# SC4010

# APPLIED CRYPTOGRAPHY

**Fermat Factorizaation Attack - Jun Sheng**

**EGImaal with elliptic curves - Haozheng**

# ATTACK ON 2048-BIT RSA

```python
def generate_two_close_primes():
    # Define the minimum gap between the two primes
    min_diff = 10**150

    # Using cryptographically secure RNG
    secure_rand = SystemRandom()

    # Generate a random number with 2048 bit length
    n = secure_rand.getrandbits(1023) | (1 << 1023) # force the first bit to 1

    # Find the next prime greater than or equal to n
    p = nextprime(n)

    # Find the next prime after (p + diff)
    q = nextprime(p + min_diff)

    # Compute the actual difference
    D = abs(p - q)

    return p, q, D

# Generate the primes
p, q, D = generate_two_close_primes()

# Print results
print("Prime 1 (p):", p)
print("Prime 2 (q):", q)
print("Difference (D):", D)
print("Bit length of p:", p.bit_length())
print("Bit length of q:", q.bit_length())

```

```python
# Checks if a number n is a perfect square. Return True if n is a perfect square, False otherwise.
def is_perfect_square(n):

    if n < 0:
        return False  # Negative numbers cannot be perfect squares in real numbers

    root = math.isqrt(n)
    return root * root == n # Return True or False
```

✓ 0.0s

```python
# Fermat's factorization
def fermat_factor(N):
    a = math.isqrt(N) + 1
    iters = 0

    while True:
        b2 = a * a - N

        if is_perfect_square(b2): # Check if b2 is perfect square
            b = math.isqrt(b2)
            q_found = a + b
            p_found = a - b
            return q_found, p_found, b, iters

        print("b^2 is not a perfect square.")
        a += 1
        iters += 1

q_found, p_found, b, iters = fermat_factor(N)
print("b:",b)
print("\nFactors found:")
print("q found =", q_found)
print("p found =", p_found)
print("\nIterations:", iters)
```

✓ 0.0s

# ATTACK ON 2048-BIT RSA

Modulus: N (public)
Two large prime factors: p & q (private)
Public Exponent: e (public)
Private Exponent: d (private)

- Strength of RSA comes from the computational difficulty of factoring the product of 2 large prime numbers (p and q)
- However, if the 2 prime numbers are close, we can easily find them using Fermat's Factorisation
- Objective is to find p and q (d can be derived if p and q are known)

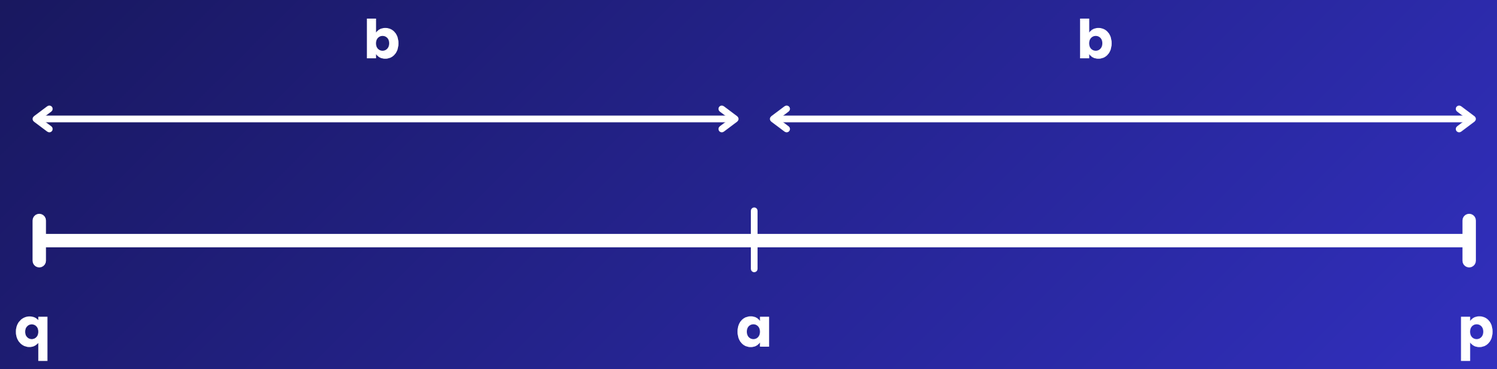
# ATTACK ON 2048-BIT RSA

$$N = p \times q$$

$$= (a + b)(a - b)$$

$$= a^2 - b^2$$

$$b^2 = a^2 - N$$



# ATTACK ON 2048-BIT RSA

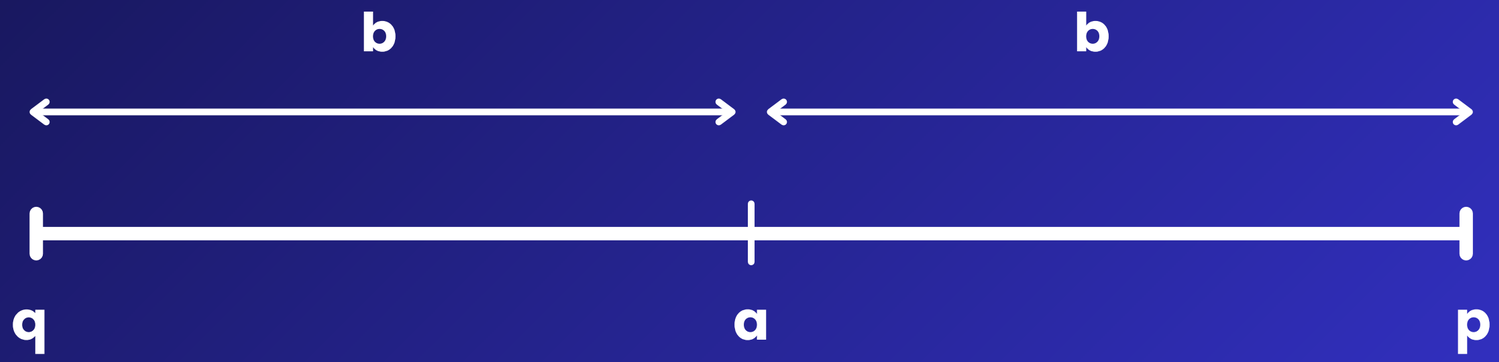
How do we find a?

a = (p+q)/2 = Arithmetic Mean (midpoint)

√N = √pq = Geometric Mean

*Geometric Mean always <= Arithmetic Mean.

→ Can use Geometric Mean to find a



# ATTACK ON 2048-BIT RSA

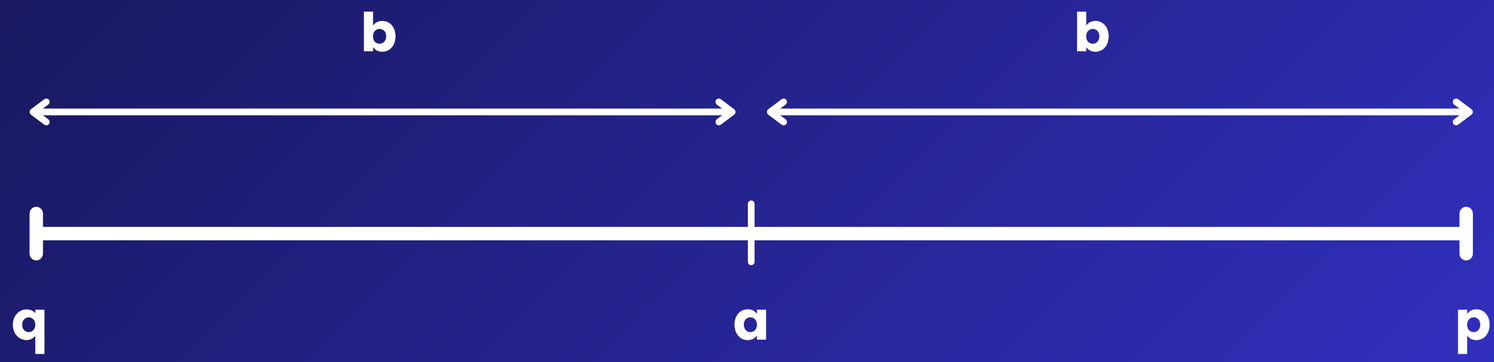
How do we find a?

a is always an integer

*All prime numbers (except 2) are odd

→ odd + odd = even

→ even / 2 = integer



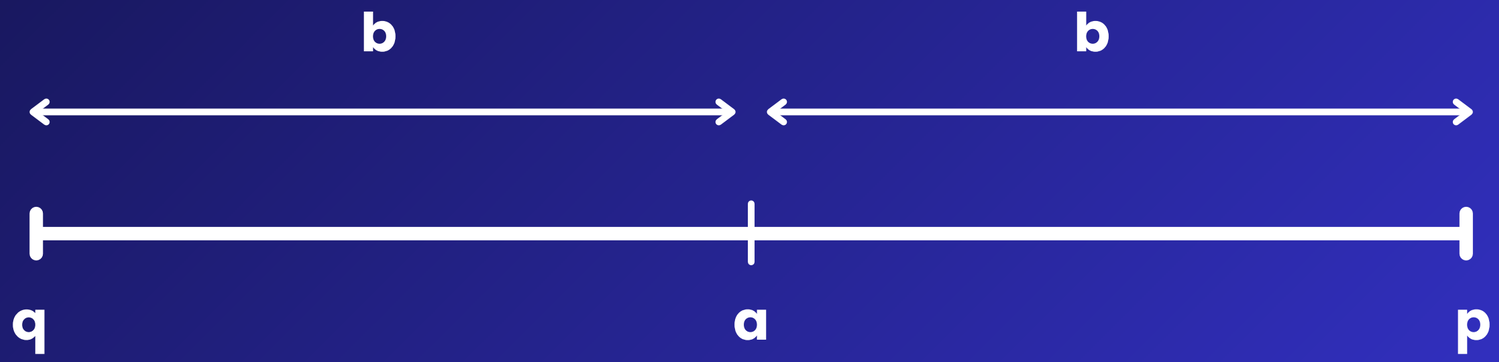
# ATTACK ON 2048-BIT RSA

$$N = p \times q$$
$$= (a + B)(a - b)$$
$$= a^2 - b^2$$
$$b^2 = a^2 - N$$

Core Logic

$\sqrt{N} = \sqrt{pq} <= a$,
which is an integer

brute force $\sqrt{N}$ to a and check that $b^2$ is a perfect square each time



# ELGAMAL WITH ECC

# ELGAMAL WITH ECC

```python
class ELGamal_with_ECC:
    def __init__(self, curve, base_point, n):
        self.curve = curve
        self.base_point = base_point
        self.n = n
        if not curve.point_on_curve(base_point):
            raise ValueError("Based point is not on curve!")

    def generate_key(self):
        private_key = secrets.randbelow(self.n - 1) + 1
        public_key = self.curve.scalar_point(private_key, self.base_point)
        return private_key, public_key

    def encrypt(self, message, public_key):
        k = secrets.randbelow(self.n-1) + 1
        cipher1 = self.curve.scalar_point(k, self.base_point)

        kA = self.curve.scalar_point(k, public_key)
        cipher2 = self.curve.add_points(kA, message)

        return cipher1, cipher2

    def decrypt(self, cipher1, cipher2, private_key):
        shared_secret = self.curve.scalar_point(private_key, cipher1)
        if shared_secret:
            neg_x_value = shared_secret[0]
            neg_y_value = -shared_secret[1] % self.curve.p
            neg_S = (neg_x_value, neg_y_value)
        else:
            neg_S = None
        M = self.curve.add_points(cipher2, neg_S)
        return M
```

```python
class ECC:
    def __init__(self, a, b, p):
        self.a = a
        self.b = b
        self.p = p

    def point_on_curve(self, point):
        if point is None:
            return True
        x = point[0]
        y = point[1]
        left = (y ** 2) % self.p
        right = (x ** 3 + (self.a * x) + self.b) % self.p
        return left == right

    def extended_gcd(self,a, b):
        if b == 0:
            return (a, 1, 0)
        gcd, x1, y1 = self.extended_gcd(b, a % b)

        x = y1
        y = x1 - (a // b) * y1

        return (gcd, x, y)

    def inverse(self, value_a, mod_value):
        gcd, x, y = self.extended_gcd(value_a, mod_value)
        if gcd != 1:
            raise ValueError("GCD != 1, there is no inverse value.")

        inverse_value = x % mod_value
        return inverse_value

    def add_points(self, point1, point2):
        if point1 is None:
            return point2
        if point2 is None:
            return point1

        point_1_x, point_1_y = point1
        point_2_x, point_2_y = point2

        if (point_1_x == point_2_x and point_1_y == point_2_y):
            # case where it is 2P
            m = (3 * (point_1_x ** 2) + self.a) * self.inverse((2* point_1_y) ,self.p)
        else:
            m = (point_2_y - point_1_y) * self.inverse((point_2_x-point_1_x) ,self.p)

        m = m % self.p

        point_3_x = (m**2 - point_1_x - point_2_x) % self.p
        point_3_y = (m * (point_1_x - point_3_x) - point_1_y) % self.p

        new_point = (point_3_x, point_3_y)
        return new_point

    def scalar_point(self, n, point):
        R = None
        Q = point
        while n:
            if n & 1:
                R = self.add_points(Q, R)
            Q = self.add_points(Q, Q)
            n >>= 1
        return R
```

# ELGAMAL WITH ECC - ALGORITHM

**Suppose bob wants to send alice an encrypted message (M)**

**Step 1 - Alice will first generate her Public and Private key:**

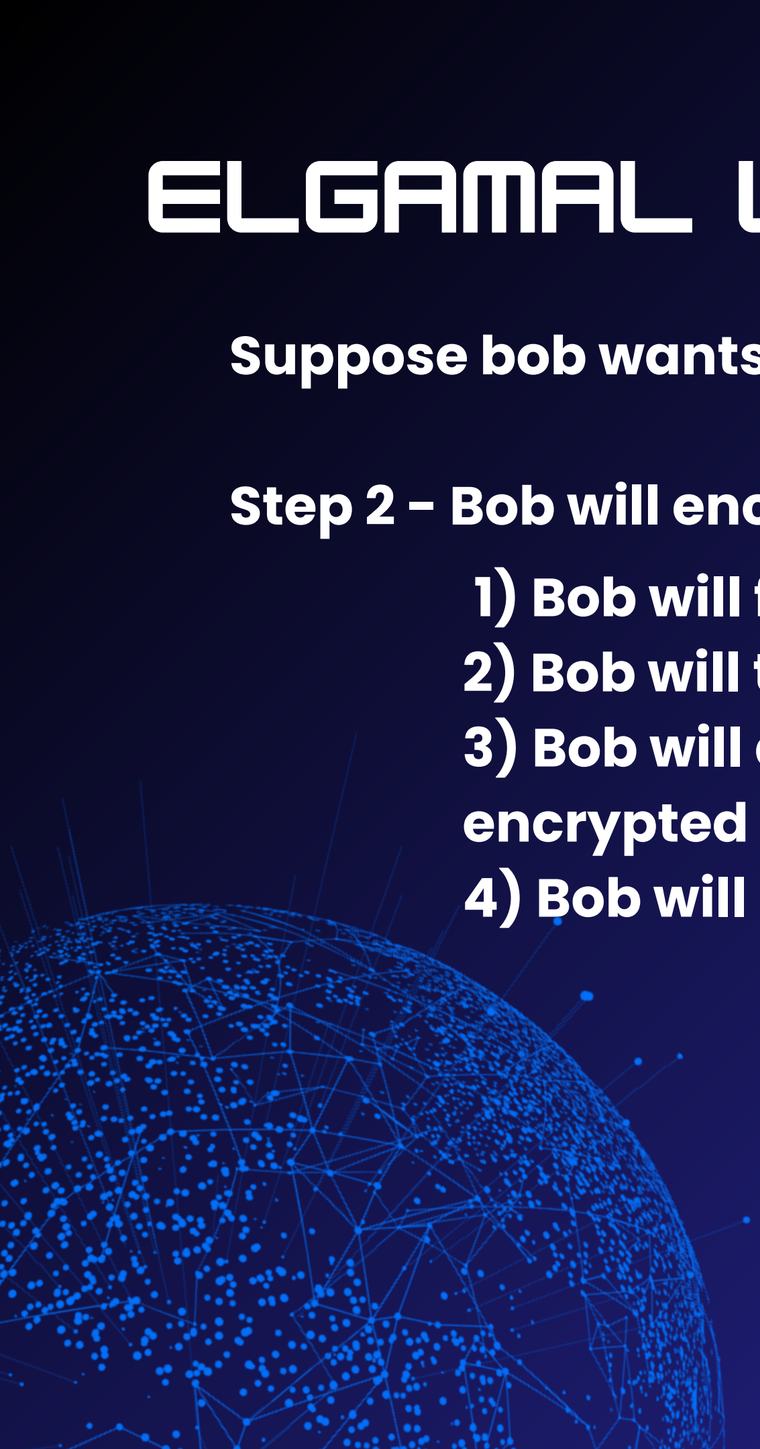
1) **Alice will first generate** $a$ **where** $a \in$ **[1, n-1]**
2) **Alice's Private key will be** $a$
3) **Alice's Public key will be** $A = a * p$ **where** $p$ **is a point on the *elliptic curve***

# ELGAMAL WITH ECC - ALGORITHM

**Suppose bob wants to send alice an encrypted message (M)**

**Step 2 – Bob will encrypt his message with Alice's Public key:**

1) **Bob will first generate a random** $k$
2) **Bob will then produce C1 =** $k * p$ **where** $p$ **is a point on the elliptic curve**
3) **Bob will also produce C2 =** $(k * A) + M$ **where M is the message to be encrypted**
4) **Bob will send Alice point (C1, C2)**



# ELGAMAL WITH ECC - ALGORITHM

**Suppose bob wants to send alice an encrypted message (M)**

**Step 3 - Alice will decrypt Bob's message:**

**1) Alice will generate S =** *a * C1*

**2) Alice can recover the message with** *C2 - S*

*Reason for why it works:*

$C1 = kP$

$C2 = (kA) + M = kaP + M$

$A = aP$

$S = akP$

$M = C2 - S = kaP + M - akP = kaP - kaP + M$

# ELGAMAL WITH ECC - EXAMPLE

**We have a curve such that** $y^2 = x^3 + 2x + 2 mod 17$
**Base point of (5, 1), n value of 19 and message mapped to point (16, 4)**

**Alice's Private key** *a = 7*
**Alice's Public key will be 7(5, 1) = (0,6)**

**Bob generate** *k = 11*
**C1 = 11(5,1) = (13, 10)**
**C2 = M + 11(0,6) = (16,4) + 11(0,6) = (9,1)**
**Bob sends ((13,10), (9,1)) to Alice**

**Alice computes S = 7(C1) = 7(13,10)**
**Message = (9,1) - 7(13,10) = (16,4) = M**

# DEMO WITH SECP256K1 CURVE


THANK
YOU