CP 460 - Applied Cryptography

The RSA Cryptosystem

Department of Physics and Computer Science
Faculty of Science, Waterloo

Abbas Yazdinejad, Ph.D.

Fall 2024

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
- Until now, RSA is the most widely use asymmetric cryptosystem although elliptic curve cryptography (ECC) becomes 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 operations are done over the integer ring Z_n (i.e., arithmetic modulo n), where $n = p * q$, with p, q being large primes
- Encryption and decryption are simply exponentiations in the ring

Definition

Given the public key $(n,e) = k_{pub}$ and the private key $d = k_{pr}$ we write

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

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

where $x, y \in Z_n$.

We call $e_{k_{pub}}()$ the encryption and $d_{k_{pr}}()$ the decryption operation.

- In practice x, y, n and d are very long integer numbers (≥ 1024 bits)
- The security of the scheme relies on the fact that it is hard to derive the „private exponent" d given the public-key (n, e)

Key Generation

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

<u>Algorithm: RSA Key Generation</u>

Output: public key: $k_{pub} = (n, e)$ and private key $k_{pr} = d$

1. Choose two large primes p, q
2. Compute $n = p * q$
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 $d * e \equiv 1 \bmod \Phi(n)$
6. **RETURN** $k_{pub} = (n, e), k_{pr} = d$

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. $d \equiv e^{-1} \equiv 7 \bmod 20$

$K_{pub} = (33,3)$

$\longleftarrow$

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

$y = 31$

$\longrightarrow$

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

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

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

- **Basic principle**: Scan exponent bits from left to right and square/multiply operand accordingly

Algorithm: Square-and-Multiply for x^H *mod n*

Input: Exponent H, base element x, Modulus n

Output: $y = x^H \bmod n$

1. Determine binary representation $H = (h_t, h_{t-1}, \ldots, h_0)_2$
2. **FOR** $i = t-1$ **TO** 0
3. $\quad y = y^2 \bmod n$
4. $\quad$ **IF** $h_i = 1$ **THEN**
5. $\quad\quad y = y * x \bmod n$
6. **RETURN** y

This algorithm is used to perform fast exponentiation by breaking the problem into a series of squaring and multiplication steps.

- Rule: Square in every iteration (Step 3) and multiply current result by x if the exponent bit $h_i = 1$ *(Step 5)*
- Modulo reduction after each step keeps the operand y small

Example: Square-and-Multiply

- Computes x^{26} without modulo reduction
- Binary representation of exponent: $26 = (1,1,0,1,0)_2 = (h_4,h_3,h_2,h_1,h_0)_2$

Step		Binary exponent	Op	Comment
1	$x = x^1$	$(1)_2$		Initial setting, h_4 processed
1a	$(x^1)^2 = x^2$	$(10)_2$	SQ	Processing h_3
1b	$x^2 * x = x^3$	$(11)_2$	MUL	$h_3 = 1$
2a	$(x^3)^2 = x^6$	$(110)_2$	SQ	Processing h_2
2b	-	$(110)_2$	-	$h_0 = 0$
3a	$(x^6)^2 = x^{12}$	$(1100)_2$	SQ	Processing h_1
3b	$x^{12} * x = x^{13}$	$(1101)_2$	MUL	$h_1=1$
4a	$(x^{13})^2 = x^{26}$	$(11010)_2$	SQ	Processing h_0
4b	-	$(11010)_2$	-	$h_0 = 0$

- Observe how the exponent evolves into $x^{26} = x^{11010}$

Complexity of Square-and-Multiply Alg.

- The square-and-multiply algorithm has a logarithmic complexity, i.e., its run time is proportional to the bit length (rather than the absolute value) of the exponent
- Given an exponent with t+1 bits

  $$H = (h_t, h_{t-1}, ..., h_0)_2$$

  with $h_t = 1$, we need the following operations
  - *# Squarings* $= t$
  - Average # multiplications $= 0.5\, t$
  - Total complexity: *#SQ + #MUL* $= 1.5\, t$
- Exponents are often randomly chosen, so *1.5 t* is a good estimate for the average number of operations
- Note that each squaring and each multiplication is an operation with very long numbers, e.g., 2048 bit integers.

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!

If you use e=3, calculating *m^3 mod n* involves fewer multiplications compared to a larger exponent. e=65537

A good balance between security and computational efficiency.

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*
- Based on the CRT we can replace the computation of

$$x^{d \bmod \Phi(n)} \bmod n$$

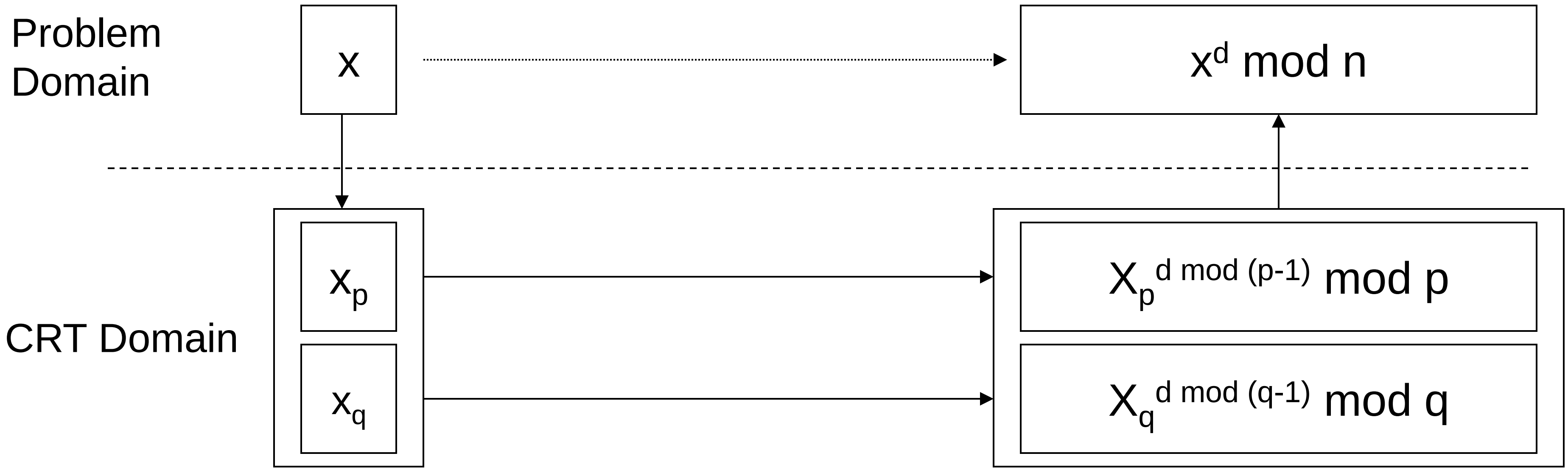
by two computations

$$x^{d \bmod (p-1)} \bmod p \quad \text{and} \quad x^{d \bmod (q-1)} \bmod q$$

where *q* and *p* are „small“ compared to *n*

The CRT can be applied to RSA to speed up the decryption process. Instead of performing modular exponentiation with a large modulus n, RSA can split the computation into two smaller computations using p and q (the prime factors of n), which are much smaller.

Basic principle of CRT-based exponentiation



- CRT involves three distinct steps

  (1) Transformation of operand into the CRT domain

  (2) Modular exponentiation in the CRT domain

  (3) Inverse transformation into the problem domain

- These steps are equivalent to one modular exponentiation in the problem domain

CRT: Step 1 – Transformation

- Transformation into the CRT domain requires the knowledge of *p* and *q*
- *p* and *q* are only known to the owner of the private key, hence CRT cannot be applied to speed up encryption
- The transformation computes (x_p, x_q) which is the representation of x in the CRT domain. They can be found easily by computing

$$x_p \equiv x \bmod p \qquad \text{and} \qquad x_q \equiv x \bmod q$$

CRT: Step 2 – Exponentiation

- Given d_p and d_q such that

$$d_p \equiv d \bmod (p-1) \quad \text{and} \quad d_q \equiv d \bmod (q-1)$$

one exponentiation in the problem domain requires two exponentiations in the CRT domain

$$y_p \equiv x_p^{d_p} \bmod p \quad \text{and} \quad y_q \equiv x_q^{d_q} \bmod q$$

- In practice, p and q are chosen to have half the bit length of n, i.e., $|p| \approx |q| \approx |n|/2$

This approach accelerates RSA decryption while maintaining the same level of security, as the operations are split into smaller, more efficient calculations involving p and q.

CRT: Step 3 – Inverse Transformation

- Inverse transformation requires modular inversion twice, which is computationally expensive

$$c_p \equiv q^{-1} \bmod p \qquad \text{and} \qquad c_q \equiv p^{-1} \bmod q$$

- Inverse transformation assembles y_p, y_q to the final result $y \bmod n$ in the problem domain

$$y \equiv [\, q * c_p \,] * y_p + [\, p * c_q \,] * y_q \bmod n$$

- The primes p and q typically change infrequently, therefore the cost of inversion can be neglected because the two expresssions

  $[\, q * c_p \,]$ and $[\, p * c_q \,]$

  can be precomputed and stored

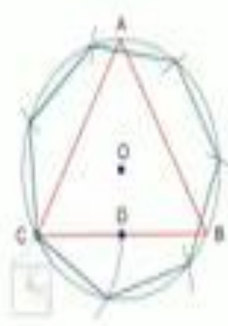
Complexity of CRT

- We ignore the transformation and inverse transformation steps since their costs can be neglected under reasonable assumptions
- Assuming that *n* has *t+1* bits, both *p* and *q* are about *t/2* bits long
- The complexity is determined by the two exponentiations in the CRT domain. The operands are only *t/2* bits long. For the exponentiations we use the square-and-multiply algorithm:
  - # squarings (one exp.): $\#SQ = 0.5\,t$
  - # aver. multiplications (one exp.): #MUL = 0.25t
  - Total complexity: $2 * (\#MUL + \#SQ) = 1.5t$
- This looks the same as regular exponentations, but since the operands have half the bit length compared to regular exponent., each operation (i.e., multipl. and squaring) is 4 times faster!
- Hence CRT is **4 times** faster than straightforward exponentiation

Detailed Example: Decrypting RSA Using the Chinese Remainder Algorithm

Jeff Suzuki

Math for Everyone
jeff.a.suzuki@gmail.com

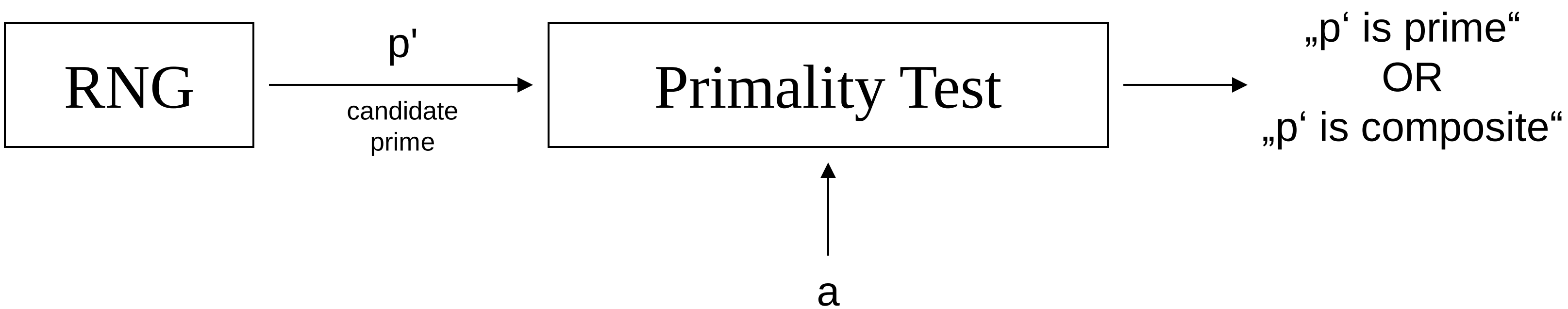


Content of this Chapter

- 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*

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

Fermat Primality-Test

- Basic idea: Fermat‘s Little Theorem holds for all primes, i.e., if a number *p‘* is found for which $a^{p'-1} \equiv 1 \bmod p'$, it is not a prime

<u>Algorithm: Fermat Primality-Test</u>

Input: Prime candidate *p‘*, security parameter *s*

Output: „*p‘* is composite“ or „*p‘* is likely a prime“

1. **FOR** *i = 1* **TO** *s*

2. choose random $a \in \{2,3, \ldots, p'-2\}$

3. **IF** $a^{p'-1} \equiv 1 \bmod p'$ **THEN**

4. **RETURN** „*p‘* is composite“

5. **RETURN** „*p‘* is likely a prime“

- For certain numbers („Carchimchael numbers“) this test returns „*p‘* is likely a prime“ often – although these numbers are composite. Therefore, the Miller-Rabin Test is preferred

Efficiency: It is a very fast test, especially for large numbers, as it involves modular exponentiation, which can be computed efficiently using algorithms like square-and-multiply.

Probabilistic Nature: The Fermat test can incorrectly identify a composite number as prime (these are called **Fermat liars**).

Theorem for Miller-Rabin‘s test

- The more powerful Miller-Rabin Test is based on the following theorem

Theorem

Given the decomposition of an odd prime candidate p'

$$p' - 1 = 2^u * r$$

where r is odd. If we can find an integer a such that

$$a^r \equiv 1 \bmod p' \qquad \text{and} \qquad a^{r2^j} \equiv p' - 1 \bmod p'$$

For all $j \in \{0,1, ..., u-1\}$, then p' is composite.

Otherwise it is probably a prime.

- This theorem can be turned into an algorithm

The **Miller-Rabin Primality Test** is also a probabilistic test, but it is considered more reliable than the Fermat test. It builds on ideas from Fermat's theorem but adds additional checks to avoid some of the weaknesses (such as Fermat liars).

Miller-Rabin Primality-Test

Algorithm: Miller-Rabin Primality-Test

Input: Prime candidate p' with $p'-1 = 2^{u} * r$ security parameter s

Output: „p' is composite" or „p' is likely a prime"

1. **FOR** $i = 1$ **TO** s

2. choose random $a \varepsilon \{2,3, ..., p'-2\}$

3. $z \equiv a^r \bmod p'$

4. **IF** $z \neq 1$ **AND** $z \neq p'-1$ **THEN**

5. **FOR** $j = 1$ **TO** $u-1$

6. $z \equiv z^2 \bmod p'$

7. **IF** $z = 1$ **THEN**

8. **RETURN** „p' is composite"

9\. **IF** $z \neq p'-1$ **THEN**

10. **RETURN** „p' is composite"

11. **RETURN** „p' is likely a prime"

Content of this Chapter

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

Attacks and Countermeasures 1/3

- There are two distinct types of attacks on cryptosystems
  - **Analytical attacks** try to break the mathematical structure of the underlying problem of RSA, Factoring Attacks, Small Exponent Attack.
  - **Implementation attacks** try to attack a real-world implementation by exploiting inherent weaknesses in the way RSA is realized in software or hardware. T**iming Attacks**: These attacks exploit the fact that the time taken to perform certain operations in RSA

Attacks and Countermeasures 2/3

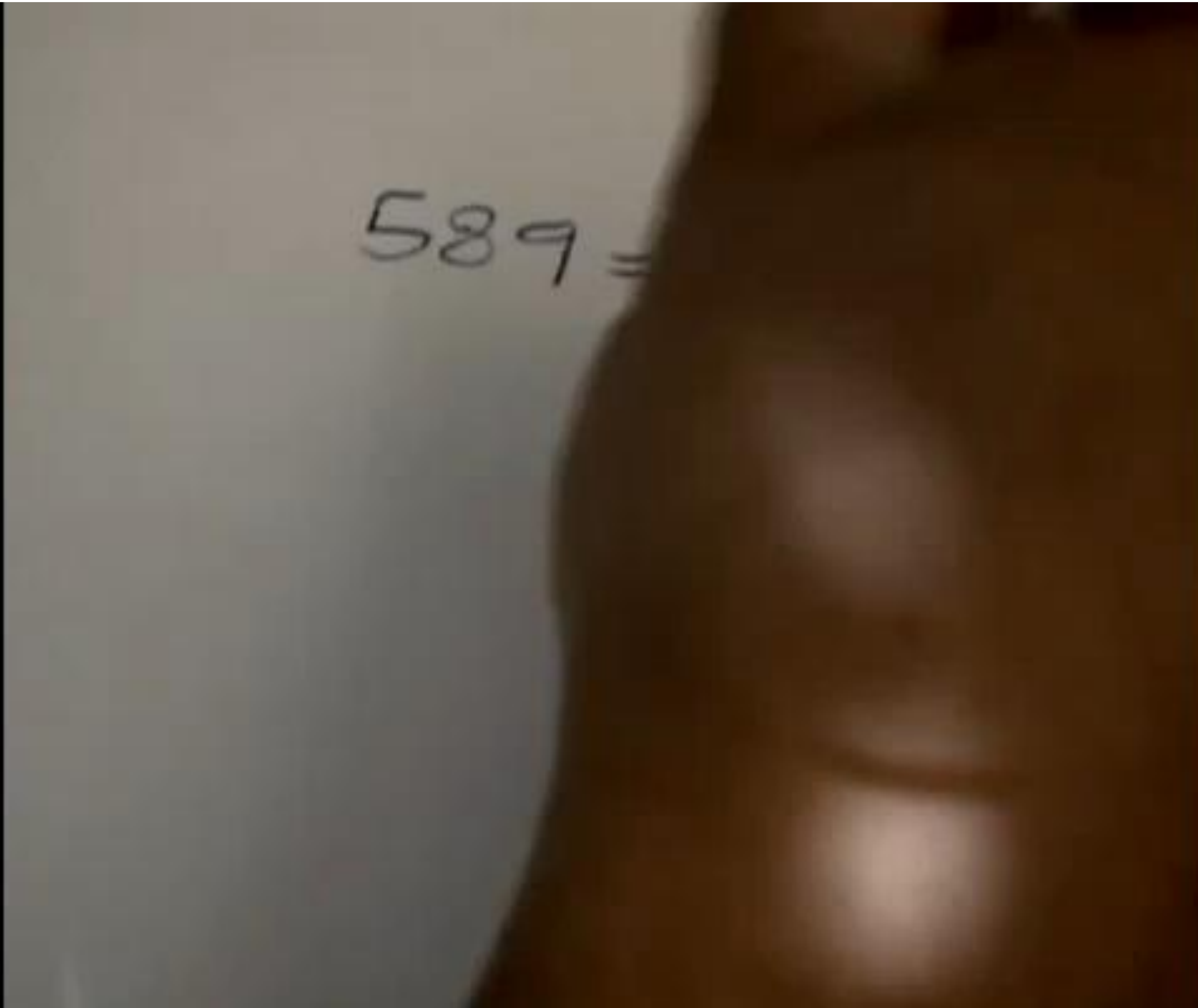
RSA is typically exposed to these analytical attack vectors

- **Mathematical attacks**
  - The best known attack is factoring of *n* in order to obtain *Φ(n)*
  - Can be prevented using a sufficiently large modulus *n*
  - The current factoring record is 664 bits. Thus, it is recommended that *n* should have a bit length between 1024 and 3072 bits
- **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

Attacks and Countermeasures 3/3

- Implementation attacks can be one of the following
  - **Side-channel analysis**
    - Exploit physical leakage of RSA implementation (e.g., power consumption, EM emanation, etc.)
  - **Fault-injection attacks**
    - Inducing faults in the device while CRT is executed can lead to a complete leakage of the private key

More on all attacks can be found in Section 7.8 of *Understanding Cryptography*


589 =

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


Thank
You