

# CMSC/Math 456: Cryptography (Fall 2023)

Lecture 8

Daniel Gottesman

# Administrative

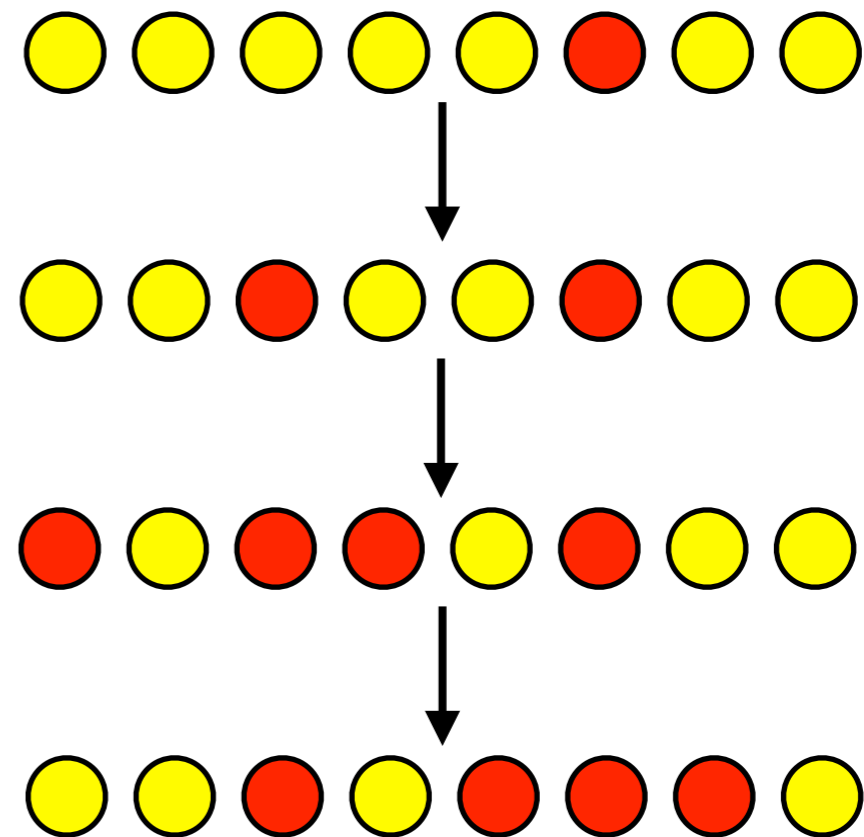
Problem set #4 is out and problem set #3 was due at noon today.

Problem set #2 grades are available, as is the solution set for #2 (on ELMS).

# Goals of Block Cipher Design

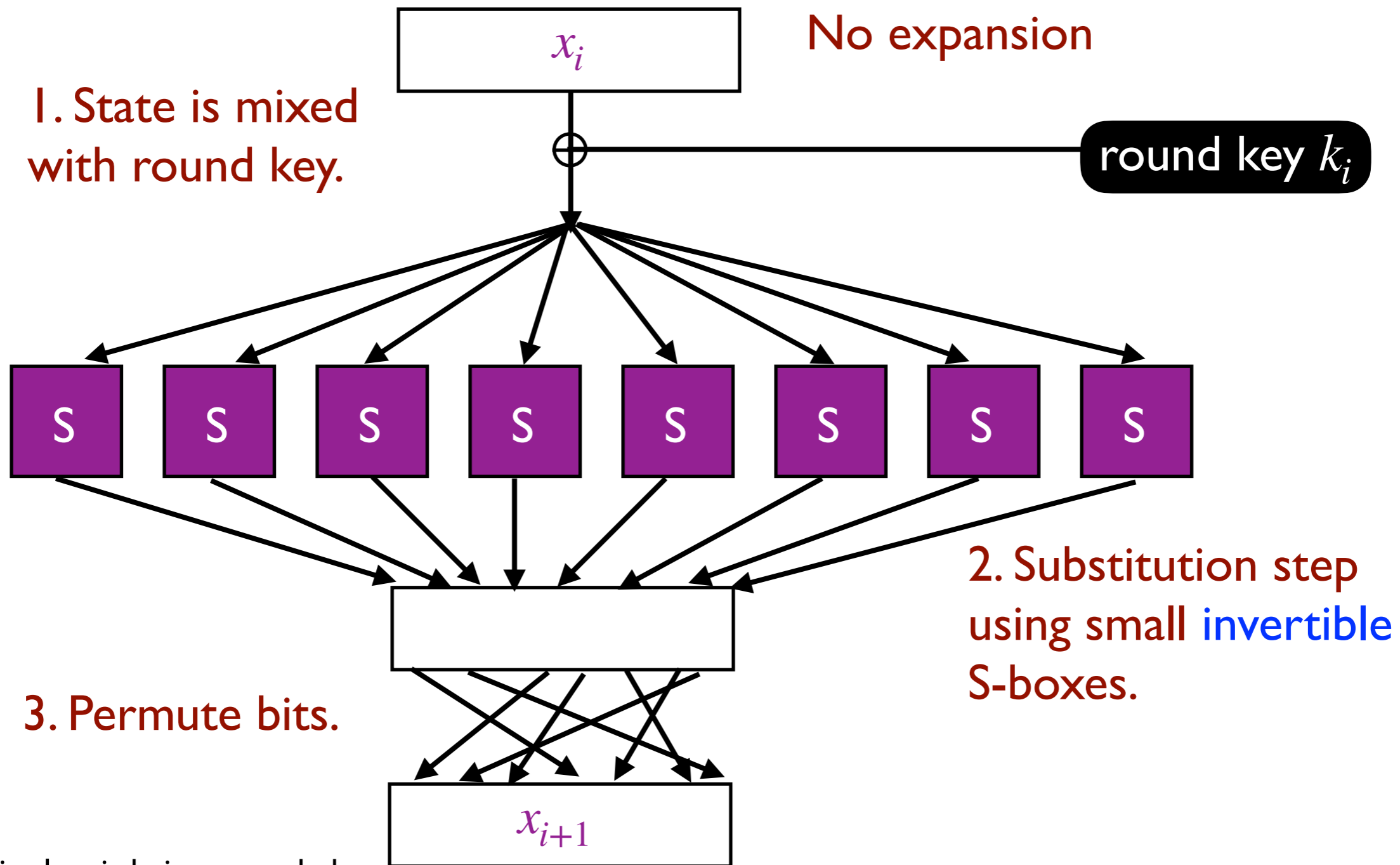
- Must be invertible to use with CBC mode (i.e., **pseudorandom permutation** rather than **pseudorandom function**).
- Even when the inputs are related, the outputs should be very different.

In particular, the change of even a single bit of the input should result in a totally different output. This is known as the “**avalanche effect**.” It is often achieved by having multiple rounds, each of which magnifies small changes.



# Substitution-Permutation Networks

AES is basically a **substitution-permutation network**, which was also essentially the design for the DES mangler function.



This class is being recorded

# Confusion-Diffusion

The **S-boxes** introduce **confusion**: They change their inputs into totally different strings and magnify single-bit changes. However, the S-box is small and acts on only a few bits, so the confusion is **only local**.

Then the **permutation step** causes **diffusion**: whatever local confusion was introduced by the S-boxes spreads out to many different locations.

Multiple rounds of substitution and permutation cause the confusion to be magnified further and continue to spread around.

**We need both to get an avalanche effect.**

You also need **key mixing**: This is a permutation, and without the key, Eve can just trace the permutation backwards to get the input.

# AES Key Schedule and Key Mixing

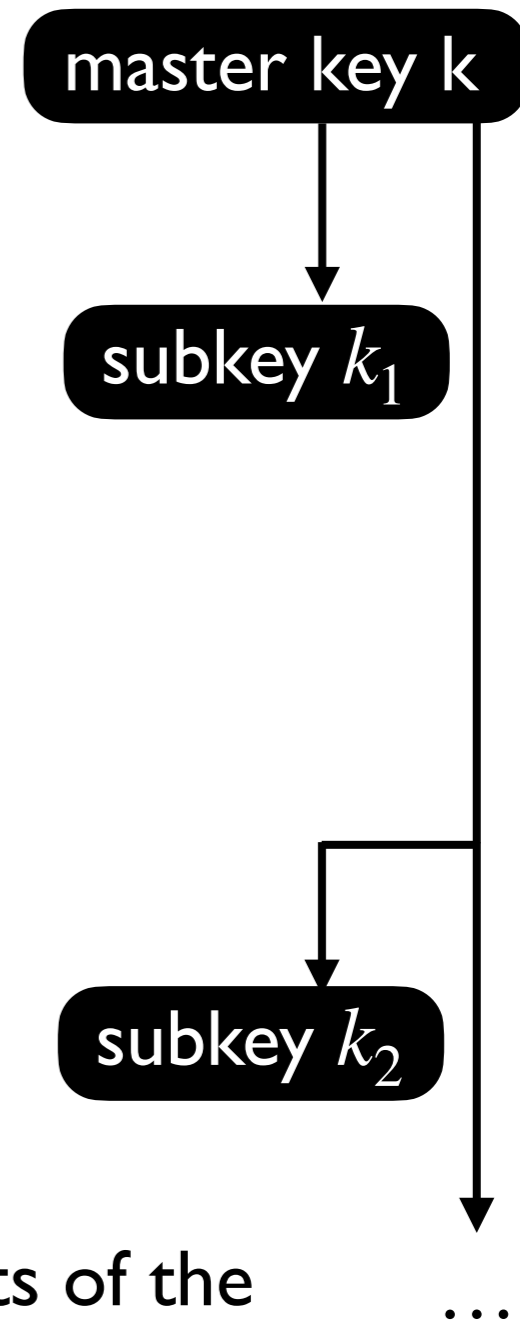
AES is standardized for 3 key lengths: **128 bits**, **192 bits**, and **256 bits**.

Each subkey is 128 bits and is derived from the master key by more complex transformations than in DES. In particular, the later subkeys are derived using the AES S-box.

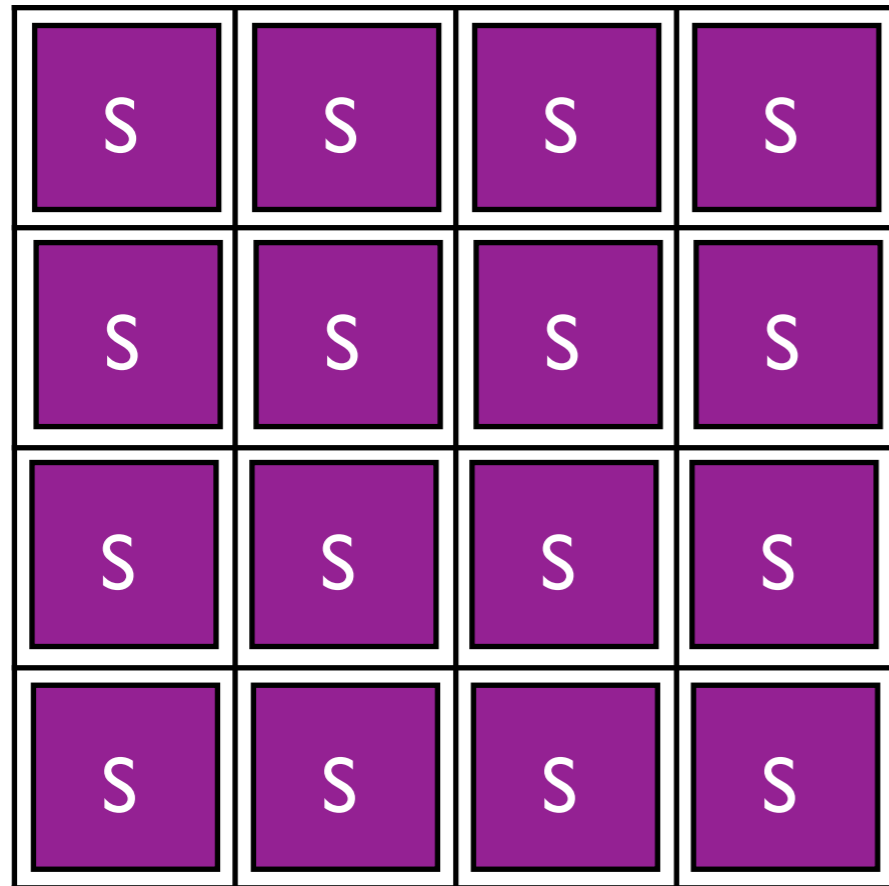
The number of rounds also depends on the key length (longer key = more rounds = more secure):

- 128-bit key: 10 rounds
- 192-bit key: 12 rounds
- 256-bit key: 14 rounds

The 128-bit subkey is then XORed with the 128 bits of the state at the key mixing stages.



# Applying AES S-Boxes



The AES S-box takes a 1-byte (8 bit) input and 1-byte output. It is invertible and again invoked via a table lookup.

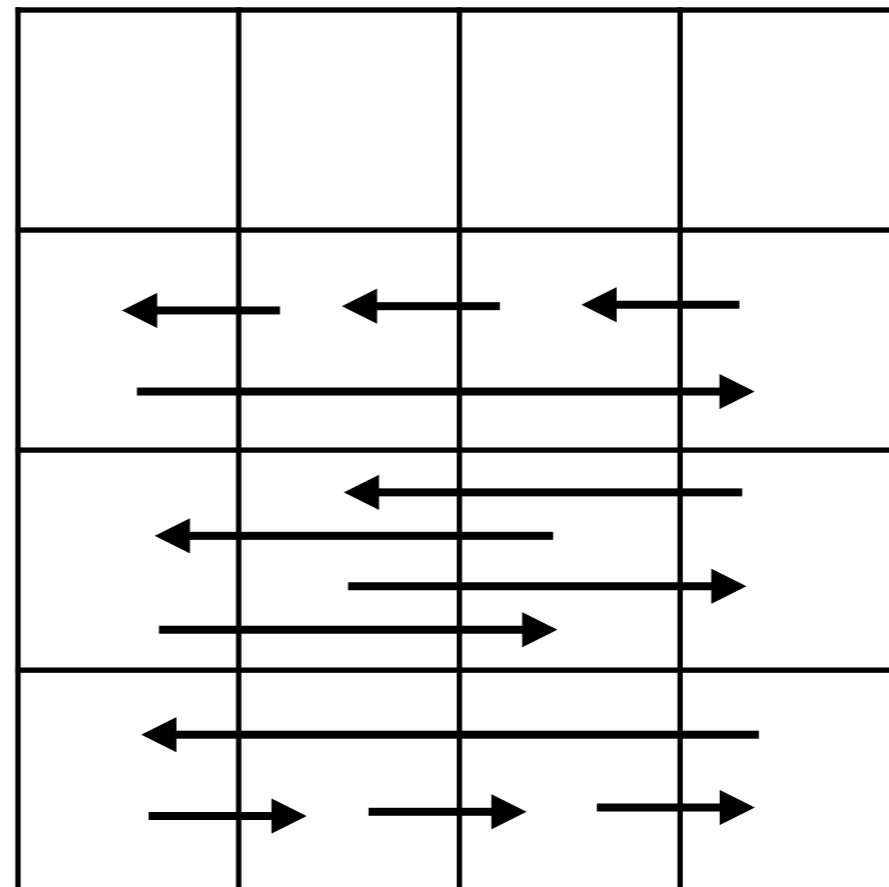
The same S-box is applied to each entry of the matrix.

Again, the S-box is chosen to introduce **confusion** by magnifying small changes in the input.

# Shifting Rows in AES

The **diffusion** step in AES consists of two pieces. In the first part, the rows are shifted independently.

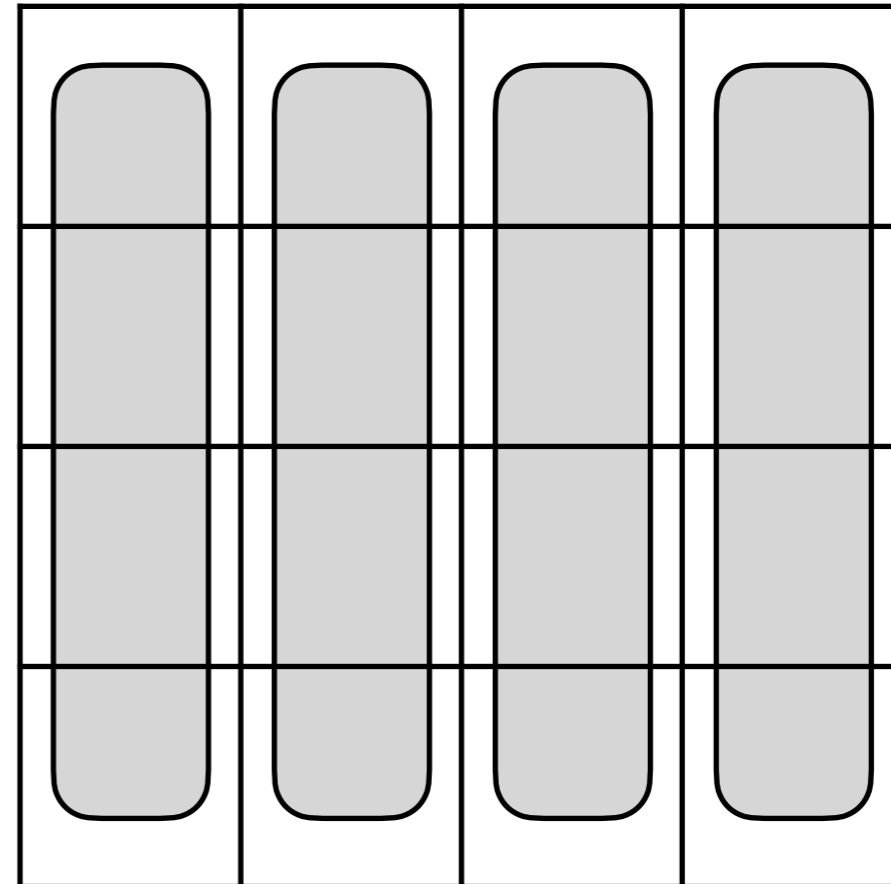
In the **ShiftRows** step, the  $i$ th row is shifted cyclically by  $i$  spaces.





# Column Mixing in AES

The **MixColumns** deviates from a substitution-permutation network by using a linear transformation on each column instead of permuting the bits. But it has a similar effect. (Linear transformations are also easy to invert, like permutations.)



There would not be much point in ending with **ShiftRows** and **MixColumns** steps, since by themselves they can be easily inverted by Eve. Instead, the last round replaces **MixColumns** with another key mixing step (requiring an extra subkey).

# Breaking AES

**Vote:** Is there a known way to break AES? (Yes/No/Other)

# Breaking AES

**Vote:** Is there a known way to break AES? (Yes/No/Other)

All three answers are correct in a sense.

There is no (publicly) known practical attack against AES in its ideal implementation.

But ... if it is not implemented properly, the encryption algorithm can sometimes leak additional information that can help narrow down the key.

This is known as a **side-channel** attack.

This seems like cheating — but yes, Eve cheats. **If she can't win playing by your rules, she will try to change the rules.**

# Side Channel Attacks

There are a wide variety of known side-channel attacks. E.g.:

**Timing Attacks:** Some computations take longer than others. If we're not **very careful**, different keys or messages will lead to quicker or slower computations, and Eve can detect that.

**Power Analysis Attacks:** Similarly, some computations may draw more power (or produce more heat) than others. This can also be used by Eve to narrow down the key or message.

**Cache Attacks:** By monitoring cache use, an attacker may be able to determine information about what computation the encryption process is using.

**Electromagnetic or Acoustic Attacks:** EM radiation or sounds may leak from the computer, revealing some information about the encryption process.

# How to Access a Side Channel

In some cases, it is possible to perform a side channel attack **over a network**. For instance, in a timing attack, Eve can interact with a server (or monitor Bob's interaction with the server) and **measure the length of time** it takes for the server to respond to each query. This reveals something about the encryption time.

In other cases, it may be necessary for Eve to **monitor the physical vicinity of Alice**. For instance, in an electromagnetic attack, Eve needs some way of seeing the leaked radiation.

In other cases, Eve may need a process on the **same computer as Alice**. For instance, Eve may manage to get some low-privilege malware on the machine which is unable to read Alice's message but can **see how its own cache usage depends on the encryption**. Or perhaps Eve's program is simply running on the **same cloud server as Alice**.

# Side Channel Attacks on AES

AES's S-boxes and column mixing operations are defined via operations on finite fields. These can be computed, but to optimize speed, they are usually pre-computed into lookup tables, which are cached during an encryption.

This enables a [cache side-channel attack](#).

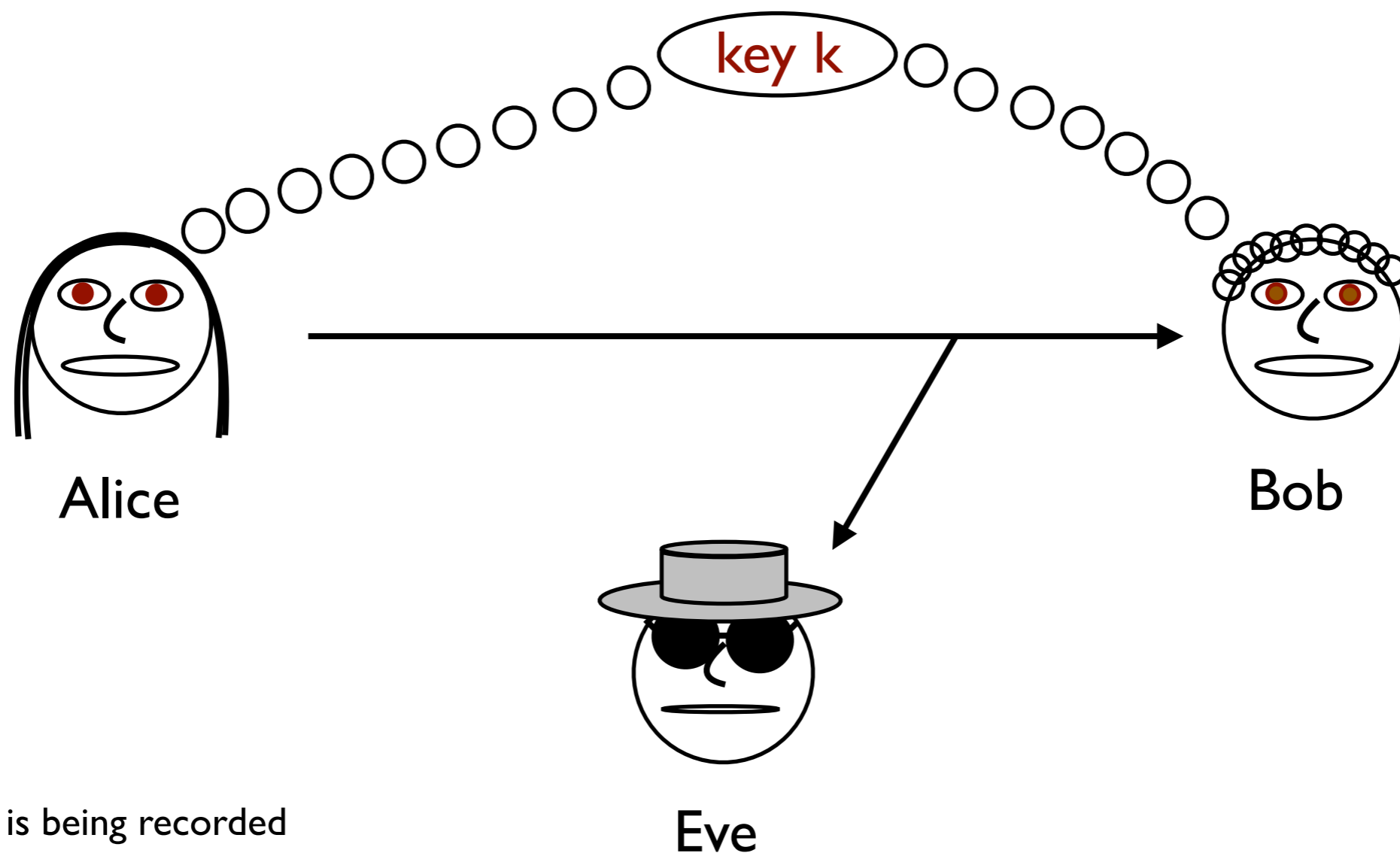
These attacks seem doable in practice given a poor implementation and a process running on the same device (e.g., same cloud server) as Alice.

But there are also countermeasures, so in most cases, AES should be considered secure.

# How Do We Use Symmetric Crypto?

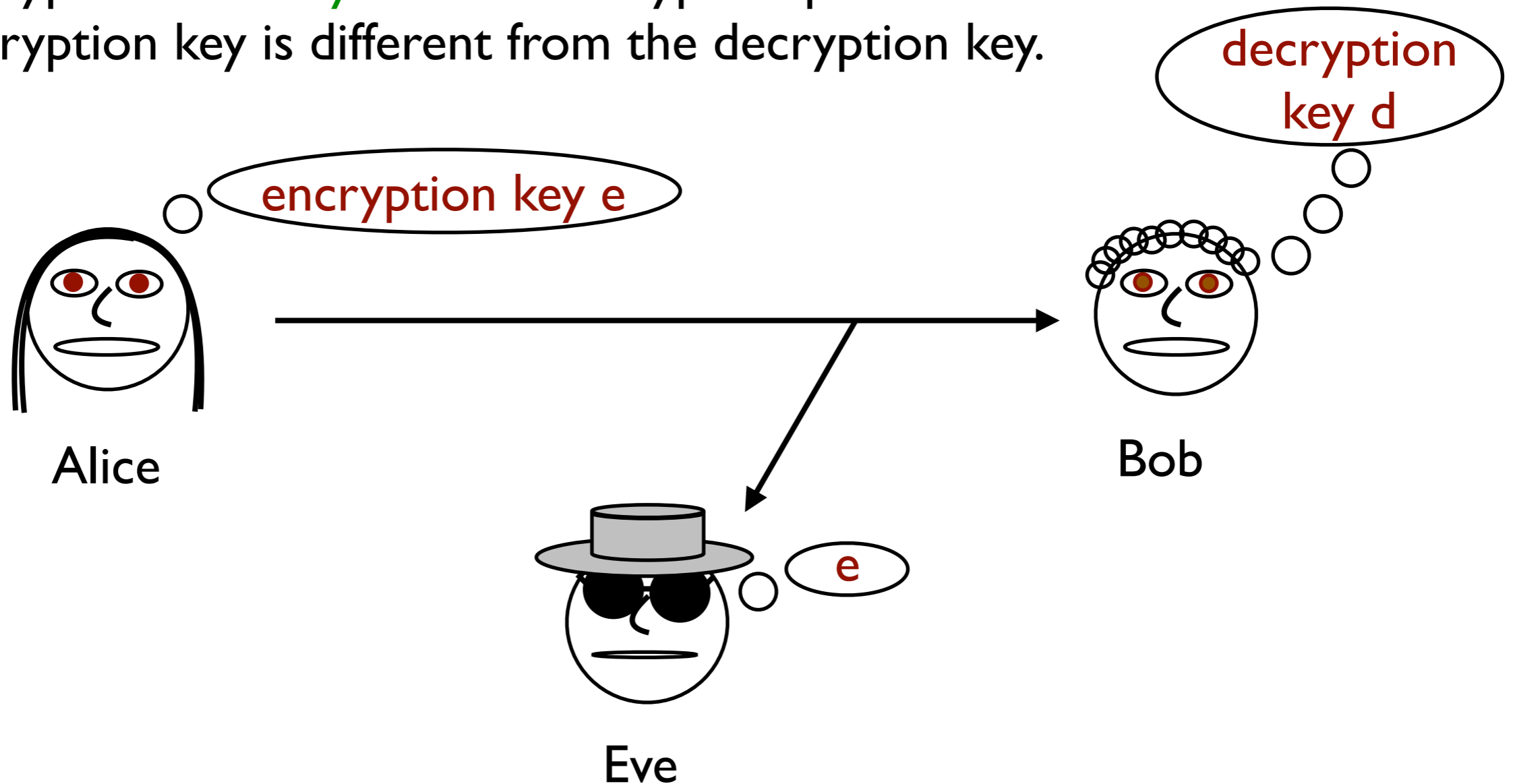
The **symmetric** cryptosystems we have seen so far require Alice and Bob to share a key unknown to Eve. This is fine if they occasionally meet in person and communicate regularly, but what if Alice and Bob have not met before?

How can they establish their first key remotely?



# Public Key Cryptography

The solution is to use **public key cryptography**. Public key encryption is an **asymmetric** encryption protocol where the encryption key is different from the decryption key.



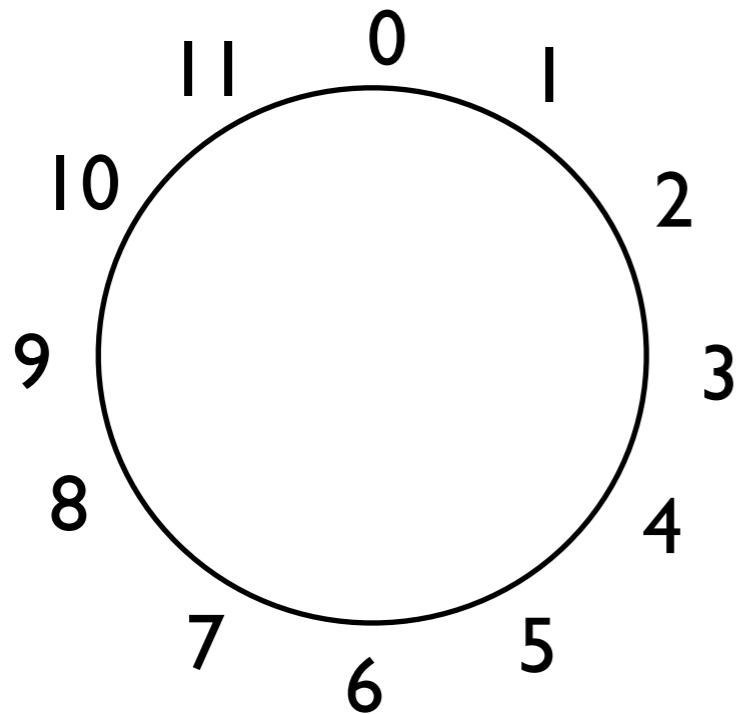
Public key encryption and key agreement protocols allow Alice and Bob to communicate securely without having a previously agreed secret key.



# Modular Arithmetic

To discuss public key cryptography, we need to know a lot more about modular arithmetic.

Modular arithmetic involves number systems that are cyclic, like a clock.



It's best to think of modular arithmetic as a new **type** of number, with different operations.

Numbers **mod N** form a new type for each different value of **N**.

We can do a **type conversion** from **integers** to numbers **mod N** by dividing by **N** and keeping only the remainder.

# Modular Arithmetic Operations

The basic arithmetic operations in modular arithmetic have the same names as for integer arithmetic and mostly inherit the same properties through the type conversion:

**Addition:**

# Modular Arithmetic Operations

The basic arithmetic operations in modular arithmetic have the same names as for integer arithmetic and mostly inherit the same properties through the type conversion:

**Addition:** Works the same. E.g.:


$$(36 + 58) + 15 \bmod 71 = 38 = 36 + (58 + 15) \bmod 71$$

# Modular Arithmetic Operations

The basic arithmetic operations in modular arithmetic have the same names as for integer arithmetic and mostly inherit the same properties through the type conversion:

**Addition:** Works the same. E.g.:

$$(36 + 58) + 15 \bmod 71 = 38 = 36 + (58 + 15) \bmod 71$$



indicates  
**mod 71** type  
for whole  
equation


# Modular Arithmetic Operations

The basic arithmetic operations in modular arithmetic have the same names as for integer arithmetic and mostly inherit the same properties through the type conversion:

**Addition:** Works the same. E.g.:

$$(36 + 58) + 15 \bmod 71 = 38 = 36 + (58 + 15) \bmod 71$$

**Subtraction:**



indicates  
**mod 71** type  
for whole  
equation

# Modular Arithmetic Operations


The basic arithmetic operations in modular arithmetic have the same names as for integer arithmetic and mostly inherit the same properties through the type conversion:

**Addition:** Works the same. E.g.:

$$(36 + 58) + 15 \text{ mod } 71 = 38 = 36 + (58 + 15) \text{ mod } 71$$

**Subtraction:** Works the same. E.g.:

$$36 - 58 = 49 \text{ mod } 71$$



indicates  
**mod 71** type  
for whole  
equation

# Modular Arithmetic Operations

The basic arithmetic operations in modular arithmetic have the same names as for integer arithmetic and mostly inherit the same properties through the type conversion:

**Addition:** Works the same. E.g.:


$$(36 + 58) + 15 \text{ mod } 71 = 38 = 36 + (58 + 15) \text{ mod } 71$$

**Subtraction:** Works the same. E.g.:

$$36 - 58 = 49 \text{ mod } 71$$

**Multiplication:**

indicates  
**mod 71** type  
for whole  
equation



# Modular Arithmetic Operations

The basic arithmetic operations in modular arithmetic have the same names as for integer arithmetic and mostly inherit the same properties through the type conversion:

**Addition:** Works the same. E.g.:


$$(36 + 58) + 15 \text{ mod } 71 = 38 = 36 + (58 + 15) \text{ mod } 71$$

**Subtraction:** Works the same. E.g.:

$$36 - 58 = 49 \text{ mod } 71$$

**Multiplication:** Works the same. E.g.:

$$36 \cdot 58 \text{ mod } 71 = 29 = 58 \cdot 36 \text{ mod } 71$$



indicates  
**mod 71** type  
for whole  
equation



# Modular Arithmetic Operations

The basic arithmetic operations in modular arithmetic have the same names as for integer arithmetic and mostly inherit the same properties through the type conversion:

**Addition:** Works the same. E.g.:

$$(36 + 58) + 15 \text{ mod } 71 = 38 = 36 + (58 + 15) \text{ mod } 71$$


**Subtraction:** Works the same. E.g.:

$$36 - 58 = 49 \text{ mod } 71$$

**Multiplication:** Works the same. E.g.:

$$36 \cdot 58 \text{ mod } 71 = 29 = 58 \cdot 36 \text{ mod } 71$$

**Division:**



indicates  
**mod 71** type  
for whole  
equation

# Modular Arithmetic Operations

The basic arithmetic operations in modular arithmetic have the same names as for integer arithmetic and mostly inherit the same properties through the type conversion:

**Addition:** Works the same. E.g.:

$$(36 + 58) + 15 \text{ mod } 71 = 38 = 36 + (58 + 15) \text{ mod } 71$$

**Subtraction:** Works the same. E.g.:

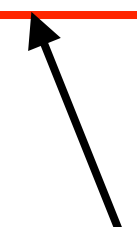
$$36 - 58 = 49 \text{ mod } 71$$

**Multiplication:** Works the same. E.g.:

$$36 \cdot 58 \text{ mod } 71 = 29 = 58 \cdot 36 \text{ mod } 71$$

**Division:** There are some issues. E.g.:

$$35/58 \text{ mod } 60 = ?$$



indicates  
**mod 71** type  
for whole  
equation

# Modular Division Definition

What does division  $\text{mod } N$  mean?

Division is supposed to be the inverse of multiplication. That is, if

$$ab = c \text{ mod } N \quad \text{then} \quad c/a = b \text{ mod } N$$

In particular, the definition of  $c/a \text{ mod } N$  is a number  $b \text{ (mod } N)$  with the property that

$$ab = c \text{ mod } N$$

This means that the question  $35/58 \text{ mod } 60 = ?$

has no answer.  $\nexists x \text{ s.t. } 58 \cdot x = 35 \text{ mod } 60$

# Modular Divison

But  $36/58 \bmod 71$  *is* well-defined:

$$58 \cdot 30 = 36 \bmod 71$$

Thus,

$$36/58 \bmod 71 = 30$$

# Modular Division

But  $36/58 \bmod 71$  *is* well-defined:

$$58 \cdot 30 = 36 \bmod 71$$

Thus,

$$36/58 \bmod 71 = 30$$

How do we determine if division is allowed or not?

# Modular Divison

But  $36/58 \bmod 71$  *is* well-defined:

$$58 \cdot 30 = 36 \bmod 71$$

Thus,

$$36/58 \bmod 71 = 30$$

How do we determine if division is allowed or not?

$$c/a = b \bmod N \iff c = ab + kN$$

Convert  
back to  
integer type

# Modular Division

But  $36/58 \bmod 71$  *is* well-defined:

$$58 \cdot 30 = 36 \bmod 71$$

Thus,

$$36/58 \bmod 71 = 30$$

How do we determine if division is allowed or not?

$$c/a = b \bmod N \iff c = ab + kN$$

Convert  
back to  
integer type

Suppose there is some  $p$  such that  $p \mid a$  and  $p \mid N$ . Then the right-hand side of the equation on the right is also a multiple of  $p$ .

Thus, division by  $a$  is only possible if  $c$  is a multiple of  $p$  as well.

This shows  $35/58 \bmod 60$  is undefined:  $58$  and  $60$  are both even.

# Modular Division

But  $36/58 \bmod 71$  *is* well-defined:

$$58 \cdot 30 = 36 \bmod 71$$

Thus,

$$36/58 \bmod 71 = 30$$

How do we determine if division is allowed or not?

$$c/a = b \bmod N \iff c = ab + kN$$

Convert  
back to  
integer type

Suppose there is some  $p$  such that  $p \mid a$  and  $p \mid N$ . Then the right-hand side of the equation on the right is also a multiple of  $p$ .

Thus, division by  $a$  is only possible if  $c$  is a multiple of  $p$  as well.

This shows  $35/58 \bmod 60$  is undefined:  $58$  and  $60$  are both even.

What about if  $a$  and  $N$  are *relatively prime*? (I.e., they have no common factors.)



# Finding GCDs

**Definition:** Let  $\gcd(a,b)$  be *greatest common divisor* of positive integers  $a$  and  $b$ : namely, the largest integer  $c$  such that  $c \mid a$  and  $c \mid b$ . Note that if  $a$  and  $b$  are relatively prime,  $\gcd(a,b) = 1$ .

# Finding GCDs

**Definition:** Let  $\gcd(a,b)$  be *greatest common divisor* of positive integers  $a$  and  $b$ : namely, the largest integer  $c$  such that  $c \mid a$  and  $c \mid b$ . Note that if  $a$  and  $b$  are relatively prime,  $\gcd(a,b) = 1$ .

**Theorem:** For any two positive integers  $a$  and  $b$ , there exists a polynomial-time algorithm to find  $X$  and  $Y$  such that

$$aX + bY = \gcd(a, b)$$

**Note:** If  $d < \gcd(a, b)$ , then  $aX + bY \neq d$  for any integers  $X, Y$ .

# Finding GCDs

**Definition:** Let  $\gcd(a,b)$  be *greatest common divisor* of positive integers  $a$  and  $b$ : namely, the largest integer  $c$  such that  $c \mid a$  and  $c \mid b$ . Note that if  $a$  and  $b$  are relatively prime,  $\gcd(a,b) = 1$ .

**Theorem:** For any two positive integers  $a$  and  $b$ , there exists a polynomial-time algorithm to find  $X$  and  $Y$  such that

$$aX + bY = \gcd(a, b)$$

**Note:** If  $d < \gcd(a, b)$ , then  $aX + bY \neq d$  for any integers  $X, Y$ .

**Proof:** The proof is an analysis of the algorithm to find  $X$  and  $Y$ . This is *Euclid's algorithm*.

Euclid's algorithm appeared in Euclid's *Elements* in around 300 BCE. That makes it **one of the world's oldest algorithms!**

# Modular Division

If  $\gcd(a, N) = 1$ , then we can always divide by  $a$  in mod  $N$  arithmetic:

Using Euclid's algorithm, find  $X, Y$  such that

$$aX + NY = 1$$

Then  $aX = 1 \pmod N$ .

$X$  is then the **multiplicative inverse** of  $a$ . Let  $b = cX \pmod N$ . Then

$$ab = a(cX) = c(Xa) = c \pmod N$$

so  $c/a = b = cX \pmod N$ .

And moreover, we can divide in polynomial time.

**Example:**  $1 = -5 \cdot 57 + 13 \cdot 22$

Thus,  $c/22 \pmod{57} = 13c \pmod{57}$ . E.g.,  $5/22 = 8 \pmod{57}$ .

# Dos and Don'ts of Division

When  $a$  and  $N$  are relatively prime, it is OK to cancel  $a$  from an equation:

$$ab = ac \pmod{N} \implies b = c \pmod{N}$$

But this is *not* OK in general if  $\gcd(a, N) \neq 1$ .

Examples:

$$2 \cdot 4 = 2 \cdot 9 \pmod{10} \text{ but } 4 \neq 9 \pmod{10}.$$

$$3 \cdot 4 + 3 \cdot 4 = 4 \pmod{10} = 3 \cdot 8 \pmod{10}$$

$$\implies 4 + 4 = 8 \pmod{10}$$

# Euclid's Algorithm Concept

Suppose we want to find  $c = \gcd(a,b)$ .

We know  $c \mid a$  and  $c \mid b$ . Can we find another smaller number that is also a multiple of  $c$ ?

If  $a > b$ , then  $a' = a - b$  is smaller than  $a$  and must still be a multiple of  $c$ .

If we keep subtracting one number from the other, our pair of numbers will get steadily smaller until eventually we get down to  $c$ .

Example:

$$a = 58$$

$$b = 36$$

$c = 2$  (but we don't know that yet)

$$a - b = 22$$

(still a multiple of  $c$ )

$$36 - 22 = 14$$

$$22 - 14 = 8$$

$$14 - 8 = 6$$

$$8 - 6 = 2$$

$6$  is a multiple of  $2$ , so we are done.

# Euclid's Algorithm Refinements

When we subtract off  $b$  from  $a$ , the result might still be bigger than  $b$ . Instead we should take  $a \bmod b$ , which means subtract off as many  $b$ 's as we can. This will give us a number  $a'$  which is less than  $b$ , so next time we reduce  $b$  instead.

# Euclid's Algorithm Refinements

When we subtract off  $b$  from  $a$ , the result might still be bigger than  $b$ . Instead we should take  $a \bmod b$ , which means subtract off as many  $b$ 's as we can. This will give us a number  $a'$  which is less than  $b$ , so next time we reduce  $b$  instead.

To get the coefficients  $X$  and  $Y$ , we should also keep track of how *many*  $b$ 's we subtracted:

$$a' = a - Y_0 b$$



# Euclid's Algorithm Refinements

When we subtract off  $b$  from  $a$ , the result might still be bigger than  $b$ . Instead we should take  $a \bmod b$ , which means subtract off as many  $b$ 's as we can. This will give us a number  $a'$  which is less than  $b$ , so next time we reduce  $b$  instead.

To get the coefficients  $X$  and  $Y$ , we should also keep track of how *many*  $b$ 's we subtracted:

$$a' = a - Y_0 b$$

At each step, this will allow us to write our current replacements for  $a$  and  $b$  in the form  $aX_i + bY_i$ .

# Euclid's Algorithm Refinements

When we subtract off  $b$  from  $a$ , the result might still be bigger than  $b$ . Instead we should take  $a \bmod b$ , which means subtract off as many  $b$ 's as we can. This will give us a number  $a'$  which is less than  $b$ , so next time we reduce  $b$  instead.

To get the coefficients  $X$  and  $Y$ , we should also keep track of how *many*  $b$ 's we subtracted:

$$a' = a - Y_0 b$$

At each step, this will allow us to write our current replacements for  $a$  and  $b$  in the form  $aX_i + bY_i$ .

In particular, if our current pair is  $a_i = aX_i + bY_i$  and  $b_i = aX'_i + bY'_i$ , and we subtract  $m_i$  copies of  $b_i$ , then

$$a_{i+1} = a_i - m_i b_i = a(X_i - m_i X'_i) + b(Y_i - m_i Y'_i)$$

# Euclid's Algorithm Refinements

When we subtract off  $b$  from  $a$ , the result might still be bigger than  $b$ . Instead we should take  $a \bmod b$ , which means subtract off as many  $b$ 's as we can. This will give us a number  $a'$  which is less than  $b$ , so next time we reduce  $b$  instead.

To get the coefficients  $X$  and  $Y$ , we should also keep track of how *many*  $b$ 's we subtracted:

$$a' = a - Y_0 b$$

At each step, this will allow us to write our current replacements for  $a$  and  $b$  in the form  $aX_i + bY_i$ .

In particular, if our current pair is  $a_i = aX_i + bY_i$  and  $b_i = aX'_i + bY'_i$ , and we subtract  $m_i$  copies of  $b_i$ , then

$$a_{i+1} = a_i - m_i b_i = a(X_i - m_i X'_i) + b(Y_i - m_i Y'_i)$$

We don't need to keep  $a_i$  and  $b_i$  separate: We can combine them into a single sequence  $r_i$ .

# Euclid's Algorithm

Let  $r_0 = a$  and  $r_1 = b$ . Assume  $a > b$ .  
 $i = 1, X_0 = 1, Y_0 = 0, X_1 = 0, Y_1 = 1$

Repeat:

$$r_{i+1} = r_{i-1} \bmod r_i$$

$$m_i = \lfloor r_{i-1}/r_i \rfloor$$

$$X_{i+1} = X_{i-1} - m_i X_i$$

$$Y_{i+1} = Y_{i-1} - m_i Y_i$$

$$i = i + 1$$

Until  $r_i = 0$

Output:

$$\gcd(a, b) = r_{i-1}$$

$$X = X_{i-1}, Y = Y_{i-1}$$

Example:

$$r_0 = 57, r_1 = 22$$

$$r_2 = 13,$$

$$X_2 = 1, Y_2 = -2$$

$$r_3 = 9,$$

$$X_3 = -1, Y_3 = 3$$

$$r_4 = 4,$$

$$X_4 = 2, Y_4 = -5$$

$$r_5 = 1,$$

$$X_5 = -5, Y_5 = 13$$

$$r_6 = 0$$

$$\gcd(57, 22) = 1,$$
$$1 = -5 \cdot 57 + 13 \cdot 22$$

# Euclid's Algorithm Analysis

At every iteration of the algorithm, the following statements are true:

$$0 \leq r_i < r_{i-1}$$

$$r_i = aX_i + bY_i$$

$$\gcd(a, b) \mid r_i$$

If these statements are true for  $i$ , the statements also hold true for  $i+1$  (by the arguments before). They are true for  $i=0$  and thus we prove by induction that the statements are true for all  $i$ .

# Euclid's Algorithm Analysis

At every iteration of the algorithm, the following statements are true:

$$0 \leq r_i < r_{i-1}$$

$$r_i = aX_i + bY_i$$

$$\gcd(a, b) \mid r_i$$

If these statements are true for  $i$ , the statements also hold true for  $i+1$  (by the arguments before). They are true for  $i=0$  and thus we prove by induction that the statements are true for all  $i$ .

Since  $r_i$  strictly decreases, the algorithm must eventually reach  $r_i = 0$ , at which point it terminates with  $i - 1 = i_f$ . At that point,  $r_{i_f} \mid r_{i_f-1}$ . But that means  $r_{i_f} \mid r_{i_f-2} = m_{i_f-1}r_{i_f-1} + r_{i_f}$  and so on. By induction, we also have  $r_{i_f} \mid r_j$  for all  $j$ .

# Euclid's Algorithm Analysis

At every iteration of the algorithm, the following statements are true:

$$0 \leq r_i < r_{i-1}$$

$$r_i = aX_i + bY_i$$

$$\gcd(a, b) \mid r_i$$

If these statements are true for  $i$ , the statements also hold true for  $i+1$  (by the arguments before). They are true for  $i=0$  and thus we prove by induction that the statements are true for all  $i$ .

Since  $r_i$  strictly decreases, the algorithm must eventually reach  $r_i = 0$ , at which point it terminates with  $i-1 = i_f$ . At that point,  $r_{i_f} \mid r_{i_f-1}$ . But that means  $r_{i_f} \mid r_{i_f-2} = m_{i_f-1}r_{i_f-1} + r_{i_f}$  and so on. By induction, we also have  $r_{i_f} \mid r_j$  for all  $j$ .

In particular,  $r_{i_f} \mid a$  and  $r_{i_f} \mid b$ . But  $\gcd(a, b) \mid r_{i_f}$ , so

$$r_{i_f} = \gcd(a, b)$$

# Efficiency of Euclid's Algorithm

How quickly does  $r_i$  decrease in Euclid's algorithm?

If  $r_i \geq r_{i-1}/2$ , then  $r_{i+1} \leq r_{i-1}/2$ .

If  $r_i \leq r_{i-1}/2$ , then  $r_{i+1} \leq r_i \leq r_{i-1}/2$ .

Either way,  $r_{i+1} \leq r_{i-1}/2$ .

Since  $r_i$  is at least halved every 2 steps, the algorithm can run at most  $2 \log_2 a$  steps before halting.



# Meaning of Efficient

It's important to remember that **efficient** (or **polynomial time**) means polynomial time as a function of **the input size**.

When doing arithmetic or finding the gcd, the **input size is the length** (i.e., **number of bits**) of the numbers being computed with.

**Not polynomial in the numbers themselves!**

Integer addition, subtraction, multiplication, division (with remainder) are all efficient in this sense using standard grade school algorithms. **Still true for modular  $+$ ,  $-$ ,  $*$ .**

$\log_2 a$  is the input size, so Euclid's algorithm has a polynomial number of steps, each of which is efficient. Therefore it is efficient overall.

