digital Signature Standard (DSS), 1994
    - Technique for creating and verifying digital signatures
      - Only the signer can produce his signature on a document
      - A signed document cannot be altered without invalidating the signature

Symmetric-Key vs. Public-Key Cryptography

- Symmetric-key:
  - To communicate securely, users must have a previously-established shared secret key
    - Sender encrypts message with the shared key
    - Receiver uses the same key to decrypt
- Public-key:
  - User generates a public-key/private-key pair:
    - Public key is made public
    - Private key is kept secret
  - Senders encrypt message with recipient's public key
  - Only the user that generated the key pair knows the private key and can perform decryption

Motivation for Public-Key Cryptography

- Symmetric-key cryptosystem:
  - Requires prearrangement
    - Need a unique secret key for each communication partner
  - Number of keys grows exponentially
    - A group of m people requires $(m^2 - m)/2$ keys
- Public-key cryptosystems:
  - Facilitates secure communications with someone you have never communicated with before
    - Only need to know that user's public key
  - Number of keys grows linearly
    - A group of m people requires $2m$ keys

Public-Key Cryptography

- Each user has a **pair** of keys that are inverses of each other:
  - P**ublic** key
    - Made public
    - Can decrypt anything encrypted with the private key
  - P**rivate** key
    - Kept secret
    - Can decrypt anything encrypted with the public key

Public-Key Cryptography – Requirements

- A unique public/private key pair for every user
- For every message, *M*, decrypting (using the corresponding private key) a message encrypted with a public key yields *M*
- Deriving the private key from the public key or the plaintext from the ciphertext is difficult
- Key generation, encryption, and decryption routines must be relatively fast

Implementing a Public-Key Cryptosystem

- Usually based on *trap-door one-way functions*, $f(x) = y$:
  - $f(x)$ is *one-way* if given x it is easy to compute y, but given y it difficult to determine x
  - $f(x)$ has a *trap-door* if there is a piece of information that allows x to be computed easily from y
  - Encryption = forward direction (anyone)
    - Public key
  - Decryption = backwards direction (only someone who knows the trap door)
    - Private key

The RSA Cryptosystem

- Rivest, Shamir, and Adleman, 1978
- Trap-door one-way function is factoring large integers (100 or 200 decimal digits)
  - Not proven that numbers must be factored to break RSA
  - Not proven that factoring large numbers will continue to be difficult
- RSA thought to be secure
- Widely used public-key cryptosystem

RSA - Overview

- Based on discrete exponentiation
- Encryption: $C = P^e \bmod n$
  - C and P are blocks of ciphertext and plaintext
  - e is a positive integer called the encryption exponent
  - n is a positive integer called the modulus
- The trap-door is p and q, the two prime factors of n
  - $n = p \times q$
- Knowledge if p and q allow one to compute d
  - d is a positive integer called the decryption exponent
- Decryption: $C^d \bmod n = P$

RSA – Mathematical Background

- A **prime** integer, *x*, has no factors by which it is evenly divisible except 1 and *x*:
  - 2, 3, 67, 491, and 2,347 are all prime
- A **composite** integer, *x*, has at least one other factor besides 1 and *x*:
  - 4 (2×2), 20 (2×2×5), 231 (3×7× 11), and 26,473 (23×1,151) are all composite
- Two integers, *x* and *y*, are **relatively prime** if their greatest common divisor is 1:
  - 2 and 5 are relatively prime, 4 and 35 are relatively prime

RSA – Mathematical Background (cont)

- Strategy #1 for determining whether or not two integers are relatively prime:
  - Create a prime factorization of each
  - Verify that the greatest common divisor (GCD) is 1
  - Examples:
    - 4 (1×2×2) and 35 (1×5×7) are relatively prime (GCD = 1)
    - 26,473 (1×23×1,151) and 249,711 (1×3×7×11×23×47) are not relatively prime (GCD = 23)
- Problem: Integer factorization is thought to be a hard problem
- Strategy #2 for determining whether or not two integers are relatively prime: Euclid's algorithm

RSA – Mathematical Background (cont)

- **Euclid's algorithm** - finds the GCD of two integers without factoring
- Example #1: 10,857 and 25,415
  - Reduce the larger modulo the smaller:
    25,415 mod 10,857 = 3,701
  - Reduce the modulus by the result:
    10,857 mod 3,701 = 3,455
  - Continue until the result is 0:
    3,701 mod 3,455 = 246
    3,455 mod 246 = 11
    246 mod 11 = 4
    11 mod 4 = 3
    4 mod 3 = 1 (GCD)
    3 mod 1 = 0
  - Second to last line is the GCD

RSA – Mathematical Background (cont)

- Euclid's algorithm - finds the GCD of two integers without factoring them
- Example #2: 2,856 and 1,320

  2,856 mod 1,320 = 216

  1,320 mod 216 = 24 (GCD)

  216 mod 24 = 0
- 2,856 and 1,320 are **not** relatively prime – their GCD is 24

RSA – Key Generation

- Randomly choose two large (probably) prime numbers, p and q
  - To make factoring "hard":
    - p and q should be of roughly equal length
    - p and q should be more than 100 decimal digits
    - p and q should be "hard" integers
- Example (using small integers): $p = 17$ and $q = 37$
- Compute the modulus, n, the product of p and q
  - Example: $n = p \times q = 17 \times 37 = 629$

RSA – Key Generation (cont)

- Randomly choose a large (probably) prime integer, *d*, as the decryption exponent:
  - *d* should be larger than *p* or *q*
  - *d* must be relatively prime to ((*p*-1) × (*q*-1))
  - Example
    - Recall: *p* = 17 and *q* = 37
    - So ((*p*-1) × (*q*-1)) = 16 × 36 = 576
    - *d* should be relatively prime to 576
      - GCD(*d*,576) must equal 1
  - Choose a random starting value for *d* (say 50) and start checking

RSA – Key Generation (cont)

- Use Euclid's Algorithm to find GCD(50,576):

  576 mod 50 = 26
  50 mod 26 = 24
  26 mod 24 = 2 (GCD)
  24 mod 2 = 0

- 50 and 576 are **not** relatively prime (GCD = 2)
- We cannot use *d*=50

RSA – Key Generation (cont)

- Use Euclid's Algorithm to find GCD(51,576):

  576 mod 51 = 15
  51 mod 15 = 6
  15 mod 6 = 3 (GCD)
  6 mod 3 = 0

- 51 and 576 are **not** relatively prime (GCD = 2)
- We cannot use *d*=51

RSA – Key Generation (cont)

- Use Euclid's Algorithm to find GCD(52,576):

  576 mod 52 = 4 (GCD)

  52 mod 4 = 0

- 52 and 576 are **not** relatively prime (GCD = 4)
- We cannot use *d*=52

RSA – Key Generation (cont)

- Use Euclid's Algorithm to find GCD(53,576):

576 mod 53 = 46
53 mod 46 = 7
46 mod 7 = 4
7 mod 4 = 3
4 mod 3 = 1 (GCD)
3 mod 1 = 0

- 53 and 576 are relatively prime (GCD = 1)
- Let the decryption exponent, *d*, be 53

RSA – Key Generation (cont)

- Generate the encryption exponent, e, such that e is the multiplicative inverse of d modulo $((p - 1) \times (q - 1))$
- A number, x, is the **multiplicative inverse** of another number, y, if the product of x and y is 1
  - E.g. 2 and ½, 9 and 1/9, 77/42 and 42/77
- A number, x, is y's **multiplicative inverse modulo z** if: $(x \times y) \bmod z = 1$
  - Example
    - 9 is a multiplicative inverse modulo 26 of 3 since $(9 \times 3) \bmod 26 = 1$
    - 35 is also a multiplicative inverse modulo 26 of 3 since $(35 \times 3) \bmod 26 = 1$
    - There is no multiplicative inverse modulo 26 for 4 since there is no integer, x, that satisfies $(x \times 4) \bmod 26 = 1$

RSA – Key Generation (cont)

- Facts:
  - If y and z are relatively prime then y has a multiplicative inverse modulo z
  - If y and z are not relatively prime then y has no multiplicative inverse modulo z
- Recall:
  - d and $((p\text{-}1) \times (q\text{-}1))$ were specifically chosen to be relatively prime
- Therefore:
  - d has a multiplicative inverse modulo $((p\text{-}1) \times (q\text{-}1))$

RSA – Extended Euclidean Algorithm

- **Extended Euclidean algorithm** - finds the multiplicative inverse of one integer modulo another
- Recall:

576 mod 53 = 46
53 mod 46 = 7
46 mod 7 = 4
7 mod 4 = 3
4 mod 3 = 1
3 mod 1 = 0

Another view:

(1) 576 – (10 × 53) = 46
(2) 53 – (1 × 46) = 7
(3) 46 – (6 × 7) = 4
(4) 7 – (1 × 4) = 3
(5) 4 – (1 × 3) = 1
(6) 3 – (3 × 1) = 0

RSA – Extended Euclidean Algorithm (cont)

- Start with line (5):
  - $4 – (1 \times 3) = 1$
- Substitute:
  - $(7 – (1 \times 4))$, a value equivalent to 3 according to line (4)
- For:
  - 3
- Gives:
  - $4 – (1 \times (7 – (1 \times 4))) = 1$
- Simplify (sum of 7s and 4s):
  - $((–1 \times 7) + (2 \times 4)) = 1$

(1) 576 – (10 × 53) = 46
(2) 53 – (1 × 46) = 7
(3) 46 – (6 × 7) = 4
(4) 7 – (1 × 4) = 3
(5) 4 – (1 × 3) = 1
(6) 3 – (3 × 1) = 0

RSA – Extended Euclidean Algorithm (cont)

- Previous result:
  - $((-1 \times 7) + (2 \times 4)) = 1$
- Substitute:
  - $(46-(6\times7))$, a value equivalent to 4 according to line (3)
- For:
  - 4
- Gives:
  - $((-1 \times 7) + (2 \times (46 - (6 \times 7)))) = 1$
- Simplify (sum of 46s and 7s):
  - $((2 \times 46) + (-13 \times 7)) = 1$

(1) $576 - (10 \times 53) = 46$
(2) $53 - (1 \times 46) = 7$
(3) $46 - (6 \times 7) = 4$
(4) $7 - (1 \times 4) = 3$
(5) $4 - (1 \times 3) = 1$
(6) $3 - (3 \times 1) = 0$

RSA – Extended Euclidean Algorithm (cont)

- Previous result:
  - $((2 \times 46) + (-13 \times 7)) = 1$
- Substitute:
  - $(53 – (1 \times 46))$, a value equivalent to 7 according to line (2)
- For:
  - 7
- Gives:
  - $((2 \times 46) + (-13 \times (53 – (1 \times 46)))) = 1$
- Simplify (sum of 53s and 46s):
  - $((-13 \times 53) + (15 \times 46)) = 1$

(1) 576 – (10 × 53) = 46
(2) 53 – (1 × 46) = 7
(3) 46 – (6 × 7) = 4
(4) 7 – (1 × 4) = 3
(5) 4 – (1 × 3) = 1
(6) 3 – (3 × 1) = 0

RSA – Extended Euclidean Algorithm (cont)

- Previous result:
  - $((-13 \times 53) + (15 \times 46)) = 1$
- Substitute:
  - $(576 – (10 \times 53))$, a value equivalent to 46 according to line (1)
- For:
  - 46
- Gives:
  - $((-13 \times 53) + (15 \times (576 – (10 \times 53)))) = 1$
- Simplify (sum of 576s and 53s):
  - $((15 \times 576) + (-163 \times 53)) = 1$

(1) $576 – (10 \times 53) = 46$
(2) $53 – (1 \times 46) = 7$
(3) $46 – (6 \times 7) = 4$
(4) $7 – (1 \times 4) = 3$
(5) $4 – (1 \times 3) = 1$
(6) $3 – (3 \times 1) = 0$

RSA – Extended Euclidean Algorithm (cont)

- Previous result:
  - $((15 \times 576) + (-163 \times 53)) = 1$
- Fact:
  - An expression of the form $ax + by = 1$ (with $a > 0$) tells us that a is x's multiplicative inverse modulo y
- Therefore, we know that:
  - 15 is 576's multiplicative inverse modulo 53
  - $(15 \times 576) \bmod 53 = 1$
- However, we are looking for 53's multiplicative inverse modulo 576

RSA – Extended Euclidean Algorithm (cont)

- Given:

  $((15 \times 576) + (-163 \times 53)) = 1$
- We know that:

  $(53 \times 576) + (-53 \times 576) = 0$
- Add $(53 \times 576) + (-53 \times 576)$ to left-hand side of the equation:

  $(15 \times 576) + (-163 \times 53) + (53 \times 576) + (-53 \times 576) = 1$
- Simplify:

  $((576 - 163) \times 53) + ((15 - 53) \times 576) = 1$
- Simplify further:
  - $((413 \times 53) + (-38 \times 576)) = 1$

RSA – Extended Euclidean Algorithm (cont)

- Previous result:
  - $((413 \times 53) + (-38 \times 576)) = 1$
- Fact:
  - An expression of the form $ax + by = 1$ (with $a > 0$) tells us that a is x's multiplicative inverse modulo y
- Therefore, we know that:
  - 413 is 53's multiplicative inverse modulo 576
  - $(413 \times 53) \bmod 576 = 1$
- Let the encryption exponent, e, be 413

RSA – Key Generation Summary

- Choose two large primes: p and q
  - $p = 17$ and $q = 37$
- Calculate the modulus, n:
  - $n = p \times q = 17 \times 37 = 629$
- Choose the decryption exponent, d, relatively prime to $((p\text{-}1) \times (q\text{-}1))$:
  - $d = 53$
- Compute e, d's multiplicative inverse mod $((p\text{-}1) \times (q\text{-}1))$:
  - $e = 413$
- Public key is (e, n), private key is d

RSA - Encryption

- Step 1:
  - Obtain the public key with which to encrypt the message
  - Let the public key be ($e = 413$, $n = 629$)
- Step 2:
  - Represent the plaintext as an integer, m, where $0 \leq m \leq n$
  - Let $m = 250$
- Step 3:
  - Create the ciphertext by computing: $C = m^e \bmod n$
  - $C = 250^{413} \bmod 629 = 337$

RSA - Decryption

- Need:
  - Ciphertext: $C = 337$
  - Public key: $e = 413$, $n = 629$
  - Private key: $d = 53$
- Decrypt by computing:

  $m = C^d \bmod n$

  $m = 337^{53} \bmod 629$

  $m = 250$

Attacks on RSA

- Assume an attacker knows:
  - The ciphertext ($C = 337$)
  - The public key ($e = 413$, $n = 629$) used to create C
- The attacker might attempt to determine:
  - A value for m that satisfies $m^{413} \bmod 629 = 337$
    - No known way to easily compute m given e, n, and C
    - Brute-force search for m is infeasible (if m is large)
  - A value for d
    - No known way to easily compute d given e and n
    - Brute-force search for d is infeasible (if d and n are large)

Attacks on RSA (cont)

- In general, it is believed that the most efficient way to attack RSA is to factor *n*, the modulus
  - Factoring *n* results in *p* and *q*
  - With *e*, *n*, *p*, and *q* the extended Euclidean algorithm can be used to compute *d*
- Factoring integers is widely believed to be an intractable problem

RSA - Security

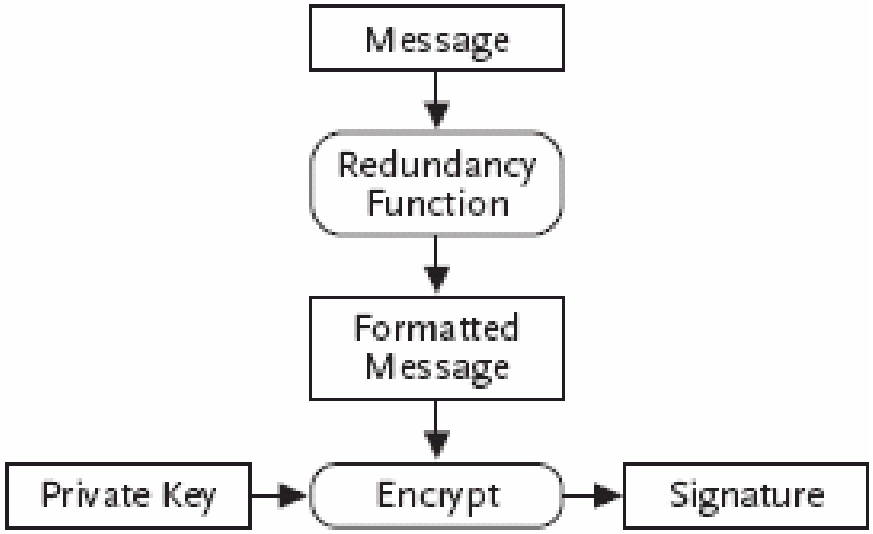
- We believe that:
  - In general, the most efficient way to attack RSA is to factor *n*, the modulus
  - In general, factoring large, "hard" integers is intractable
- However:
  - There may be an efficient way to attack RSA without factoring *n*, or
  - There may be an efficient algorithm for factoring *n*

Digital Signatures

- A **digital signature** indicates the signer's agreement with the contents of an electronic document
- Digital signatures should be: **authentic**, **unforgeable**, **non-reusable**, and **non-repudiable:**
  - Signer must deliberately sign a document
  - Only the signer can produce his/her signature
  - Cannot move a signature from one document to another document or alter a signed document without invalidating the signature
  - Signatures can be validated by other users, and the signer cannot reasonably claim that he/she did not sign a document bearing his/her signature

Digital Signatures - RSA

- Given an RSA public/private key pair and a message:
  - $e = 413, n = 629, d = 53, m = 250$
- Signature generation:

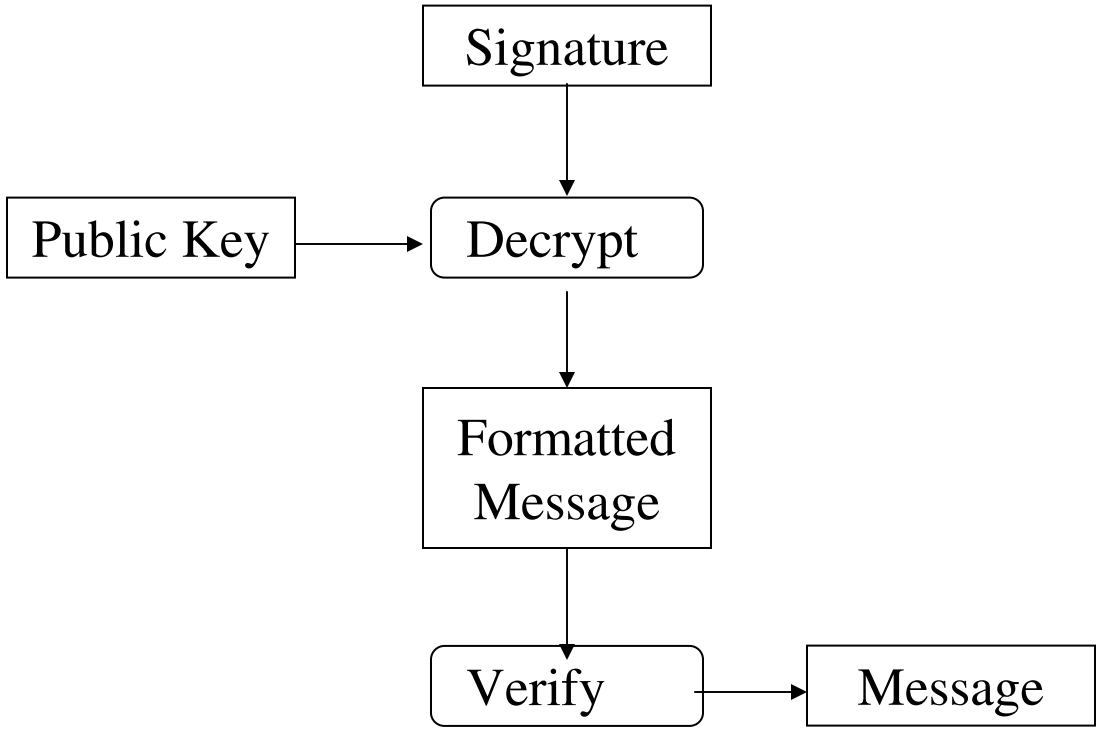


Digital Signatures – RSA (cont)

- Signature generation:
  - Step 1: Apply redundancy function, R
    - Redundancy function helps protect against signature forgery (as we shall see)
    - For now, we will use the simple (and insecure) identity redundancy function: $R(x) = x$
      - $m = 250$, $R(m) = 250$
  - Step 2: Encrypt $R(m)$ using the private key
    - $S = 250^{53} \bmod 629 = 411$
    - The digital signature, S, is 411

Digital Signatures – RSA (cont)

- Signature verification:



Digital Signatures – RSA (cont)

- RSA is a digital signature scheme with **message recovery**:
  - A signature can be verified without knowing the original message that was signed
  - Signature verification results in a copy of the original message
- Other digital signature schemes use an **appendix**:
  - The original message is required in order to verify the signature

Digital Signatures – RSA (cont)

- Signature verification:
  - Step 1: Decrypt the signature with the signer's public key
    - $R(m) = 411^{413} \bmod 629 = 250$
  - Step 2: Verify that the result has the proper redundancy specified by R (none in this case) and recover m
    - $R(m) = 250$
    - $m = 250$

Digital Signatures – RSA (cont)

- Problem: the redundancy function used in the last example is a bad one because it makes it easy to forge a signature
  - Choose a random value between 0 and n-1 for S
    - $S = 323$
  - Use the signer's public key to decrypt S:
    - $R(m) = 323^{413} \bmod 629 = 85$
  - Invert R to recover m:
    - $m = 85$
- Therefore:
  - A valid signature (323) can be created for a random message (85) without knowledge of the signer's private key

Digital Signatures – RSA (cont)

- Choosing a better redundancy function:
  - Consider: $R'(x) = \{x \text{ concatenated to } x\}$
    - To sign the message $m = 7$ we first apply R' to m:

      $R'(7) = 77$
    - Create the digital signature by encrypting $R'(m)$ with the private key

      $S = 77^{53} \bmod 629 = 25$
    - To verify this signature, we use the public key to decrypt:

      $R'(m) = 25^{413} \bmod 629 = 77$
    - Verify that $R'(m)$ is of the form xx for some message x
    - Invert R' and recover the original message: $m = 7$

Digital Signatures – RSA (cont)

- Choosing a better redundancy function:
  - Try to forge a signature with *R'* as the redundancy function
    - Choose a random value between 0 and *n*-1 for *S*
      - $S = 323$
    - Use the signer's public key to decrypt *S*:
      - $R(m) = 323^{413} \bmod 629 = 85$
  - Result:
    - 85 is not a legal value for $R'(m)$
    - 323 is not a valid signature
    - A good redundancy function (i.e. PKCS) makes forging a signature very difficult

The Digital Signature Standard (DSS)

- **Digital Signature Standard**
  - 1994 FIPS adopted by NIST
  - Includes a Digital Signature Algorithm (DSA) based on ElGamal
  - Cannot be used for encryption – only for digital signatures
  - Digital signature scheme with appendix
    - The original message is required in order to verify the signature

DSS – Key Generation

- A public/private key pair must be generated:
  - A 160-bit prime number, q, is selected
    - Small example: $q = 72$
  - A prime number, p, is selected
    - p must be either 512, 576, 640, 704, 768, 832, 896, 960, or 1,024 bits
    - q must be a factor of $(p - 1)$
  - Example using small numbers:
    - $q = 72$, $p = 58{,}537$
    - Note: 58,536 / 72 = 813 so q is a factor of $(p-1)$

DSS – Key Generation (cont)

- An integer, h, is randomly selected from the range $1 \ldots p - 1$
- g is computed from h, p, and q:

  $g = h^{(p-1)/q} \bmod p$
- Example using small numbers:

  $q = 72$, $p = 58{,}537$, $h = 471$

  $g = 471^{58536/72} \bmod 58{,}537$

  $= 471^{813} \bmod 58{,}537$

  $= 26{,}994$

DSS – Key Generation (cont)

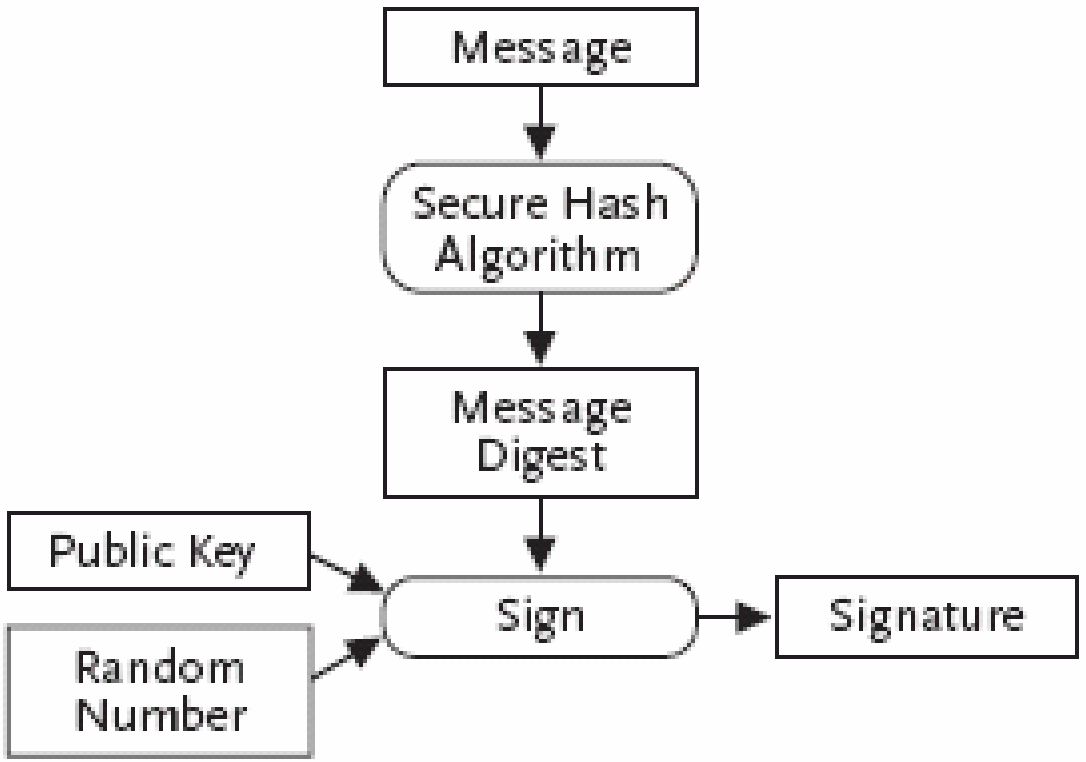
- A random integer, x, is chosen such that $0 < x < q$
- y is computed using g, x, and p

  $y = g^x \bmod p$
- Example using small numbers:

  $q = 72$, $p = 58{,}537$, $h = 471$, $g = 26{,}994$, $x = 61$

  $y = 26{,}994^{61} \bmod 58{,}537 = 4{,}105$
- Public key = (p, q, g, y), private key = x

DSS – Signature Generation



DSS – Signature Generation (cont)

- Given the public key:
  - $p = 58{,}537$, $q = 72$, $g = 26{,}994$, $y = 4{,}105$
- Select a positive random integer, k, that is less than q
  - Example using small numbers: $k = 29$
- A different value for k must be chosen each time a message is to be signed
- Compute one part of the signature:

$$r = (g^k \bmod p) \bmod q$$
$$= (26{,}994^{29} \bmod 58{,}537) \bmod 72$$
$$= 49$$

DSS – Signature Generation (cont)

- Compute the multiplicative inverse of k (29) mod q (72)
  - $(5 \times 29) \bmod 72 = 1$
  - $k^{-1} = 5$
- The message to be signed, m, is hashed using the Secure Hash Algorithm
  - $MD = \text{SHA}(m)$
  - Example using small numbers: $\text{SHA}(m) = 6{,}034$

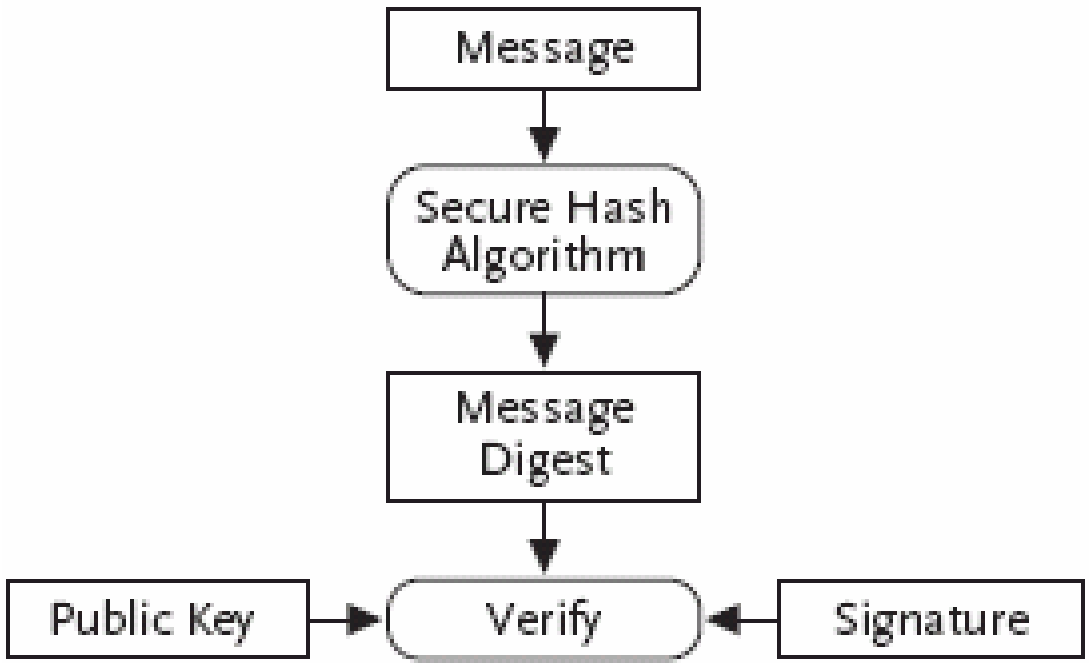
DSS – Signature Generation (cont)

- Using the public and private keys:
  - Public: $p = 58{,}537$, $q = 72$, $g = 26{,}994$, $y = 4{,}105$
  - Private: $x = 61$
- Compute the second part of the signature:

$$s = (k^{-1} \times (MD + (x \times r))) \bmod q$$
$$= (5 \times (6{,}034 + (61 \times 49))) \bmod 72$$
$$= (5 \times (6{,}034 + 2{,}989)) \bmod 72$$
$$= (5 \times 9{,}023) \bmod 72$$
$$= 45{,}115 \bmod 72$$
$$= 43$$

- The two values, r (49) and s (43), are the digital signature of m

DSS – Signature Verification



DSS – Signature Verification (cont)

- DSS is a digital signature scheme with appendix
  - The original message is required in order to verify the signature
- Given r, s, m, and the signer's public key
- Anyone can verify that (r, s) is a valid signature on m:
  - Verify that $0 < r < q$ and $0 < s < q$
  - Compute the message digest of m using SHA
    - MD = 6,034

DSS – Signature Verification (cont)

- Compute w, the multiplicative inverse of s (42) modulo q (72):

  $(67 \times 42) \bmod 72 = 1$

  $w = 67$

- Compute $u_1 = (MD \times w) \bmod q$

  $u_1 = (6{,}034 \times 67) \bmod 72$

  $= 404{,}278 \bmod 72$

  $= 70$

- Compute $u_2 = (r \times w) \bmod q$

  $u_2 = (49 \times 67) \bmod 72$

  $= 3{,}283 \bmod 72$

  $= 43$

DSS – Signature Verification (cont)

- Compute the value v:

  $v = ((g^{u1} \times y^{u2}) \bmod p) \bmod q$

  $= ((26{,}994^{70} \times 4{,}105^{43}) \bmod 58{,}537) \bmod 72$

  $= 14{,}809 \bmod 72$

  $= 49$

- If v (49) equals r (49) then the signature is verified
  - The message m was signed by someone who knows x, the private key corresponding to y

Symmetric vs. Asymmetric Cryptosystems

- Public-key cryptosystems usually:
  - Have keys that are about 10 times bigger
    - 1,024 bits vs. 56-128 bits
  - Performs encryption 100-1000 times slower
    - Due to more complicated operations
  - Simplifies key management: requires no previously established, shared secrets
  - Improves scalability: a group of m agents needs only $2m$ total keys (vs. m^2)
  - Allows digital signatures to be created and verified

Summary

- Public-key cryptosystems use different keys to encrypt and decrypt messages
  - Simplifies key distribution and management
  - Facilitates the creation of digitally signed messages
- RSA
  - Proposed in1978
  - Can be used for encryption and digital signatures
- DSS
  - Adopted in 1994
  - Can be used for digital signatures

Overview

- Beyond basic cryptography:
  - **Secret splitting** - divide a message into *n* pieces, such that all *n* pieces must be combined to recover the message
  - **Cryptographic protocols** make use of cryptography to accomplish some task securely
    - Authentication
    - Key-exchange

Motivation for Secret Splitting

- A professor, Carol, encrypts her grade file with a symmetric-key cryptosystem
  - Good: only Carol can read grades (privacy)
  - Good: only Carol can modify grades (integrity)
  - Bad: if Carol becomes incapacitated nobody else can recover the grades
- Carol needs some kind of a mechanism that will allow someone other than her to decrypt the grade file in case of an emergency

Secret Splitting

- **Secret splitting** makes it possible to divide a message into *n* pieces called **shadows**, such that:
  - Combining less than *n* shadows yields nothing
  - Combining all *n* shadows yields the message
- Carol can split her key into *n* shadows and give one to *n* different people she trusts:
  - Good: if Carol becomes incapacitated all *n* people can get together and recover the grade file
  - Good: Unlikely that all *n* people would betray Carol's trust

Secret Splitting Using One-Time Pads

- M = "THEKEYISTHREE"
- Create four shadows:
- Generate n-1 one-time pads (as long as M):
  - P_1 = PDJEUVNSKTUEG
  - P_2 = NBEXUYYKPAQJZ
  - P_3 = ICMKELDAOFGMC
- Encrypt M with P_1:
  - C_1 = JLOPZUYLEBMJL
    - T (20) + P (16) mod 26 = J (10)
    - H (8) + D (4) mod 26 = L (12)
    - E(5) + J (10) mod 26 = O (15)

    …

Secret Splitting Using One-Time Pads (cont)

- Encrypt C_1 with P_2:
  - C_2 = XNTNUTXWUCDTL
- Encrypt C_2 with P_3:
  - C_3 = GQGYZFBXJIKGO
- P_1, P_2, P_3, and C_3 are the four shadows
- Good: all four shadows are required to reconstruct M:
  - Use P_3 to decrypt C_3 yielding C_2
  - Use P_2 to decrypt C_2 yielding C_1
  - Use P_1 to decrypt C_1 yielding M
- Bad: What happens if Carol and one of the shadow holders become incapacitated?

Secret Sharing

- **Secret sharing** (also called a **threshold scheme**) makes it possible to divide a message into *n* shadows, such that:
  - Combining less than *k* shadows yields nothing
  - Combining any *k* (or more) yields the message
- Example:
  - Carol uses a (3-8)-threshold scheme to divide her key into eight shadows (any three required to recover *M*)
  - Give three to Alice, one to Bob, two to Dave, and one each to Elvis and Fred
  - Carol's key can be recovered by:
    - Alice (3)
    - Dave (2) and Bob (1)
    - Bob (1), Elvis (1), and Fred (1)
    - Etc.
- Good: Need some, not all, shadows to recover the key

Cryptographic Protocols

- **Protocol** an agreed-upon sequence of actions performed by two or more principals
  - **Cryptographic protocols** make use of cryptography to accomplish some task securely
- Example:
  - How can Alice and Bob agree on a session key to protect a conversation?
  - Answer: use a key-exchange cryptographic protocol

Key Exchange with Symmetric Cryptography

- Assume Alice and Bob each share a key with a Key Distribution Center (KDC)
  - K_A is the key shared by Alice and the KDC
  - K_B is the key shared by Bob and the KDC
- To agree on a session key:
  - Alice contacts the KDC and requests a session key for Bob and her
  - KDC generates a random session key, encrypts it with both K_A and K_B, and sends the results to Alice

Key Exchange with Symmetric Cryptography (cont)

- Agreeing on a session key (cont):
  - Alice decrypts the part of the message encrypted with K_A and learns the session key
  - Alice sends the part of the message encrypted with K_B to Bob
  - Bob receives Alice's message, decrypts it, and learns the session key
  - Alice and Bob communicate securely using the session key

Key Exchange with Symmetric Cryptography (cont)

- The key-exchange protocol:

A: => KDC (A,B);

KDC: => A ($E(K_{AB},K_A)$, $E(K_{AB},K_B)$);

A: => B ($E(K_{AB},K_B)$);

Key Exchange with Symmetric Cryptography (cont)

- Issues:
  - Security depends on secrecy of K_A and K_B
    - KDC must be secure and trusted by both Alice and Bob
    - K_A and K_B should be used sparingly
  - The use of a new session key for each conversation limits the chances/value of compromising a session key

Attacking the Protocol

- Alice and Bob set up a secure session protected by K_{AB}
- An intruder, Mallory, watches them do this and stores the KDC's message to Alice and all the subsequent messages between Alice Bob encrypted with K_{AB}
- Mallory cryptanalyzes the session between Alice and Bob and eventually recovers K_{AB}
- The next time Alice and Bob want to talk Mallory intercepts the KDC's reply and replays the old message containing K_{AB}
- Alice and Bob conduct a "secure" conversation which is protected by K_{AB} which is known to Mallory

Attacking the Protocol (cont)

A: => KDC (A,B);

KDC: => A ($E(K_{AB},K_A)$, $E(K_{AB},K_B)$);

A: => B ($E(K_{AB},K_B)$);

// Alice and Bob encrypt their messages using K_{AB}

// Mallory recovers K_{AB} by analyzing Alice and Bob's session

A: => KDC (A,B);

KDC: => A ($E(K_{AB'},K_A)$, $E(K_{AB'},K_B)$);

// Mallory intercepts the above message and replaces it

M: => A ($E(K_{AB},K_A)$, $E(K_{AB},K_B)$);

A: => B ($E(K_{AB},K_B)$);

// Mallory reads all traffic session between Alice and Bob

What Went Wrong?

- Alice and Bob need to be able to distinguish between a current (or **fresh**) response from the KDC and an old one
- Solutions:
  - Alice and Bob could keep track of all previously-used session keys and never accept an old session key
  - KDC could include freshness information in its messages
    - Timestamps
    - Nonces

Using Timestamps to Establish Freshness

A: => KDC (A,B);

KDC: => A ($E((K_{AB},T_{KDC}),K_A)$, $E((K_{AB},T_{KDC}),K_B)$);

A: => B ($E((K_{AB},T_{KDC}),K_B)$);

Where T_{KDC} is a timestamp from the KDC's clock and:

- Alice and Bob's clocks are both synchronized with the KDC's
- Alice and Bob both check the KDC's message to make sure it was generated recently

Using Nonces to Establish Freshness

- A **nonce** is a randomly-generated value that:
  - Is never reused
  - Can be used to prove the freshness of a message

A: => KDC (A,B,N_A);

B: => KDC (A,B, N_B);

KDC: => A (E((K_{AB},N_A),K_A));

KDC: => B (E((K_{AB},N_B),K_B));

Key-Exchange with Public-Key Cryptography

- Alice learns Bob's public key (by either asking Bob or some third party)
- Alice generates a random session key, K_{AB}
- Alice encrypts the session key with Bob's public key
- Alice sends Encrypt(K_{AB},B_{Public}) to Bob
- Bob receives Alice's message and decrypts it with his private key
- Alice and Bob encrypt their subsequent communications with K_{AB}

Attacking the Protocol

- Recall the man-in-the-middle attack
  - If Mallory can trick Alice into thinking that M_{Public} is Bob's public key
    - Mallory can decrypt Alice's first message to Bob Encrypt(K_{AB},M_{Public})
    - Mallory learns the proposed session key K_{AB}
    - Mallory can send Bob: Encrypt(K_{AB},B_{Public})
    - Alice and Bob will encrypt their subsequent communications with K_{AB} thinking that it is secure
- This is a very serious problem because it's often difficult to be sure you know somebody's public key

The Interlock Protocol

- Combating the man-in-the-middle attack:
  - Alice and Bob exchange public keys
  - Alice encrypts her message using Bob's public key. Alice sends half the encrypted message to Bob (e.g. every other bit)
  - Bob encrypts his message using Alice's public key. Bob sends half the encrypted message to Alice (e.g. every other bit)
  - Alice sends the other half of her encrypted message to Bob. Bob puts the two halves together and decrypts them using his private key
  - Bob sends the other half of his encrypted message to Alice. Alice puts the two halves together and decrypts them using her private key

The Interlock Protocol (cont)

- Foiling the man-in-the-middle:
  - Assume Mallory can trick Alice into using M_{Public} instead of B_{Public}
  - When Mallory receives the first half of Alice's message she won't be able to decrypt it and re-encrypt it with B_{Public}
  - Mallory must invent a completely new message, encrypt it and send half of it to Bob
  - When the second half of Alice's message arrives, Mallory can put the two halves together, decrypt, and learn what Alice's original message was
  - However, Mallory has already committed to the first half of the message and it is too late to change
  - Therefore, Bob will not get the message Alice sent, and Alice and Bob will probably be able to figure out that there is an intruder between them

Authentication

- Process of proving your identity to someone else
  - One-way
  - Two-way
- Often, authentication protocols are designed using a **challenge and response** mechanism
  - Authenticator creates a random challenge
  - Authenticatee proves identity by replying with the appropriate response

One-way Authentication Using Symmetric-Key Cryptography

- Assume that Alice and Bob share a secret symmetric key, K_{AB}
- One-way authentication protocol:
  - Alice creates a nonce, N_A, and sends it to Bob as a challenge
  - Bob encrypts Alice's nonce with their secret key and returns the result, $Encrypt(N_A, K_{AB})$, to Alice
  - Alice can decrypt Bob's response and verify that the result is her nonce

A: => B(N_A);

B: => A($Encrypt(N_A, K_{AB})$);

One-way Authentication Using Symmetric-Key Cryptography

- Problem: an adversary, Mallory, might be able to impersonate Bob to Alice:
  - Alice sends challenge to Bob (intercepted by Mallory)
  - Mallory does not know K_{AB} and thus cannot create the appropriate response
  - Mallory may be able to trick Bob (or Alice) into creating the appropriate response for her:

A: => M(N_A);
M: => B(N_N);
B: => M(*Encrypt*(N_A, K_{AB}));
M: => A(*Encrypt*(N_A, K_{AB}));

One-way Authentication Using Public-Key Cryptography

- Alice sends a nonce to Bob as a challenge
- Bob replies by encrypting the nonce with his private key
- Alice decrypts the response using Bob's public key and verify that the result is her nonce

  A: => B(N_A);

  B: => A(*Encrypt*(N_A, $B_{Private}$));

- Encrypting any message that someone sends as an authentication challenge might not be a good idea

One-way Authentication Using Public-Key Cryptography

- Another challenge-and-response authentication protocol:
  - Alice performs a computation based on some random numbers (chosen by Alice) and her private key and sends the result to Bob
  - Bob sends Alice a random number (chosen by Bob)
  - Alice makes some computation based on her private key, her random numbers, and the random number received from Bob and sends the result to Bob
  - Bob performs some computations on the various numbers and Alice's public key to verify that Alice knows her private key
- Advantage: Alice never encrypts a message chosen by someone else

Authentication and Key-Exchange Protocols

- Combine authentication and key-exchange
- Assume Carla and Diane are on opposite ends of a network and want to talk securely
  - Want to agree on a new session key securely
  - Want to each be sure that they are talking to the other and not an intruder
- Wide-Mouth Frog
  - Assumes a trusted third-party, Sam, who shares a secret keys, K_C and K_D, respectively, with Carla and Diane

Authentication and Key-Exchange Protocols

- Wide-Mouth Frog

C: => S(C,*Encrypt*((D,K_{CD},T_C),K_{CS}));
S: => D(*Encrypt*((C, K_{CD}, T_S), K_{DS}));

- Observations:
  - Reliance on synchronized clocks to generate timestamps
  - Depends on a third-party that both participants trust
  - Initiator is trusted to generate good session keys

Authentication and Key-Exchange Protocols

- Yahalom

  $C \Rightarrow D\ (C, N_C)$;

  $D \Rightarrow S\ (D, \text{Encrypt}((C, N_C, N_D), K_D))$;

  $S \Rightarrow C\ (\text{Encrypt}((D, K_{CD}, N_C, N_D), K_C), \text{Encrypt}((C, K_{CD}), K_D))$;

  $C \Rightarrow D\ (\text{Encrypt}((C, K_{CD}), K_D), \text{Encrypt}(N_D, K_{CD}))$;

- Note: Diane is the first one to contact Sam who only sends one message to Carla

Authentication and Key-Exchange Protocols

- Denning and Sacco (public-key)
  - Carla sends a message to Sam including her name and Diane's name
  - Sam replies with signed copies of both Carla and Diane's public key

    C: => S(C,D);

    S: => C(Encrypt((C,C_{Public},T_S),$S_{Priavte}$),Encrypt((D,D_{Public},T_S),$S_{Priavte}$));

    C: => D(Encrypt((C,C_{Public},T_S),$S_{Priavte}$),Encrypt((D,D_{Public},T_S),$S_{Priavte}$));
  - Carla generates the session key, K_{CD}, and signed a message containing it and a timestamp with her private key

    C: => D(*Encrypt*(*Encrypt*((K_{CD},T_C),$C_{Private}$),D_{Public}));

Authentication and Key-Exchange Protocols

- A weakness of the Denning and Sacco protocol
  - Harry can trick Diane into thinking that she is communicating with Carla when she is really communicating with Harry
  - Harry establishes a session key, K_{CH}, with Carla

    C: => H(*Encrypt*(*Encrypt*((K_{CH},T_C),$C_{Private}$),H_{Public}));
  - Harry decrypts Carla's message and learns K_{CH}
  - Harry encrypts Carla's signed message with Diane's public key, and sends the result to Diane claiming to be Carla

    H: => D(*Encrypt*(*Encrypt*((K_{CH},T_C),$C_{Private}$),D_{Public}));
  - Diane will decrypt the message, check the signature and timestamp, and believe that she is talking to Carla with K_{CH} as the session key

Authentication and Key-Exchange Protocols

- Fixing the Denning and Sacco protocol:
  - Add the other party's name to the key exchange message:
    - C: => D($Encrypt(Encrypt((D,K_{CD},T_C),C_{Private}),D_{Public})$);