Thus far, we have <u>assumed</u> that parties have a <u>shared</u> key. Where does the shared key come from?

Can we do this using the tools we have developed so far?

So far in this course:

PRFs $\Rightarrow$ CPA-secure encryption
PRFs $\Rightarrow$ MAC
$\Rightarrow$ authenticated encryption

Can we use PRFs to construct secure key-agreement protocols?

Key agreement:

Alice $\quad\quad$ Bob

$\longrightarrow$
$\longleftarrow$
$\vdots$
$\longrightarrow$
$\longleftarrow$

$\downarrow$ k_1 $\quad\quad$ $\downarrow$ k_2

Requirements:
1) $k_1 = k_2 = k$ with high probability
2) Eavesdropper cannot learn k (efficiently)

<u>Merkle puzzles</u>: Suppose $f : \mathcal{X} \to \mathcal{Y}$ is a function that is hard to invert ("one-way function")

$\hookrightarrow$ for example, a secure PRG $G : \{0,1\}^\lambda \to \{0,1\}^n$ is one-way

<u>Alice</u> $\quad\quad$ <u>Bob</u>

$x_1, \dots, x_n \leftarrow \mathcal{X}$

$\xrightarrow{y_1 = f(x_1), \dots, y_n = f(x_n)}$

$i \xleftarrow{R} [n]$

find x_i such that $f(x_i) = y_i$ [solve the "puzzle"]

$\hookrightarrow$ we assume that the solution is unique

derive a key k from x_i

$\xleftarrow{\text{Encrypt}_{AE}(k, m)}$

$\downarrow$ try each key k_i to decrypt ciphertext

derived from x_i

Suppose it takes time t to solve a puzzle. Adversary needs time $O(nt)$ to solve all puzzles and identify key. Honest parties work in time $O(n+t)$.

$\hookrightarrow$ Only provides <u>linear gap</u> between honest parties and adversary

Can we get a super-polynomial gap just using PRGs? Very difficult! [Impagliazzo-Rudich]

Can we get a super-linear gap just using PRGs? Very difficult! [Barak-Mahmoody]

Impagliazzo-Rudich: <u>Proving</u> the existence of key-agreement that makes <u>black-box</u> use of PRG implies $P \neq NP$.

result holds even if start with a one-way permutation

We will turn to algebra/number theory for new sources of hardness to build key agreement protocols.

Definition. A group consists of a set $\mathbb{G}$ together with an operation $*$ that satisfies the following properties:

- Closure: If $g_1, g_2 \in \mathbb{G}$, then $g_1 * g_2 \in \mathbb{G}$
- Associativity: For all $g_1, g_2, g_3 \in \mathbb{G}$, $g_1 * (g_2 * g_3) = (g_1 * g_2) * g_3$
- Identity: There exists an element $e \in \mathbb{G}$ such that $e * g = g = g * e$ for all $g \in \mathbb{G}$
- Inverse: For every element $g \in \mathbb{G}$, there exists an element $g^{-1} \in \mathbb{G}$ such that $g * g^{-1} = e = g^{-1} * g$

In addition, we say a group is commutative (or abelian) if the following property also holds:

- Commutative: For all $g_1, g_2 \in \mathbb{G}$, $g_1 * g_2 = g_2 * g_1$

called "multiplicative" notation

Notation: Typically, we will use "$\cdot$" to denote the group operation (unless explictly specified otherwise). We will write g^x to denote $\underbrace{g \cdot g \cdot g \cdots g}_{x \text{ times}}$ (the usual exponential notation). We use "1" to denote the multiplicative identity

Examples of groups: $(\mathbb{R}, +)$: real numbers under addition

$(\mathbb{Z}, +)$: integers under addition

$(\mathbb{Z}_p, +)$: integers modulo p under addition [sometimes written as $\mathbb{Z}/p\mathbb{Z}$]

here, p is prime

The structure of $\mathbb{Z}_p^*$ (an important group for cryptography):

$\mathbb{Z}_p^* = \{x \in \mathbb{Z}_p : \text{there exists } y \in \mathbb{Z}_p \text{ where } xy = 1 \pmod p\}$

the set of elements with multiplicative inverses modulo p

What are the elements in $\mathbb{Z}_p^*$?

greatest common divisor

<u>Bezout's identity</u>: For all positive integers $x, y \in \mathbb{Z}$, there exists integers $a, b \in \mathbb{Z}$ such that $ax + by = \gcd(x, y)$.

<u>Corollary</u>: For prime p, $\mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$.

<u>Proof</u>. Take any $x \in \{1, 2, \dots, p-1\}$. By Bezout's identity, $\gcd(x, p) = 1$ so there exists integers $a, b \in \mathbb{Z}$ where $1 = ax + bp$. Modulo p, this is $ax = 1 \pmod{p}$ so $a = x^{-1} \pmod{p}$.

Coefficients a, b in Bezout's identity can be efficiently computed using the extended Euclidean algorithm:

<u>Euclidean algorithm</u>: algorithm for computing $\gcd(a, b)$ for positive integers $a > b$:

relies on fact that $\gcd(a, b) = \gcd(b, a \pmod{b})$:

to see this: take any $a > b$

↳ we can write $a = b \cdot q + r$ where $q \geq 1$ is the quotient and $0 \leq r < b$ is the remainder

↳ d divides a and b $\Longleftrightarrow$ d divides b and r

↳ $\gcd(a, b) = \gcd(b, r) = \gcd(b, a \pmod{b})$

gives an explicit algorithm for computing gcd: repeatedly divide:

$\gcd(60, 27)$:

$60 = 27(2) + 6$ $[q = 2,\ r = 6]$ $\rightsquigarrow$ $\gcd(60, 27) = \gcd(27, 6)$

$27 = 6(4) + 3$ $[q = 4,\ r = 3]$ $\rightsquigarrow$ $\gcd(27, 6) = \gcd(6, 3)$

$6 = 3(2) + 0$ $[q = 2,\ r = 0]$ $\rightsquigarrow$ $\gcd(6, 3) = \gcd(3, 0) = 3$

"rewind" to recover coefficients in Bezout's identity:

extended Euclidean algorithm
$$\left\{\begin{array}{l} 60 = 27(2) + 6 \\ 27 = 6(4) + 3 \\ 6 = 3(2) + 0 \end{array}\right. \longrightarrow 3 = 27 - 6 \cdot 4$$

$6 = 60 - 27(2)$

$27 - (60 - 27(2))4$

$= 27(9) + 60(-4)$

coefficients

<u>Iterations needed</u>: $O(\log a)$ – i.e., bit-length of the input [worst case inputs: Fibonacci numbers]

<u>Implication</u>: Euclidean algorithm can be used to compute modular inverses (faster algorithms also exist)