# INTRODUCTION TO PUBLIC-KEY CRYPTOGRAPHY

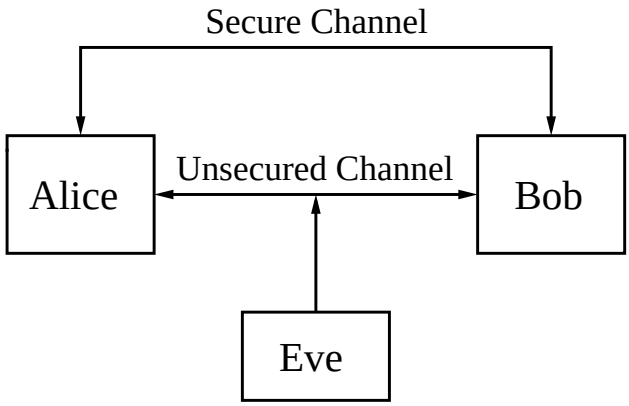


## Outline

1. Drawbacks with Symmetric-Key Cryptography
   (a) Key Establishment Problem
   (b) Key Management Problem
   (c) Non-Repudiation is Difficult to Achieve
2. Public-Key Cryptography
   (a) Public-Key Encryption
   (b) Digital Signatures
   (c) Hybrid Schemes

# Drawbacks with Symmetric-Key Cryptography

Symmetric-key cryptography: Communicating parties a priori share some *secret* information.

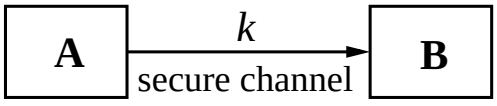





# Key Establishment Problem

How do Alice and Bob establish the secret key $k$?

Method 1: Point-to-point key distribution.
(Alice selects the key and sends it to Bob over a secure channel)




The secure channel could be:

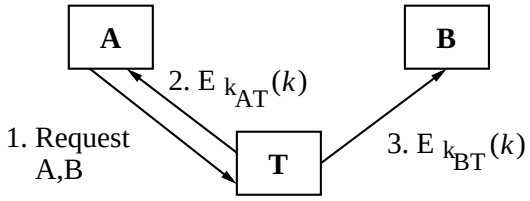
- A trusted courier.
- A face-to-face meeting in a dark alley, etc.

This is not practical for large-scale applications.

# Key Establishment Problem (2)

Method 2: Use a Trusted Third Party $T$.

- Each user $A$ shares a secret key $k_{AT}$ with $T$ for a symmetric-key encryption scheme $E$.
- To establish this key, $A$ must visit $T$ once.
- $T$ serves as a key distribution centre (KDC):




1. $A$ sends $T$ a request for a key to share with $B$.
2. $T$ selects a session key $k$, and encrypts it for $A$ using $k_{AT}$.
3. $T$ encrypts $k$ for $B$ using $k_{BT}$.



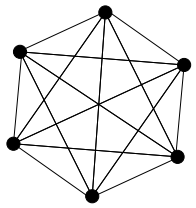
# Key Establishment Problem (3)

Drawbacks of using a KDC:

- The TTP must be unconditionally trusted.
  - Makes it an attractive target.
- Requirement for an on-line TTP.
  - Potential bottleneck.
  - Critical reliability point.

# Key Management Problem

- In a network of $n$ users, each user has to share a different key with every other user.



- Each user thus has to store $n - 1$ different secret keys.
- The total number of secret keys is $\binom{n}{2} \approx n^2/2$.

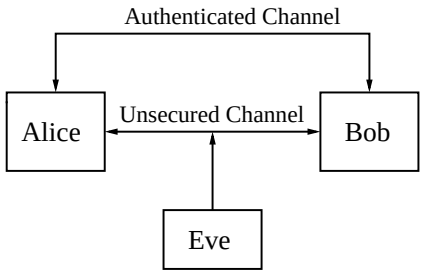


# Non-Repudiation is Impractical

- *Non-repudiation*: Preventing an entity from denying previous actions or commitments.
  - Denying being the source of a message.
- Symmetric-key techniques can be used to achieve non-repudiation, but typically requires the services of an on-line TTP (e.g., use a MAC algorithm where each user shares a secret key with the TTP), or different keying material for each signature (so-called *one-time signature schemes*).

# Public-Key Cryptography

- Public-key cryptography: Communicating parties a priori share some *authenticated* (but non-secret) information.




- Invented by Ralph Merkle, Whitfield Diffie, Martin Hellman in 1975.
  (And in 1970 by researchers at GCHQ.....)

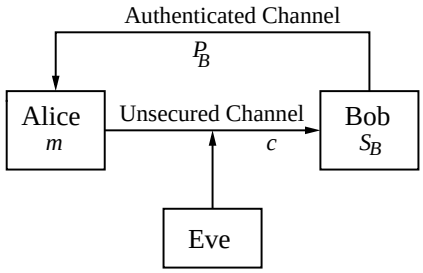


# Key Pair Generation for Public-Key Crypto

- Each entity $A$ does the following:
  1. Generate a key pair $(P_A, S_A)$.
  2. $S_A$ is $A$'s *secret key*.
  3. $P_A$ is $A$'s *public key*.

- Security requirement: It should be infeasible for an adversary to recover $S_A$ from $P_A$.

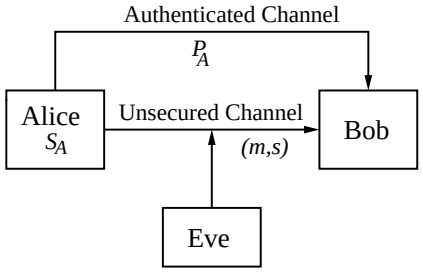
# Public-Key Encryption




- To *encrypt* a secret message $m$ for Bob, Alice does:
  1. Obtain an authentic copy of Bob's public key $P_B$.
  2. Compute $c = E(P_B, m)$; $E$ is the encryption function.
  3. Send $c$ to Bob.
- To *decrypt* $c$, Bob does:
  1. Compute $m = D(S_B, c)$; $D$ is the decryption function.



# Digital Signatures




- To *sign* a message $m$, Alice does:
  1. Compute $s = \text{Sign}(S_A, m)$.
  2. Send $m$ and $s$ to Bob.
- To *verify* Alice's signature $s$ on $m$, Bob does:
  1. Obtain an authentic copy of Alice's public key $P_A$.
  2. Accept if $\text{Verify}(P_A, m, s) = \text{Accept}$.

# Public-Key Versus Symmetric-Key

Advantages of public-key cryptography:

- No requirement for a secret channel.
- Each user has only 1 key pair, which simplifies key management.
- Facilitates the provision of non-repudiation services (with digital signatures).

Disadvantages of public-key cryptography:

- Public keys are typically larger than symmetric keys.
- Public-key schemes are slower than their symmetric-key counterparts.



# Hybrid Schemes

In practice, symmetric-key and public-key schemes are used together. An example:

To *encrypt a secret signed message* $m$, Alice does:

1. Compute $s = \text{Sign}(S_A, m)$.
2. Select a secret key $k$ for a symmetric-key encryption scheme such as the AES.
3. Obtain an authentic copy of Bob's public key $P_B$.
4. Send $c_1 = E(P_B, k)$, $c_2 = \text{AES}_k(m, s)$.

To *recover* $m$ *and verify its authenticity*, Bob does:

1. Decrypt $c_1$: $k = D(S_B, c_1)$.
2. Decrypt $c_2$ using $k$ to obtain $(m, s)$.
3. Obtain an authentic copy of Alice's public key $P_A$.
4. Verify Alice's signature $s$ on $m$.

# ALGORITHMIC NUMBER THEORY



## Outline

1. Fundamental Theorem of Arithmetic
2. Basic Concepts from Complexity Theory
3. Basic Integer Operations
4. Basic Modular Operations

# Fundamental Theorem of Arithmetic

Theorem: Every integer $n \geq 2$ has a unique prime factorization (up to ordering of factors).

Interesting questions:

- Given an integer $n$, how do we find its prime factorization *efficiently*?
- How do we *efficiently* verify an alleged prime factorization of an integer $n$?



# Basic Concepts from Complexity Theory

- An *algorithm* is a "well-defined computational procedure" (e.g., a Turing machine) that takes a variable input and eventually halts with some output.
  - For an integer factorization algorithm, the input is a positive integer $n$, and the output is the prime factorization of $n$.
- The *efficiency* of an algorithm is measured by the scarce resources it consumes (e.g. time, space, number of processors).
- The *input size* is the number of bits required to write down the input using a reasonable encoding.
  - The size of a positive integer $n$ is $\lfloor \log_2 n \rfloor + 1$ bits.

# Basic Concepts from Complexity Theory (2)

- The *running time* of an algorithm is an upper bound as a function of the input size, of the worst case number of basic steps the algorithm takes over all inputs of a fixed size.
- An algorithm is a *polynomial-time* (efficient) algorithm if its (expected) running time is $O(k^c)$, where $c$ is a fixed positive integer, and $k$ is the input size.
- Recall that if $f(n)$ and $g(n)$ are functions from the positive integers to the positive real numbers, then $f(n) = O(g(n))$ means that there exists a positive constant $c$ and a positive integer $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.
  - For example, $7.5n^3 + 1000n^2 - 99 = O(n^3)$.



# Basic Integer Operations

Input: Two $k$-bit integers $a$ and $b$.

Input size: $O(k)$ bits.

| Operation | Running time of naive algorithm (in bit operations) |
|---|---|
| Addition: $a + b$ | $O(k)$ |
| Subtraction: $a - b$ | $O(k)$ |
| Multiplication: $a \cdot b$ | $O(k^2)$ |
| Division: $a = qb + r$ | $O(k^2)$ |
| GCD: $\gcd(a, b)$ | $O(k^2)$ |

# Basic Modular Operations

Input: A $k$-bit integer $n$, and integers $a, b, m \in [0, n-1]$.

Input size: $O(k)$ bits.

| Operation | Running time of naive algorithm (in bit operations) |
|---|---|
| Addition: $a + b \bmod n$ | $O(k)$ |
| Subtraction: $a - b \bmod n$ | $O(k)$ |
| Multiplication: $a \cdot b \bmod n$ | $O(k^2)$ |
| Inversion: $a^{-1} \bmod n$ | $O(k^2)$ |
| Exponentiation: $a^m \bmod n$ | $O(k^3)$ |



# Modular Exponentiation

Input: A $k$-bit integer $n$, and integers $a, m \in [1, n-1]$.
Output: $a^m \bmod n$.

- Naive algorithm 1:
  Compute $d = a^m$.
  Return($d \bmod n$).

- Naive algorithm 2:
  $A \leftarrow a$
  For $i$ from 2 to $m$ do: $A \leftarrow A \times a \bmod n$.
  Return($A$).

# Modular Exponentiation

*Repeated square-and-multiply algorithm*

Let the binary representation of $m$ be $m = \sum_{i=0}^{k-1} m_i 2^i$. Then

$$a^m \equiv a^{\sum_{i=0}^{k-1} m_i 2^i} \equiv \prod_{i=0}^{k-1} a^{m_i 2^i} \equiv \prod_{\substack{0 \le i \le k-1 \\ m_i = 1}} a^{2^i} \pmod{n}.$$

This suggests the following algorithm for computing $a^m \bmod n$:

Write $m$ in binary: $m = \sum_{i=0}^{k-1} m_i 2^i$.
If $m_0 = 1$ then $B \leftarrow a$; else $B \leftarrow 1$.
$A \leftarrow a$.
For $i$ from 1 to $k-1$ do:
    $A \leftarrow A^2 \bmod n$.
    If $m_i = 1$ then $B \leftarrow B \times A \bmod n$.
Return($B$).

Analysis: At most $k$ modular squarings and $k$ modular multiplications, so worst-case running time is $O(k^3)$ bit operations.



# THE RSA ENCRYPTION SCHEME

# Outline

1. RSA Key Pair Generation
2. Basic RSA Public-Key Encryption Schcme
3. Security of the RSA Key Generation Process
4. Security of RSA Public-Key Encryption

See class notes



# Toy Example: RSA Key Generation

Alice does the following:

1. Selects primes $p = 23$ and $q = 37$.
2. Computes $n = pq = 851$ and $\phi(n) = (p-1)(q-1) = 792$.
3. Selects $e = 631$ satisfying $\gcd(631, 792) = 1$.
4. Solves $631d \equiv 1 \pmod{792}$ to get $d \equiv -305 \equiv 487 \pmod{792}$, and selects $d = 487$.
5. Alice's public key is ($n = 851$, $e = 631$); her private key is $d = 487$.

# Toy Example: RSA Encryption

To encrypt a message $m = 13$ for Alice, Bob does:

1. Obtains Alice's public key ($n = 851$, $e = 631$).
2. Computes $c = 13^{631} \bmod 851$ using the repeated-square-and-multiply algorithm:
   (a) Write $e = 631$ in binary:

$$e = 2^9 + 2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0.$$

   (b) Compute successive squarings of $m = 13$ modulo $n$:

$$\begin{aligned} 13 &\equiv 13 \pmod{851} & 13^2 &\equiv 169 \pmod{851} \\ 13^{2^2} &\equiv 478 \pmod{851} & 13^{2^3} &\equiv 416 \pmod{851} \\ 13^{2^4} &\equiv 303 \pmod{851} & 13^{2^5} &\equiv 752 \pmod{851} \\ 13^{2^6} &\equiv 440 \pmod{851} & 13^{2^7} &\equiv 423 \pmod{851} \\ 13^{2^8} &\equiv 219 \pmod{851} & 13^{2^9} &\equiv 305 \pmod{851}. \end{aligned}$$



# Toy Example: RSA Encryption (2)

(c) Multiply together the squares $13^{2^i}$ for which the $i$th bit (where $0 \le i \le 9$) of the binary representation of 631 is 1:

$$\begin{aligned} 13^{631} &= 13^{2^9+2^6+2^5+2^4+2^2+2^1+2^0} \\ &= 13^{2^9} \cdot 13^{2^6} \cdot 13^{2^5} \cdot 13^{2^4} \cdot 13^{2^2} \cdot 13^{2^1} \cdot 13^{2^0} \\ &\equiv 305 \cdot 440 \cdot 752 \cdot 303 \cdot 478 \cdot 169 \cdot 13 \pmod{851} \\ &\equiv 616 \pmod{851}. \end{aligned}$$

3. Bob sends $c = 616$ to Alice.

To decrypt $c = 616$, Alice uses her private key $d = 487$ as follows:

1. Compute $m = 616^{487} \bmod 851$ to get $m = 13$.

# STATUS OF INTEGER FACTORIZATION



## Outline

1. Definitions from Complexity Theory
2. Special-Purpose Factoring Algorithms
3. General-Purpose Factoring Algorithms
4. History of Factoring

# Big-O and Little-o Notation

- Let $f(n)$ and $g(n)$ be functions from the positive integers to the positive real numbers.
- (*Big-O notation*) We write $f(n) = O(g(n))$ if there exists a positive constant $c$ and a positive integer $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.
  - Example: $3n^3 + 4n^2 + 79 = O(n^3)$.
- (*Little-o notation*) We write $f(n) = o(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

  - Example: $\frac{1}{n} = o(1)$.



# Measures of Running Time

- (*Polynomial-time algorithm*) One whose worst-case running time function is of the form $O(n^c)$, where $n$ is the input size and $c$ is a constant.
- (*Exponential-time algorithm*) One whose worst-case running time function is not of the form $O(n^c)$.
  - In this course, fully exponential-time functions are of the form $2^{cn}$, where $c$ is a constant.
  - (*Subexponential-time algorithm*) One whose worst-case running time function is of the form $2^{o(n)}$, and not of the form $O(n^c)$ for any constant $c$.

Roughly speaking, “polynomial-time = efficient”,
“fully exponential-time = terribly inefficient”,
“subexponential-time = inefficient but not terribly so”.

# Example (Trial Division)

- Consider the following algorithm (trial division) for factoring RSA-moduli $n$.
- Trial divide $n$ by the primes $2, 3, 5, \ldots, \sqrt{n}$. If any of these, say $l$, divides $n$, then stop and output the factor $l$ of $n$.
- The running time of this method is at most $\sqrt{n}$ trial divisions, which is $O(\sqrt{n}\,)$.
- Is this a polynomial-time algorithm for factoring RSA moduli?



# Subexponential Time

- Let $A$ be an algorithm whose inputs are elements of the integers modulo $n$, $\mathbb{Z}_n$, or an integer $n$ (so the input size is $O(\log n)$).
- If the expected running time of $A$ is of the form

$$L_n[\alpha, c] = O\left(\exp\left((c + o(1))(\log n)^{\alpha}(\log\log n)^{1-\alpha}\right)\right),$$

where $c$ is a positive constant, and $\alpha$ is a constant satisfying $0 < \alpha < 1$, then $A$ is a subexponential-time algorithm.

Note ($\alpha = 0$): $L_n[0, c] = O((\log n)^{c+o(1)})$ (polytime).

Note ($\alpha = 1$): $L_n[1, c] = O(n^{c+o(1)})$ (fully exponential time).

# Special-Purpose Factoring Algorithms

- Examples: Trial division, Pollard's p-1 algorithm, Pollard's $\rho$ algorithm, elliptic curve factoring algorithm, special number field sieve.
- These are only efficient if the number $n$ being factored has a special form (e.g., $n$ has a prime factor $p$ such that $p-1$ has only small factors; or $n$ has a prime factor $p$ that is relatively small).
- To maximize resistance to these factoring attacks on RSA moduli, one should select the RSA primes $p$ and $q$ *at random* and *of the same bitlength*.



# General-Purpose Factoring Algorithms

- These are factoring algorithms whose running times do not depend of any properties of the number being factored.
- Two major developments in the history of factoring:
  1. (1982) *Quadratic sieve factoring algorithm (QS)*: Running time: $L_n[\frac{1}{2}, 1]$.
  2. (1990) *Number field sieve factoring algorithm (NFS)*: Running time: $L_n[\frac{1}{3}, 1.923]$.

# History of Factoring

| Year | Number | Bits | Method | Notes |
|---|---|---|---|---|
| 1903 | $2^{67}-1$ | 67 | Naive | F. Cole (3 years of Sundays)<br>In 2006: 0.4 secs in Maple |
| 1988 | $\approx 10^{100}$ | 332 | QS | Distributed computation by 100's of computers; communication by email |
| 1994 | RSA-129 | 425 | QS | 1600 computers around the world; 8 months |
| 1999 | RSA-155 | 512 | NFS | 300 workstations + Cray; 5 months |
| 2002 | RSA-158 | 524 | NFS | $\approx 30$ workstations + Cray; 3 months |
| 2003 | RSA-174 | 576 | NFS | |
| 2005 | RSA-200 | 663 | NFS | (55 years on a single workstation) |

RSA Factoring challenge:
`www.rsa.com/rsalabs/node.asp?id=2092`



# RSA-200

The largest 'hard' number factored to date is RSA-200 (200 decimal digits, 663 bits), which was factored in 2005.

```
27997833911221327870829467638722601621070446786955
42853756000992932612840010760934567105295536085606
18223519109513657886371059544820065767750985805576
13579098734950144178863178946295187237869221823983
=
35324619344027701212726049781984643686711974001976
25023649303468776121253679423200058547956528088349
*
79258699544783330333470858414800596877379758573642
19960734330341455767872818152135381409304740185467
```

# Conclusions

- Factoring is *believed* to be a hard problem. However, we have no *proof* or *theoretical evidence* that factoring is indeed hard.
- Note however that factoring is *known* to be *easy* on a quantum computer.
  - In December 2001, the number 15 was factored on a quantum computer by a team of IBM scientists.

  The big open question is whether large-scale quantum computers can ever be built.
- 512-bit RSA is considered insecure today.
- 1024-bit RSA is considered secure today & widely deployed.
- Applications are slowly moving to 2048-bit RSA.



# Equivalent Security Levels

| Security in bits | Block cipher | Hash function | RSA $\|n\|_2$ |
|---|---|---|---|
| 80 | SKIPJACK | (SHA-1) | 1024 |
| 112 | Triple-DES | SHA-224 | 2048 |
| 128 | AES Small | SHA-256 | 3072 |
| 192 | AES Medium | SHA-384 | 7680 |
| 256 | AES Large | SHA-512 | 15360 |

# THE RSA SIGNATURE SCHEME



## Outline

1. Basic RSA Signature Scheme
2. Security of the RSA Signature Scheme
   (a) Hardness of RSAP
   (b) Security Properties of the Hash Function
   (c) Objectives of the Adversary
   (d) Attack Model
   (e) Security Definition
3. Full Domain Hash (FDH)

# Basic RSA Signature Scheme

*Key generation*: Each entity $A$ does the following:

1. Randomly select 2 large distinct primes $p$ and $q$ of the same bitlength.
2. Compute $n = pq$ and $\phi(n) = (p-1)(q-1)$.
3. Select arbitrary $e$, $1 < e < \phi(n)$, such that $\gcd(e, \phi(n)) = 1$.
4. Compute $d$, $1 < d < \phi(n)$, such that $ed \equiv 1 \pmod{\phi(n)}$.
5. $A$'s *public key* is $(n, e)$; $A$'s *private key* is $d$.



# Signature Generation and Verification

*Signature generation*: To sign a message $m \in \{0,1\}^*$, $A$ does the following:

1. Compute $M = H(m)$, where $H$ is a hash function.
2. Compute $s = M^d \bmod n$.
3. $A$'s signature on $m$ is $s$.

*Signature verification*: To verify $A$'s signature $s$ on $m$, $B$ does the following:

1. Obtain an authentic copy of $A$'s public key $(n, e)$.
2. Compute $M = H(m)$.
3. Compute $M' = s^e \bmod n$
4. Accept $(m, s)$ if and only if $M = M'$.

# Security of the Basic RSA Signature Scheme

*Hardness of RSAP*
We require that RSAP be intractable, since otherwise $E$ could forge $A$'s signature as follows:

1. Select $m$.
2. Compute $M = H(m)$.
3. Solve $s^e \equiv M \pmod{n}$ for $s$.
4. Then $s$ is $A$'s signature on $m$.



# Security Properties of the Hash Function

The following are desired security properties of $H$:

*Preimage resistance*: If $H$ is not preimage resistant, and the range of $H$ is $[0, n-1]$, $E$ can forge signatures as follows:

1. Select $s \in [0, n-1]$.
2. Compute $M = s^e \bmod n$.
3. Find $m$ such that $H(m) = M$.
4. Then $s$ is $A$'s signature on $m$.

*2nd preimage resistance*: If $H$ is not 2nd preimage resistant, $E$ could forge signatures as follows:

1. Suppose that $(m, s)$ is a valid signed message.
2. Find an $m'$, $m \neq m'$, such that $H(m) = H(m')$.
3. Then $(m', s)$ is a valid signed message.

# Security Properties of the Hash Function (2)

*Collision resistance*: If $H$ is not collision resistant, $E$ could forge signatures as follows:

1. Select $m_1$, $m_2$ such that $H(m_1) = H(m_2)$, where $m_1$ is an "innocent" message, and $m_2$ is a "harmful" message.
2. Induce $A$ to sign $m_1$: $s = H(m_1)^d \bmod n$.
3. Then $s$ is also $A$'s signature on $m_2$.



# Goals of the Adversary

1. *Total break*: $E$ recovers $A$'s private key, or a method for systematically forging $A$'s signatures.
2. *Selective forgery*: $E$ forges $A$'s signature for a selected subset of messages.
3. *Existential forgery*: $E$ forges $A$'s signature for a single message; $E$ may not have any control over the content or structure of this message.

# Attack Model

Types of attacks $E$ can launch:

1. *Key-only attack*: The only information $E$ has is $A$'s public key.
2. *Known-message attack*: $E$ knows some message/signature pairs.
3. *Chosen-message attack*: $E$ has access to a signing oracle which it can use to obtain $A$'s signatures on some messages of its choosing.



# Security Definition

Definition: A signature scheme is said to be *secure* if it is existentially unforgeable by a computationally bounded adversary who launches a chosen-message attack.

Note: The adversary has access to a signing oracle. Its goal is to compute a single valid message/signature pair for any message that was not previously given to the signing oracle.

Question: Is the basic RSA signature scheme secure?
Answer:
NO, if $H$ is SHA-1; [Details not covered in this course.]
YES if $H$ is a 'full domain' hash function.

# Full Domain Hash RSA (RSA-FDH)

- Same as the basic RSA signature scheme, except that the hash function is $H : \{0,1\}^* \longrightarrow [0, n-1]$.
- In practice, one could use:
  $H(M) =$ SHA-1$(0, M) \,\|$ SHA-1$(1, M) \,\| \cdots \|$ SHA-1$(t, M)$.

- Theorem (Bellare & Rogaway, 1996): If RSAP is intractable and $H$ is a random function, then RSA-FDH is a secure signature scheme.



# ELECTRONIC CASH

# Outline

1. What is Electronic Cash?
2. Simplified Electronic Cash Protocols
3. RSA Blind Signatures
4. On-Line Electronic Cash Protocol
5. Off-Line Electronic Cash Protocol
6. Miscellaneous Notes



# What is Electronic Cash?

- *Electronic cash* is an electronic payment system modeled after our paper cash system.
- Some features of paper cash:
  - Recognizable (as legal tender)
  - Portable (easily carried)
  - Transferable (without involvement of the financial network)
  - Divisible (has the ability to make change)
  - Unforgeable (difficult to duplicate)
  - Untraceable (difficult to keep a record of where money is spent)
  - Anonymous (no record of who spent the money)
- Note: Many of these features are not available with credit cards.

# Basic Concepts

- There are three players in an electronic payment system:
  - A *payer* or *consumer* (called Alice)
  - A *payee* or *merchant* (called Bob)
  - A *financial network* (called the Bank)
- The electronic representation of cash is called a *token* or *(electronic) coin*.
- A device that stores the coin is called a *card*.



# Payment System

- The sequence of events in a payment system is:
  - *Withdrawal*: Alice transfers some funds from her Bank account to her card.
  - *Payment*: Alice transfers money from her card to Bob's.
  - *Deposit*: Bob transfers the money he has received to his Bank account.
- *On-line payments*: Bob calls the Bank to verify the validity of Alice's coin before accepting her payment.
- *Off-line payments*: Bob submits Alice's coin for verification and deposit sometime after the payment transaction is completed.

# Desirable Security Properties of E-Cash

- For the payer:
  - *Payer anonymity* during payment.
  - *Payment untraceability* so that the Bank cannot tell whose money is used in a particular payment.
- For the payee and Bank:
  - *Unforgeable coins*: creating a valid-looking coin without making a corresponding Bank withdrawal should be infeasible.
  - *No double-spending*: A coin cannot be used more than once for making a payment.



# Other Desirable Properties of E-Cash

- Off-line payment system is preferable to on-line
- Payment mechanism should be cheap
- Payment mechanism should be efficient
- Cash should be transferable
- Cash should be divisible

# Simplified Electronic Cash Protocols

Assumptions:

- The Bank has an RSA public key $(n, e)$ and a corresponding private key $d$.
- The Bank's public key is known to Alice and Bob.
- If $M$ is a message, then $\{M\}$ denotes the message together with the Bank's signature on the message. That is, $\{M\} = (M, s)$, where $s = H(M)^d \bmod n$.



# Simplified On-Line Protocol

*Withdrawal protocol*

1. Alice requests a $100 withdrawal from the Bank.
2. Bank debits Alice's account by $100, and gives Alice the coin {This is a $100 bill, #123456789}. Here, 123456789 is a (randomly selected) serial number.

*Payment/Deposit protocol*

1. Alice gives the coin to Bob.
2. Bob forwards the coin to the Bank.
3. Bank verifies the signature.
4. Bank verifies that the coin has not already been spent.
5. Bank enters coin in a *spent-coin database*.
6. Bank credits Bob's account with $100 and informs Bob.
7. Bob completes the transaction with Alice.

# Security Properties

Security properties of the simplified on-line protocol:

- Coins are unforgeable.
- Double-spending is *prevented*.
- Payments are traceable (since the Bank can link the coin to Alice's withdrawal request and the serial number).
- No payer anonymity (since the Bank can link the coin to Alice's withdrawal request and the serial number).



# Simplified Off-Line Protocol

*Withdrawal protocol*

1. Alice requests a $100 withdrawal from the Bank.
2. Bank debits Alice's account by $100, and gives Alice the coin {This is a $100 bill, #123456789}.

*Payment protocol*

1. Alice gives the coin to Bob.
2. Bob verifies the signature.
3. Bob completes the transaction with Alice.

*Deposit protocol*

1. Bob forwards the coin to the Bank.
2. Bank verifies the signature.
3. Bank verifies that the coin has not already been spent.
4. Bank enters coin in a *spent-coin database*.
5. Bank credits Bob's account with $100.

# Security Properties

Security properties of the simplified off-line protocol:

- Coins are unforgeable.
- Payments are traceable.
- No payer anonymity.
- Double-spending is not prevented. However, double-spending is *detected* after it has occurred; the culprit is identified as either Alice or Bob.



# RSA Blind Signatures

Used for payer anonymity and payment untraceability.

*Withdrawal protocol*

1. Alice prepares the message $M =$(This is a \$100 bill, #123456789).
2. Alice selects $r \in_R \mathbb{Z}_n^*$. [$r$ = *blinding factor*]
3. Alice computes $m' = H(M)r^e \bmod n$.
4. Alice requests a \$100 withdrawal from the Bank, and gives $m'$ to the Bank.
5. Bank debits Alice's account by \$100, and gives Alice the signature $s' = (m')^d \bmod n$.
   Note that $s' \equiv H(M)^d r^{ed} \equiv H(M)^d r \pmod n$.
6. Alice computes $s = s'r^{-1} \bmod n$. The coin is $\{M\} = (M, s)$.

# RSA Blind Signatures (2)

Notes:

- Note that $s = H(M)^d \bmod n$ is the Bank's signature on $M$. However, the Bank does not know $M$.
- The payment and deposit protocols are the same as those in the simplified schemes.



# Security Properties

- Payer anonymity and payment untraceability are provided since the Bank cannot link a specific withdrawal with a specific deposit.
- Two problems remain:
  1. Coins can be forged. For example, the Bank may think it is signing a $100 message, when it fact it is signing a $1,000 message.
  2. Double-spending is detected in the off-line scheme, but the culprit cannot be identified.

# Preventing Coin Forgery

There is a simple solution for preventing the forgery of coins.

- The Bank has one key pair for each denomination (e.g., one key pair for $20 coins, one key pair for $100 coins).
- The public key for each denomination is only valid for generating coins of that denomination.



# On-Line Electronic Cash Protocol

*Withdrawal protocol*

1. Alice prepares the message $M =$(This is a $100 bill, #123456789).
2. Alice obtains the Bank's public key $(n, e)$ for generating $100 coins.
3. Alice selects $r \in_R \mathbb{Z}_n^*$.
4. Alice computes $m' = H(M)r^e \bmod n$.
5. Alice requests a $100 withdrawal from the Bank, and gives $m'$ to the Bank.
6. Bank debits Alice's account by $100, and gives Alice the signature $s' = (m')^d \bmod n$ (where $d$ is the Bank's private key for generating $100 coins).
7. Alice computes $s = s'r^{-1} \bmod n$. The coin is $(M, s)$.

# On-Line Electronic Cash Protocol (2)

*Payment/Deposit protocol*

1. Alice gives the coin to Bob.
2. Bob forwards the coin to the Bank.
3. Bank verifies the signature (with its public key for generating $100 coins).
4. Bank verifies that the coin has not already been spent.
5. Bank enters coin in a spent-coin database.
6. Bank credits Bob's account with $100 and informs Bob.
7. Bob completes the transaction with Alice.



# Security Properties

- Payer anonymity and payment untraceability are provided.
- Coins are unforgeable.
- Doubling spending is prevented.

# Off-Line Electronic Cash Protocol

- Double-spending of electronic cash in an off-line protocol cannot be (easily) prevented.
- Many protocols have been invented for detecting double-spending and identifying the culprit.
- The protocol described here is somewhat inefficient, but illustrates the basic ideas.
  - Protocols that are more efficient are known (but not described in this course).
- Also, the probability of cheating (without getting caught) is not very low, namely $\frac{1}{1024}$.
  - Protocols that have have much smaller cheating probabilities are known (but not described in this course).



# Withdrawal Protocol

1. Alice prepares a string $ID_A$ which contains her identifying information. $ID_A$ should be known only to Alice and the Bank.
2. Alice obtains the Bank's public key $(n, e)$.
3. For each $1 \leq i \leq 1024$, $1 \leq j \leq 10$, Alice does:
   Select a random bit string $x_{i,j}$ [*mask*]
   Let $x'_{i,j} = x_{i,j} \oplus ID_A$ [*masked ID*]
   Compute $y_{i,j} = H(x_{i,j})$ and $y'_{i,j} = H(x'_{i,j})$ [*commitment*]
4. For each $1 \leq i \leq 1024$, Alice does:
   Prepare the message: $M_i =$ (This is a \$100 bill, serial#$_i$, $y_{i,1}, y'_{i,1}, \ldots, y_{i,10}, y'_{i,10}$).
   Select $r_i \in_R \mathbb{Z}_n^*$ and compute $m'_i = H(M_i) r_i^e \bmod n$.
5. Alice requests a \$100 withdrawal from the Bank, and gives $m'_1, m'_2, \ldots, m'_{1024}$ to the Bank.

# Withdrawal Protocol (2)

6. The Bank selects $k \in_R [1, 1024]$ and asks Alice to "unblind" all the $m'_i$ except for $m'_k$.
7. Alice gives the Bank all the $M_i$, $x_{i,j}$, $x'_{i,j}$, and $r_i$ except for $i = k$.
8. For each $i \in [1, 1024]$, except for $i = k$, the Bank checks:
   $M_i$ indeed is for \$100.
   $m'_i = H(M_i) r_i^e \bmod n$.
   $y_{i,j} = H(x_{i,j})$, $y'_{i,j} = H(x'_{i,j})$ for $1 \le j \le 10$.
   $x_{i,j} \oplus x'_{i,j} = ID_A$ for $1 \le j \le 10$.
9. Bank debits Alice's account by \$100, and gives Alice the signature $s' = (m'_k)^d \bmod n$.
10. Alice computes $s = s' r_k^{-1} \bmod n$.
11. The coin is $(M_k, s)$.



# Payment Protocol

1. Alice gives $(M_k, s)$ to Bob.
2. Bob verifies the signature using the Bank's public key.
3. Bob selects random bits $b_1, b_2, \ldots, b_{10}$ and sends them to Alice as a challenge.
4. Alice sends $(z_1, z_2, \ldots, z_{10})$ to Bob, where $z_j = x_{k,j}$ if $b_j = 0$ and $z_j = x'_{k,j}$ if $b_j = 1$.
5. For each $1 \le j \le 10$, Bob checks that $y_{k,j} = H(z_j)$ if $b_j = 0$ or $y'_{k,j} = H(z_j)$ if $b_j = 1$.
6. Bob completes the transaction with Alice.

# Deposit Protocol

1. Bob sends $(M_k, s)$ and $z = (z_1, \ldots, z_{10})$ to the Bank.
2. Bank verifies the signature.
3. Bank verifies that the coin has not already been spent. If the coin has been spent, the Bank compares the $z$ vectors of the two coins. If they are the same, the Bank concludes that Bob is trying to deposit the coin twice. If they are different, say the first components of the vectors are different, then the Bank XORs these components to reveal Alice's identity and concludes that Alice double-spent the coin.
4. Bank enters the coin and $z$ in the spent-coin database.
5. Bank credits Bob's account with \$100.



# Notes on the Off-Line E-Cash Protocol

- If Alice tries to double-spend a coin, then with probability $\frac{1023}{1024}$ the second challenge vector is different from the first. Hence the $z$ vectors that Alice returns will reveal her identity.
- Bob cannot create a valid $z$ vector different from the one that Alice gave him the first time she spent the coin.
- Payer anonymity and payment untraceability are provided.
- Coins are unforgeable.
- Double-spending is detected, and the culprit identified.

# Miscellaneous Notes

- Preventing double-spending in off-line electronic cash schemes is possible with tamper-resistant cards.
- Divisibility of electronic cash is relatively easy to achieve.
- Transferability of electronic cash is not easy to achieve without tamper-resistant cards:
  - The coin must contain information about every individual who has spent it so that the Bank maintains the ability to identify multiple spenders.
  - Multiple spending will not be noticed until two copies of the same coin are eventually deposited.
- *Fair* electronic cash schemes have been proposed
  - Anonymity can be removed given an appropriate court order (analogous to key escrow schemes).