Public Key Cryptography (PKC) & RSA Cryptosystem

Cryptography - CS 411 / CS 507

Erkay Savaş

Department of Computer Science and Engineering
Sabancı University

October 31, 2019

Need for PKC

- Distribution and management of secret keys are difficult.
- Need for a secure distribution of secret keys,
  - a secure channel.
- In a networked environment, each pair of users has to have a different key;
  - resulting in too many keys in the system ($n \times (n-1)/2$ if there are n users)
- PKC solution was first proposed in 1976 by Diffie/Hellman

Principle & Protocol

- Every user has a pair of related keys
  - Public key
    - known to everyone in the system with assurance
  - Private key
    - known only to its owner
- Protocol
  1. Alice and Bob agrees on a PKC
  2. Bob sends his public key to Alice
  3. Alice encrypts her message with Bob's public key and sends the ciphertext to Bob
  4. Bob decrypts the ciphertext using his private key.

Hard Problems

- Integer factorization problem - IF (RSA)
- Discrete Logarithm problem - DL (Diffie-Helman, ElGamal)
- Elliptic curve discrete logarithm problem - ECDL (Elliptic Curve Cryptosystems)

Algorithm family	**Bit length**
Integer Factorization (IF)	1024/2048/3072
Discrete Logarithm (DL)	1024/2048/3072
Elliptic Curve Discrete Logarithm (ECDL)	160/224/256
Block Cipher	80/112/128
Cryptographic Hash Functions	160/224/256

Table: Security levels of PKCs

See https://www.keylength.com/en/4/

PKC Applications

- Encryption/decryption
  - Only short messages are encrypted by the receiver's public key,
  - The receiver decrypts it by its private key.
- Digital signature
  - A message digest is encrypted by the message owner's private key
  - Anyone who knows the public key of the message owner can verify that the message and its origin are authentic.
- Key exchange

RSA

- Most popular PKC
- Invented by Rivest/Shamir/Adleman in 1977 at MIT.
- Its patent expired in 2000.
- Based on Integer Factorization problem
- Each user has public and private key pair.

RSA Setup Stage

1. Choose two large primes p and q
2. Compute $n = p \times q$
3. Compute $\Phi(n) = (p-1) \times (q-1)$
4. Choose a random integer, $0 < e < \Phi(n)$ with $\gcd(e, \Phi(n)) = 1$
5. Compute the inverse $d = e^{-1} \bmod \Phi(n)$,
   - i.e., $e \times d \equiv 1 \bmod \Phi(n)$,

- Public key: (e, n)
- Private key: (d, p, q)

RSA Encryption & Decryption

- Encryption done by using public key

$$y = x^e \bmod n, \text{ where } x < n$$

- Decryption done by using private key

$$x = y^d \bmod n$$

Example

Alice

1. Message: $x = 4$
2. $y = x^e \bmod n = 31$
3. Sends y to Bob

Bob

1. chooses $p = 3$, $q = 11$
2. $n = p \cdot q = 33$
3. $\Phi(n) = (p-1)(q-1) = 20$
4. Chooses $e = 3$; $gcd(3, 20) = 1$
5. Computes $d = e^{-1} \bmod \Phi(n)$, $d = 7$
6. Sends (e, n) to Alice
7. $x = y^d \bmod n = 4$

Why does RSA work?

- We want to show that $y^d \bmod n = x$.

- $y^d \bmod n \equiv (x^e \bmod n)^d \bmod n \equiv x^{ed} \mod n$
- $e \cdot d \equiv 1 \mod \Phi(n) \rightarrow e \cdot d = 1 + k \cdot \Phi(n)$
- $x^{ed} \bmod n \equiv x^{1+k\Phi(n)} \bmod n \equiv x^1 \cdot x^{k\Phi(n)} \bmod n$.
- If $x^{\Phi(n)} \equiv 1 \bmod n$
- Then, $x^1 \cdot x^{k\Phi(n)} \bmod n \equiv x \cdot (1)^k \bmod n \equiv x$.

Why does RSA work?

- Euler's theorem:

  If $\gcd(x, n) = 1$ then $x^{\Phi(n)} \equiv 1 \bmod n$

- What if $\gcd(x, n) \neq 1$ (i.e. $\gcd(x, p \cdot q) \neq 1$)
- Assume x is a multiple of q ($x = k_1 q$)
  - $x^{k\Phi(n)} \bmod q = 0$
  - $x^{k\Phi(n)} = x^{k(p-1)(q-1)} = x^{\Phi(p)k(q-1)} \equiv 1^{k(q-1)} \bmod p \equiv 1 \bmod p$
- Using CRT,
  - $x^{k\Phi(n)} = (0 \times p \times (p^{-1} \bmod q) + 1 \times q \times (q^{-1} \bmod p)) \bmod n$
  - $x^{k\Phi(n)} = (q \times (q^{-1} \bmod p)) \bmod n = (1 + k_2 p) \bmod n$
  - $x \cdot x^{k\Phi(n)} = k_1 q \times (1 + k_2 p) \bmod n = x + k_1 k_2 pq \bmod n$

Computational Aspects

- Problem:
  - finding two large primes (> 2048 bits at least)
- Method:
  - pick a large integer and apply a primality test, which does not require factorization.
- Miller-Rabin Algorithm for primality testing
  - Input:n
  - Output:
    - "n is composite" $\rightarrow 100\%$ assurance
    - "n is probably prime" $\rightarrow$ prime with probability > 0.75
- Idea: Use this algorithm many times to get comfortable level of confidence about the primeness.

M-R Test: Method

- We repeat the TEST
- If, at any point, the TEST returns "composite", then n is determined to be nonprime.
- If the TEST returns "inconclusive" t times, then the probability that n is prime is at least $(1 - 4^{-t})$

Distribution of Primes 1/2

- Concern
  - how many integers are likely to be rejected before a prime number is found using a primality test.
- Prime Number Theorem: Let $\pi(x)$ be the # of primes less than x. Then
$$\pi(x) \to x/\ln x$$
  - the primes near x are spaced, on average, one every $(\ln x)$.
- Then, on average, one would have to test about (on the order of) $\ln x$ integers before a prime is found.
- **Example:** $n = 2^{256}$, then the percentage of primes smaller than n is $\frac{n/\ln(n)}{n} = 0.56\%$
- **Example:** $n = 2^{1024}$, then the percentage of primes smaller than n is $\frac{n/\ln(n)}{n} = 0.14\%$

Distribution of Primes 2/2

- Because of all even integers and all integers ending with digit 5 can be immediately rejected,
  - the exact number of the trials is $0.4 \times \ln(x)$.
- For 200-bit prime, the trial number on the average is
  - $0.4 \times \ln(2^{200}) \approx 55$
- For 512-bit prime, average trial number
  - $0.4 \times \ln(2^{512}) \approx 142$

Security of RSA 1/2

- Brute force attack
  - Given $y = x^e \bmod n$, try all possible keys d;
    - $0 < d < \Phi(n)$ to obtain $x = y^d \bmod n$.
  - In practice, the key space
    - $|K| = \Phi(n) \approx n > 2^t$ it is impossible apply brute force for even moderate values of t.
- Finding $\Phi(n)$
  - Given $n, e, y = x^e \bmod n$, find $\Phi(n)$ and compute $d = e^{-1} \bmod \Phi(n)$.
  - Computing $\Phi(n)$ is believed to be as difficult as factoring n.

Security of RSA 2/2

- Factoring n
  - Given $n, e, y = x^e \bmod n$, find p and q such that $n = p \cdot q$ and compute
  - $\Phi(n) = (p-1) \cdot (q-1)$
  - $d = e^{-1} \bmod \Phi(n)$
  - $x = y^d \bmod n$
  - Factoring n is the only practical approach.
  - We need efficient integer factorization algorithms

Factoring Algorithms

- Quadratic Sieve (QS): speed depends on the size of the modulus n. In 1994, RSA129 challenge (RSA with modulus of 129 digit (≈ 428 bits)) is broken by QS
- Elliptic Curve Method: speed depends on the size of the smallest factor of n, i.e. p or q.
- Number Field Sieve: Asymptotically better than QS. In 1999, RSA140 challenge (RSA with modulus of 140 digit (≈ 465 bits)) is broken by generalized number field sieve.

Factoring Algorithms

- The computational complexity of factoring algorithms

Algorithm	Complexity
Quadratic Sieve	$O(e^{(1+o(1))\sqrt{\ln(n)\ln(\ln(n))}})$
Elliptic Curve	$O(e^{(1+o(1))\sqrt{2\ln(p)\ln(\ln(p))}})$
Number Field Sieve	$O(e^{(1.92+o(1))(\ln(n))^{1/3}(\ln(\ln(p)))^{2/3}})$

Largest Number Factored So Far

- RSA-768 (768-bit modulus)
  - December 12, 2009 by T. Kleinjung, K. Aoki, J. Franke, A. K. Lenstra, E. Thomé, P. Gaudry, A. Kruppa, P. Montgomery, J. W. Bos, D. A. Osvik, H. te Riele, A. Timofeev, and P. Zimmermann
  - Method: NFS
  - The sieving effort is estimated to have taken the equivalent of 1500 years on a single 2.2 GHz Opteron CPU. (2000 years in total)
  - 2^{67} instructions were carried out.
  - http://www.crypto-world.com/announcements/rsa768.txt
  - http://en.wikipedia.org/wiki/RSA-768#RSA-768
  - http://en.wikipedia.org/wiki/RSA_Factoring_Challenge

RSA Challenges

Challenge no	Approx. # of bits	Date	Prize
RSA-576	576	Dec 3, 2002	US$10.000
RSA-640	640	Nov 2, 2005	US$20.000
RSA-704	704	Jul 2, 2012	US$30.000
RSA-768	768	Dec 11, 2009	US$50.000
RSA-896	896	Not Yet	US$75.000
RSA-1024	1024	Not Yet	US$100.000

RSA is not semantically secure

- Eve picks two arbitrary mesages $x_0, x_1 < n$, $x_0 \neq x_1$
- Eve is challenged to guess a uniformly randomly chosen $b \in \{0, 1\}$
  - Given $c = x_b^e \pmod{n}$
- Can she guess b correctly?
- As RSA is deterministic, yes she can
  - she computes $c_0 = x_0^e \pmod{n}$
  - if $c_0 = c_b$ then $b = 0$ else $b = 1$
- RSA in practice is probabilistic

Optimal Asymmetric Encryption Padding → RSA-OAEP

$$R \leftarrow \{0,1\}^{k_0}$$

2^{-k_0} and 2^{-k_1} are negligible

$$G : \{0,1\}^{k_0} \rightarrow \{0,1\}^{k-k_0}$$

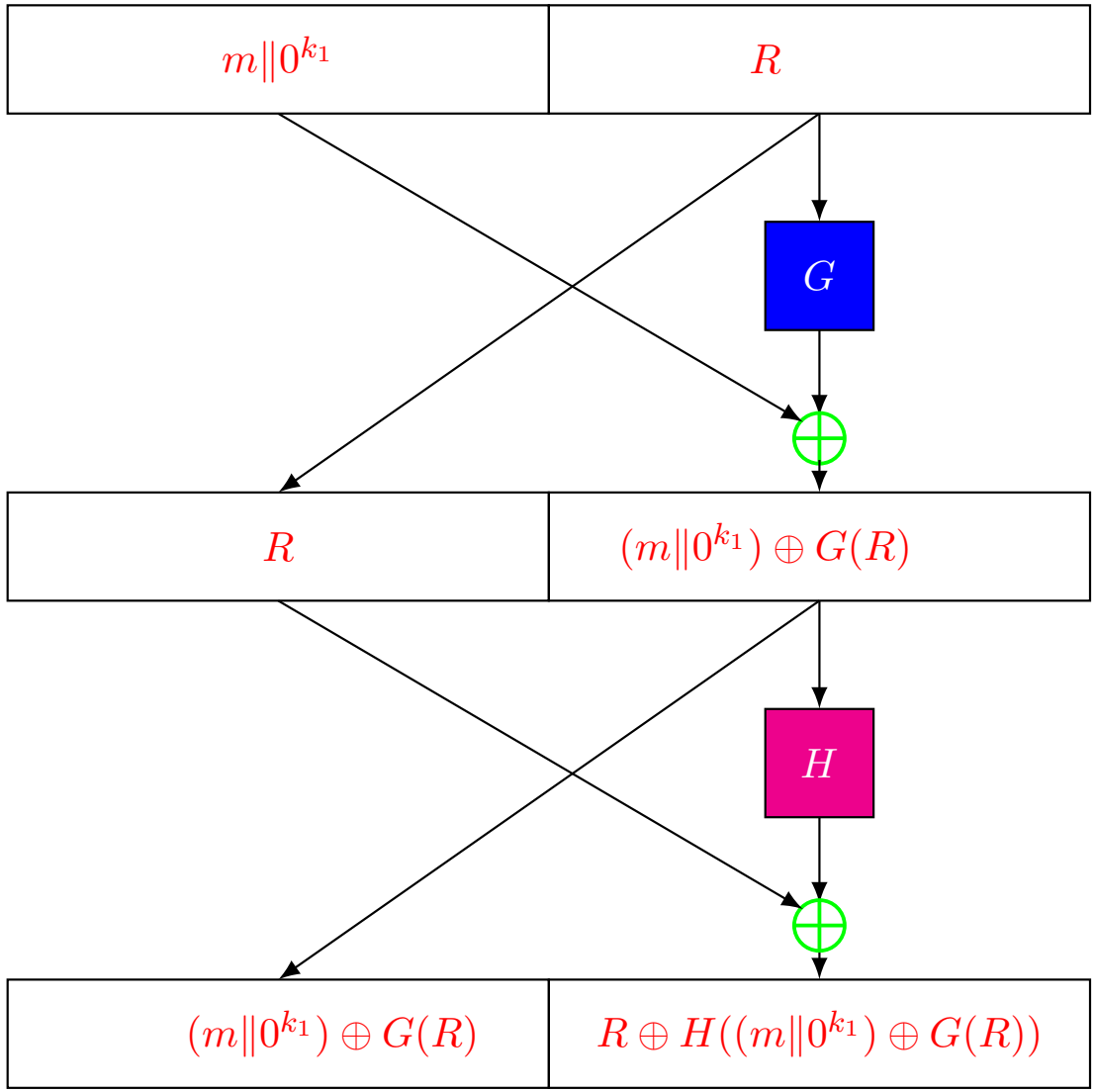
$$H : \{0,1\}^{k-k_0} \rightarrow \{0,1\}^{k_0}$$

$$k = \lceil \log_2 n \rceil$$

Two rounds

Feistel system

$$y = \tilde{m}^e \pmod{p} \qquad \tilde{m}$$



Side-Channel Attacks

- Basic RSA operation
  - modular exponentiation
- The binary left-to-right exponentiation algorithm

The binary left-to-right exponentiation algorithm

Input: $y, d = (d_{k-1}, d_{k-1}, \ldots, d_1, d_0), n, k = \lceil \log_2 n \rceil$
Output: $y^d \bmod n$

1: $s := y$
2: **for** $i = k - 2$ `downto 0` **do**
3: $\quad s := s \times s \bmod n$
4: $\quad$ **if** $d_i = 1$ **then**
5: $\quad\quad s := s \times y \bmod n$
6: $\quad$ **end if**
7: **end for**
8: **return** s

Example: RSA Decryption

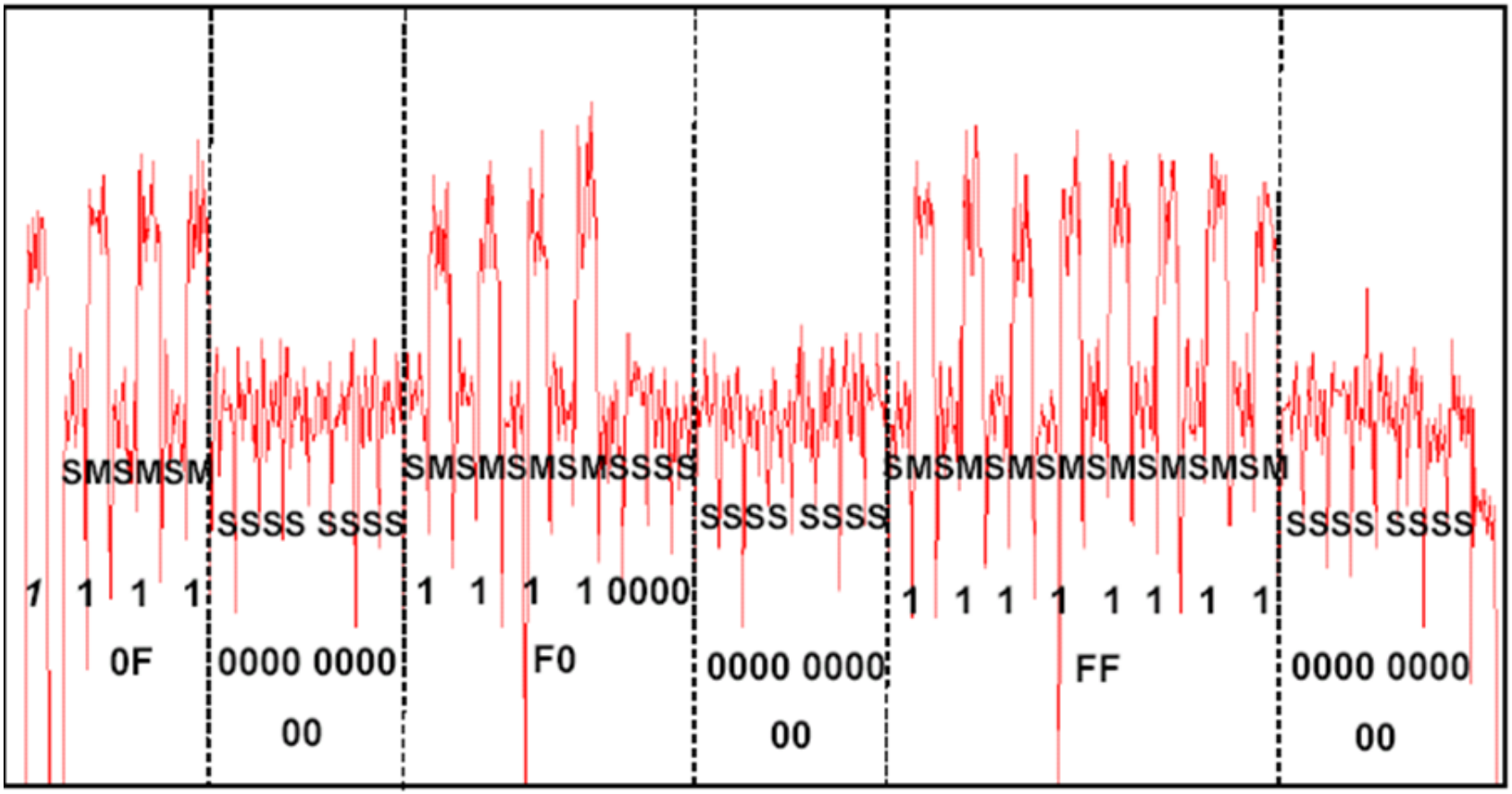
- $7^{560} \bmod 561$
- $d = (1000110000)$

Iteration	0	1	2	3	4	5	6	7	8	9
Exponent bits	1	0	0	0	1	1	0	0	0	0
Square	7	49	157	526	103	355	298	166	67	1
Multiply	7	-	-	-	160	241	-	-	-	-

- Assume that an adversary can observe the decryption of a ciphertext (or a signature) and record the power consumption
  - $y^d \bmod n$
- Attack Scenario: a smart card that relies on an external power
  - Power supplied by the reader

Simple Power Analysis (SPA) 1/2

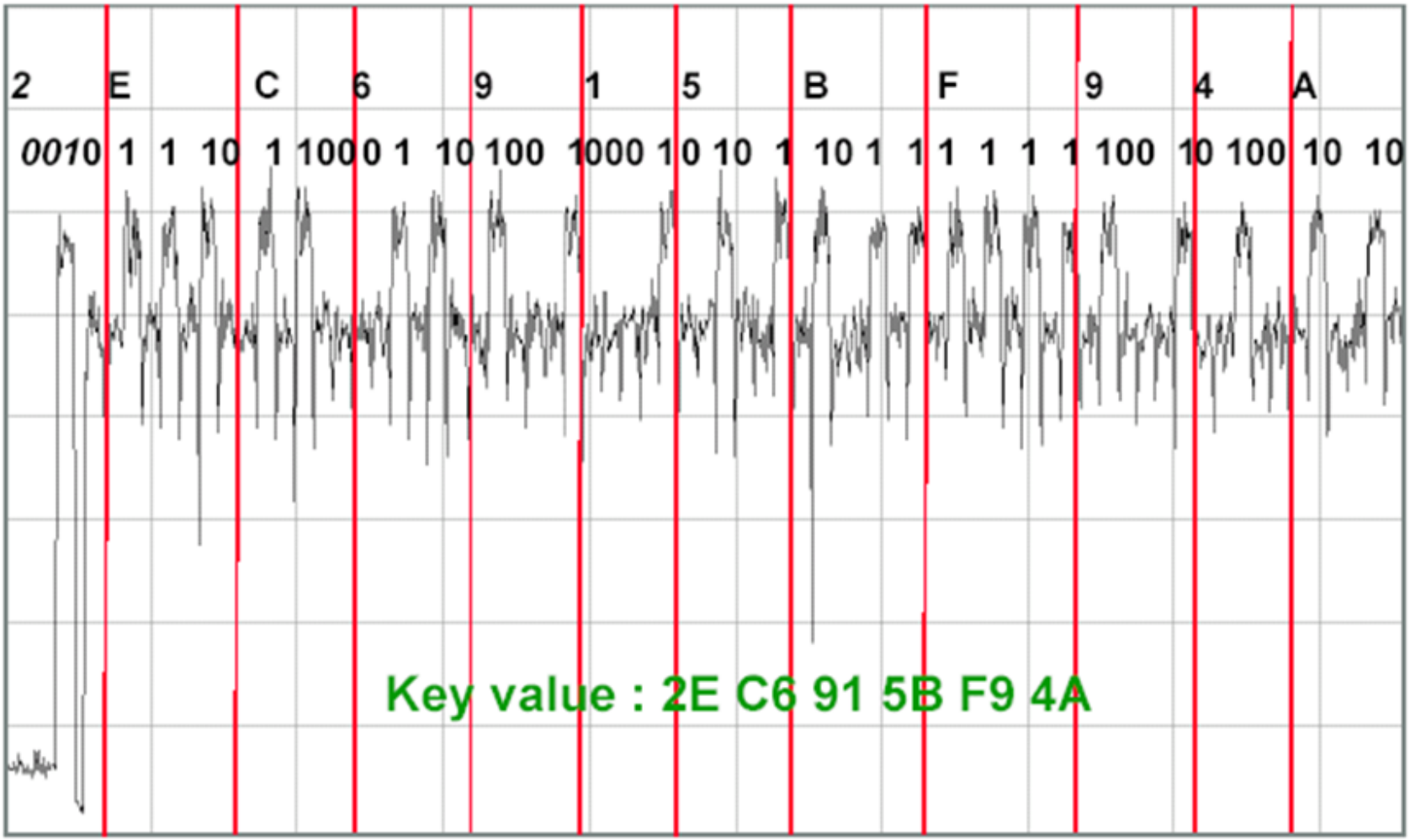
- private key: d = 0F 00 F0 00 FF 00



Power Spectrum

Simple Power Analysis (SPA) 2/2

- private key: d = 2E C6 91 5B F9 4A



Countermeasure Against SPA

- Double-and-Add Always Algorithm

Algorithm 1 Double-and-Add Always Algorithm

Input: $y, d = (d_{k-1}, d_{k-1}, \ldots, d_1, d_0), n, k = \lceil \log_2 n \rceil$
Output: $y^d \bmod n$

1: $s_0 := y, s_1 := 1$
2: **for** $i = k - 2$ `downto 0` **do**
3: $\quad s_0 := s_0 \times s_0 \bmod n$
4: $\quad s_1 := s_0 \times y \bmod n$
5: $\quad b := d_i$
6: $\quad s_0 := s_b$
7: **end for**
8: **return** s