**Justin Ciocoi Nov. 6, 2023**

# CSCI 360 Textbook Notes

## Chapter 6: Intro to Public Key Cryptography

#### 6.1: Symmetric vs. Asymmetric Cryptography

- Asymmetric, public-key algorithms are very different from symmetric algorithms such as AES or DES
- The symmetric algorithms that are widely in use today are very secure and fast, yet they do pose a number of disadvantages
    - *Key distribution problem*, such that sending the key over the secure channel is not possible given that the key is necessary to make the channel secure in the first place
    - *Number of Keys*, since for large symmetric networks, he number of keys can become a burden on the network capabilities
        - In a network with $n$ users, the number of keys is equal to
        - $$\frac{n \cdot (n-1)}{2}$$
        - Additionally, every user has to securely store $n-1$ keys
    - Symmetric cryptography also does not provide any *nonrepudiation*, meaning a user cannot deny having sent a message
        - For example, since each user has the same private key, one of the users could fraudulently produce a message seemingly coming from the other
    - In public key cryptography each user has a public encryption key, $k_{pub}$ which they publicly broadcast, as well as a matching private key, $k_{priv}$, which is used for decryption
    - This is shown in a very basic sense in the below diagram
    -

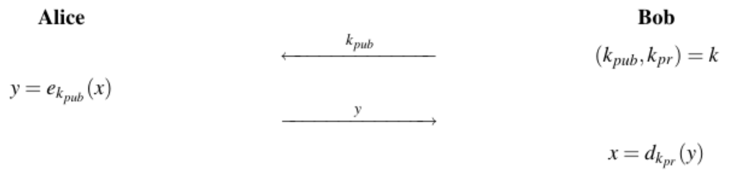


**Fig. 6.4** Basic protocol for public-key encryption

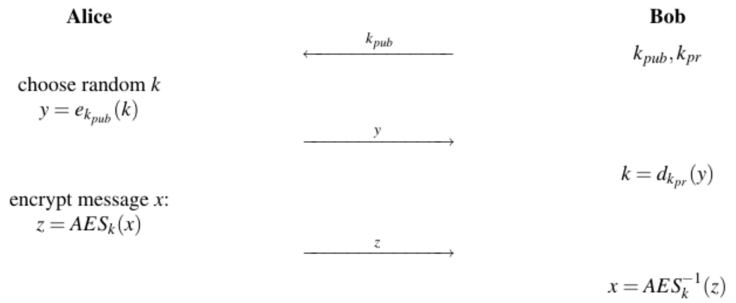
- The asymmetric protocol described here can also be modified in order to allow for symmetric key cryptography over an asymmetric protocol
  - Using the public-key algorithm, $y = e_{k_{pub}}(x)$, we can encrypt a symmetric key, like an AES key
  - Then, we can have the users securely exchange that symmetric key over the asymmetric protocol
  - Users can then use it to securely communicate
- This is shown below
- 

**Fig. 6.5** Basic key transport protocol with AES as an example of a symmetric cipher

- **Definition 6.1.1:**
  - A function $f()$ is a *one-way* function if:
    - $y = f(x)$ is computationally easy
    - $x = f^{-1}(y)$ is computationally infeasible
- There are two popular one way functions which are used in practical public-key cryptography schemes
- The first is the integer factorization problem, on which RSA is based
- The other is the discrete logarithm function

### 6.2: Practical Aspects of Public-Key Cryptography

- **6.2.1: Security Mechanisms**
  - The main functions that public-key cryptography algorithms provide are
    - *Key Establishment*, such that a secret key can be established over insecure channels
    - *Nonrepudiation*, which is realized with digital signature algorithms
    - *Identification* using challenge-and-response protocols along with digital signatures
    - *Encryption* as outlined in the previous section
  - The major drawback of using public-key algorithms is that encryption is very computationally intensive, and thus extremely slow
    - Because of this, we often use symmetric cryptography to provide encryption and identification while asymmetric cryptography provides key establishment and nonrepudiation, resulting in a *hybrid network*
- **6.2.2: The Remaining Problem: Authenticity of Public Keys**
  - The problem of ensuring a particular public key belongs to a certain person can be solved using *certificates*, which, roughly speaking, bind public keys to certain identities
  - This is often used for e-commerce applications over the internet
- **6.2.3: Important Public-Key Algorithms**
  - In the world of public key cryptography, there are only three major families of public-key algorithm which are of practical relevance
  - These problems are all based on one of the three computational problems outlined below
    - *Integer Factorization Schemes*, which is based on the idea that it is difficult to factor large integers and is prominently used in RSA
    - *Discrete Logarithm Schemes*, which include the Diffie-Hellman key exchange or the Digital Signatures Algorithm
    - *Elliptical Curve (EC) Schemes*, which are a generalization of the discrete logarithm algorithm
- **6.2.4: Key Lengths and Security Levels**

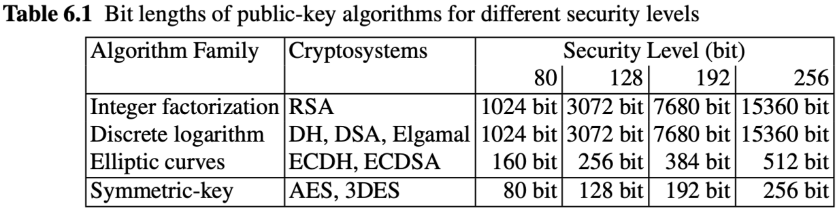
    - Not surprisingly, the longer the operands and keys are in a public-key cryptographic scheme, the more secure the algorithms become
    - An algorithm is said to have $n$ bits of security when the best known attack on the algorithm requires $2^n$ operations
    - In symmetric cryptography, a cipher with key length of 256 bits has 256 bits of security, but the situation with asymmetric cryptography is not quite as simple
    - 

**Table 6.1** Bit lengths of public-key algorithms for different security levels

| Algorithm Family | Cryptosystems | Security Level (bit) | | | |
|---|---|---|---|---|---|
| | | 80 | 128 | 192 | 256 |
| Integer factorization | RSA | 1024 bit | 3072 bit | 7680 bit | 15360 bit |
| Discrete logarithm | DH, DSA, Elgamal | 1024 bit | 3072 bit | 7680 bit | 15360 bit |
| Elliptic curves | ECDH, ECDSA | 160 bit | 256 bit | 384 bit | 512 bit |
| Symmetric-key | AES, 3DES | 80 bit | 128 bit | 192 bit | 256 bit |

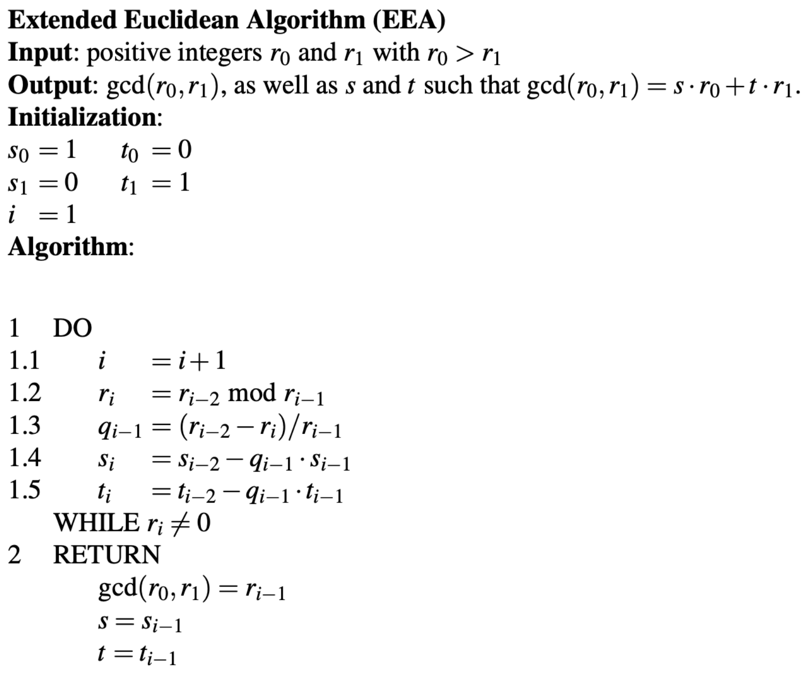
### 6.3: Essential Number Theory for Public Key Algorithms

- **6.3.1: Euclidean Algorithm**
    - We will begin with the problem of computing the greatest common divisor, or GCD, of two positive integers, $r_0$ and $r_1$, which is denoted by $gccd(r_0, r_1)$
    - For example, let $r_0 = 84$ and $r_1 = 30$
    - Factoring yields:
        - $r_0 = 84 = 2 * 2 * 3 * 7$
        - $r_1 = 30 = 2 * 3 * 5$
    - The GCD is the product of all common prime factors, so in this case
        - $gcd(30, 84) = 2 * 3 = 6$
    - Euclid's Algorithm posits that $gcd(r_0, r_1) = gcd(r_0 \ mod \ r_1, r_1)$
    - Let us now define Euclid's algorithm in a more formal fashion
        - **Input:** Positive integers $r_0$ and $r_1$ with $r_0 > r_1$
        - **Output:** $gcd(r_0, r_1)$
        - **Initialization:** $i = 1$
        - **Algorithm:**

        - Do
            - $i = i + 1$
            - $r_i = r_{i-2} \; mod \; r_{i-1}$
        - While $r_i \neq 0$
        - Return
            - $gcd(r_0, r_1) = r_{i-1}$

- **6.3.2: Extended Euclidean Algorithm**
    - In addition to computing the GCD of two integers, the extended Euclidean algorithm also computes a linear combination of the form
    - $$gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$$
    - Let us demonstrate with the problem $gcd(973, 301)$
        - $gcd(973, 301) = 973s + 301t$
        - $1 * 973 - 3 * 301 = 973 - 903 = 70$
        - $1 * 301 - 4 * 70 = 301 - 280 = 21$
            - $21 = 1 * 301 - 4 * (1 * 973 - 3 * 301)$
        - $1 * 70 - 3 * 21 = 70 - 63 = 7$
            - $7 = (973 - 3 * 301) - 3(301 - 4(973 - 3 * 301))$
            - let $973 = x, 301 = y$
            - $7 = (x - 3y) - 3(y - 4(x - 3y))$
                - $= (x - 3y) - 3(13y - 4x)$
                - $= (x - 3y) - 39y + 12x$
                - $7 = 13x - 42y$
    - Let us now try to find $12^{-1} \; mod \; 67$
        - $gcd(67, 12) = gcd(12, 7) = gcd(7, 5) = gcd(5, 2) = gcd(2, 1) = 1$

    - $gcd(67, 12) = 67s + 12t$
    - $1 * 67 - 5 * 12 = 7$
    - $1 * 12 - 1 * 7 = 5$
      - $5 = 1 * 12 - 1(1 * 67 - 5 * 12)$
    - $2 = 1 * 7 - 1 * 5$
      - $2 = (1a - 5b) - (6b - a)$
    - let $67 = a, 12 = b$
    - $1a - 5b = 7$
    - $1b - 1(1a - 5b) = 5$
      - $5 = 6b - a$
    - $2 = 2a - 11b$
    - $1 = 1 * 5 - 2 * 2$
      - $1 = (6b - a) - 2(2a - 11b)$
      - $1 = 28b - 5a$
      - $1 = 28(12) - 5(67)\ mod\ 67 = 28(12) - 0 = 1$
      - Thus, 28 is the inverse of 12 in mod 67
  - Below is the generalized algorithm for the extended Euclidean algorithm
  -



**Extended Euclidean Algorithm (EEA)**
**Input**: positive integers $r_0$ and $r_1$ with $r_0 > r_1$
**Output**: $\gcd(r_0, r_1)$, as well as $s$ and $t$ such that $\gcd(r_0, r_1) = s \cdot r_0 + t \cdot r_1$.
**Initialization**:

```
$s_0 = 1$    $t_0 = 0$
$s_1 = 0$    $t_1 = 1$
$i = 1$
```

**Algorithm**:

```
1     DO
1.1       $i = i + 1$
1.2       $r_i = r_{i-2} \bmod r_{i-1}$
1.3       $q_{i-1} = (r_{i-2} - r_i)/r_{i-1}$
1.4       $s_i = s_{i-2} - q_{i-1} \cdot s_{i-1}$
1.5       $t_i = t_{i-2} - q_{i-1} \cdot t_{i-1}$
      WHILE $r_i \neq 0$
2     RETURN
          $\gcd(r_0, r_1) = r_{i-1}$
          $s = s_{i-1}$
          $t = t_{i-1}$
```

- The main application of this algorithm in asymmetric cryptography is to compute the inverse modulo of an integer
- If we recall that a modular inverse exists if and only if $gcd(r_0, r_1) = 1$
  - Hence, when we apply the extended Euclidean algorithm, we get $s \cdot r_0 + t \cdot r_1$
  - When we take this equation in modulo $r_0$, we are left with
  - $$s \cdot r_0 + t \cdot r_1 = 1$$ $$s \cdot 0 + t \cdot r_1 = 1 \; mod \; r_0$$ $$r_1 \cdot t = 1 \; mod \; r_0$$
  - Which means that $t$ is the inverse of $r_1$ in modulo $r_0$

- **6.3.3: Euler's Phi Function**
  - Let's now consider the ring $\mathbb{Z}_m$, or the set of integers $[0, 1, ..., m-1]$
  - We will be interested in the problem of figuring out *how many* numbers in this set are relatively prime to $m$
  - We can solve this problem using Euler's Phi function, which is defined below
    - **Definition 6.3.1:** *Euler's Phi Function*
    - The number of integers in $\mathbb{Z}_m$ relatively prime to $m$ is denoted by
    - $$\Phi(m)$$

    - Calculating Euler's phi function by running through all the elements and computing the GCD becomes extremely slow when the numbers are large, and completely infeasible when dealing with the magnitude of numbers involved in public-key cryptography
    - Fortunately, we can calculate $\Phi(m)$ if we know the factorization of $m$, which is given in the following theorem
        - Given the following canonical factorization of $m$
        - $$m = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_n^{e_n}$$
        - where the $p_i$ are distinct prime factors and $e_i$ are positive integers, then
        - $$\Phi(m) = \prod_{i=1}^{n} (p_i^{e_i} - p_i^{e_i - 1})$$
- **6.3.4: Fermat's Little Theorem and Euler's Theorem**
    - *Fermat's Little Theorem* is a useful theorem in public-key cryptography that deals with primality testing and many other things
    - **Theorem 6.3.2:** *Fermat's Little Theorem*
        - Let $a$ be an integer and $p$ be a prime, then:
        - $$a^p = a \; mod \; p$$
        - The theorem can also be stated in this form
        - $$a^{p-1} \equiv 1 \; mod \; p$$
    - This theorem also gives us a simple way to find the modular inverse of a number
    - For an integer $a$, in mod $p$, $a^{-1} \equiv a^{p-2} \; mod \; p$
    - We can also generalize Fermat's Little Theorem to any integer modulo, not just primes
    - This generalization is *Euler's Theorem*
    - **Theorem 6.3.3:** *Euler's Theorem*
        - Let $a$ and $m$ be integers with $gcd(a, m) = 1$, then:
        - $$a^{\Phi(m)} \equiv 1 \; mod \; m$$

**Justin Ciocoi**

**Nov. 9, 2023**

# CSCI 360 Textbook Notes

## Chapter 7: The RSA Cryptosystem

#### 7.1: Introduction

- The RSA Crypto scheme is currently the most widely used asymmetric cryptographic scheme
- There are many different applications for RSA, but in practice it is most often used for
    - Encryption of *small* pieces of data, especially when dealing with *key transport*
    - Digital signatures, which are used for digital certificates on the internet
- It should be noted however, that RSA is not intended to replace symmetric cryptography due to the fact that RSA is several times slower as ciphers such as AES
    - This is because of the many computations which are involved in the RSA algorithm
- The underlying one-way function of RSA is the integer factorization problem:
    - Multiplying two large primes is computationally easy
    - Factoring the resulting problem is computationally hard

#### 7.2: Encryption and Decryption

- **RSA Encryption**
    - Given the public key $(n, e) = k_{pub}$ and the plaintext, $x$, the encryption function is:

$$y = e_{k_{pub}}(x) \equiv x^e \ mod \ n$$

where $x, y \in \mathbb{Z}_n$

---

- **RSA Decryption**

- Given the private key $d = k_{pr}$ and the ciphertext $y$, the decryption function is:

  $$x = d_{k_{pr}}(y) \equiv y^d \; mod \; n$$

  where $x, y \in \mathbb{Z}_n$

---

- In practice, the variables, $x, y, n, d$, are very long numbers, usually 1024 bits long or more
- $e$ is often referred to as the *encryption exponent* or *public exponent*, and $d$ is often referred to as the *decryption exponent* or *private exponent*
- Let us say user A wants to send a message to user B
  - User A must first have user B's public key, $(n, e)$
  - User B will decrypt the message using their private key $d$
- We can now state a few requirements for the RSA cryptosystem
  - Since an attacker has access to the public key, it must be computationally infeasible to calculate $d$ given public key values $e$ and $n$
  - Since $x$ is only unique up to the size of the modulus $n$, more than $l$ bits cannot be encrypted with one RSA encryption, where $l$ is the bit length of $n$
  - It should be relatively easy to calculate $x^e \; mod \; n$ (encryption) and $y^d \; mod \; n$ (decryption)
    - This means we need a method for fast exponentiation with very long numbers
  - For a given $n$, there should be many private/public key pairs, otherwise an attacker might be able to perform a brute force attack

### 7.3: Key Generation and Proof of Correctness

- A distinctive feature of all asymmetric cryptosystems is that there is a set-up phase during which the public and private keys are computed
- Depending on the scheme, key generation can be quite complex, which is usually not an issue for block or stream ciphers
- **RSA Key Generation**
  - **Output:** public key: $k_{pub} = (n, e)$ and private key $k_{pr} = (d)$

1. Choose two large primes, $p$, and $q$
2. Compute $n = p \cdot q$
3. Compute $\Phi(n) = (p-1)(q-1)$
4. Select the public exponent $e \in [1, 2, ..., \Phi(n) - 1]$ such that

$$gcd(e, \Phi(n)) = 1$$

5. Compute the private key $d$ such that

$$d \cdot e \equiv mod\ \Phi(n)$$

---

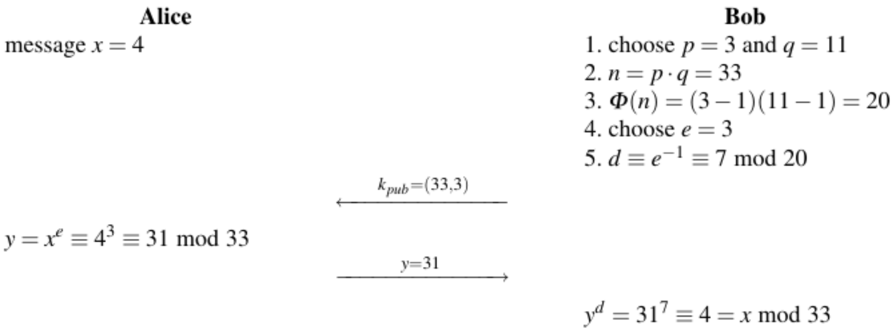
- The condition for public exponent selection will ensure that the inverse of $e$ exists in modulo $\Phi(n)$, such that there will always be a private key $d$
- The computation of $d$ and $e$ can be done at once using the extended Euclidean algorithm
    - In practice, one first selects a public parameter $e$ in the range $0 < e < \Phi(n)$, where the value of $e$ satisfies the relationship $gcd(e, \Phi(n)) = 1$
    - We can apply the extended Euclidean algorithm with the input parameters $n$ and $e$ and obtain the following relationship

$$gcd(\Phi(n), e) = s \cdot \Phi(n) + t \cdot e$$

    - Where we know that $e$ is a valid public key if $gcd(e, \Phi(n)) = 1$
    - We also know that the parameter $t$ calculated using the extended Euclidean algorithm is the inverse of $e$ and thus

$$d = t\ mod\ \Phi(n)$$

    - In the case that $e$ does not satisfy the above condition, we simply choose a new value for $e$ and repeat the process
- Here we can see an example



**Alice**

message $x = 4$

**Bob**

1. choose $p = 3$ and $q = 11$
2. $n = p \cdot q = 33$
3. $\Phi(n) = (3-1)(11-1) = 20$
4. choose $e = 3$
5. $d \equiv e^{-1} \equiv 7 \bmod 20$

$\xleftarrow{k_{pub}=(33,3)}$

$y = x^e \equiv 4^3 \equiv 31 \bmod 33$

$\xrightarrow{y=31}$

$y^d = 31^7 \equiv 4 = x \bmod 33$

- Expressed as an equation, we can consider the process as:

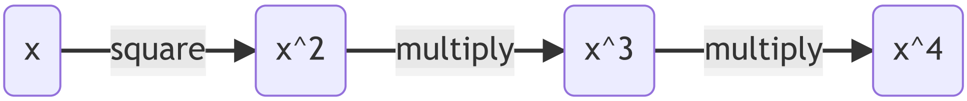
$$d_{k_{pr}}(y) = d_{k_{pr}}(e_{pub}(x)) \equiv (x^e)^d \equiv x^{de} \equiv x \; mod \; n$$

### 7.4: Encryption and Decryption: Fast Exponentiation

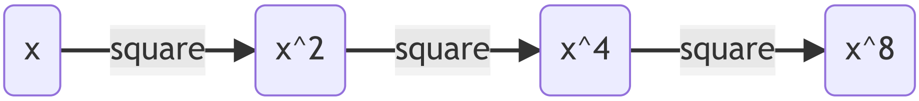
- Since asymmetric algorithms are based on arithmetic with very long numbers, we need a method by which to quickly exponentiate large integers such that asymmetric schemes do not use too much system overhead in encryption and decryption
- Recall the RSA encryption and decryption functions

$$y = e_{k_{pub}}(x) \equiv x^e \; mod \; n$$
$$x = d_{k_{pr}}(y) \equiv y^d \; mod \; n$$

- A straightforward exponentiation method looks as follows



- For example, to compute the value of $x^8$ would require one squaring and six multiplications
- Instead, we could do the following



- This requires only 3 operations as opposed to the seven from the naive approach
- However, for a specific given exponent, we do not know the exact order of the square and multiply operations, and we thus have an algorithm which provides a systematic way to check the order of these operations
- **Square-and-Multiply Algorithm for Modular Exponentiation**

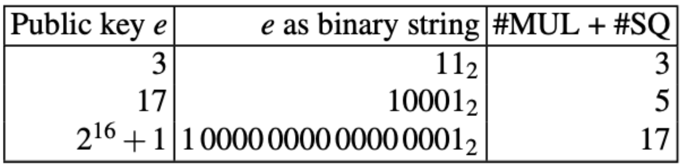
- *Input:*
    - Base element: $x$
    - Exponent: $H = \sum_{i=0}^{t} h_i 2^i$ with $h_i \in [0, 1]$ and $h_t = 1$ and modulus $n$
- *Output:* $x^H \ mod\ n$
- *Initialization:* $r = x$
- *Algorithm:*

```
for(i=t-1; i!=0; i++)
{
    r = (r*r) mod n

    if(h_i == 1)
        r = (r*x) mod n
}
return r
```

**7.5: Speed-up Techniques for RSA**

- **7.5.1: Fast Encryption with Short Public Exponents**
    - When concerned with the public exponent, $e$, which is used for encryption, we can choose a very small value for $e$
        - In practice, the values $3$, $17$, and $2^{16} + 1$ are of particular importance for RSA
        - Below we can see a table of the complexities of the three important public exponents



| Public key $e$ | $e$ as binary string | #MUL + #SQ |
|---:|---:|---:|
| 3 | $11_2$ | 3 |
| 17 | $10001_2$ | 5 |
| $2^{16}+1$ | $1\,0000\,0000\,0000\,0001_2$ | 17 |

    - RSA is still secure when using these particularly short public exponents
    - This scheme is useful for speeding up the encryption portion of RSA, but when the public key, $d$ is involved, i.e. for decryption or signature generation, it is not possible to speed RSA up with this technique
- **7.5.2: Fast Decryption with the Chinese Remainder Theorem**

- In RSA, a short private key cannot be chosen without compromising the security of encryption
- The goal here is to perform $x^d \ mod \ n$ efficiently
    - The party who possesses the private key also possesses the primes $p$ and $q$
    - The basic idea of the Chinese Remainder Theorem is that rather than doing arithmetic with one long modulus, $n$ we can perform two individual exponentiations modulus the shorter primes $p$ and $q$
    - We have to transform *into* the CRT domain, perform computations within the CRT domain, and inverse transform to find the result
- First we reduce x to obtain what is known as the *modular* representation of $x$

$$x_p \equiv x \ mod \ p \\ x_q \equiv x \ mod \ q$$

- Now, we will perform the following exponentiations

$$y_p = x_p^{d_p} \ mod \ p \\ y_q = x_q^{d_q} \ mod \ q$$

- Where the exponents $d_i$ are given by

$$d_p \equiv d \ mod \ (p-1) \\ d_q \equiv d \ mod \ (q-1)$$

- To perform the inverse transformation we must first compute two coefficients given by

$$c_p \equiv q^{-1} \ mod \ p \\ c_q \equiv p^{-1} \ mod \ q$$

- And then assemble the final result, $y$

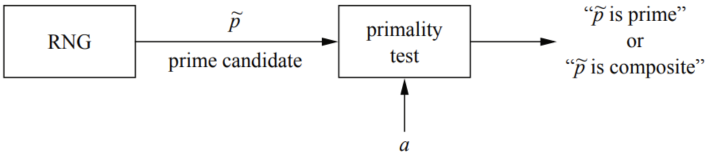
$$y = [q * c_p]y_p + [p * c_q]y_q \ mod \ n$$

- The total speed-up obtained by this implementation of the Chinese Remainder Theorem is a factor of 4, and there are hardly any drawbacks, so it is very frequently used in real-world cryptographic applications

### 7.6: Finding Large Primes

- One thing that hasn't been discussed is how the two primes $p$ and $q$ are generated

- Since their product is the RSA modulus, $n = p \cdot q$, each of the two primes should have about half the bit length of $n$



- The practicality of this approach is put into question by two factors
    - The likelihood of finding a prime number using an RNG
    - The speed at which we can check whether the randomly generated number is prime or composite
- **Fermat Primality Test**
    - One primality test which is commonly used is based on *Fermat's Little Theorem*
    - *Input:* prime candidate $\tilde{p}$ and security parameter $s$
    - *Output:* statement "$\tilde{p}$ is composite" or "$\tilde{p}$ is likely prime"
    - *Algorithm:*

    ```
    for(i=1; i<s; i++)
    {
        choose random a in {2, 3, 4, ..., p-2}
        if a^{p-1} != 1
            return p is composite
    }
    return p is likely prime
    ```

    - Since this algorithm is still capable of producing incorrect results via Carmichael numbers, or *Fermat liars*
- **Miller-Rabin Primality Test**
    - For a given odd integer, $n > 2$, we can write $n - 1$ as $2^s d$ where $s$ is a positive integer and $d$ is an odd positive integer
    - We will then consider an integer $a$ which we can call the *base*, which is co-prime to $n$
    - Then, $n$ is said to be a *strong probable prime* to base $a$ if one of these congruence relations holds

$$a^d \equiv 1 \ (mod \, n)$$
$$a^{2^r d} \equiv -1 \ (mod \, n)$$

where $0 \leq r < s$

### 7.7: RSA in Practice: Padding

- In practice, RSA has to be used with a *padding scheme* which is extremely important and can lead to insecurity if they are not implemented properly
- This is due to the problematic properties of RSA encryption listed below
    - RSA encryption is deterministic, i.e., for a specific key, a particular plaintext is always mapped to a particular ciphertext, creating vectors for a potential statistical attack
    - Plaintext values $x = 0, x = 1, x = -1$ produce ciphertexts equal to $0, 1, -1$
    - Small public exponents $e$ and small plaintexts $x$ might be subject to attacks if no padding, or weak padding, is used
- Another undesirable property of RSA is that it is *malleable*
    - This means that the attacker is capable of transforming the ciphertext into another ciphertext which leads to a known transformation of the plaintext
    - The attacker cannot decrypt the ciphertext here, but they are capable of manipulating the plaintext in a predictable manner
- One possible solution to all of the problems mentioned above is the use of padding
- Modern techniques, such as the *Optimal Asymmetric Encryption Padding (OAEP)* for padding RSA messages are specified and standardized in *Public Key Cryptography Standard #1 (PKCS#1)*
- Let $M$ be the message which will be padded, let $k$ be the length of modulus $n$ in bytes, let $|H|$ be the length of the hash function output in bytes, and let $|M|$ be the length of the message in bytes
- Let $L$ be an optional label which is left as an empty string if not used
- According to the most recent standard, padding a message within the RSA Encryption scheme is done in the following way:
    1. Generate a string, $PS$ of length $k - |M| - 2|H| - 2$ of zeroed byte

    - The length of $PS$ may be 0

2. Concatenate $hash(L), PS$, a single byte with ha hexadecimal value of $0x01$, and the message $M$, to form a data block, $DB$ of length $k-|H|-1$ bytes
    - $$DB = Hash(L)||PS||0x01||M$$

3- Generate a random byte string *seed* of length $|H|$

4- Let $dbMask = MGF(seed, k-|H|-1)$, where MGF is the mask generation function which, in practice, is often a hash function such as SHA-1

5- Let $maskedDB = DB \oplus dbMask$

6- Let $seedMask = MGF(maskeedDB, |H|)$

7- Let $maskedSeed = seed \oplus seedMask$

8- Concatenate a single byte with hexadecimal value $0x00$, $maskedSeed$, and $maskedDB$ to form an encoded message, $EM$, of length $k$ bytes

- $$EM = 0x00||maskedSeed||maskedDB$$

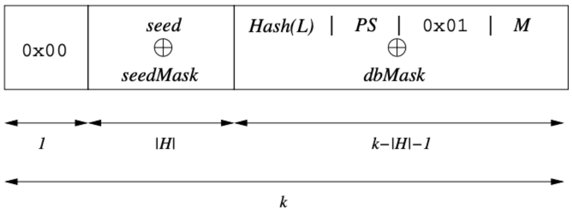


**Fig. 7.3** RSA encryption of a message *M* with Optimal Asymmetric Encryption Padding (OAEP)

**Justin Ciocoi Nov. 16, 2023**

# CSCI 360 Textbook Notes

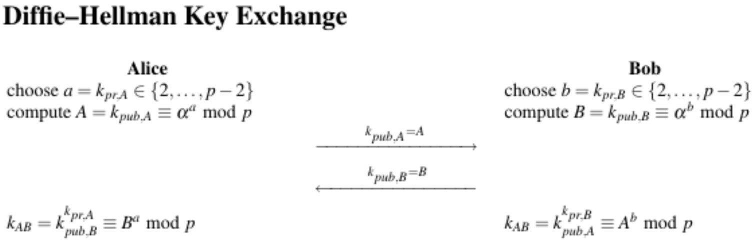
## Chapter 8: Public-Key Cryptosystems Based on the Discrete Logarithm Problem

#### 8.1: Diffie-Hellman Key Exchange

- The *Diffie-Hellman Key Exchange* was the first asymmetric cryptographic scheme that was published in open literature
- It provides a solution to the key distribution problem by allowing two parties to derive a common secret key over an insecure channel
- The basic idea behind DHKE is that exponentiation in $\mathbb{Z}_p^*$, where $p$ is prime is a one way function and that exponentiation is commutative:

$$k = (\alpha^x)^y \equiv (\alpha^y)^x \; mod \, p$$

- The value $k$ is the joint secret which can be used as the session key between the two parties
- We will now consider how the DHKE protocol works over $\mathbb{Z}_p^*$ where two people, *Person A*, and *Person B* would like to establish a shared secret key
- There are two steps to the DHKE protocol
  - **Diffie-Hellman Set-Up**
    1. Choose a large prime, $p$
    2. Choose an integer, $\alpha \in \{2, 3, 4, ....., p-2\}$
    3. Publish $p$ and $\alpha$
  - **Diffie-Hellman Key Exchange**



**Diffie–Hellman Key Exchange**

**Alice**
choose $a = k_{pr,A} \in \{2, \ldots, p-2\}$
compute $A = k_{pub,A} \equiv \alpha^a \bmod p$

**Bob**
choose $b = k_{pr,B} \in \{2, \ldots, p-2\}$
compute $B = k_{pub,B} \equiv \alpha^b \bmod p$

$\xrightarrow{k_{pub,A}=A}$

$\xleftarrow{k_{pub,B}=B}$

$k_{AB} = k_{pub,B}^{k_{pr,A}} \equiv B^a \bmod p$

$k_{AB} = k_{pub,A}^{k_{pr,B}} \equiv A^b \bmod p$

        - Here, *Alice* computes $B^a \equiv (\alpha^b)^a \equiv \alpha^{ab} \; mod \; p$
        - *Bob* computes $A^b \equiv (\alpha^a)^b \equiv \alpha^{ab} \; mod \; p$
        - And thus they share the key $k_{AB} \equiv \alpha^{ab} \; mod \; p$

    **8.2: Some Algebra**

- **8.2.2: Cyclic Groups**
    - In cryptography, we are almost always concerned with finite structures
    - **Definition 8.2.2**: *Finite Group*

      A group, $(G, \circ)$ is finite if it has a finite number of elements. we denote the *cardinality* or *order* of the group $G$ using $|G|$

    - **Definition 8.2.3**: *Order of an element*

      The *order*, $ord(a)$ of an element, $a$, in group $(G, \circ)$ is the smallest positive integer, $k$ such that:

      $$a^k = a \circ a \circ a \circ ... \circ a \; (k \; times) = 1$$

      where $1$ is the identity element of $G$

    - **Definition 8.2.4**: *Cyclic Group*

      A group, $G$, which contains an element $\alpha$, with a maximum order, $ord(a) = |G|$, is said to be cyclic

      Elements with maximum order are called *primitive elements* or *generators*

    - An element $\alpha$ of a group $G$ is called a generator when every element, $a$, in the set can be written as a power of $\alpha^i = a$ where $i$ is also in the group
    - There are several interesting properties of cyclic groups, and the most important ones for cryptographic applications are as follow:

        - **Theorem 8.2.2:** For every prime $p$, the group ($\mathbb{Z}_p^*, \cdot$ ) is an abelian finite cyclic group

          This essentially means that every prime field is cyclic, which is a fact with far reaching consequences in cryptography

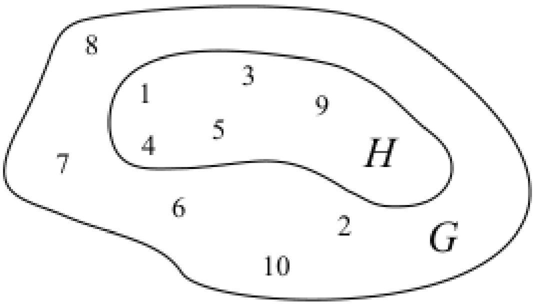
        - **Theorem 8.2.3:** Let $G$ be a finite group. Then for every $a \in G$, it holds that:
            - $a^{|G|} = 1$
            - $|G| \; mod \; order(a) = 0$
        - Theorem **8.2.4:** Let $G$ be a finite cyclic group; then it holds that:
            - The number of primitive elements of $G$ is $\Phi(|G|)$
            - If $|G|$ is prime, then all elements $a \neq 1 \in G$ are primitive

- **8.2.3: Subgroups**
    - Now we discuss subsets of groups which are groups themselves, also referred to as *subgroups*
    - In the case of cyclic groups, there is an easy way to generate subgroups, which comes from the following theorem
    - **Theorem 8.2.5:** *Cyclic Subgroup Theorem*

      Let $(G, \circ)$ be a cyclic group

      Then, every element with $a \in G$ with $order(a) = s$ is the primitive element of a cyclic subgroup with $s$ elements

    - This essentially tells us that any element of a cyclic group is the generator of a subgroup which is, in turn, also cyclic
    - Below we can see an illustration of group $G$, enclosing its subgroup, $H$



- **Theorem 8.2.6:** *Lagrange's Theorem*

  Let $H$ be a subgroup of $G$. Then, $|H|$ evenly divides $|G|$

- **Theorem 8.2.7:**

  Let $G$ be a finite cyclic group of order $n$ and let $\alpha$ be a generator of $G$

  Then, for every integer $k$ that divides $n$ evenly, there exists exactly one cyclic subgroup $H$ of $G$ of order $k$

  This subgroup is generates by $\alpha^{n/k}$

  $H$ consists exactly of the elements $a \in G$ which satisfy the condition $a^k = 1$

  There are no other subgroups

  **8.3: The Discrete Logarithm Problem**

- **8.3.1: The Discrete Logarithm Problem in Prime Fields**
  - **Definition 8.3.1:** *Discrete Logarithm Problem (DLP) in $\mathbb{Z}_p^\star$*

    Given the finite cyclic group $\mathbb{Z}_p^*$ of order $p-1$, a primitive element $\alpha \in \mathbb{Z}_p^*$, and another element $\beta \in \mathbb{Z}_p^*$

    The DLP is the problem of determining the integer $1 \leq x \leq p-1$ such that

$$\alpha^x \equiv \beta \; mod \; p$$

  - We can also formally write that

$$x = log_\alpha \beta \; mod \; p$$

- **8.3.2: The Generalized Discrete Logarithm Problem**
  - The feature which makes the DLP particularly useful in cryptography is that it is not restricted to the multiplicative group $\mathbb{Z}_p^*$, where $p$ is prime, but can instead be defined over any cyclic groups
  - This is called the *generalized discrete logarithm problem* and can be stated as follows
  - **Definition 8.3.2:** *Generalized Discrete Logarithm Problem*

    Given a finite cyclic group $G$ with the group operation $\circ$ and cardinality $n$

    We consider a primitive element $\alpha \in G$ and another element $\beta \in G$

The discrete logarithm problem is finding the integer $x$, where $1 \leq x \leq n$ such that:

$$\beta = \alpha \circ \alpha \circ \alpha \circ ... \circ \alpha(x\ times) = \alpha^x$$

- **8.3.3: Attacks Against the Discrete Logarithm Problem**
    - *Brute-Force Search*
        - In a brute-force attack, we can simply compute powers of the generator $\alpha$ successively until the result equals $\beta$ i.e.

$$\begin{aligned} \alpha^1 &= \beta \\ \alpha^2 &= \beta \\ \alpha^3 &= \beta \\ &\vdots \\ \alpha^x &= \beta \end{aligned}$$

    - *Shanks' Baby-Step Giant-Step Method*
        - Shanks' algorithm is a time-memory tradeoff method, which reduces the overall time it will take to run a brute-force algorithm at the cost of extra storage
        - The idea is based on rewriting the discrete logarithm $x = log_\alpha \beta$ in a two-digit representation as follows:

$$x = x_g m + x_b \text{ for } 0 \leq x_g,\ x_b < m$$

        - The value $m$, here is chosen to be the square root of the group order, i.e. $m = \sqrt{|G|}$
        - Now, the discrete logarithm can be written as $\beta = \alpha^x = \alpha^{x_g m + x_b}$ which leads to:

$$\beta \cdot (\alpha^{-m})^{x_g} = \alpha^{x_b}$$

        - The core idea here is that this equation can be solved by searching for $x_g$ and $x_b$ separately, i.e. using a divide-and-conquer approach
        - In the first phase, or *baby-step phase* we compute and store all values $\alpha^{x_b}$ where $0 \leq c_b < m$
            - This phase requires $m \approx \sqrt{|G|}$ steps, or group operations, and needs to store $m \approx \sqrt{|G|}$ group elements
        - In the *giant-step phase*, the algorithm checks for all $x_g$ in the range $0 \leq x_g < m$ whether the following condition is fulfilled

  $$\beta \cdot (\alpha^{-m})^{x_g} = \alpha^{x_b}$$

  for some stored entry $\alpha^{x_b}$ which was computed in the prior phase

  - If a match for some pair $(x_{g,0}, x_{b,0})$ exists, then the discrete logarithm is given by

    $$x = x_{g,0}m + x_{b,0}$$

  - This method requires $O(\sqrt{|G|})$ steps and an equal amount of memory
  - In a group of order $2^{80}$, an attacker would only need approximately $2^{40} = \sqrt{2^{80}}$ computations and memory locations, which is easily achievable with todays computer systems and storage mediums
  - Thus, in order to obtain an attack complexity of $2^{80}$, a group must have a cardinality of at least $2^{160}$
    - In the case of groups $G = \mathbb{Z}_p^*$ the prime $p$ should have a length of at least 160 bit

- *Pollard's Rho Method*
  - In Pollard's rho method, the runtime is approximately the same as Shanks' algorithm, but it has only negligible space requirements
  - The basic idea here is to pseudo-randomly generate group elements in the form $\alpha^i \cdot \beta^i$, where for every element we keep track of the elements $i$ and $j$ and continue until we obtain a collision of two elements such that

    $$\alpha^{i_1} \cdot \beta^{i_1} = \alpha^{i_2} \cdot \beta^{i_2}$$

  - If we substitute $\beta = \alpha^x$ and compare the exponents on both sides of the equation, the collision will lead us to the relation

    $$i_1 + xj_1 \equiv i_2 + xj_2 \bmod |G|$$

  - From here, we can compute the discrete logarithm easily using

    $$x \equiv \frac{i_2 - i_1}{j_1 - j_2} \bmod |G|$$

  - Pollard's rho method is of great practical importance because it is currently the best known algorithm for computing discrete logarithms in elliptic curve groups

- *Non-generic Algorithms: The Index-Calculus Method*

        - The algorithms defined thus far work for discrete logarithms defined over any cyclic group

        - The index-calculus method does not work for any cyclic group, but it is a very efficient algorithm for computing discrete logarithms in the cyclic groups $\mathbb{Z}_p^*$ and $GF(2^m)^*$

        - The index-calculus method relies on the property that a significant factor of elements in $G$ can be efficiently expressed as products of elements of a small subset of $G$

        - For the group $\mathbb{Z}_p^*$, this means that many elements should be expressible as a product of small primes

        - In order to provide 80 bits of security for DLP based cryptography in the group $\mathbb{Z}_p^*$, the prime $p$ must be at 1024 bits long

**8.4: Security of the Diffie-Hellman Key Exchange**

- Let us consider an attacker on the DHKE protocol who can listen to, but not alter, messages

- His goal here, would be to compute the session key $k_{AB}$ shared by two users

- The attacker clearly knows the public parameters $\alpha$ and $p$

- Additionally, by eavesdropping on the channel during the key exchange, the attacker can obtain the values $A = k_{pub,A}$ and $B = k_{pub,B}$

- So, the question is whether the attacker can compute $k = \alpha^{ab}$ from $\alpha, p, A \equiv \alpha^a \bmod p \text{ and } B \equiv \alpha^b \bmod p$

- This problem is called the *Diffie-Hellman Problem*, and can be generalized to the following

- **Definition 8.4.1:** *Generalized Diffie-Hellman Problem*

    - Given a finite cyclic group $G$ of order $n$, a primitive element $\alpha \in G$ and two elements $A = \alpha^a$ and $B = \alpha^b$ in $G$, the Diffie-Hellman problem is to find the group element $\alpha^{ab}$

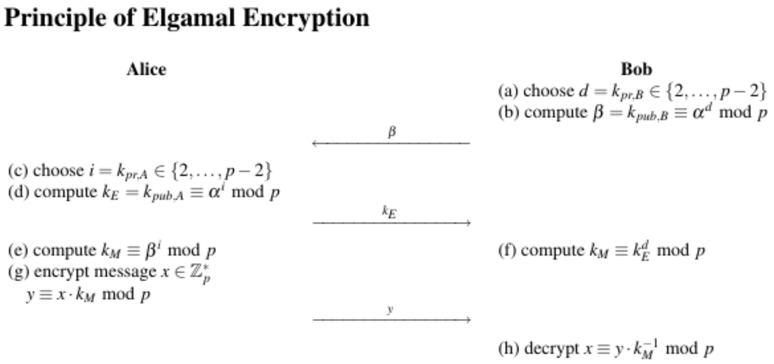
- We suppose the attacker knows an efficient method for computing discrete logarithms in $\mathbb{Z}_p^*$, and can thus solve the Diffie-Hellman problem and obtain the key $k_{AB}$ using the following steps

1. Compute user A's private key $a = k_{pr,A}$ by solving the discrete logarithm problem $a \equiv log_{\alpha} A \bmod p$
2. Compute the session key $k_{AB} \equiv B^a \bmod p$

- From the earlier section on DLP attacks, we know that solving the discrete logarithm problem here is infeasible if $p$ is sufficiently large

**8.5: Elgamal Encryption Scheme**

- The *Elgamal* encryption scheme, proposed by Taher Elgamal in 1985, often referred to as *Elgamal encryption*, can be viewed as an extension of the DHKE protocol
- Its security is also based on the intracability of the discrete logarithm problem and the Diffie-Hellman problem
- In this section, we will consider the Elgamal encryption scheme over the group $\mathbb{Z}_p^*$
- **8.5.1: From Diffie-Hellman Key Exchange to Elgamal Encryption**
    - Let us consider two parties, person A and person B
    - Person A wants to send an encrypted message, $x$, to person B
    - First, both parties perform the Diffie-Hellman key exchange to derived a shared key $k_M$
        - We can assume the public parameters $p$ and $\alpha$ have been generated
    - Now, person A will used this shared key as a multiplicative mask with which to encrypt $x$ as $y \equiv x \cdot k_M \bmod p$
    - The process is as follows:

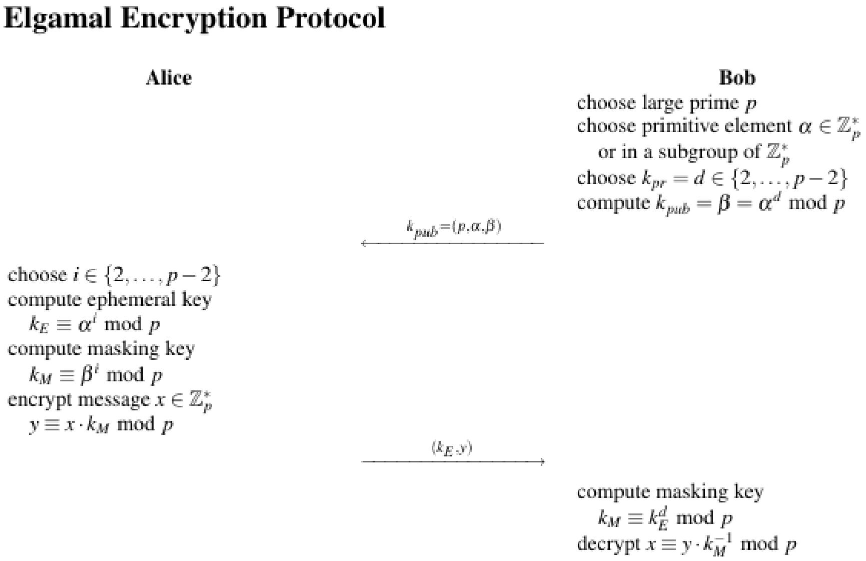


**Principle of Elgamal Encryption**

| Alice | | Bob |
|---|---|---|
| | | (a) choose $d = k_{pr,B} \in \{2,\ldots,p-2\}$<br>(b) compute $\beta = k_{pub,B} \equiv \alpha^d \bmod p$ |
| | $\xleftarrow{\beta}$ | |
| (c) choose $i = k_{pr,A} \in \{2,\ldots,p-2\}$<br>(d) compute $k_E = k_{pub,A} \equiv \alpha^i \bmod p$ | | |
| | $\xrightarrow{k_E}$ | |
| (e) compute $k_M \equiv \beta^i \bmod p$<br>(g) encrypt message $x \in \mathbb{Z}_p^*$<br>$y \equiv x \cdot k_M \bmod p$ | | (f) compute $k_M \equiv k_E^d \bmod p$ |
| | $\xrightarrow{y}$ | |
| | | (h) decrypt $x \equiv y \cdot k_M^{-1} \bmod p$ |

- Here, if the key $k_M$ is randomly drawn from $\mathbb{Z}_P^*$, every cipher text $y \in \{1, 2, 3, ..., p-1\}$ is equally likely

- **8.5.2: The Elgamal Protocol**



**Elgamal Encryption Protocol**

**Alice** | **Bob**

Bob:
choose large prime $p$
choose primitive element $\alpha \in \mathbb{Z}_p^*$
or in a subgroup of $\mathbb{Z}_p^*$
choose $k_{pr} = d \in \{2, \ldots, p-2\}$
compute $k_{pub} = \beta = \alpha^d \bmod p$

$\xleftarrow{k_{pub}=(p,\alpha,\beta)}$

Alice:
choose $i \in \{2, \ldots, p-2\}$
compute ephemeral key
$k_E \equiv \alpha^i \bmod p$
compute masking key
$k_M \equiv \beta^i \bmod p$
encrypt message $x \in \mathbb{Z}_p^*$
$y \equiv x \cdot k_M \bmod p$

$\xrightarrow{(k_E,y)}$

Bob:
compute masking key
$k_M \equiv k_E^d \bmod p$
decrypt $x \equiv y \cdot k_M^{-1} \bmod p$

  - This protocol serves to rearrange the steps from the Diffie-Hellman protocol to where person A needs to send only one message to person B as opposed to two messages

- **8.5.3: Computational Aspects of The Elgamal Protocol**
  - *Key Generation*
    - During the key generation phase, the receiver will generate prime $p$ and then compute both the public and private key
    - Since the Elgamal protocol's security relies on the discrete logarithm problem, $p$ needs to have length of at least 1024 bits
  - *Encryption*
    - The square-and-multiply algorithm will be used to speed up encryption using exponentiation of large numbers

**Justin Ciocoi**

**Nov. 27, 2023**

# CSCI 360 Textbook Notes

## Chapter 9: Elliptic Curve Cryptosystems

### 9.1: How to Compute with Elliptic Curves

- Elliptic curve cryptosystems, like other public-key cryptosystems, is based on the generalized discrete logarithm problem
- Thus, we must first find a cyclic group on which we can build our cryptosystem
- The mere existence of such a cyclic group is not enough though, as the group must be computationally hard to prevent against brute-force attacks
- **9.1.1: Definition of Elliptic Curves**
    - The *elliptic curve* over $\mathbb{Z}_p,\ p > 3$, is the set of all pairs $(x, y) \in \mathbb{Z}_p$ which fulfill

      $$y^2 \equiv x^3 + a \cdot x + b \bmod p$$

      together with an imaginary point of infinity, $\infty$, where

      $$a, b \in \mathbb{Z}_p$$

      and the condition $4 \cdot a^3 + 27 \cdot b^2 \neq 0 \bmod p$

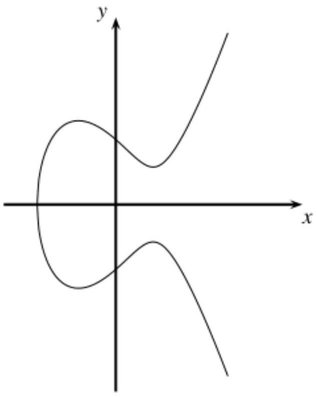
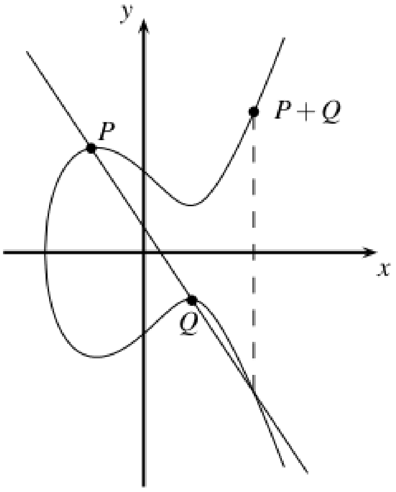
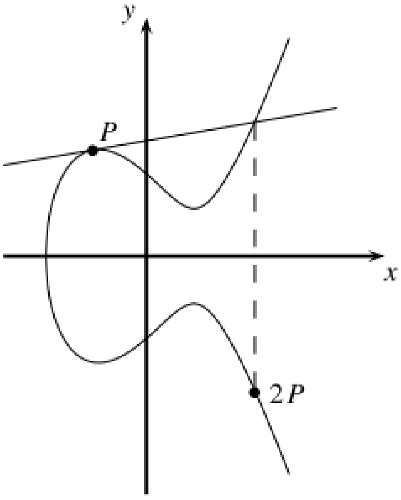


**Fig. 9.3** $y^2 = x^3 - 3x + 3$ over $\mathbb{R}$

- **9.2.2: Group Operations on Elliptic Curves**

- Point addition, $P+Q$ and point doubling, $P+P$ can be achieved using the following methods respectively





- In addition, we can find the inverse of a point on an elliptic curve by finding its reflection over the x-axis

### 9.2: Building a Discrete Logarithm Problem with Elliptic Curves

- **Theorem 9.2.1**

  The points on an elliptic curve together with $\infty$ have cyclic subgroups, and under certain conditions all points on an elliptic curve form a cyclic group

### 9.3: Diffie-Hellman Key Exchange with Elliptic Curves

- **Elliptic Curve Diffie-Hellman Domain Parameters**

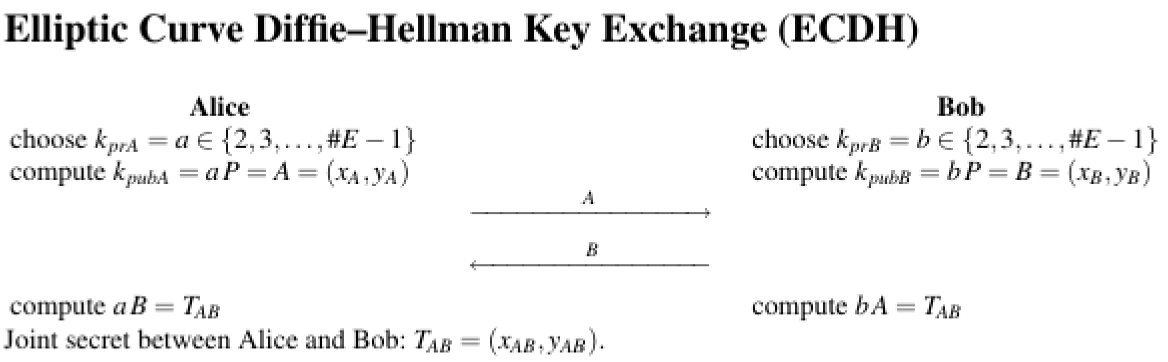
  1. Choose a prime $p$ and the elliptic curve

$$E : y^2 \equiv x^3 + a \cdot x + b \bmod p$$

2. Choose a primitive element $P = (x_p, y_p)$

The prime $p$, the curve given by its coefficients $a, b$, and the primitive element $P$ are the domain parameters

- Key exchange here is done in essentially the same way as conventional Diffie-Hellman key exchange



**Elliptic Curve Diffie–Hellman Key Exchange (ECDH)**

| Alice | | Bob |
|---|---|---|
| choose $k_{prA} = a \in \{2,3,\ldots,\#E-1\}$<br>compute $k_{pubA} = aP = A = (x_A, y_A)$ | | choose $k_{prB} = b \in \{2,3,\ldots,\#E-1\}$<br>compute $k_{pubB} = bP = B = (x_B, y_B)$ |
| | $\xrightarrow{A}$ | |
| | $\xleftarrow{B}$ | |
| compute $aB = T_{AB}$ | | compute $bA = T_{AB}$ |

Joint secret between Alice and Bob: $T_{AB} = (x_{AB}, y_{AB})$.

### 9.4: Security

- The reason why elliptic curves are used in modern cryptography is the fact that they have very good one way properties
- As opposed to the simpler discrete logarithm problems based in $\mathbb{Z}_p^*$, discrete logarithm problems in elliptic curve groups are not vulnerable to index calculus attacks
- Thus, the best remaining algorithms when attacking an elliptic curve discrete logarithm problem are Shanks' baby-step giant-step methos and Pollard's rho method
- With these attacks, the number of computations needed is the square root of the cardinality of the group
  - Therefore, a prime $p$ should be chosen to be 256 bits in order to provide 128 bits of security, since $\sqrt{2^{256}} = 2^{128}$

### 9.5: Implementation in Software and Hardware

- In practice, a core requirement for using ECC in cryptography is that the cyclic group formed by the curve points has prime order
- When implementing elliptic curve cryptography, it is useful to view an ECC scheme as a structure with four layers

- On the bottom layer, modular arithmetic is performed
- On the next layer, the two group operations, point addition and point doubling, are realized
- On the third layer, scalar multiplication is realized, which uses the group operations of the previous layer
- The top layer implements the protocol, such as ECDH (Elliptic Curve Diffie-Hellman) or ECDSA (Elliptic Curve Digital Signature Algorithm)

**Justin Ciocoi**

**Dec. 3, 2023**

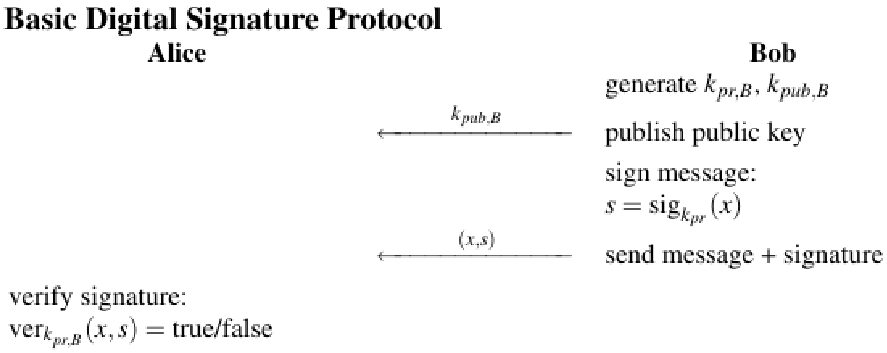
# CSCI 360 Textbook Notes

## Chapter 10: Digital Signatures

**10.1: Introduction**

- This section will describe why digital signatures are needed, why they must be based on asymmetric cryptography, and various fundamental principles of digital signature schemes
- **10.1.1: Why Symmetric Cryptography is not Sufficient**
  - Of the crypto schemes that have thus far been covered, all have had one of two goals:
    - To encrypt some data
    - To establish a shared key between two parties
  - While this provides some services, there are other security needs which we will deem security services
  - Under modern symmetric cryptography schemes, we have a level of security protecting communicating parties from outside attacks
  - However, symmetric schemes do not provide either party with any protection against *each other*
  - In order to prove to a neutral third party that one of the two parties sent a specific message, symmetric cryptography is not sufficient, since both parties share the key $k_{ab}$ and can thus both generate messages encrypted in the same manner with the same key
- **10.1.2: Principles of Digital Signatures**
  - The process of digital signatures starts with one party signing a message, $x$
  - The signature algorithm is a function of this party's private key, $k_{pr}$, so assuming this key is kept private, this party is the only one who can sign message $x$ on their behalf

  - $x$ and $k_{pr}$ are inputs into the signature function, and after signature, the signature $s$ is appended to the message and the pair $(x, s)$ is sent
  - The signature is only of use to the receiving party if they have a method by which they can *verify* the signature's validity
  - Thus, the receiving party has a verification function which will accept $x, s,$ and, $k_{pub}$ as inputs
    - If $x$ was actually signed with the private key which belongs to the public verification key, then the function returns `true`
    - Otherwise, it returns `false`
  - Here, we can see a basic visualization of the protocols outlined above:


Basic Digital Signature Protocol
Alice
Bob
generate $k_{pr,B}, k_{pub,B}$
$k_{pub,B}$
publish public key
sign message:
$s = \mathrm{sig}_{k_{pr}}(x)$
$(x,s)$
send message + signature
verify signature:
$\mathrm{ver}_{k_{pr,B}}(x,s) = \text{true/false}$

  - A signed message can unambiguously be traced back to its originator since a valid signature can only be computed with the unique signer's private key
  - Each of the three popular public-key algorithm families (integer factorization, discrete logarithms, and elliptic curves) allows us to construct digital signature schemes
- **10.1.3: Security Services**
  - Let us first discuss the various security services that may be provided by cryptographic schemes in general
    - **Confidentiality:** Information is kept secret from all but authorized parties
    - **Integrity:** Messages have not been modified in transit
    - **Message Authentication:** The sender of a message is authentic. An alternative term is *data origin authentication*

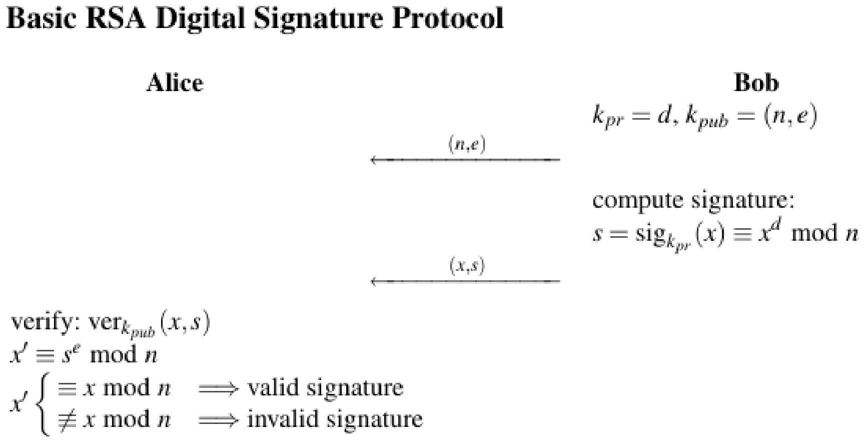
        - **Nonrepudiation:** The sender of a message can not deny the creation of the message
        - **Identification/Entity Authentication:** Establish and verify the identity of an entity, e.g., a person, a computer or a credit card
        - **Access Control:** Restrict access to the resources to privileged entities
        - **Availability:** Assures that the electronic system is reliably available
        - **Auditing:** Provide evidence about security-relevant activities, e.g., by keeping logs about certain events
        - **Physical Security:** Provide protection against physical tampering and/or responses to physical tampering attempts
        - **Anonymity:** Provide protection against discovery and misuse of identity
    - Which services are desired in a given system is heavily application specific and can vary greatly in different environments or implementations

### 10.2: The RSA Signature Scheme

- **10.2.1: Schoolbook RSA Digital Signature**
    - First, let us imagine a sending party wants to send a signed message and after set up has the following parameters:

$$\text{Private Key: } k_{pr} = (d)$$
$$\text{Public Key: } k_{pub} = (n, e)$$

    - The basic protocol will be achieved as shown in the following diagram


Basic RSA Digital Signature Protocol
Alice
Bob
$k_{pr} = d,\ k_{pub} = (n,e)$
$(n,e)$
compute signature:
$s = \mathrm{sig}_{k_{pr}}(x) \equiv x^d \bmod n$
$(x,s)$
verify: $\mathrm{ver}_{k_{pub}}(x,s)$
$x' \equiv s^e \bmod n$
$x' \begin{cases} \equiv x \bmod n & \Longrightarrow \text{valid signature} \\ \not\equiv x \bmod n & \Longrightarrow \text{invalid signature} \end{cases}$

  - As we can see, the sender computes their signature, $s$ by RSA-encrypting $x$ with his private key $k_{pr}$
  - The receiver then calculates $x'$, and if $x = x'$ , confirms the validity of the signature
  - Let us look at the verification operations:

    $$s^e = (x^d)^e = x^{de} \equiv x \bmod n$$

    which we can confirm the validity of due to the mathematical relationship between the public and private key, namely:

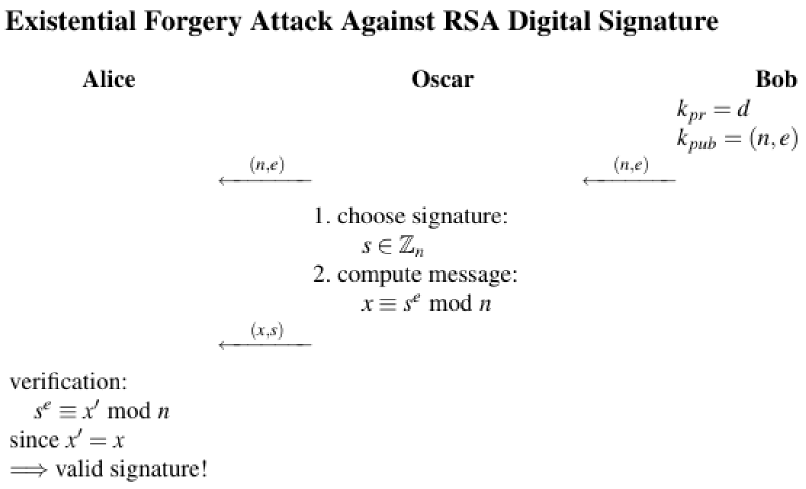
    $$d \cdot e \equiv 1 \bmod \phi(n)$$

    which means that raising any integer, $x$ to the $(de)^{th}$ power yields itself again, showing that

    $$x^{de} \equiv x \bmod n$$

- **10.2.2: Computational Aspects**
  - The signature in RSA digital signature schemes is as long as the modulus $n$, or roughly $log_2(n)$ bits
  - Using RSA with short public keys is particularly useful, since while a message may only be signed once, verification could happen many times, and thus the short public key can greatly speed up the verification function
- **10.2.3: Security**
  - *Algorithmic Attacks*
    - This group of attacks attempts to break RSA by computing the private key, $d$
    - The most general of these attacks is an attacker factoring the modulus $n$ into primes $p$ and $q$, after which the private key $d$ can be computed using the public key $e$
    - Thus, the modulus $n$ must be sufficiently large to prevent factoring attacks
  - *Existential Forgery*
    - This attack allows an attacker to generate a valid signature for a *random* message $x$

    - Here we can see a basic visualization of these types of attacks



Existential Forgery Attack Against RSA Digital Signature

    - In this scheme, the attacker chooses the signature and then computes the message, so while the semantics of the message cannot be controlled, this is still a clearly undesirable quality o have in a secure cryptographic scheme

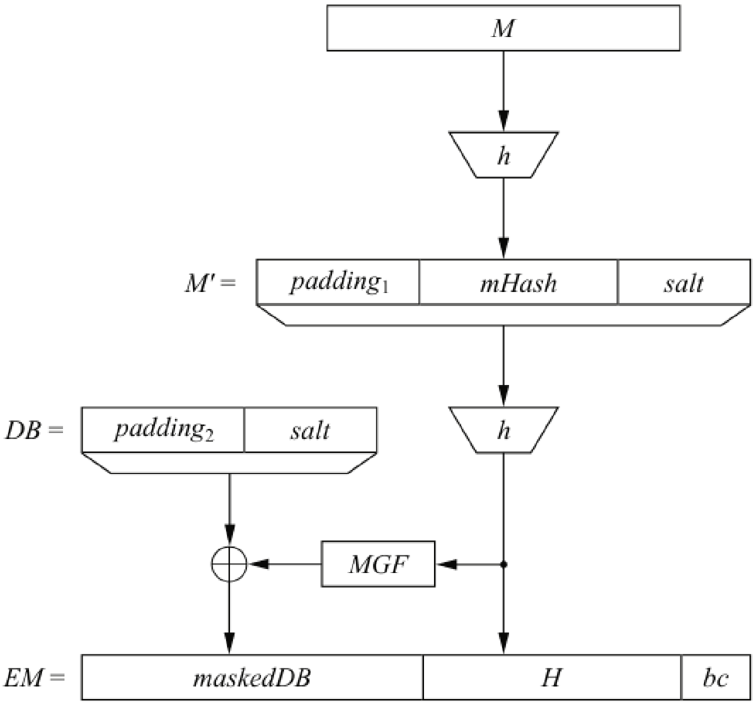
- **RSA Padding: The Probabilistic Signature Standard**
  - In this scheme, the message itself is not signed, but rather the hashed version of the message
  - *Encoding for the EMSA Probabilistic Signature Scheme*

    Let $|n|$ be the size of the RSA modulus in bits. The encoded message $EM$ has a length $[(|n|-1)/8]$ bytes such that the bit length of $EM$ is at most $|n|-1$ bits

    1. Generate a random value, *salt*
    2. Form a string, $M'$ by concatenating a fixed padding, $pad_1$, the hash value, $mHash = h(M)$, and *salt*
    3. Compute a hash value $H$ of the string $M'$
    4. Concatenate a fixed padding, $pad_2$ and *salt* to form a data block, $DB$
    5. Apply a mask generation function $MGF$ to the string $M'$ to compute the mask value $dbMask$
        - In practice, a hash function such as SHA-1 is often used
    6. XOR the mask value $dbMask$ and the data block $DB$ to compute $maskedDB$

7. The encoded message $EM$ is obtained by concatenating $maskedDB$, the hash value $H$, and the fixed padding $bc$

- After the encoding, the actual signature function is carried out with the encoded message and private key as inputs



### 10.3: The Elgamal Digital Signature Schemes

- **10.3.1: Schoolbook Elgamal Digital Signature**
  - Normal Elgamal key generation protocols will be followed such that:

$$k_{pub} = (p, \alpha, \beta)$$
$$k_{pr} = d$$

  - *Signature Generation*

1. Choose a random ephemeral key $k_E \in \{0, 1, 2, ..., p-1\}$ such that $gcd(k_E, p-1) = 1$

2. Compute the signature parameters:

$$r \equiv \alpha^{k_E} \mod p$$
$$s \equiv (x - d \cdot r)k_E^{-1} \mod p - 1$$

  - *Signature Verification*

1. Compute the value

$$t \equiv \beta^r \cdot r^s \bmod p$$

2. The verification compares $t$ to $\alpha^x \bmod p$, and if they match validates the signature

### 10.4: The Digital Signature Algorithm (DSA)

- **10.4.1: The DSA Algorithm**
    - Here we will introduce DSA using the 1024 bit standard, but other longer bit lengths are also possible in the standard
    - *Key Generation*
        1. Generate a prime $p$ with $2^{1023} < p < 2^{1024}$
        2. Find a prime divisor $q$ of $p-1$ with $2^{159} < q < 2^{260}$
        3. Find an element $\alpha$ with $ord(\alpha) = q$, i.e. $\alpha$ generates the subgroup with $q$ elements
        4. Choose a random integer $d$ between $0$ and $q$
        5. Compute $\beta \equiv \alpha \bmod p$

      The keys are now:

$$\begin{aligned} k_{pub} &= (p, q, \alpha, \beta) \\ k_{pr} &= (d) \end{aligned}$$

    - The central idea here is that there are two cyclic groups involved, namely the large cyclic group $\mathbb{Z}_p^*$ which has an order of bit length 1024, and its smaller subgroup which has an order of bit length 160
    - *Signature Generation*
        1. Choose an integer between $0$ and $q$ as the random ephemeral key, $k_E$
        2. Compute $r \equiv (\alpha^{k_E} \bmod p) \bmod q$
        3. Compute $s \equiv (SHA(x) + d \cdot r)k_E^{-1} \bmod q$
    - *Signature Verification*

1. Compute auxiliary value $w \equiv s^{-1} \bmod q$
2. Compute auxiliary value $u_1 \equiv w \cdot SHA(x) \bmod q$
3. Compute auxiliary value $u_2 \equiv w \cdot r \bmod q$
4. Compute $v \equiv (\alpha^{u_1} \cdot \beta^{u_2} \bmod p) \bmod q$
5. If $v$ is equivalent to $r \bmod q$, then the signature is validated, otherwise it is not

**10.5: The Elliptic Curve Digital Signature Algorithm (ECDSA)**

- **10.5.1: The ECDSA Algorithm**
  - *Key Generation*
    1. Use an elliptic curve $E$ with
       - modulus $p$
       - coefficients $a$ and $b$
       - a point $A$ which generates a cyclic group of prime order $q$
    2. Choose a random integer $d$ between $0$ and $q$
    3. Compute $B = dA$

    The keys are now:

    $$k_{pub} = (p, a, b, q, A, B) \\ k_{pr} = (d)$$
  - *Signature Generation*
    1. Choose an integer between $0$ and $q$ as the random ephemeral key, $k_E$
    2. Compute $R = k_E A$
    3. Let $r = x_R$
    4. Compute $s \equiv (h(x) + d \cdot r)k_E^{-1} \bmod q$
  - *Signature Verification*
    1. Compute auxiliary value $w \equiv s^{-1} \bmod q$

2. Compute auxiliary value $u_1 \equiv w \cdot h(x) \bmod q$

3. Compute auxiliary value $u_2 \equiv w \cdot r \bmod q$

4. Compute $P = u_1 A + u_2 B$

5. If $x_P$ (the x-coordinate of point $P$) is equivalent to $r \bmod q$ then the signature is validated