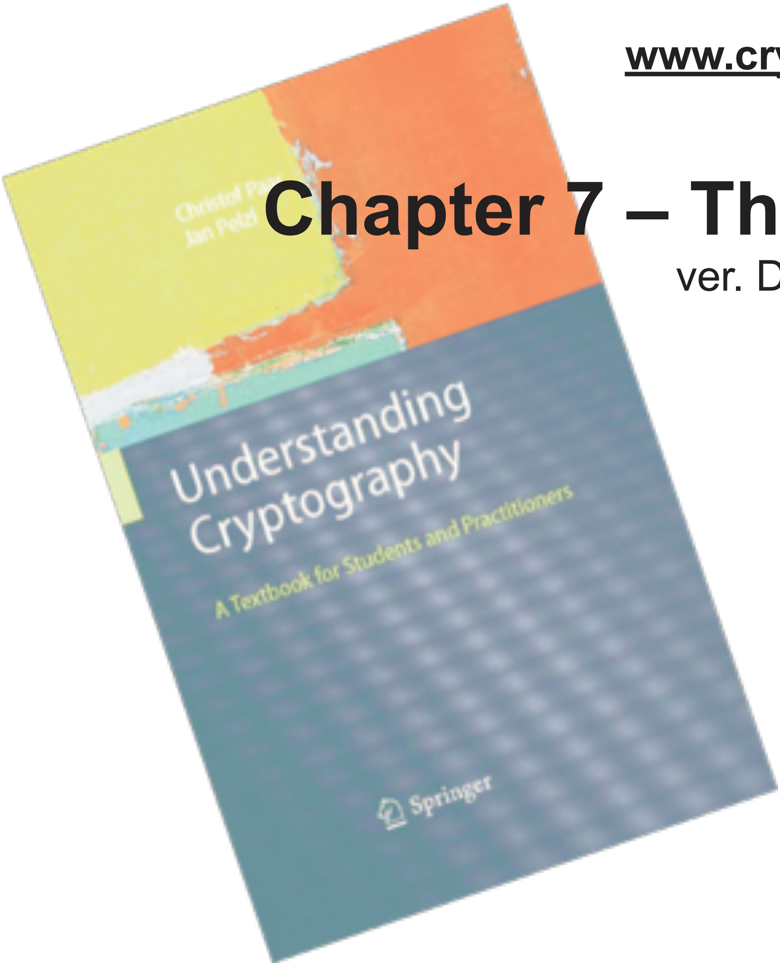
Understanding Cryptography – A Textbook for Students and Practitioners

by Christof Paar and Jan Pelzl

www.crypto-textbook.com



Chapter 7 – The RSA Cryptosystem

ver. December 7, 2010

These slides were prepared by Benedikt Driessen, Christof Paar and Jan Pelzl and modified by Sam Bowne

Some legal stuff (sorry): Terms of use

- The slides can used free of charge. All copyrights for the slides remain with Christof Paar and Jan Pelzl.
- The title of the accompanying book “Understanding Cryptography” by Springer and the author’s names must remain on each slide.
- If the slides are modified, appropriate credits to the book authors and the book title must remain within the slides.
- It is not permitted to reproduce parts or all of the slides in printed form whatsoever without written consent by the authors.

Content of this Chapter

- The RSA Cryptosystem
- Implementation aspects
- Finding Large Primes
- Attacks and Countermeasures
- Lessons Learned

Content of this Chapter

- **The RSA Cryptosystem**
- Implementation aspects
- Finding Large Primes
- Attacks and Countermeasures
- Lessons Learned

The RSA Cryptosystem

- Martin Hellman and Whitfield Diffie published their landmark public-key paper in 1976
- Ronald Rivest, Adi Shamir and Leonard Adleman proposed the asymmetric RSA cryptosystem in1977
- RSA is the most widely used asymmetric cryptosystem although elliptic curve cryptography (ECC) is becoming increasingly popular
- RSA is mainly used for two applications
  - Transport of (i.e., symmetric) keys (cf. Chptr 13 of *Understanding Cryptography*)
  - Digital signatures (cf. Chptr 10 of *Understanding Cryptography*)

Encryption and Decryption

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

$$y = e_{k_{pub}}(x) \equiv x^e \bmod n \tag{7.1}$$

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

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

$$x = d_{k_{pr}}(y) \equiv y^d \bmod n \tag{7.2}$$

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

Key Generation

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 $e \in \{1, 2, \ldots, \Phi(n) - 1\}$ such that

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

5. Compute the private key d such that

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

RSA Encryption in Python

```
>>> from Crypto.PublicKey import RSA
>>> key = RSA.generate(2048)
>>> publickey = key.publickey()
>>> plain = 'encrypt this message'
>>> ciphertext = publickey.encrypt(plain, 0)[0]
>>> print ciphertext.encode("hex")
536eda071ab9e526442f2b56e71fa5abfc603c88c2eac03d91f22bab6d0ea14bab2e8c8247df477c
5f15ce3ccc551227799d1f4f8943fa8bd278639bd90292c5799d11f9f6601c94d88f10fc314317fb
1d75f55e20d1c5dd4e7448ff39018dab44091b6664610657516bfaf95a3f0e63e9194f1e08343421
f7cf8c35550ed951b240e4c42f94b8bfc73ec3ccd519f7c489c28aaf799c78d6a695707423f72c05
4edfd8f4c2ac0f5c25a996647b8958f160983db8bdf2214fe131b0f3d558aeb7560e67f0621f0224
fd21f18034eebb9c8773e6310f80975539765d7235235a446f037179e94e504b21f9ffac6679570a
95848f238cdd3243723ed4722e549498
```

RSA Decryption in Python

```
>>> decrypted = key.decrypt(ciphertext)
>>> print decrypted
encrypt this message
```

Speed of Calculations

```
LENGTH:  1024
0.150621891022  sec. for one RSA key generation
0.026349067688  sec. for 400 RSA encryptions
0.0133030414581  sec. for 5 RSA decryptions
```

- Encryption is **fastest**
- Decryption is **much slower**
- Key generation is **slowest**

Key Generation

- Like all asymmetric schemes, RSA has set-up phase during which the private and public keys are computed

Remarks:

- Choosing two large, distinct primes *p, q* (in Step 1) is non-trivial
- gcd$(e, \Phi(n)) = 1$ ensures that *e* has an inverse and, thus, that there is always a private key *d*

Example: RSA with small numbers

ALICE

Message $x = 4$

BOB

1. Choose $p = 3$ and $q = 11$
2. Compute $n = p * q = 33$ 3.

$$\Phi(n) = (3-1) * (11-1) = 20$$

4. Choose $e = 3$
5. $d \equiv e^{-1} \equiv 7 \bmod 20$

$K_{pub} = (33,3)$ ←

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

$y = 31$ →

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

Content of this Chapter

- The RSA Cryptosystem
- **Implementation aspects**
- Finding Large Primes
- Attacks and Countermeasures
- Lessons Learned

Implementation aspects

- The RSA cryptosystem uses only one arithmetic operation (modular exponentiation) which makes it conceptually a simple asymmetric scheme
- Even though conceptually simple, due to the use of very long numbers, RSA is orders of magnitude slower than symmetric schemes, e.g., DES, AES
- When implementing RSA (esp. on a constrained device such as smartcards or cell phones) close attention has to be paid to the correct choice of arithmetic algorithms
- The square-and-multiply algorithm allows fast exponentiation, even with very long numbers…

Square-and-Multiply

$$y = e_{k_{pub}}(x) \equiv x^e \bmod n \quad \text{(encryption)}$$
$$x = d_{k_{pr}}(y) \equiv y^d \bmod n \quad \text{(decryption)}$$

- Consider RSA with a 1024-bit key
- We need to calculate x^e where e is 1024 bits long
- x * x * x * x .... 2^{1024} multiplications
- Competely impossible -- we can't even crack a 72-bit key yet (2^{72} calculations)

Square-and-Multiply

- Use memory to save time
- Do these ten multiplications
  - x2 = x * x
  - x4 = x2 * x2
  - x8 = x4 * x4
  - x16 = x8 * x8
  - ...
  - x1024 = x512 * x512
  - ...
- Combine the results to make any exponent

Square-and-Multiply

- With this trick, a 1024-bit exponent can be calculated with only 1536 multiplications
- But each number being multiplied is 1024 bits long, so it still takes a lot of CPU

Speed-Up Techniques

- Modular exponentiation is computationally intensive
- Even with the square-and-multiply algorithm, RSA can be quite slow on constrained devices such as smart cards
- Some important tricks:
    - Short public exponent e
    - Chinese Remainder Theorem (CRT)
    - Exponentiation with pre-computation *(not covered here)*

Fast encryption with small public exponent

- Choosing a small public exponent *e* does not weaken the security of RSA
- A small public exponent improves the speed of the RSA encryption significantly

Public Key	**e as binary string**	**#MUL + #SQ**
$2^1+1 = 3$	$(11)_2$	$1 + 1 = 2$
$2^4+1 = 17$	$(1\ 0001)_2$	$4 + 1 = 5$
$2^{16} + 1$	$(1\ 0000\ 0000\ 0000\ 0001)_2$	$16 + 1 = 17$

- This is a commonly used trick (e.g., SSL/TLS, etc.) and makes RSA the fastest asymmetric scheme with regard to encryption!

Fast decryption with CRT

- Choosing a small private key *d* results in security weaknesses!
    - In fact, d must have at least *0.3t* bits, where *t* is the bit length of the modulus *n*
- However, the Chinese Remainder Theorem (CRT) can be used to (somewhat) accelerate exponentiation with the private key *d*
- **It gets 4 times faster**

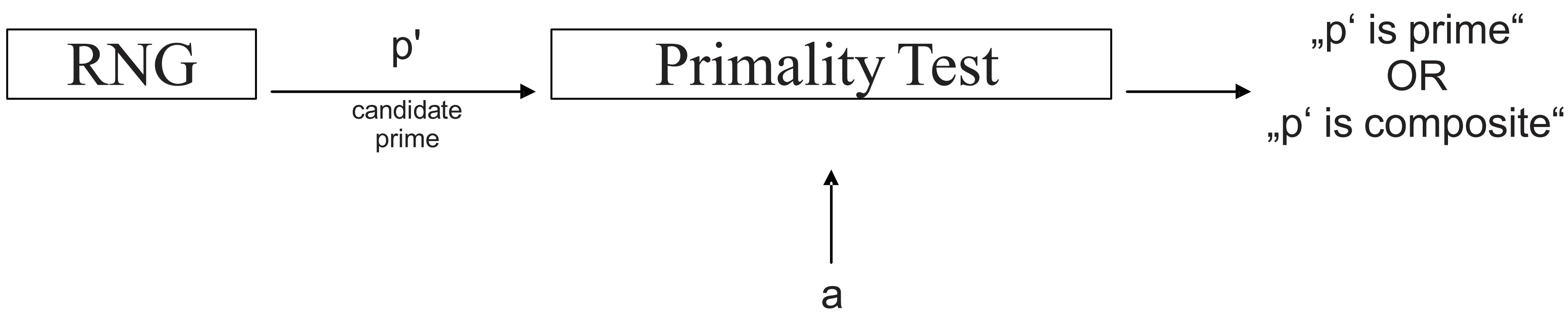
Kahoot!

Content of this Chapter

- The RSA Cryptosystem
- Implementation aspects
- **Finding Large Primes**
- Attacks and Countermeasures
- Lessons Learned

Finding Large Primes

- Generating keys for RSA requires finding two large primes *p* and *q* such that *n* = *p* * *q* is sufficiently large
- The size of *p* and *q* is typically half the size of the desired size of *n*
- To find primes, random integers are generated and tested for primality:



- The random number generator (RNG) should be non-predictable otherwise an attacker could guess the factorization of *n*

How Common Are Primes?

$$P(\tilde{p} \text{ is prime}) \approx \frac{2}{\ln(\tilde{p})}$$

- For a 1024-bit key, p and q will be around 512 bits long
- So the density of primes near p and q will be

$$P(\tilde{p} \text{ is prime}) \approx \frac{2}{\ln(2^{512})} = \frac{2}{512\ln(2)} \approx \frac{1}{177}$$

- So guessing a few hundred times should be enough

Primality Tests

- Factoring *p* and *q* to test for primality is typically not feasible
- However, we are not interested in the factorization, we only want to know whether *p* and *q* are composite
- Typical primality tests are probabilistic, i.e., they are not 100% accurate but their output is correct with very high probability
- A probabilistic test has two outputs:
  - „p‘ is composite“ – always true
  - „p‘ is a prime“ – only true with a certain probability
- Among the well-known primality tests are the following
  - Fermat Primality-Test
  - Miller-Rabin Primality-Test

Miller–Rabin Primality Test

Input: prime candidate $\tilde{p}$ with $\tilde{p}-1=2^u r$ and security parameter s

Output: statement "$\tilde{p}$ is composite" or "$\tilde{p}$ is likely prime"

Algorithm:

1 FOR $i=1$ TO s
    choose random $a \in \{2,3,\ldots,\tilde{p}-2\}$
1.2 $z \equiv a^r \bmod \tilde{p}$
1.3   IF $z \not\equiv 1$ and $z \not\equiv \tilde{p}-1$
1.4   FOR $j=1$ TO $u-1$
            $z \equiv z^2 \bmod \tilde{p}$
            IF $z=1$
                RETURN ("$\tilde{p}$ is composite")
1.5   IF $z \neq \tilde{p}-1$
            RETURN ("$\tilde{p}$ is composite")
2 RETURN ("$\tilde{p}$ is likely prime")

Number of Tests Required

Table 7.2 Number of runs within the Miller–Rabin primality test for an error probability of less than 2^{-80}

Bit lengths of $\tilde{p}$	Security parameter s
250	11
300	9
400	6
500	5
600	3

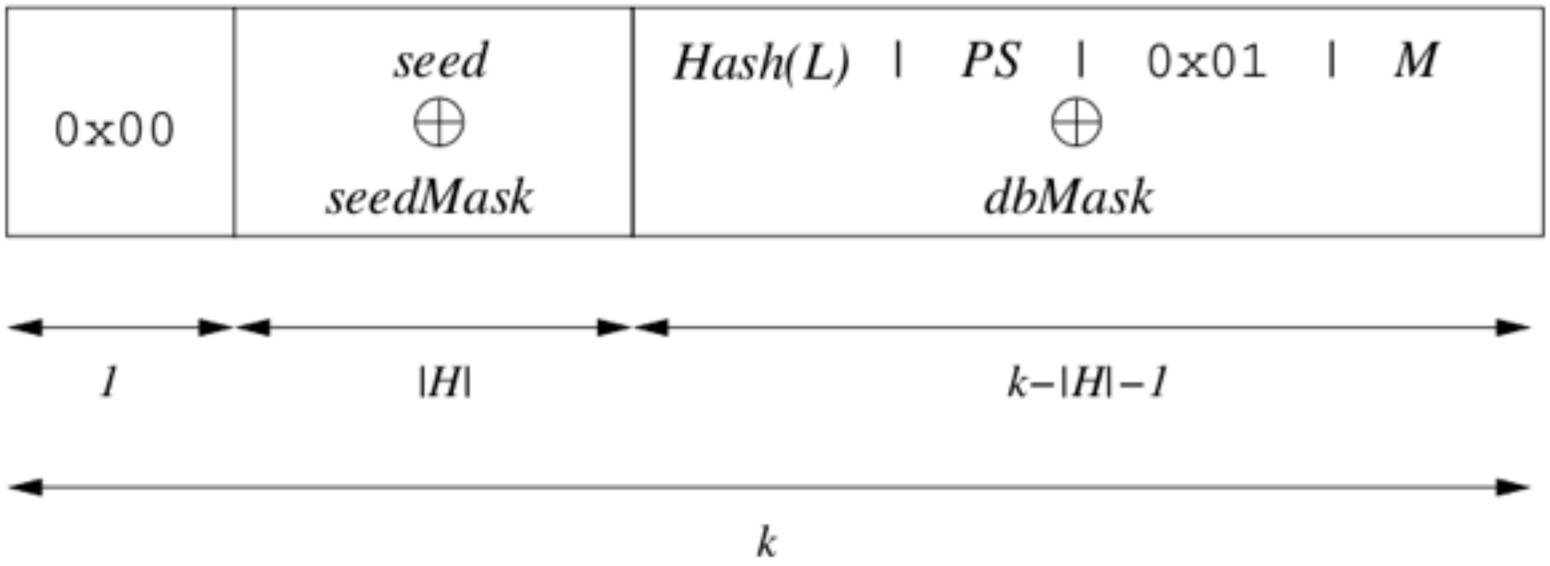
RSA in Practice: Padding

- Problems with "scholbook RSA"
  1. RSA encryption is **deterministic**
     - Repeated plaintext results in repeated ciphertext
  2. Paintext x=0, x=1, or x=-1 produce ciphertext y=0, y=1, or y=-1
  3. RSA is **malleable**
     - Multiplying ciphertext by an integer without decrypting it can lead to readable plaintext
     - Could be used to change the amount of a transaction
     - Replace **y** with $\mathbf{s^e * y}$

$$(s^e y)^d \equiv s^{ed} x^{ed} \equiv s\,x \bmod n$$

PKCS#1 (v2.1) Padding

- Put 0, a "MaskedSeed", a Hash, 1, and more zeroes before the message M
- Total padded length = same as **n**
    - e.g. 1024 or 2048 bits



PKCS#1 (v2.1) Padding

- When decrypting, verify structure of the message
- This removes these weaknesses in RSA:
  1. **Deterministic**
  2. **1**, **0**, and **-1**
  3. **Malleable**

Content of this Chapter

- The RSA Cryptosystem
- Implementation aspects
- Finding Large Primes
- **Attacks and Countermeasures**
- Lessons Learned

Attacks and Countermeasures 1/3

There are three general attack families against RSA:

1. Protocol attacks
2. Mathematical attacks
3. Side-channel attacks

Protocol attacks

- Exploit the **malleability** of RSA, i.e., the property that a ciphertext can be transformed into another ciphertext which decrypts to a related plaintext – without knowing the private key
- Can be prevented by proper padding

Mathematical attacks

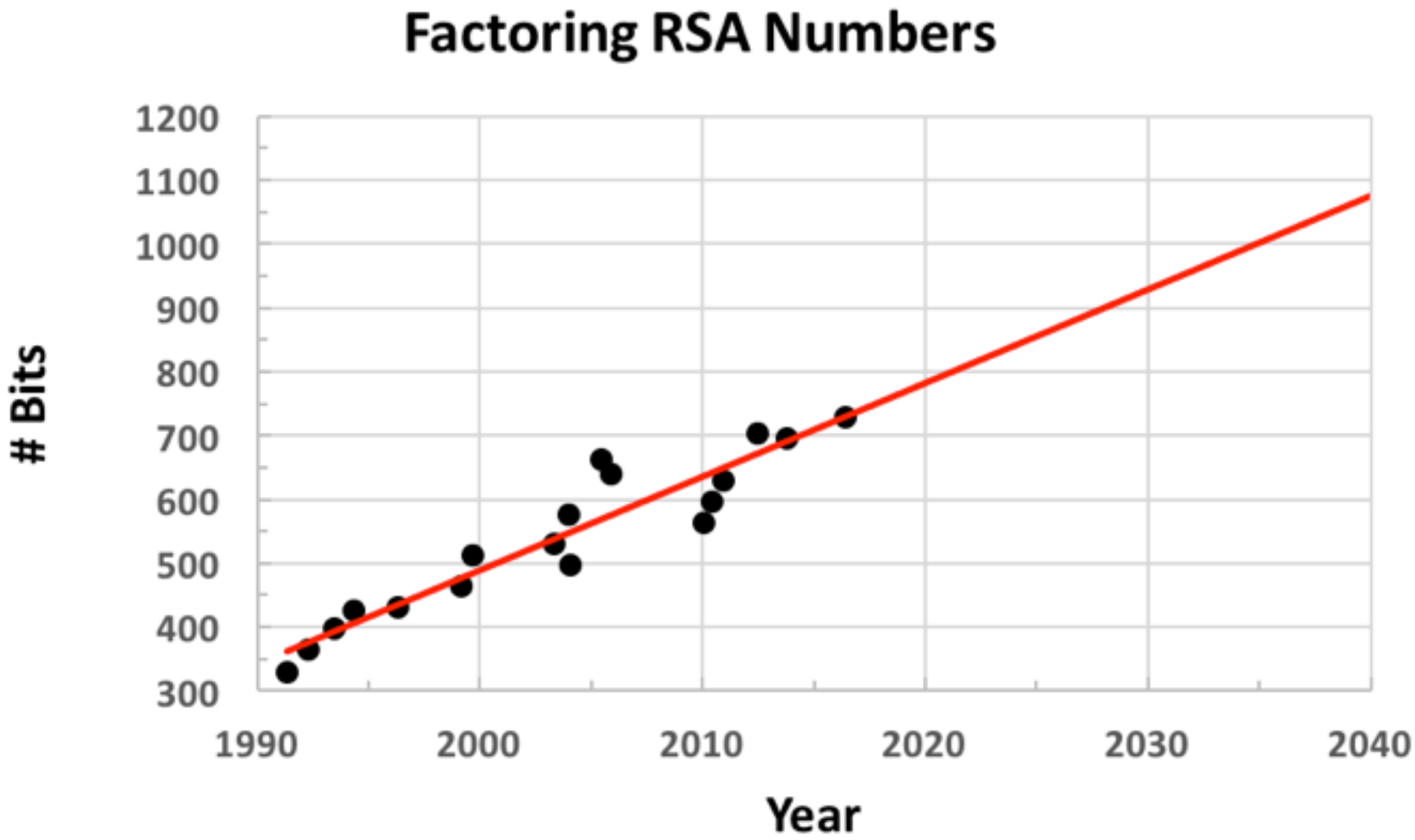
- The best known attack is **factoring** of ***n*** into ***p*** and ***q***
  - Attacker can then decrypt the message

$$\Phi(n) = (p-1)(q-1)$$
$$d^{-1} \equiv e \bmod \Phi(n)$$
$$x \equiv y^d \bmod n.$$

- Can be prevented using a sufficiently large modulus ***n***
- Current record: **729 bits** factored in 2016
  - Link Ch 7a

RSA Numbers

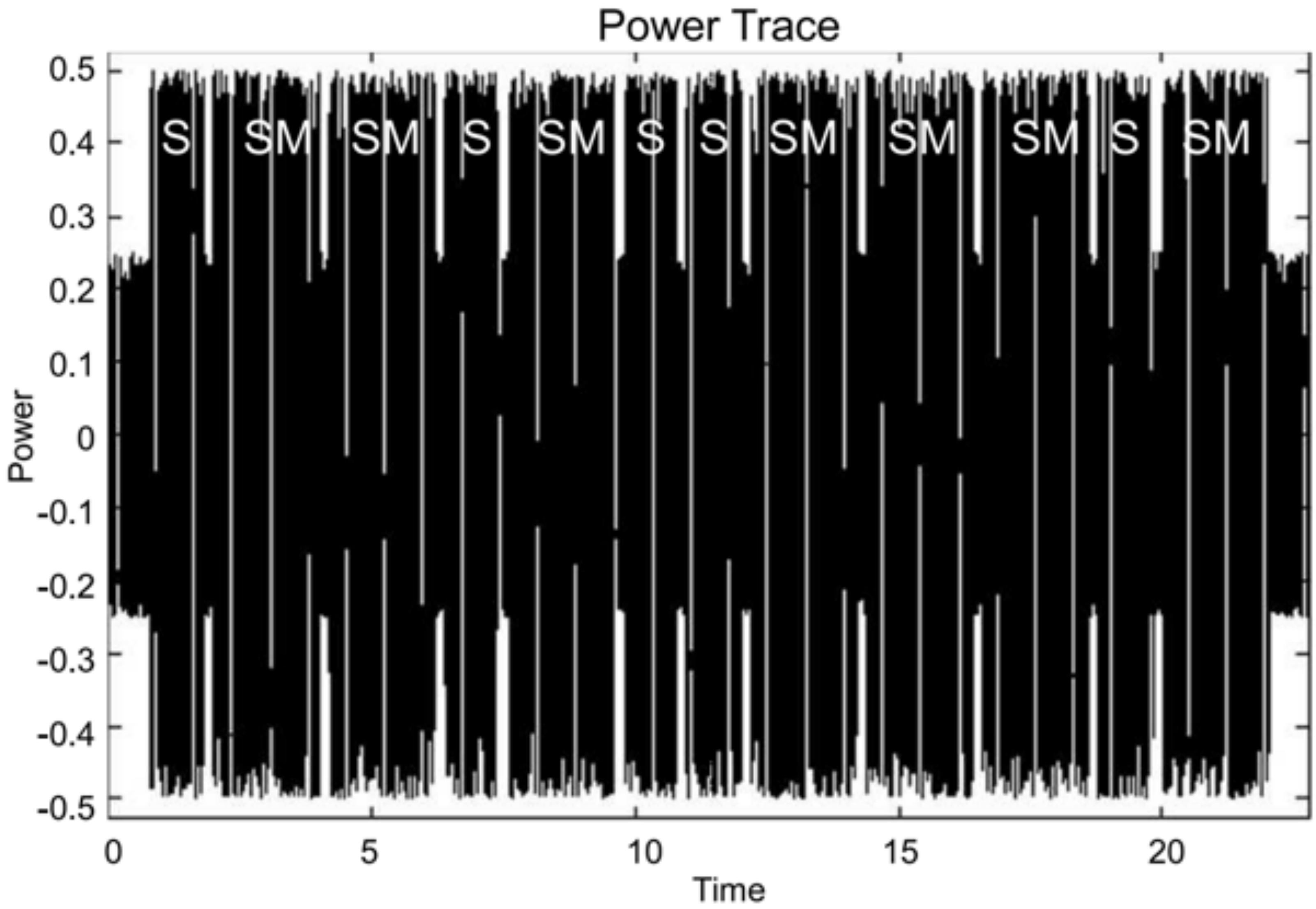
- A challenge to test security of RSA encryption



Side-Channel Attacks

- Exploit physical leakage of RSA implementation (e.g., power consumption, EM emanation, etc.)
- Ex: **Power Consumption**
    - Square and Multiply operations take a lot of power
    - Two bursts of power consumption: key bit is 1
    - One burst of power consumption: key bit is 0

operations:	*S*	*SM*	*SM*	*S*	*SM*	*S*	*S*	*SM*	*SM*	*SM*	*S*	*SM*
private key:	0	1	1	0	1	0	0	1	1	1	0	1

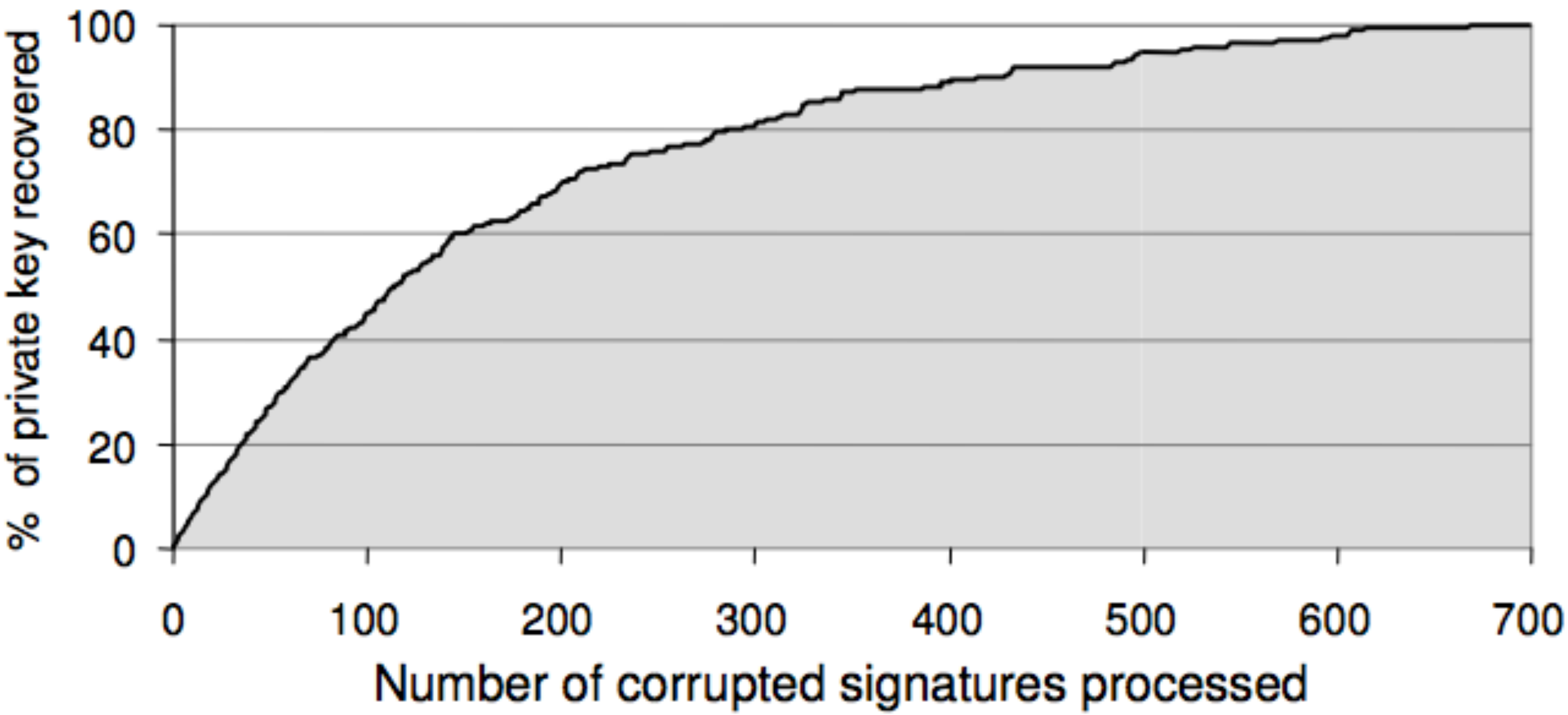


Power Consumption

- Countermeasure:
  - Perform a dummy multiplication operation for each 0 bit
  - So the power consumption remains the same

Fault-Injection Attacks

- Inducing faults in the device while decryption is executed can lead to a complete leakage of the private key
- In 2010, researchers extracted a 1024-bit key in 24 hours
  - Links Ch 7b, 7c



Content of this Chapter

- The RSA Cryptosystem
- Implementation aspects
- Finding Large Primes
- Attacks and Countermeasures
- **Lessons Learned**

Lessons Learned

- RSA is the most widely used public-key cryptosystem
- RSA is mainly used for key transport and digital signatures
- The public key *e* can be a short integer, the private key *d* needs to have the full length of the modulus *n*
- RSA relies on the fact that it is hard to factorize *n*
- Currently 1024-bit cannot be factored, but progress in factorization could bring this into reach within 10-15 years. Hence, RSA with a 2048 or 3076 bit modulus should be used for long-term security
- A naïve implementation of RSA allows several attacks, and in practice RSA should be used together with padding

Kahoot!