Information Security Lab: Module 4

Lab Session Week 3: Coppersmith's Attack

Kenny Paterson, Jan Gilcher, Kien Tuong Truong

Your tasks

- For this week you have the following tasks
  - Recover a message from an RSA implementation that uses PKCS7 padding
  - Forge signatures against a key-store for RSA keys
  - Recover a message from an elliptic curve scheme over Z mod N
- You are expected to implement Coppersmith's attack using only LLL
- Do **not** copy implementations from the internet
- Do **not** use the .small_roots() method from SageMath

Task 0: RSA, the Padding-ton Bear

- In Week 2’s Exercise session, you’ve discussed why textbook (unpadded) RSA is not secure
- For this task, we have a padding scheme
- Try to recover the message despite the padding scheme!

$$(\ \text{message} \ \ \text{padding}\ )^e \bmod N$$

Task 0: PKCS7 Padding

- A message must be padded to about the same size of N

  e.g. |N| ~ 1024 bits = 128 B
- Let m be the message, we need to create k = |N| - |m| bytes of padding
- We simply encode k as a byte and repeat it k times
- We prepend a 0 byte before the message, to ensure that the padded message is not reduced modulo N

\x00	A 124-byte message	\x03\x03\x03

Task 1: Export-Grade Cryptography

- The server allows you to generate and export RSA private keys
- Think of a TPM (Trusted Platform Module)
- Generate keys for an identifier
- The server exports only p, one of the prime factors of N
- Unfortunately, the exported p is encrypted (with a key not known to you)
- Can you still somehow recover the private key and forge an RSA signature?

Task 1: RSA Signatures

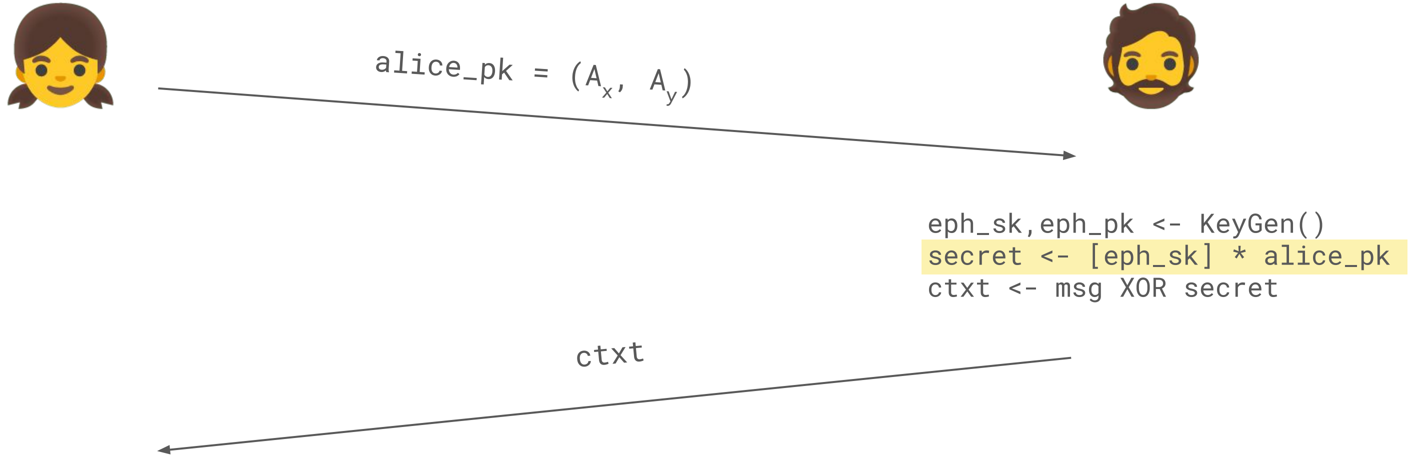
- A reminder of how RSA hash-then-sign signatures work
- Let (N, e) be a public key, and let d be the private key
- Sign(d, m):
    - h = SHA-256(m) mod N
    - $s = h^d \bmod N$
    - Return s
- Vfy(e, m, s):
    - h = SHA-256(m) mod N
    - $h' = s^e \bmod N$
    - if h = h’ then return 1 else 0

Task 2: Elliptic Curve Craptography

- Elliptic curves can also be defined over integers modulo N for N **not** prime, though some operations may break down
- If N = p*q, then it works well most of the times... (when can it break?)
- ...but we can't compute i-th roots (why?)

Task 2: Elliptic Curve Craptography

- Let’s do some cryptography: we first do a Diffie-Hellman and then we use the resulting shared secret point to encrypt a message



Polynomials in SageMath

- SageMath has 3 (!) different polynomial implementations for $Z_N[x]$
  - libSINGULAR: supports multivariate polynomials
  - FLINT: fast, but only univariate and N small (< 2^64)
  - NTL: slower than FLINT, only univariate but N arbitrarily large
- We suggest using libSINGULAR (later we see why)
- https://doc.sagemath.org/html/en/reference/polynomial_rings/sage/rings/polynomial/multi_polynomial_libsingular.html

Polynomials in SageMath

- libSINGULAR polynomial rings can be created explicitly or implicitly
- We will often be working on polynomials over the ring Zmod(n) or the ring of integers ZZ

```
# Univariate polynomial, explicit init
R = PolynomialRing(Zmod(n), ‘x’, implementation=’libSINGULAR’)

# Univariate polynomial, implicit init (see second param)
R = PolynomialRing(Zmod(n), 1, ‘x’)

# Multivariate polynomial
R = PolynomialRing(Zmod(n), [‘x’, ‘y’])
```

Polynomials in SageMath

- Afterwards, you can define polynomials and play around with them

```
R = PolynomialRing(Zmod(23), 1, ‘x’)
x = R.gen() # Define the variable x
P = x**3 + 17*x + 2
print(P(3)) # prints 11
print(10 * P) # 10x^3 + 9x + 20
print(P**2) # x^6 + 11*x^4 + 4*x^3 + 13*x^2 + 22*x + 4
```

Extracting Coefficients

- The method coefficients() does **not** give you zero coefficients
- You can use e.g. P.coefficient(x**3) to obtain the coefficient for x**3
- Caveat: this does not return an integer, but a **constant polynomial**

```
P = x**3 + 17*x + 2
print(P.coefficients()) # [1, 17, 2]
print(P.coefficient(x**3)) # 1
print(P.coefficient(x**2)) # 0
```

Coefficients Matrices with Sequence

- An alternative to extract a coefficient matrix is the Sequence class
  - Only supports libSINGULAR polynomials
- See the documentation

```
R = PolynomialRing(ZZ, 1, ‘x’)
x = R.gen()
S = Sequence([], R)
S.append(x**3 + x)
S.append(x + 3)
S.append(3*x**3 + 1)
matrix, monomials = S.coefficient_matrix(sparse=False)
```

Coefficients Matrices with Sequence

```
# matrix is a (dense) matrix over the base ring of R (i.e. ZZ)
print(matrix)
# [1 1 0]
# [0 1 3]
# [3 0 1]
print(matrix.LLL()) # Dense integer matrices have the LLL method

# monomials is a column vector with entries in R
print(monomials)
# [x^3]
# [x]
# [1]
```

Changing the base ring

- You may often want to change the base ring of polynomials/matrices
    - LLL is only defined for integer or rational matrices
- To do so, you can use the change_ring() method
    - Works for both polynomials and matrices

```
R = PolynomialRing(Zmod(23), 1, ‘x’)
P = x**3 + 17*x + 2
print(P.base_ring()) # Ring of integers modulo 23
P_zz = P.change_ring(ZZ)
print(P_zz.base_ring()) # Integer Ring
```

Finding roots of polynomials

- One can find roots of polynomials using ideals and varieties
  - You don’t need to exactly understand what this means, take it as a black-box

```
R = PolynomialRing(ZZ, 1, ‘x’)
x = R.gen()
# P is a polynomial in ZZ
P = x**5 - 8 * x**4 - 25 * x**3 + 44 * x**2 + 60*x
# Only defined over fields, so we have to change ring
I = ideal(P.change_ring(QQ))
print(I.variety(ring=ZZ))    # Give back the roots in ZZ
# [{x: 0}, {x: 10}, {x: 2}, {x: -1}, {x: -3}]
```

A few tips

- Coppersmith requires **monic** polynomials (i.e. the coefficient of the highest-degree term must be 1)
- Coppersmith may perform better than the provable bound: smaller matrices may work as well!