Aufgaben

32. Beweisen sie die Korrektheit der RSA Ver- und Entschl¨usselungsformel f¨ur (mi , n) = 1 und (mi , n) 6= 1.
Fall 1: ggT(mi, n) = 1
dann gilt z.z.: m' = m für $m' \equiv c^d(\mod n)$ wo $c \equiv m^e(\mod n)$

wir wissen das $ed \equiv 1(\mod \phi(n))$ was bedeutet, dass es ein k gibt so, dass: $ed = 1 + k * \phi(n)$ da m und n Teilerfremd sind

somit:
$m' \equiv (m^e)^d(\mod n)$
$m' \equiv m^{ed}(\mod n)$
$m' \equiv m^{1+k*\phi(n)}(\mod n)$
$m' \equiv m * (m^{\phi(n)})^k(\mod n)$
Nach dem kleinen Satz von Fermat gilt hier, dass $m^{\phi(n)} \equiv 1(\mod n)$ =>
$m' \equiv m * 1^k(\mod n)$
$m' \equiv m(\mod n)$

somit ist die Korrektheit bewiesen.

Fall 2: ggT(mi, n) $\neq$ 1
Erneut berechnen wir:
$m' \equiv (m^e)^d(\mod n)$

da m ein Teiler von p ist schreiben wir $m = k * p$ für ein $k \in \mathbb{Z}$
somit ist $m^e(\mod n) = (kp)^e = k^e * p^e$
wir wissen außerdem: $k^e * p^e \equiv 0(\mod p)$

nun:
$m' \equiv (m^e)^d(\mod n)$
$m' \equiv m^{ed}(\mod n)$
$m' \equiv (kp)^{ed}(\mod n)$

da $ed \equiv 1(\mod \phi(n))$ gilt $ed = 1 + k * \phi(n)$:
$m' \equiv (kp)^{1+k*\phi(n)}(\mod n)$

$kp * (kp)^{k*\phi(n)} \equiv (kp)^{1+k*\phi(n)}(\mod n)$

da kp ein Teiler von p ist gilt:

$kp * (kp)^{k*\phi(n)} \equiv 0(\mod p)$
$0 * (0)^{k*\phi(n)} \equiv 0(\mod p)$

da $n = p * q$ gilt:
$0 \equiv (kp)^{1+k*\phi(n)}(\mod n)$
$0 \equiv (kp)^{ed}(\mod n)$
$m' \equiv 0(\mod n)$

Somit ist die richtige Entschlüsselung von RSA unter der annahme ggT(mi, n) $\neq$ 1 wiederlegt.

Kleiner Satz von Fermat

---

33. Warum ist RSA in der bisherigen Beschreibung (sog. Textbook RSA) nicht INDCPA ? Wie wird mit RSA diese Sicherheitsstufe erreicht?

Wie bereits in der Vorlesung angesprochen wurde ist Textbook RSA nicht IND-CPA, da es deterministisch ist, s.h.: hat man einen Ciphertext c und soll bestimmen ob A oder B der zugehörige Plaintext ist, kann man bei einer Chosen Plaintext Attacke einfach beide mit dem Publickey verschlüsseln und überprüfen welches c_A/c_B = c.
Um dies zu verhindern muss man dafür sorgen, dass für denselben Plaintext nicht immer derselbe Ciphertext ensteht. Dafür gibt es mehrere Möglichkeiten, eine davon ist Padding mit OAEP (Optimal asymetric encryption padding)[1].
Das ganze funktioniert so, dass man zwei Masken erstellt, die erste XORed man mit dem Plaintext und die zweite XORed man mit dem Seed der ersten Maske, diese beiden mit den Masken bearbeiteten Teile konkateniert man dann und verschlüsselt diese mit RSA:

$$E_{RSA}((M1 \oplus m) + (M2 \oplus Seed_{M1})$$

Hierbei gibt es eine Funktion welche mit demselben Seed immer dieselbe Maske generiert, also z.B.: $MGF(Seed_{M1}) = M1$

Somit kommen bei zwei Verschlüsselugen desselben Plaintexts verschiedene Ciphertexte heraus und es ist einfach nach dem entschlüsseln, das ganze zu reversieren: da der Seed von Maske 2 $M1 \oplus m$entspricht und man mit dem dadurch gewonnen Seed von M1 die Nachricht wiederherstellen kann.
[1] Wikipedia 19.05.2024

---

34. Implementieren sie die Chosen Ciphertext Attacke 3 gegen RSA (Slide 157 der VO). Bedenken sie, dass sie dafür auch die Abbildung des Plaintexts auf numerische Blöcke realisieren müssen. Weiters wird Alice nur sinnvolle Dokumente unterschreiben!

Zuerst generieren wir unsere Keys:

```
def generate_keys(bits=1024):
    e = 65537
    p = number.getPrime(bits)
    q = number.getPrime(bits)
    n = p * q
    phi = (p - 1) * (q - 1)
    d = pow(e, -1, phi)
    return (e, n), (d, n)
```

vermutlich eine schlechte Quelle aber für habe ich nach einer kurzen Google suche 65537 aufgrund dieser Website gewählt: Is there a preferred size for e in an RSA cryptosystem? 22.05.2024
Nachdem es funktioniert und das ganze recht bliebt scheint bleibe ich dabei.

Jetzt braucht es noch eine funktion welche Plaintextblöcke signiert:

```
def sign(message, private_key):
    d, n = private_key
    message_int = int.from_bytes(message.encode('utf-8'), byteorder='big')
    signature = pow(message_int, d, n)
    return signature
```

Dann muss nurnoch die Attacke durchgeführt werden:

```
def chosen_ciphertext_attack(public_key, private_key):
    m1 = "Hello"
    m2 = "World"

    # In numerische Blöcke konvertieren
    m1_int = int.from_bytes(m1.encode('utf-8'), byteorder='big')
    m2_int = int.from_bytes(m2.encode('utf-8'), byteorder='big')
```

```
    # Berechnen von numerischen m3 für späteres überpfüfen
    n = public_key[1]
    m3_int = (m1_int * m2_int) % n

    # An Alice "senden"

    # Alice signiert m1 und m2
    s1 = sign(m1, private_key)
    s2 = sign(m2, private_key)

    # Alice "sendet" zurück

    # Eve berechnet signiertes m3 aus s1 und s2
    s3 = (s1 * s2) % n

    # Überprüfen ob m3^d = m1^d * m2^d
    signed_message_int = pow(s3, public_key[0], n) # m^e*d = m
    is_valid = signed_message_int == m3_int
    print(f"Forged signature valid: {is_valid}")

public_key, private_key = generate_keys()
chosen_ciphertext_attack(public_key, private_key)
```

Hierbei folgen wir genau dem Ablauf auf Slide 157 und überprüfen danach noch ob tatsächlich das richtige herauskommt in dem wir die Signatur mithilfe des Public keys reversieren (nachdem $m^{e*d} \equiv m(\mod n)$).

Zusatz

- Repository