# Lecture 30: Hybrid Encryption and Prime Number Generation

# Recall: ElGamal Encryption I

- We begin by recalling the ElGamal Public-key Encryption
- Recall that to describe a private-key encryption scheme we had to provide the algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. Similarly, to describe a public-key encryption scheme, we will have to provide the $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ algorithms
- Assume that the DDH Assumption holds for the group $(G, \circ)$ of size $N$, and the group $G$ has a generator $g$
- For perspective, $N = 2^n$, where $n = 1024$. Our algorithms have to be polynomial in $n$ and the adversary, to break the scheme, has to invest roughly $2^{\text{constant} \cdot n}$ effort

# Recall: ElGamal Encryption II

## Generation Algorithm.

- Recall that in the private-key encryption scheme the generation algorithm Gen() outputs the secret-key for the encryption scheme. In public-key encryption, the generation algorithm has to output the public-key pk for the scheme. Additionally, it has to output the "trapdoor" trap that assists the receiver to decrypt the cipher-text. If such a trapdoor does not exist, then Bob gets no additional advantage over an eavesdropper to decrypt the cipher-text.

Gen():

1. Sample $b \xleftarrow{\$} \{0, 1, 2, \ldots, N-1\}$
2. Compute $B = g^b$ (using repeated squaring technique)
3. Return $(\text{pk} = B, \text{trap} = b)$

Now, the receiver can broadcast the pk to everyone and keep trap secret with herself to assist in the decryption algorithm

# Recall: ElGamal Encryption III

## Encryption Algorithm.

- Recall that in the private-key encryption scheme the encryption algorithm takes two inputs (the secret-key and the message) $\mathsf{Enc}_{\mathsf{sk}}(m)$. In the public-key encryption, it will take the public-key and the message as input and output the encryption.

$\mathsf{Enc}_{\mathsf{pk}}(m)$:

1. Sample $a \xleftarrow{\$} \{0, 1, 2, \ldots, N-1\}$
2. Compute $A = g^a$ (using repeated squaring technique)
3. Compute mask $= \mathsf{pk}^a$ (using repeated squaring technique)
4. Return the cipher-text $c = (A, m \circ \text{mask})$

In the ElGamal encryption scheme $\mathsf{pk} = B$. Note that each time the encryption algorithm is invoked, it will create a new random mask.

If the same mask is generated in two different invocations of the encryption algorithm, then it must be the case that the same $A$ was generate in those two

# Recall: ElGamal Encryption IV

invocations. That implies that the same $a$ was generate in those two invocations, which has probability $\sqrt{2^{-n}} = 2^{-n/2}$ by the birthday bound)

# Recall: ElGamal Encryption V

**Decryption Algorithm.**

- Recall that in the private-key encryption scheme the decryption algorithm takes two inputs (the secret-key and the cipher-text) $\mathsf{Dec}_{\mathsf{sk}}(c)$. In the public-key encryption, it will take the cipher-text and the trapdoor generated during the generation procedure as input.

$\mathsf{Dec}_{\mathsf{trap}}(\widetilde{A}, \widetilde{c})$:

1. Compute $\widetilde{\mathsf{mask}}(\widetilde{A})^{\mathsf{trap}}$
2. Return $\widetilde{c} \circ \mathsf{inv}(\widetilde{\mathsf{mask}})$

Recall that $\mathsf{trap} = b$. If $\widetilde{A} = g^a$, then $\widetilde{\mathsf{mask}} = g^{ab}$.

# Hybrid Encryption I

- We will combine any public-key encryption scheme with any private-key encryption scheme to create a new public-key encryption (called, the hybrid-encryption scheme)
- We emphasize that any public-key encryption scheme can be used. It need not be the ElGamal Scheme. You can choose any encryption scheme that you prefer.
- The benefit of hybrid-encryption is that it allows us to combine two encryption scheme in a modular fashion.
- Suppose the public-key encryption scheme is provided by the triplet of algorithms

$$(\mathsf{Gen}^{(\mathsf{pub})}, \mathsf{Enc}^{(\mathsf{pub})}, \mathsf{Dec}^{(\mathsf{pub})})$$

# Hybrid Encryption II

- Suppose the private-key encryption scheme is provided by the triplet of algorithms

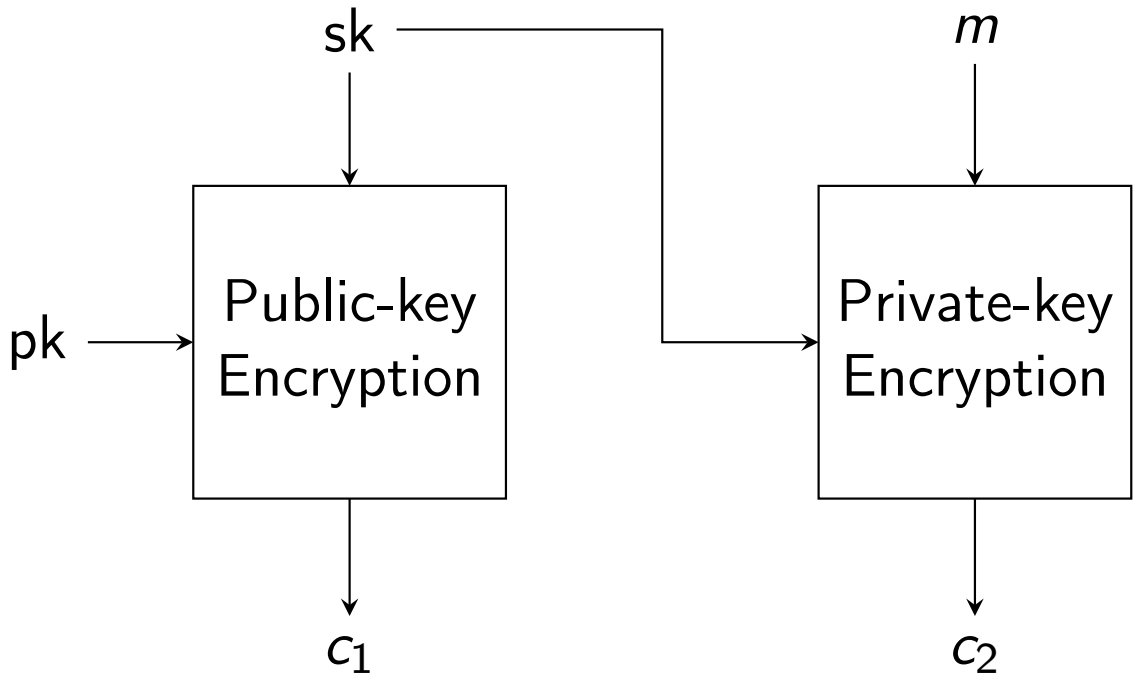
$$(\mathsf{Gen}^{(\mathsf{priv})}, \mathsf{Enc}^{(\mathsf{priv})}, \mathsf{Dec}^{(\mathsf{priv})})$$

- Now, we need to describe the hybrid-encryption scheme algorithms

$$(\mathsf{Gen}^{(\mathsf{hyb})}, \mathsf{Enc}^{(\mathsf{hyb})}, \mathsf{Dec}^{(\mathsf{hyb})})$$

# Hybrid Encryption III

Let us first draw a block-diagram for intuition purpose




- The secret-key sk will be encrypted by the public-key encryption
- The secret-key sk will be used to encryption the actual message $m$ using the private-key encryption

# Hybrid Encryption IV

**Generation Algorithm for Hybrid-Encryption.**

$\mathsf{Gen}^{(\mathsf{hyb})}()$:

1. Return $(\mathsf{pk}, \mathsf{trap}) = \mathsf{Gen}^{(\mathsf{pub})}()$

The receiver broadcasts pk and keeps trap safe with herself

# Hybrid Encryption V

**Encryption Algorithm for Hybrid-Encryption.**

$\mathsf{Enc}_{\mathsf{pk}}(m)$:

1. Generate $\mathsf{sk} = \mathsf{Gen}^{(\mathsf{priv})}()$
2. Encrypt the secret-key $c_1 = \mathsf{Enc}^{(\mathsf{pub})}_{\mathsf{pk}}(\mathsf{sk})$
3. Encrypt the message $c_2 = \mathsf{Enc}^{(\mathsf{priv})}_{\mathsf{sk}}(m)$
4. Return the cipher-test $(c_1, c_2)$

**Decryption Algorithm for Hybrid-Encryption.**

$\mathsf{Dec}_{\mathsf{trap}}(\widetilde{c}_1, \widetilde{c}_2)$:

1. Decrypt the secret-key $\widetilde{\mathsf{sk}} = \mathsf{Dec}^{(\mathsf{pub})}_{\mathsf{trap}}(\widetilde{c}_1)$
2. Return the decrypted the message $\widetilde{m} = \mathsf{Dec}^{(\mathsf{priv})}_{\widetilde{\mathsf{sk}}}(\widetilde{c}_2)$

# Transition Slide

In the remaining of the lecture, we shall transition to the introduction of RSA algorithm. We shall begin by understanding how to generate large prime numbers, because RSA algorithm needs access to large prime numbers. In the next lecture, we shall see how to test whether a number is a prime number.

# Generating Large Prime Numbers I

- To read results in this slide, assume $n = 1024$
- Our goal is to generate prime numbers that require $n$ bits in their binary representation.

  Clarification: The number 5, for example, can be represented as 101 in binary. So, the number 5 needs at least 3 bits in its binary representation. The number 5 can be written as 0101 or 00101. But, the number 5 does not need 4 or 5 bits for its binary representation. Note that a number needs $n$ bits in its binary representation if its $n$-bit binary representation begins with 1; otherwise not.
- Now, prime numbers that need $n$-bits in their binary representation are in the range $\left\{2^{n/2}, 2^{n/2}+2, \ldots, 2^n - 1\right\}$.
  To understand how large these numbers are, always remember that the total number of atoms in the universe is less than $2^{266}$

# Generating Large Prime Numbers II

- For our algorithms to be efficient, we need our algorithms to be polynomial in $n$, the number of bits needed to represent the numbers

# Generating Large Prime Numbers III

- For today's algorithm we shall assume that we are already provided with an algorithm IsPrime($x$) that tests whether the number $x$ is a prime number or not. We shall assume that this algorithm runs in time polynomial in $n$, where $x$ needs $n$-bits in its binary representation.
- For perspective, only very recently, we discovered a polynomial time deterministic algorithm for IsPrime($x$)
- Before the discovery of this algorithm, we knew of an efficient probabilistic algorithm that was correct with very high probability
- Despite the discovery of the polynomial time deterministic algorithm, we still use the probabilistic algorithm in practice, because the probabilistic algorithm is significantly faster!

# Generating Large Prime Numbers IV

- We shall use a very big result from mathematics: Prime Number Theorem. We write an intuitive version of this result.

**Theorem (Prime Number Theorem)**

*There exists a universal constant $c$ such that the total number of primes $< N$ is roughly*

$$c \cdot \frac{N}{\log N}$$

- Let us understand the intuition of this result. There are $N$ natural numbers $< N$. Of them $cN/\log N$ are primes. So, the "density of primes" is

$$\frac{cN/\log N}{N} = \frac{c}{n},$$

where $N = 2^n$. That is the density is "inverse-polynomial-in-$n$," the number of bits needed to represent $N$

# Generating Large Prime Numbers V

Let us write our algorithm

GenerateRandomPrime($1^n$):

1. For $i = 1$ to $t = n^2/c$:
   1. Generate a random number $x$ in the range $\{2^{n-1}, 2^{n-1}+1, \ldots, 2^n - 1\}$
   2. If IsPrime(x) then return $x$
2. Return $-1$ (to indicate failure)

We will see the justification of the value of $t$ during the analysis of our algorithm

# Generating Large Prime Numbers VI

**Intuition of the algorithm.**

- We shall attempt $t$ times to generate a prime number. If we fail in all of our $t$ attempts, then we declare failure (by returning $-1$)
- In each attempt, we generate a random number $x$ that needs $n$ bits for its binary representation. Observe that such a number lies in the range $\{2^{n-1}, 2^{n-1}+1, \ldots, 2^n - 1\}$.

  Clarification: Recall that we had mentioned that a number that needs $n$ bits in its binary representation begins with 1 in its binary representation. So, these number lie between (and including) the two numbers $A$ and $B$, where $A$ has binary representation $1\overbrace{00\cdots0}^{(n-1)\text{-times}}$ and $B$ has binary representation $1\overbrace{11\cdots1}^{(n-1)\text{-times}}$. That is, $A = 2^{n-1}$ and $B = 2^n - 1$.

  And we test this number $x$ whether it is a prime number or not. If find $x$ is a prime number in any of our attempts, then we return this value.

# Analysis of our Prime Number Generation Algorithm I

**First, let us find the number of choices for the number $x$.**

- Note that the size of the set

$$\left\{2^{n-1}, 2^{n-1}+1, \ldots, 2^n - 1\right\}$$

  is $(2^n - 1) - (2^{n-1}) + 1 = 2^{n-1} = N/2$.
- So, we have $N/2$ choices for the number $x$, where $N = 2^n$

# Analysis of our Prime Number Generation Algorithm II

**Secondly, let us find the number of prime numbers that require $n$-bits in their binary representation.**

- Applying the Prime Number Theorem, the number of prime numbers $< N = 2^n$ is given by

$$\frac{cN}{\log N} = \frac{cN}{n}$$

- Applying the Prime Number Theorem, the number of prime numbers $< (N/2) = 2^{n-1}$ is given by

$$\frac{c(N/2)}{\log(N/2)} = \frac{cN}{2(n-1)}$$

- So, the number of prime numbers in the range $\{2^{n-1}, 2^{n-1}+1, \ldots, 2^n - 1\}$ is given by

$$\frac{cN}{n} - \frac{cN}{2(n-1)} = cN\frac{n-2}{2n(n-1)} \approx cN\frac{1}{2n}$$

# Analysis of our Prime Number Generation Algorithm III

**Finally, let us compute the probability of finding a prime number in one attempt.**

- The probability of finding a prime number in one attempt is given by

$$\frac{\text{Number of prime numbers in the set} \{2^{n-1}, 2^{n-1}+1, \ldots, 2^n - 1\}}{\text{Size of the set} \{2^{n-1}, 2^{n-1}+1, \ldots, 2^n - 1\}}$$

- By our first two calculations, this expression evaluates to

$$\frac{cN/2n}{N/2} = \frac{c}{n}$$

- So, the probability that our algorithm finds a prime number in an attempt is $c/n$

# Analysis of our Prime Number Generation Algorithm IV

- The probability that the algorithm fails to find a prime in one attempt is

$$\left(1 - \frac{c}{n}\right)$$

- The probability that the algorithm fails to find a prime in two attempts is

$$\left(1 - \frac{c}{n}\right)^2$$

# Analysis of our Prime Number Generation Algorithm V

- Similarly, the probability that the algorithm fails to find a prime in $t$ attempts is

$$\begin{aligned}\left(1-\frac{c}{n}\right)^{t} &= \left(1-\frac{c}{n}\right)^{n^2/c} \\ &= \left[\left(1-\frac{c}{n}\right)^{n/c}\right]^{n} \\ &= \left[\frac{1}{e}\right]^{n} = \exp(-n)\end{aligned}$$

- Here we used the fact that $\lim_{x\to\infty}\left(1-\frac{1}{x}\right)^{x} = \frac{1}{e}$

# Analysis of our Prime Number Generation Algorithm VI

- So, the probability that our algorithm fails to find a prime number is $\exp(-n)$
- Recall that the number of atoms in the universe is less than $2^{266}$ or $\exp(185)$. This explains that the probability that our algorithm fails is extremely small!