Cybersecurity: Public-key Cryptography and the RSA Algorithm

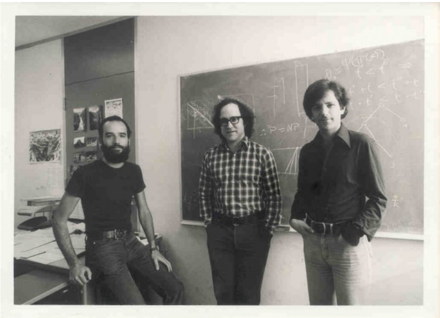
Ozalp Babaoglu

Public-Key Cryptography

- "Is it possible to exchange information confidentially without having to first agree on a key?"
- Breakthrough idea due to Diffie, Hellman and Merkle in their 1976 works
- Respond "yes" to the interrogative as long if the "one-way trap-door" concept can be implemented mathematically

RSA Algorithm

- One of the first practical responses to the challenge posed by Diffie-Hellman was developed by ***Ron Rivest***, ***Adi Shamir***, and ***Len Adleman*** of MIT in 1977
- Resulting algorithm is known as ***RSA***
- Based on properties of ***prime numbers*** and results from ***number theory***



Notation

Let

$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$ denote the set of integers

$\mathbb{Z}_n = \{0, 1, 2, 3, \ldots, n-1\}$ denote the set of integers modulo n

$GCD(m,n)$ denote the ***greatest common divisor*** of m and n

$\mathbb{Z}_n^*$ denote the integers ***relatively prime*** with n

$\varphi(n) = |\mathbb{Z}_n^*|$ denote ***Euler's totient*** function

Some Facts

If $GCD(n, m) = 1$ (n and m are ***relatively prime*** or ***coprime***) then

$$\varphi(nm) = \varphi(n)\varphi(m)$$

If p and q are two primes, then

$$\varphi(p) = (p - 1)$$
$$\varphi(pq) = (p - 1)(q - 1)$$

Example

- Let n=15
- What is $\varphi(15)$=?
- Integers ***relatively prime*** with 15: $\{1, 2, 4, 7, 8, 11, 13, 14\}$
- Therefore, $\varphi(15)=8$
- Observe that $15=3\times5$
- Therefore, $\varphi(n)=\varphi(3\times5)$

$=\varphi(3)\times\varphi(5)$

$=(3-1)\times(5-1)$

$=2\times4$

$=8$

RSA

- To define RSA, we need to specify the following operations:
  - How to generate the keys
  - How to encrypt: $C(m)$
  - How to decrypt: $D(c)$

RSA: Generation of the keys

- Choose two *very large* primes p, q
- Compute $n = p\times q$
- Compute $\varphi(n) = (p-1)(q-1)$
- Choose $1< e < \varphi(n)$ such that $GCD(e, \varphi(n)) = 1$ (e and $\varphi(n)$ are *coprime*)
- Compute d as the *multiplicative inverse* of e:

  $d\times e$ **mod** $\varphi(n) = 1$

- Public key = (e,n)
- Private key = (d,n)

RSA: Encryption

$$C(m) = m^e \textbf{ mod } n$$

RSA: Decryption

$$D(c) = c^d \textbf{ mod } n$$

RSA: Example 1

- Assume we choose $p=5$, $q=11$ (not realistic!!)
- Therefore $n = 5\times 11 = 55$, $\varphi(n) = (5-1)(11-1) = 40$
- Choose $e = 7$ (verify that $GCD(e, \varphi(n)) = GCD(7, 40) = 1$)
- Compute d as the multiplicative inverse of e:

$$d\times e \textbf{ mod } \varphi(n) = 1$$
$$d\times 7 \textbf{ mod } 40 = 1$$

RSA: Example 1

- d can be computed using the *extended Euclidean algorithm*
- Euclidean algorithm computes $GCD(e, \varphi(n))$
- *Extended Euclidean algorithm* expresses $GCD(e, \varphi(n))$ as a linear combination of e and $\varphi(n)$

RSA: Example 1

- Extended Euclidean algorithm for $GCD(7,40)$

$40 = (5)7 + (5)$
$7 = (1)5 + (2)$
$5 = (2)2 + (1)$ Stop when we reach 1 ($GCD(7,40)$)

- Back substitution: Start with last equation in terms of 1

$1 = 5 - 2(2)$ Substitute for 2
$1 = 5 - 2(7 - (1)5)$ Distribute the 2 and collect terms
$1 = 3(5) - 2(7)$ Substitute for 5
$1 = 3(40 - 5(7)) - 2(7)$
$1 = 3(40) - 17(7)$ Stop when we reach e (7)

- The answer is the coefficient 17
- Because it is negative, we have to subtract it from $\varphi(n)$
  $d = 40 - 17 = 23$

RSA: Example 1

- Verify:
  with d=23 e=7, 23×7 **mod** $40 = 1$ $(23\times7=161=40\times4+1)$
- Therefore, the private-public key pair becomes:
  $K[priv] = (23,40)$ $K[pub] = (7,40)$

RSA: Example 2

- Assume we choose p=53, q=61 (still not realistic!!)
- Therefore n=53×61=3233, $\varphi(n)=(53-1)(61-1)=3120$
- Choose e=17 (verify that $GCD(e, \varphi(n))=1$)
- Compute d=2753 and verify that $e\times d$ **mod** $\varphi(n) = 1$
  $e\times d = 2753 \times 17 = 46801$
  $e\times d$ **mod** $\varphi(n) = 46801$ **mod** $3120 = 1$
  since $15 \times 3120 + 1 = 46801$
- Therefore, the private-public key pair becomes:
  $K[priv] = (2753,3233)$ $K[pub] = (17,3233)$

RSA: Example 2

- Let the plaintext message be "*hi*"
- Encode message as a numeric value using the position of the letters in the alphabet: $m = 0809$
- Encryption: 809^{17} **mod** $3233 = 1171 = c$
- Decryption: 1171^{2753} **mod** $3233 = 809 = m$
- Decode numeric value as text: $08 = h$ $09 = i$

Remaining Questions

- How to encode the plaintext message as an integer m such that $0 < m < n$? (Need to divide long messages into blocks)
- How can we guarantee that encryption and decryption are indeed inverses; in other words, $D(C(m)) = m$?
- How can we argue that RSA is secure?
- What about the efficiency of RSA?
- How to carry out the various steps in the algorithm?

Correctness, Security and Efficiency of RSA

Correctness of RSA

- Need to show

$$\forall m\colon D(C(m)) = m$$

Correctness of RSA

- Classical results from number theory
- Euler's Theorem:

if $GCD(m,n) = 1$ then $m^{\varphi(n)} \textbf{ mod } n = 1$

Correctness of RSA

- Properties of modular arithmetic:
  - if $x \textbf{ mod } n = 1$, then for any integer y, we have $x^y \textbf{ mod } n = 1$
  - if $x \textbf{ mod } n = 0$, then for any integer y, we have $x^y \textbf{ mod } n = 0$
  - $(m^x \textbf{ mod } n)^y = (m^x)^y \textbf{ mod } n$
- Let m be an integer encoding of the original message such that $0 < m < n$
- By definition, we have

$$\begin{aligned} D(C(m)) = D(m^e \textbf{ mod } n) &\\ &= (m^e \textbf{ mod } n)^d \textbf{ mod } n \\ &= (m^e)^d \textbf{ mod } n \\ &= m^{ed} \textbf{ mod } n \end{aligned}$$

Correctness of RSA

- By construction, we know that $ed \textbf{ mod } \varphi(n) = 1$
- Therefore, there must exist a positive integer k such that $ed = k\varphi(n) + 1$
- Substituting, we obtain

$$\begin{aligned} D(C(m)) = m^{ed} \textbf{ mod } n &= m^{k\varphi(n)+1} \textbf{ mod } n \\ &= m m^{k\varphi(n)} \textbf{ mod } n \\ &= m \cdot 1 = m \end{aligned}$$

- follows by Euler's Theorem when m is relatively prime to n (but can be extended to hold for all m) and properties of modular arithmetic

Security of RSA

- How can the confidentiality (secrecy) property of RSA be compromised?
- Brute force attack
  - Try all possible private keys
- Defense (as for any other crypto-system)
  - Use large enough key space

Security of RSA

- Mathematical attacks:
  - Factorize n into its prime factors p and q, compute $\varphi(n)$ and then compute $d=e^{-1}(\textbf{mod } \varphi(n))$
  - Compute $\varphi(n)$ without factorizing n, and then compute $d=e^{-1}(\textbf{mod } \varphi(n))$
- Both approaches are characterized by the difficulty of *factoring* n

The Factoring Problem

- No theorems or lower-bound results
- Only empirical evidence about its difficulty
- No guarantee that what is secure today will remain secure tomorrow

The Factoring Problem

Number of decimal digits	Number of bits	Date achieved	MIPS-years	Algorithm
100	332	April 1991	7	Quadratic Sieve
110	365	April 1992	75	Quadratic Sieve
120	398	June 1993	830	Quadratic Sieve
129	428	April 1994	5000	Quadratic Sieve
130	431	April 1996	1000	Generalized number field sieve
140	465	February 1999	2000	Generalized number field sieve
155	512	August 1999	8000	Generalized number field sieve
160	530	April 2003	-	Lattice sieve
174	576	December 2003	-	Lattice sieve
200	663	May 2005	37500	Lattice sieve (18 months using 80 Opteron processors)

- 1GHz Pentium is about a 250-MIPS machine

RSA Factoring Challenge

- Launched by ***RSA Laboratories*** in 1991 to motivate research in computational number theory
- Published ***semi-primes*** (numbers with exactly two prime factors) with 100 to 617 decimal digits
- Offered cash prizes for factoring them
- Declared inactive in 2007

Some RSA Numbers

- **RSA-155**=10941738641570527421809707322040357612003732945449205990913842131476349984288934784717997257891267332497625752899781833797076537244027146743531593354333897

  =102639592829741105772054196573991675900716567808038066803341933521790711307779 × 106603488380168454820927220360012878679207958575989291522270608237193062808643

- **RSA-160**=2152741102718889701896015201312825429257773588845675980170497676778133145218859135673011059773491059602497907111585214302079314665202840140619946994927570407753

  = 45427892858481394071686190649738831656137145778469793250959984709250004157335359 × 47388090603832016196633832303788951973268922921040957944741354648812028493909367

- **RSA-174**=188198812920607963838697239461650439807163563379417382700763356422988859715234665485319060606504743045317388011303396716199692321205734031879550656996221305168759307650257059

  =398075086424064937397125500550386491199064362342526708406385189575946388957261768583317 × 472772146107435302536223071973048224632914695302097116459852171130520711256363590397527

- **RSA-200**=27997833911221327870829467638722601621070446786955428537560009929326128400107609345671052955360856061822351910951365788637105954482006576775098580557613579098734950144178863178946295187237869221823983

  =3532461934402770121272604978198464368671197400197625023649303468776121253679423200058547956528088349 × 7925869954478333033347085841480059687737975857364219960734330341455767872818152135381409304740185467

The Factoring Problem State-of-the-art

- As of November 2010, the 15 semi-primes from RSA-100 to RSA-200 plus RSA-768 had been factored
- As of the end of 2007, special-form numbers of up to 750 bits and general-form numbers of up to about 520 bits can be factored in a few months on a few PCs by a single person without any special mathematical experience

Breaking News!!!

Breaking News!!!

- "A crippling flaw in a widely used code library has fatally undermined the security of millions of encryption keys used in some of the highest-stakes settings, including national identity cards, software- and application-signing, and trusted platform modules protecting government and corporate computers"
- "The weakness allows attackers to calculate the private portion of any vulnerable key using nothing more than the corresponding public portion"
- "The flaw resides in the Infineon-developed ***RSA Library version v1.02.013***, specifically within an algorithm it implements for RSA primes generation"
- Factoring a 2048-bit RSA key generated with the faulty Infineon library takes a maximum of 100 years (on average only half that) and keys with 1024 bits take a maximum of only three months

Efficiency of RSA

- How to compute ($x^z \ \mathbf{mod}\ n$) efficiently:

x^{32}

$$x \longrightarrow x^2 \longrightarrow x^4 \longrightarrow x^8 \longrightarrow x^{16} \longrightarrow x^{32}$$

5 multiplications total since $5 = log_2(32)$

Efficiency of RSA

- What if z is not a power of two?
- Note that from x^y we can obtain x^{2y} and x^{2y+1} with at most two additional multiplications:

$$x^{2y} = (x^y)^2 = x^y \cdot x^y$$

$$x^{2y+1} = x^{2y} \cdot x = x^y \cdot x^y \cdot x$$

- How to decompose z as a linear combination of x^{2y} and x^{2y+1}

Efficiency of RSA

- Suppose we need to compute 1284^{54} **mod** 3233
- Write the exponent 54 as a binary number: 110110_2
- Now we need to compute 1284^{110110_2} **mod** 3233

Efficiency of RSA

- For the time being, ignore **mod** and consider the exponent *one bit at a time* from *msb* to *lsb*
- Example: 1284^{110110_2}

$$\begin{array}{ll} 1284^{1_2} & 1284 \\ 1284^{11_2} & 1284^2 \cdot 1284 \\ 1284^{110_2} & (1284^2 \cdot 1284)^2 \\ 1284^{1101_2} & ((1284^2 \cdot 1284)^2)^2 \cdot 1284 \\ \vdots & \\ 1284^{110110_2} & \end{array}$$

- Thus, we can compute x^y doing only $2\lceil log_2(y) \rceil$ multiplications

Efficiency of RSA

- Property of modular arithmetic:

  $(a \times b)$ **mod** $n = [(a$ **mod** $n) \times (b$ **mod** $n)]$ **mod** n
- Therefore, each of the intermediate results can be reduced by modulo n

Efficiency of RSA

- Example: $1284^{110110_2} \textbf{ mod } 3233$

$$
\begin{array}{ll}
1284^{1_2} & (1284) \textbf{ mod } 3233 \\
1284^{11_2} & (1284^2 \cdot 1284) \textbf{ mod } 3233 \\
1284^{110_2} & ((1284^2 \cdot 1284)^2) \textbf{ mod } 3233 \\
1284^{1101_2} & (((1284^2 \cdot 1284)^2)^2 \cdot 1284) \textbf{ mod } 3233
\end{array}
$$

- This makes the computation practical and avoids ***overflows***

Generation of Large Primes

Generation of Large Primes

- For small primes, we can look them up in a table
- But what if we want primes that have hundreds of digits?
- How are prime numbers distributed?
- What is the probability that a number n picked at random is prime?

$$Pr(n \text{ picked at random is prime}) \sim 1/log(n)$$

Generation of Large Primes

- For example, if n has 10 digits, then $Pr(n \text{ is prime}) \sim 1/23$
- If n has 100 digits, then $Pr(n \text{ is prime}) \sim 1/230$
- These probabilities are too small for us to use the randomly generated number as if it were prime
- If we had a test for primality, `p_test(n)`, we could use it to reject the randomly generated number if the test fails and generate a new one until the test succeeds

```
n=rand() #generate a large random number
while p_test(n) == false:
   n=rand()
```

Primality Testing

- How to implement `p_test(n)` such that it responds "`true`" if n is prime, "`false`" otherwise (composite)
- Naïve method: check wether any integer k from 2 to $n-1$ divides n
- Rather than testing all integers up to $n-1$, if suffices to test only up to $\sqrt{n}$
- Complexity: $O(\sqrt{n})$ or $O(2^{½m})$ where $m=log(n)$ is the size of the input in bits

Primality Testing

- Until recently, no polynomial (in the size of the input) algorithm existed for primality testing
- If we assume the *generalized Riemann hypothesis*, an $O((log\ n)^4)$ for primality testing exists
- In 2002, Agrawal, Kayal and Saxena (AKS) discovered an $O((log\ n)^6)$ for primality testing
- Even though these algorithms are polynomial, they are too expensive to be practical
- Resort to "probabilistic" primality testing

Probabilistic Primality Testing

- Fermat's little theorem:
  if n is prime, then for any integer $a,\ 0<a<n$
  $a^{(n-1)}\ \mathbf{mod}\ n = 1$
- Result of Pomerance (1981):
  - What is the probability that Fermat's theorem ***holds*** even when n is *not* a prime?
  - Let n be a ***large integer*** (more than 100 digits)
  - For any positive random integer a less than n
    $Pr[(n \text{ is not prime}) \text{ and } (a^{(n-1)}\ \mathbf{mod}\ n = 1)] \simeq 10^{-13}$

Probabilistic Primality Testing

```
def p_test(n):
    a = rand() mod n
    x = a^(n-1) mod n
    if x == 1:
        return "true"
    else:
        return "false"
```

Probabilistic Primality Testing

- If the test "fails", then n is not prime
- If the test "passes", then n may still not be a prime with probability 10^{-13}
- This probability is small but may still not be acceptable
- *Idea*: repeat the test k times with different values of a each time

Probabilistic Primality Testing

```
def p_test(n, k):
    repeat k times:
        a = rand() mod n
        x = a^(n−1) mod n
        if x != 1:
            return "false"
    return "true"
```


Probabilistic Primality Testing

- Probability of accepting n that is not prime is reduced to $(10^{-13})^k$
- On the average, how many numbers are tested before accepting?

$$log(n)/2$$

- Example: for a 200-bit random number, need about $log(2^{200})/2=70$ trials

Other Public-key Schemes

- While it is relatively easy to calculate exponentials modulo a prime, it is very difficult to calculate ***discrete logarithms***
- The discrete logarithm of g base b is the integer k solving the equation $b^k=g$ where b and g are elements of a finite group
- Public-key schemes based on discrete logarithms
  - Diffie-Hellman
  - El Gamal