Introduction to Software Security

Public Key Crypto
(Chapter 4)

Seong-je Cho

Spring 2018

Computer Security & Operating Systems Lab, DKU
9 Algorithms that Changed the Future

2. PageRank: The Technology That Launched Google
3. Public-key cryptography: Sending Secrets on a Postcard
4. Forward error correction: Mistakes That Fix Themselves
5. Pattern recognition: Learning from Experience
6. Data compression: Something for Nothing
7. Database: The Quest for Consistency
8. Digital signature: Who Really Wrote This Software?
9. What Is Computable? (Computability and Undecidability) --- Opening Quotation

Sources / References

- Textbook -- Chapter 4
- Public-Key Encryption by RSA Algorithm
- How RSA Works with Examples
- A toy RSA implementation by Alex Roper:
  - [https://github.com/calmofthestorm/toys/blob/master/rsa/rsa.py](https://github.com/calmofthestorm/toys/blob/master/rsa/rsa.py)
  - [https://www.youtube.com/watch?v=wXB-V_Keiu8](https://www.youtube.com/watch?v=wXB-V_Keiu8)
- N. Vlajic, CSE 3482: Introduction to Computer Security, Yorku
- Nicholas Weaver, Computer Science 161: Computer Security, Berkeley
- Myrto Arapinis, Computer Security: INFRA10067, University of Edinburgh

Please do not duplicate and distribute
Shortcomings of Symmetric Key Crypto

Public Key Crypto (PKC)

RSA (Rivest, Shamir, Adleman) in 1978
  - Problem of factoring large numbers

Uses for Public Key Crypto

Knapsack Problem
  - Merkle-Hellman knapsack cryptosystem
  - The first proposed PKC. But, it is insecure

Diffie-Hellman Key Exchange in 1976
  - Discrete log problem

Elliptic Curve Cryptography (ECC)
  - Based on the algebraic structure of elliptic curves over finite fields
PKC is Newcomer

- Different name
  - Asymmetric cryptography
    - Consider the symmetric cryptography
  - Two key cryptography
  - Non-security key cryptography

- The concept is relative newcomer
  - In the late 1960s by GCHQ of British
    - GCHQ (The Government Communications Headquarters):
      정부통신본부
  - Independently, in early 1970s by academic researchers
Misconceptions on PKC

- PKC is more secure than that of symm cipher
  - Cipher Security is depends on computational work to break a cipher – both are depends on it
- PKC made symm cipher obsolete
  - The problem of computation overhead of PKC
- Key distribution of PKC is trivial
  - The procedures of PKC are so not simpler and more efficient than those of symm cipher
  - PKI is required for the key distribution of PKC
Key Generation of PKC

- Making two keys: Based on trap door one way function
  - Easy to compute in one direction
  - Hard to compute in other direction
  - “Trap door” used to create keys
  - Example: Given $p$ and $q$, product $N = p \times q$ is easy to compute, but given $N$, it is hard to find $p$ and $q$

- A message encrypted by the public key can decrypted only with the corresponding private key
Two main branches of PKC

- **Public key Encryption**
  - Suppose we encrypt $M$ with Alice’s public key
  - Only Alice’s private key can decrypt to find $M$

- **Digital Signature**
  - Sign by “encrypting” with private key
  - Anyone can **verify** signature by “decrypting” with public key
  - But only private key holder could have signed
  - Like a handwritten signature (and then some)
Public-Key Cryptography

- Encryption with public key
  - User encrypts data using his or her public key
  - He or she will be able to decrypt the data
  - Directed toward providing confidentiality

- Encryption with private key
  - User encrypts data using his or her own private key
  - Anyone who knows the corresponding public key will be able to decrypt the message
  - Directed toward providing authentication

Figure 2.7 Public-Key Cryptography
RSA cryptosystem
(Ron Rivest, Adi Shamir, Leonard Adleman) at MIT in 1978
Turing award in 2002
### RSA

- The most difficult computation?

<table>
<thead>
<tr>
<th>Addition</th>
<th>Multiplication</th>
<th>Factorization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Easy</strong></td>
<td></td>
<td><strong>Difficult</strong></td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>221 = ?x?</td>
</tr>
<tr>
<td>+ 654</td>
<td>x 654</td>
<td>221/2 =</td>
</tr>
<tr>
<td>--------</td>
<td>---------------</td>
<td>221/3 =</td>
</tr>
<tr>
<td>777</td>
<td>492</td>
<td>221/5 =</td>
</tr>
<tr>
<td></td>
<td>615</td>
<td>221/7 =</td>
</tr>
<tr>
<td></td>
<td>738</td>
<td>221/11 =</td>
</tr>
<tr>
<td></td>
<td>---------------</td>
<td>221/13 =</td>
</tr>
<tr>
<td></td>
<td>80442</td>
<td>221 = 13 x 17</td>
</tr>
</tbody>
</table>

- Easy
- Difficult
RSA Cryptosystem

Key Generation

- Select $p$, $q$ such that $p$ and $q$ are both prime
- Calculate $n = p \times q$
- Calculate $\varphi(n) = (p-1)(q-1)$
- Select integer $e$ such that $\gcd(\varphi(n), e) = 1$ and $1 < e < \varphi(n)$
- Calculate $d = e^{-1} \mod \varphi(n)$
- Public key: $KU = \{e, n\}$
- Private key: $KR = \{d, n\}$

Encryption

- Plaintext: $M < n$
- Ciphertext: $C = M^e \mod n$

Decryption

- Ciphertext: $C$
- Plaintext: $M = C^d \mod n$

- $e$ is relatively prime to $\varphi(n)$
- $e \cdot d \equiv 1 \mod \varphi(n)$
  - $\varphi(n) = \varphi(pq) = (p-1)(q-1)$
  - $\gcd(e, d) = 1$ and $e \cdot d \mod \varphi(n) = 1$
- Trapdoor:
  - $\varphi(n) = (p-1)(q-1)$

- $M = C^d \mod n = (M^e)^d \mod n = M^{ed} \mod n = M$
- $M \equiv C^d \equiv (M^e)^d \equiv M^{ed} \mod n$
RSA

- Invented by Cocks (GCHQ), independently, by Rivest, Shamir and Adleman (MIT)

- Let $p$ and $q$ be two large prime numbers

- Let $N = p \times q$ be the modulus. \( \varphi(n) = \varphi(pq) = (p-1)(q-1) \)

- Choose $e$ relatively prime to $(p-1)(q-1)$ \( \gcd(\varphi(n), e) = 1, \ 1 < e < \varphi(n) \)

- Find $d$ s.t. \( e \times d = 1 \mod (p-1)(q-1) \)

- **Public key** is \( (N, e) \)

- **Private key** is $d$

---

- To encrypt message $M$ compute: \( C = M^e \mod N \quad (M < n) \)

- To decrypt $C$ compute: \( M = C^d \mod N \)

- Recall that $e$ and $N$ are public

- If attacker can factor $N$, he can use $e$ to easily find $d$ since \( e \times d = 1 \mod (p-1)(q-1) \)

- **Factoring the modulus breaks RSA**

- It is not known whether factoring is the only way to break RSA
Does RSA Really Work?

- Given $C = M^e \mod N$, we must show
  - $M = C^d \mod N = M^{ed} \mod N$ where $M < N$

- We’ll use **Euler’s Theorem**
  - If $M$ is relatively prime to $N$ then $M^{\phi(N)} = 1 \mod N$

- Facts:
  - $e \cdot d = 1 \mod (p - 1)(q - 1)$
  - By definition of “mod”, $e \cdot d = k(p - 1)(q - 1) + 1$
  - $\phi(N) = (p - 1)(q - 1)$
  - Then $e \cdot d - 1 = k(p - 1)(q - 1) = k\phi(N)$

- $M^{ed} = M^{(ed - 1) + 1} = M \cdot M^{ed - 1} = M \cdot M^{\phi(N)} = M \cdot (M^{\phi(N)})^k \mod N$
  - $= M \cdot 1^k \mod N = M \mod N$
Simple RSA Example

- Example of RSA
  - Select “large” primes \( p = 11, q = 3 \)
  - Then \( N = p \cdot q = 33 \) and \( (p-1)(q-1) = 20 \)
  - Choose \( e = 3 \) (relatively prime to 20)
  - Find \( d \) such that \( e \cdot d = 1 \mod 20 \), we find that \( d = 7 \) works

  - **Public key:** \((N, e) = (33, 3)\)
  - **Private key:** \(d = 7\)

- Suppose message \( M = 8 \)
- Ciphertext \( C \) is computed as
  \[
  C = M^e \mod N = 8^3 = 512 = 17 \mod 33
  \]
- Decrypt \( C \) to recover the message \( M \) by
  \[
  M = C^d \mod N = 17^7 = 410,338,673 = 12,434,505 \cdot 33 + 8 = 8 \mod 33
  \]
RSA Example 2

1. Select two primes, p = 7 and q = 17
2. Calculate n = pq = 7 \times 17 = 119
3. Calculate \( \phi(n) = (p-1)(q-1) = 96 \)
4. Select e s.t. e is relatively prime to \( \phi(n) \) and less than \( \phi(n) \); in this case, \( e = 5 \)
5. Determine d s.t. \( d \times e \mod 96 = 1 \) and d < 96. The correct value is d = 77 (77 \times 5 = 385 = 4 \times 96 + 1)
6. KU = {5, 119}, KR = {77, 119}
A Real World Example


- lets encrypt the message "attack at dawn"
  - The first thing that must be done is to convert the message into a numeric format.
  - "attack at dawn" becomes 1976620216402300889624482718775150
    - https://gist.github.com/barrysteyn/4184435#file_convert_text_to_decimal.py

- Pick two large primes, p and q
  - p
    121310724392112718973236715316124404284724276337014109256345493123019643730420856
    19324197365322416866541017057361365214171711713797974299334871062829803541
  - q
    1202752425547848885956220793734512128733387803682075433653899839551798509887978
    998691469008091316111533468170508320960221601463663463491812470987105415233

- calculate n and φ(n)
  - n
    1459067680075833232301869393490706352924018723753571643499581871019873438799005358
    938369571402670149802121818086292467422828157022922076746906543401224889672472407
    926969987100581290103199317858753663710862357656510507883714297115637342788911463
    535102712032765166518411726859837988672111837205085526346618740053
A Real World Example  (cont’)

- calculate n and \( \varphi(n) \)
  - \( \varphi(n) \)
    
    145906768007583323230186939349070635292401872375357164399581871019873438799005358
    938369571402670149802121818086292467422828157022922076746906543401224889648313811
    232279966317301397777852365301547848273478871297222058587457152891606459269718119
    268971163555070802643999529549644116811947516513938184296683521280

- \( e \) - the public key
  65537 has a gcd of 1 with \( \varphi(n) \), so let's use it as the public key.

- To calculate the private key, use extended euclidean algorithm to find the multiplicative inverse with respect to \( \varphi(n) \).

- \( d \) - the private key
  89489425009274444368228545921773093919669586065884257445497854456487674839629
  81839093494197326287961679797060891728367987549933157416111385408881327548811
  05882471930775825272784379065040156806234235500672400424666656542323835029222
  15493623289472138866445818789127946123407807725702626644091036502372545139713
A Real World Example (cont’)

- Encryption / Decryption
  - 평문: “attack at dawn” = 1976620216402300889624482718775150

```
Encryption: 1976620216402300889624482718775150^e \bmod n

Decryption:
```

```
1976620216402300889624482718775150^d \bmod n
```

(which is our plaintext "attack at dawn")
More Efficient RSA (1/2)

- Modular exponentiation example
  - $5^{20} = 95367431640625 = 25 \mod 35$

- A better way: repeated squaring
  - $20_{10} = 10100_2$
  - $(1, 10, 101, 1010, 10100)_2 = (1, 2, 5, 10, 20)_{10}$
  - Note that $2 = 1 \cdot 2$, $5 = 2 \cdot 2 + 1$, $10 = 2 \cdot 5$, $20 = 2 \cdot 10$
  - $5^{1} = 5 \mod 35$
  - $5^{2} = (5^{1})^{2} = 5^{2} = 25 \mod 35$
  - $5^{5} = (5^{2})^{2} \cdot 5^{1} = 25^{2} \cdot 5 = 3125 = 10 \mod 35$
  - $5^{10} = (5^{5})^{2} = 10^{2} = 100 = 30 \mod 35$
  - $5^{20} = (5^{10})^{2} = 30^{2} = 900 = 25 \mod 35$

- Never have to deal with huge numbers!
More Efficient RSA (2/2)

- Let $e = 3$ for all users (but not same $N$ or $d$)
  - Public key operations only require 2 multiplies
  - Private key operations remain “expensive”
  - If $M < N^{1/3}$ then $C = M^e = M^3$ and cube root attack
    - $(\text{mod } N)$ operation has no effect
  - For any $M$, if $C_1, C_2, C_3$ sent to 3 users, cube root attack works (uses Chinese Remainder Theorem)
  - Can prevent cube root attack by padding message with random bits

- Note: $e = 2^{16} + 1$ also used: Protect CRT attack
## Modular Power

\[(a \times b) \mod n = [(a \mod n) \times (b \mod n)] \mod n\]

\[x^{16} = \text{xxxxxxxxxxxxxxx} = x^8 \times x^8 \leftarrow (x^4 \times x^4) \times (x^4 \times x^4) \leftarrow\]

<table>
<thead>
<tr>
<th>19^5 Mod 119</th>
<th>5 = 101_{(2)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>19^2 = 19 \times 19 (mod 119)</td>
<td>\equiv 4 (mod 119)</td>
</tr>
<tr>
<td>19^4 \equiv 4 \times 4 (mod 119)</td>
<td>\equiv 16 (mod 119)</td>
</tr>
<tr>
<td>19^5 = 19^4 \times 19^1 (mod 119)</td>
<td>\equiv 16 \times 19 (mod 119)</td>
</tr>
<tr>
<td>\equiv 66 (mod 119)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>66^{77} Mod 119</th>
<th>77 = 1001101_{(2)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>66^2 \equiv 72 (mod 119)</td>
<td></td>
</tr>
<tr>
<td>66^4 \equiv 67 (mod 119)</td>
<td></td>
</tr>
<tr>
<td>66^8 \equiv 86 (mod 119)</td>
<td></td>
</tr>
<tr>
<td>66^{16} \equiv 18 (mod 119)</td>
<td></td>
</tr>
<tr>
<td>66^{32} \equiv 86 (mod 119)</td>
<td></td>
</tr>
<tr>
<td>66^{64} \equiv 18 (mod 119)</td>
<td></td>
</tr>
<tr>
<td>66^{77} = 66^{64} \times 66^8 \times 66^4 \times 66^1</td>
<td>\equiv 19 (mod 119)</td>
</tr>
</tbody>
</table>
Chinese Remainder Theorem

- Suppose $n_1, n_2, \ldots, n_k$ are integers which are pairwise coprime. Then, for any given integers $a_1, a_2, \ldots, a_k$, there exists an integer $x$ solving the system of simultaneous congruences

$$x \equiv a_1 \pmod{n_1}, \quad x \equiv a_2 \pmod{n_2}, \ldots,$$

$$x \equiv a_k \pmod{n_k}$$

- Furthermore, all solutions $x$ to this system are congruent modulo the product $N = n_1n_2\ldots n_k$. 

Congruence: (수학) 합동, 일치, 합치
Uses for Public Key Crypto
Integrity and MAC

- **Integrity**: prevent (or at least detect) unauthorized modification of data
- Example: Inter-bank fund transfers
  - Confidentiality is nice, but integrity is critical
- **Message Authentication Code (MAC)**
  - Used for data integrity
  - Integrity *not* the same as confidentiality

- MAC is computed as **CBC residue**
  - Compute CBC encryption, *but only save the final ciphertext block*
- MAC computation (assuming $N$ blocks)
  \[
  C_0 = E(IV \oplus P_0, K), \\
  C_1 = E(C_0 \oplus P_1, K), \\
  C_2 = E(C_1 \oplus P_2, K), \ldots, \\
  C_{N-1} = E(C_{N-2} \oplus P_{N-1}, K) = MAC
  \]
- **MAC sent along with plaintext**
- Receiver does same computation and verifies that result agrees with **MAC**
  - Receiver must also know the key $K$
Why does a MAC work?

- Suppose Alice has 4 plaintext blocks

- Alice computes \( C_0 = E(IV \oplus P_0, K), \ C_1 = E(C_0 \oplus P_1, K), \)
  \( C_2 = E(C_1 \oplus P_2, K), \ C_3 = E(C_2 \oplus P_3, K) = MAC \)

- Alice sends \( IV, P_0, P_1, P_2, P_3 \) and MAC to Bob

- Suppose Trudy changes \( P_1 \) to \( X \)

- Bob computes \( C_0 = E(IV \oplus P_0, K), \ C_1 = E(C_0 \oplus X, K), \)
  \( C_2 = E(C_1 \oplus P_2, K), \ C_3 = E(C_2 \oplus P_3, K) = MAC \neq MAC \)

- Error propagates into MAC (unlike CBC decryption)

  - Recall CBC decryption
    - If \( C_1 \) is garbled to, say, \( G \) then \( P_1 \neq C_0 \oplus D(G, K), \ P_2 \neq G \oplus D(C_2, K) \)
    - But \( P_3 = C_2 \oplus D(C_3, K), \ P_4 = C_3 \oplus D(C_4, K), \ldots \)

  - Compare the above to the following
    - \( C_0 = E(IV \oplus P_0, K), \ C_1 = E(C_0 \oplus X, K), \)
    \( C_2 = E(C_1 \oplus P_2, K), \ C_3 = E(C_2 \oplus P_3, K) = MAC \neq MAC \)

- Trudy can’t change MAC to MAC without key \( K \)
Uses for Public Key Crypto

- Confidentiality
  - Transmitting data over insecure channel
  - Secure storage on insecure media
- Authentication (later)
- Digital signature provides integrity and non-repudiation
  - No non-repudiation with symmetric keys
  - Who has the secret key is the key for non-repudiation.
No non-repudiation vs. Non-repudiation

No non-repudiation
- Alice orders 100 shares of stock from Bob
- Alice computes MAC using symmetric key
- Stock drops, Alice claims she did not order
- Can Bob prove that Alice placed the order?
- **No!** Since Bob also knows symmetric key, he could have forged message
- **Problem:** Bob knows Alice placed the order, but he can’t prove it

Non-repudiation
- Alice orders 100 shares of stock from Bob
- Alice **signs** order with her private key
- Stock drops, Alice claims she did not order
- Can Bob prove that Alice placed the order?
- **Yes!** Only someone with Alice’s private key could have signed the order
- This assumes Alice’s private key is not stolen (revocation problem)
Sign and Encrypt vs Encrypt and Sign
Public Key Notation

- **Sign** message $M$ with
  - Alice’s **private key**: $[M]_{Alice}$
  - Bob’s **private key**: $[M]_{Bob}$

- **Encrypt** message $M$ with
  - Alice’s **public key**: ${M}_{Alice}$
  - Bob’s **public key**: ${M}_{Bob}$

- Then
  \[
  \{[M]_{Alice}\}_{Alice} = M \\
  [{M}_{Alice}]_{Alice} = M
  \]
Confidentiality and Non-repudiation

- Suppose that we want confidentiality and non-repudiation
- Can public key crypto achieve both?

- Alice sends message to Bob
  - Sign and encrypt $\{[M]_{\text{Alice}}\}_{\text{Bob}}$
  - Encrypt and sign $\{[M]_{\text{Bob}}\}_{\text{Alice}}$

- Can the order possibly matter?
M = “I love you”

Q: What is the problem?
A: Charlie misunderstands crypto!

Alice \{ [M]_{Alice} \}_Bob \rightarrow Bob \rightarrow Charlie \{ [M]_{Alice} \}_Charlie
Encrypt and Sign

- **M** = “My theory, which is mine....”

**Note** that Charlie cannot decrypt **M**

**Q**: What is the problem?

**A**: Bob misunderstands crypto!

**Computer Security & OS Lab, DKU**
Summary

- In 2005, a team of researchers factored the RSA-640 challenge number using 30 2.2GHz CPU years.
- In 2004, the prize for factoring RSA-2048 was $200,000.
- Current practice is 2,048-bit keys.
Secret key (AES key) Distribution with RSA

The Internet

insecure channel

secure key distribution

A's Pub key
A's Private key

A

A's Secret key

Clear Text

AES or RC4

RSA

B's Pub Key

B

B's Pub Key
B's Private key

B's
Pub Key

B's
Private key

Directory Server

Crypt Secret Key

RSA

A's Secret key

AES or RC4

Clear Text
Confidentiality in the Real World

- **Symmetric key +’s**
  - *Speed*

- **Public Key +’s**
  - *Signatures* (non-repudiation)

- **Hybrid cryptosystem**
  - Public key crypto to establish a key
  - Symmetric key crypto to encrypt data
  - Consider the following

  - Public key notation
    - [ ] private key
    - { } public key

  - Symmetric key notation
    - \( C = E(M, K) \)
    - \( M = D(C, K) \)

- Can Bob be sure he’s talking to Alice?
PKC: Secrecy and Authentication

Z = E_{KUb}[E_{KRa}(X)]
X = D_{KUa}[D_{KRb}(Z)]

KUb: B’s public key
KRb: B’s private key
Public-key Cryptography

- based on mathematical functions
- asymmetric
- six ingredients: plaintext, encryption algorithm, public and private key, ciphertext, and decryption algorithm

Applications for asymmetric cryptosystems

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Digital Signature</th>
<th>Symmetric Key Distribution</th>
<th>Encryption of Secret Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>RSA</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Diffie-Hellman</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>DSS</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Elliptic Curve</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Appendix

APIs/library for RSA
Cryo 라이브러리에서는 RSA 알고리즘 관련 API를 제공

- openssl/rsa.h에서 해당 API들의 정의를 찾아볼 수 있음.
- RSA_new() 함수를 이용해 초기화를 수행
  - RSA 객체가 생성되며, 처리 완료 후 RSA_free(RSA *r) 함수를 호출해야
- RSA_new_method(ENGINE *engine), RSA_size(const RSA *)
  - 설정된 RSA context에 실제로 수행하는 Method와 키의 길이를 지정하기 위해서 사용
- RSA_genetate_key(int bits, unsigned long e, void (*callback)(int, int, void *), void *
cb_arg)
  - 두 키를 생성
- RSA_check_key(const RSA *): 생성된 키를 검증
- RSA_public_encrypt(...), RSA_private_encrypt(...)
  - 암호화
- RSA_public_decrypt(...), RSA_private_decrypt(...)
  - 복호화
- RSA_free(RSA *r)
Programming with RSA

**RSA_genetate_key**(int bits, unsigned long e, void (*callback)(int, int, void *), void *cb_arg)

- **bits**: 생성하려는 키의 크기를 비트 단위로 지정한다. 이 값은 16의 배수로 설정해야 하며, 최소한 1024 이상인 값을 지정해야 한다. 통상적으로 2048을 값으로 지정한다.
- **exp**: 키를 생성하기 위한 지수(exponent) 값을 지정한다. 통상적으로 3, 17 또는 65537을 지정하며, 이 값은 RSA 오브젝트에 저장되어 암호화, 복호화 과정에도 이용된다.

**int RSA_public_encrypt**(int flen, const unsigned char *from, unsigned char *to, RSA *rsa, int padding);

- **flen**: 암호화를 하기 위한 평문(PlainText)의 길이
- **from**: 암호화를 위한 평문의 포인터(평문이 저장된 버퍼 포인터)
- **To**: 암호화를 수행한 암호문(CipherText)을 저장할 버퍼의 포인터
- **Rsa**: 평문을 암호화하기 위해서 사용하는 공개 키를 포함하는 RSA 오브젝트
- **Padding**: 사용하려는 패딩의 유형
  - RSA_PKCS1_PADDING, RSA_PKCS1_OAEP_PADDING, RSA_SSLV23_PADDING, RSA_NO_PADDING