

What did you learn?

密码算法的优化方法

Number 21: How does the CRT method improve performance of RSA?

The Chinese Remainder Theorem (CRT) 中国剩余定理: 当有俩等式 $x \equiv a \pmod{M}, x \equiv b \pmod{N}$ 时, 如果 M, N 互质, 那么可以通过 CRT 计算出 x 的唯一解。

一些题外话: RSA方案中的主要运算是模幂(modular exponentiation) $M = C^d \pmod{N}$ 。一次模幂云散在改进的状态下, 可以使用 $(h-1)$ 次乘法和 $(t-1)$ 次平方操作 (t 是指数的位长度, h 是汉明权重)。乘法和平方平均次数为 $t + t/2 - 1$ 。进一步改进: 我们可以使用 binary exponentiation algorithms 或者 window methods。对于后者, 我们一次处理 w 位指数, 这种方法, 我们仍然需要 t 次平方, 但乘法次数平均减少到 t/w 。通过采用滑动窗口方法, 乘法次数可以降低到平均 $t/(w+1)$ 次。

开始正文: CRT 用于 RSA 解密的情况。因此, 我们正在考虑私钥操作, 这意味着我们可以访问私钥, 从而可以访问模数 $N = pq$ 的因式分解。如果我们假设我们想要解密一条消息, 那么我们的目标是计算 $M = C^d \pmod{N}$ 。首先我们计算:

$$M_p = C^d \pmod{p} = C^{d \pmod{p-1}} \pmod{p} \quad M_q = C^d \pmod{q} = C^{d \pmod{q-1}} \pmod{q}$$

此计算需要对 512 位取模和 512 位指数求幂, 因为 p 和 q 是 512 位。这比用 1024 位数字进行单次取模求幂要快。

使用 CRT 可以根据 M_p, M_q 计算出 M :

$$M = M_p + p \cdot (p^{-1} \pmod{q}) \cdot (M_q - M_p) \pmod{N}$$

另一种表示方法:

$$T = p^{-1} \pmod{q} \quad \text{作为私钥 } U = (M_q - M_p)T \pmod{q} \quad M = M_p + U \cdot p$$

$$\text{p.s.: } q^{-1} \pmod{p} \cdot q + p^{-1} \pmod{q} \cdot p = 1 \pmod{N}$$

Number 22: How do you represent a number and multiply numbers in Montgomery arithmetic?

这个连问题都看不懂了属于是

1. Secure and Efficient

虽然密码学的目标是设计高度安全的密码协议, 但还必须能够有效地实现这些协议, 以便可以一遍又一遍地执行这些协议, 而不会减慢用户的速度, 例如在线购物或转账通过网上银行。因此, 需要采取措施最大限度地减少这些协议中使用的每种算法的计算成本。此类算法中使用的最昂贵的运算之一是对某些 $n > 0$ 进行 reduction of integers modulo, 因为它实际上是除法运算。

2. The Cost of Modular Reductions

假如我们知道 $a \pmod{n}, b \pmod{n}$, 现在想计算 $ab \pmod{n}$ 。我们可以计算 ab , 然后再对 n 取模。一次乘法倒是还好, 但是多次这样的计算成本会很高, 比如 $c = m^e$, 需要迭代 e 次, 每次都要取模, e 可不是一个小数目。一个解决办法是先把乘法算完了最后模。但这样的话, 我们必须得存储 (并计算) 非常非常大的数字。肿么办?

3. "Montgomery Space"

Montgomery 提出了一种方法, 可以在不进行模运算的情况下进行乘法。这种方法是通过引入一个称为 Montgomery 空间的新空间来实现的。在这个空间中, 我们可以进行乘法, 而不需要进行模运算。然后, 我们可以将结果转换回普通的空间, 而不需要进行模运算。这种方法的关键是选择一个称为 Montgomery 模数的特殊模数。这个模数是一个方便的模数, 通常是 2 的幂。

为什么这个模数是方便的呢? 因为我们可以通过移位来实现模运算。假设我们有一个数 x , 与特殊模数 $r = 2^k$, 将 x 变为二进制串

- 计算 $x \pmod{r}$, 只需要保留 x 最低的 k 位
- 计算 xr , 只需将 x 左移动 k 位
- 计算 x/r , 只需将 x 右移动 k 位

4. Montgomery Algorithm

a, b, n, r , 其中 $\gcd(n, r) = 1$, 且 $r > n$, 我们可以计算 $ab \pmod{n}$:

1. 通过扩展欧几里得算法计算 $rr^{-1} = 1 + nn'$

这步很快的。而且因为 r 是 2 的幂, n 是奇数, 所以 $\gcd(n, r) = 1$, 所以肯定有解

2. 计算 $\bar{a} = ar \pmod{n}$ 与 $\bar{b} = br \pmod{n}$

$a \times r$ 这步也很快, 因为 r 是 2 的幂, 所以只需要移位 \pmod{n} 这步开销比较大, 甚至比 naive 的算法还要大, 所以本算法更适用于少数 multiplier 多次相乘的情况 (e.g. $a^m b^{m'}$), 这样结果可以多次重复使用步骤 2, 3 的结果

3. 计算 $u = \bar{a}\bar{b} \pmod{n}$

每一步都很快

- $t \leftarrow \bar{a}\bar{b}$
- $u \leftarrow (t + (n't \pmod{r})n)/r$

- if $u \geq n$ then $u \leftarrow (u - n)$
 - output u
4. Multiply u by r^{-1} and reduce modulo n

同 2

正确性:

$$u = abr \bmod n = arbr^{-1} \bmod n = tr^{-1} \bmod n = trr^{-1}/r \bmod n = t(1 + nn')/r \bmod n = ((t + nn't)/r + mn) \bmod n = (t + (n't + mr)n)/r \bmod n$$

可以发现没有 $\bmod n$ 的操作。

因为 $n > r > 0$, 可以得出 $n^2 < rn \Rightarrow n^2 + rn < 2rn \Rightarrow (n^2 + rn)/r < 2n$ 所以 $(t + (n't \bmod r)n)/r < 2n$, 因此不需要 $\bmod n$, 只需要看情况减去 n 即可。

Number 23: Write a C program to implement Montgomery arithmetic.

自己写了一个

```
#include <stdio.h>

// 求模逆
int mod_inverse(int a, int m) {
    int m0 = m, t, q;
    int x0 = 0, x1 = 1;
    if (m == 1) {
        return 0;
    }
    while (a > 1) {
        q = a / m;
        t = m;
        m = a % m, a = t;
        t = x0;
        x0 = x1 - q * x0;
        x1 = t;
    }
    if (x1 < 0) {
        x1 += m0;
    }
    return x1;
}

// 计算Montgomery reduction
int montgomery_reduction(int a, int b, int n, int r, int r_inv) {
    int n_apo = (r*r_inv - 1)/n;

    int a_bar = a * r % n;
    int b_bar = b * r % n;
    int t = a_bar * b_bar;
    int m = t * n_apo % r;
    int u = (t + m * n) / r;

    if (u >= n) {
        return u - n;
    } else {
        return u;
    }
}

int main() {
    int a = 123; // 输入参数a
    int b = 456; // 输入参数b
    int n = 1009; // 模数n
    int r = 1 << 10; // 选择r为2的8次方, 且r > n
    int r_inv = mod_inverse(r, n);

    int u = montgomery_reduction(a, b, n, r, r_inv);
    int result = u * r_inv % n;
    printf("Montgomery reduction of %d and %d modulo %d is: %d\n", a, b, n, result);

    return 0;
}
```

Number 24: Describe the binary, m-ary and sliding window exponentiation algorithms.

继续考虑一些计算模幂 (modular exponentiations) 的不同方法, 例如计算: $X^E \bmod N$ 。

1. Binary

The binary modular exponentiation 举例:

$$4^{13} \bmod 497 = 4^{(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)} \bmod 497$$

$$= (4^{2^3})^1 \bmod 497 \times (4^{2^2})^1 \bmod 497 \times (4^{2^1})^0 \bmod 497 \times (4^{2^0})^1 \bmod 497$$

可以看我

出前一个基数的幂是后一个基数幂的平方，所以我们假设第一个基数为 b_1 ，第二个为 b_2 ，依次类推

$$4^{13} \bmod 497 = 4^{(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)} \bmod 497$$

$$= (4^{2^3})^1 \bmod 497 \times (4^{2^2})^1 \bmod 497 \times (4^{2^1})^0 \bmod 497 \times (4^{2^0})^1 \bmod 497$$

$$= (b_3^2)^1 \bmod 497 \times (b_2^2)^1 \bmod 497 \times (b_1^2)^0 \bmod 497 \times (b_1)^1 \bmod 497$$

现在我

们合并一下，即每个先乘以低位mod的结构，再mod

$$4^{13} \bmod 497 = 4^{(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)} \bmod 497$$

$$= (4^{2^3})^1 \bmod 497 \times (4^{2^2})^1 \bmod 497 \times (4^{2^1})^0 \bmod 497 \times (4^{2^0})^1 \bmod 497$$

$$= (b_3^2)^1 \bmod 497 \times (b_2^2)^1 \bmod 497 \times (b_1^2)^0 \bmod 497 \times (b_1)^1 \bmod 497$$

$$= ((b_3^2)^1 \times \{ (b_2^2)^1 \times [(b_1^2)^0 \times (b_1)^1 \bmod 497 \bmod 497] \bmod 497 \} \bmod 497 \bmod 497$$

我们再优化一下，指数等于 0 是没有必要乘的，但是我们仍然需要计算基数，方便下一个直接平方

$$4^{13} \bmod 497 = 4^{(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)} \bmod 497$$

$$= (4^{2^3})^1 \bmod 497 \times (4^{2^2})^1 \bmod 497 \times (4^{2^1})^0 \bmod 497 \times (4^{2^0})^1 \bmod 497$$

$$= (b_3^2)^1 \bmod 497 \times (b_2^2)^1 \bmod 497 \times (b_1^2)^0 \bmod 497 \times (b_1)^1 \bmod 497$$

$$= ((b_3^2)^1 \times \{ (b_2^2)^1 \times [(b_1^2)^0 \times (b_1)^1 \bmod 497 \bmod 497] \bmod 497 \} \bmod 497 \bmod 497$$

$$= ((b_3^2)^1 \times \{ (b_2^2)^1 \times [1 \times (b_1)^1 \bmod 497 \bmod 497] \bmod 497 \} \bmod 497 \bmod 497$$

其实上面每次算基数时，我们仍然可以mod 497

$$4^{13} \bmod 497 = 4^{(1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0)} \bmod 497$$

$$= (4^{2^3})^1 \bmod 497 \times (4^{2^2})^1 \bmod 497 \times (4^{2^1})^0 \bmod 497 \times (4^{2^0})^1 \bmod 497$$

$$= (b_3^2)^1 \bmod 497 \times (b_2^2)^1 \bmod 497 \times (b_1^2)^0 \bmod 497 \times (b_1)^1 \bmod 497$$

$$= ((b_3^2)^1 \times \{ (b_2^2)^1 \times [(b_1^2)^0 \times (b_1)^1 \bmod 497 \bmod 497] \bmod 497 \} \bmod 497 \bmod 497$$

$$= ((b_3^2 \bmod 497)^1 \times \{ (b_2^2 \bmod 497)^1 \times [1 \times (b_1 \bmod 497)^1 \bmod 497 \bmod 497] \bmod 497 \} \bmod 497 \bmod 497$$

每一步都有两个操作

1. (假如幂是 1 执行，是 0 直接执行第二步): 基数(记为 B) 乘前一位的结果 (记为 R)
2. 基数 B 平方且 mod n 为下一次作准备 $B^2 \bmod n$

2. m-ary

m 进制与 Binary 相似，后者是把每个元素看作一个 bit，而前者是看作 m 个 bit，即一共 $M = 2^m$ 个元素。Binary 可以认为是 $M = 2$ 的 1-ary。

工作原理：首先我们为 $i = 1$ to $2^m - 1$ 的所有 X^i 计算一个查找表。然后，我们将指数 E 变为以 M 为底。然后，对于 E 中的每个“项”，我们只需从表中查找适当的值，然后我们不加倍，而是移 m 位。

例如：\$m=3, M=2^3=8, E=35=(43)_8, X^{(43)_8} = X^{(48)} \cdot X^{(3)} = X^{(42 \cdot 3)} \cdot X^{(3)}\$ 没看懂为啥是移 m 位 我的理解是连续三次平方，Binary 是平方一次 原文：Then, we work our way through the exponent E (in base M). Then, for each 'term' in E we simply look up the appropriate value from our table, and then instead of doubling we shift by m .

与 Binary 相比，这意味着更多的预计算（即计算更多的 X 次幂），因此需要更多的空间，但作为回报，可以进行更少的乘法。

3. Sliding Window

是对 m -ary 的一次优化

但我连 m -ary 都没看懂...

假设 $m=4, E=22=(0,0,0,1,0,1,1,0)_2=(1,6)(2^4)_2$

然后我们可以使用 4-ary，但如果我们“重新定位”窗口，也许可以做得更好：毕竟只有 3 个位置为了 1，并且它们都在范围为 4 的窗口内。如果我们提前知道这一点，我们就可以在该位置应用我们的查找表，因此只需要进行一次查找。

$X^{22} = X^{16} \cdot X^6 = X^{2^4} \cdot X^6 = X^{11} \cdot X^2$ 这样的话，我们需要利用预先计算的 X^{11} ， X^2

所以我们首先将 X 转为二进制，这样需要更多的预计算，但是如果 X 足够稀疏，那么我们可以通过滑动窗口法可以节省不少时间。

Number 25: Methods for modular reduction using "special" primes that define $GF(p)$ and $GF(2^n)$

1. Pseudo-Mersenne Prime Reduction

modulo operation 使用量很大，但是不好操作，之前的 Montgomery representation (22 章) 是一个优化方法，现在我们来看看另一个优化方法——Pseudo-Mersenne Prime reduction

首先看个定义：**Definition:** A prime number p is called a **Pseudo-Mersenne Prime** if p can be written in the form:

$$p = b^n - c$$

where

$$0 < |c| < 2^{\lfloor n/2 \rfloor}$$

一般情况下, $b = 2$, n 一般取一个字长 (word size) 32/64 bits

从定义可以推出:

$$p \equiv b^n - c \equiv 0 \pmod{p} \quad b^n \equiv c \pmod{p}$$

因此给定一个 k bits 的整数 z , 我们令 z' 为其最低 (Least Significant) n bits, z'' 为其最高 (Most Significant) $k - n$ bits, 即 $z = z''2^n + z'$, 我们可以将 $z \pmod{p}$ 重写为:

$$z \equiv z''2^n + z' \equiv z''c + z' \pmod{p}$$

递归地利用这种方法, 很容易就能把 z 模到 \mathbb{Z}_p 上。同时我们可以注意到以下几点:

1. z' 和 z'' 可以很容易地通过移位操作得到
2. 因为 c 是在一个字长内选择的, 所以 $z''c$ 可以通过乘法高效算出
3. 每次迭代可以将 k bits 缩小到 $\max(k - n + w, n)$ bits, 其中 w 是字长

所以一般来说, 当模数是伪梅森素数时, modulo 可以非常有效地完成, 因为它只需要移位、加法和乘法。

然而, 使用这种方法的缺点也很明显, 因为这通常需要多方使用固定的参数配置 (fixed setup), 这可能会导致互动和安全方面的问题

看过了素数域 $GF(p)$, 接下来我们看一下另一个有限域 $GF(2^n)$, 一般是三元 (trinomial) 或者五元 (pentanomial) 的多项式 (以下以三为例)

我知道“元”这个说法不对

这个想法和上面的素数域很像, 假如我们有一个多项式

$$f(x) = x^n + x^t + 1$$

其中 $0 < t < n/2$, 那么和上面很像, 可以得到

$$x^n \equiv x^t + 1 \pmod{f(x)}$$

如果有一个大于 n 度 (degree) 的多项式 $z(x)$, 我们可以将其分解为

$$z(x) = z''(x)x^n + z'(x)$$

和上面很像, 其中 $z'(x)$ 是 $z(x)$ 的最低 $n - 1$ 位, $z''(x)$ 是剩下的位。所以我们可以用上面的方案将 $z(x)$ 模到 $f(x)$ 上。

$$z(x) \equiv z''(x)x^n + z'(x) \equiv z''(x)(x^t + 1) + z'(x) \pmod{f(x)} \equiv z''(x)x^t + z''(x) + z'(x) \pmod{f(x)}$$

由于 t 是一个“小”数, 所以这个操作是非常高效的

2. Another optimization

这是多项式标准的 reduction 流程: [standard reduction](#) 现在介绍一个 $\pmod{f(x)}$ 优化方法:

假设:

$$z(x) = a_mx^m + a_{m-1}x^{m-1} + \dots + a_1x^1 + a_0x^0 \text{ 则 } z''(x) = a_mx^m + a_{m-1}x^{m-1} + \dots + a_{n+1}x^{n+1} + a_nx^n \quad f(x) = x^n + x^t + 1$$

假如我们想要 reduce a_ix^i , 会出现两者情况:

- $a_i = 0$: 什么也不会发生
- $a_i = 1$: x^t 和 1 对应的两项 a_{i-n+t}, a_{i-n} 加1

这个算法可以写成一个形式化的算法

```
INPUT: z(x)
OUTPUT: z(x)

1. for i=m to n by -1
2. {
3.     a_{i-n+t} += ai
4.     a_{i-n} += ai
5. }
6. return z(x)
```

这个算法在软件层面进步一般, 但是在硬件层面有很大的优势, 因为它只需要更新 $z(x)$ 且不需要额外存储。另一个优点是这样的代码只需要 $0 < t < n$ 并且可以在恒定时间内执行。

Number 26: Describe the NAF scalar multiplication algorithm.

NAF 标量乘法算法是标量乘法的增强形式，其中使用非相邻形式（**Non-Adjacent Form**, NAF）表示减少了算法的预期运行时间。

这里的标量乘法是指在椭圆曲线密码学中的标量乘法，即 $Q = k \cdot P$ ，其中 k 是一个整数， P 是一个点。一般是倍点运算与点加运算的组合：

point doubling and point addition

把 k 表示为二进制形式，即 $k = \sum_{i=0}^{n-1} k_i 2^i$ ，其中 $k_i \in \{0, 1\}$ ，我们可以得到 Q ：

```
// 从右到左
INPUT: P, k={k_{n-1},...,k_1,k_0}
OUTPUT: Q=k*P
  Q←∞
  for i from 0 to t-1 do
    if k_i=1 then Q←Q+P
    P←2P
  return Q

// 从左到右
INPUT: P, k={k_{n-1},...,k_1,k_0}
OUTPUT: Q=k*P
  Q←∞
  for i from t-1 to 0 do
    Q←2Q
    if k_i=1 then Q←Q+P
  return Q
```

由于一个 k bits 的字符串中 1 的数量平均下来是一半，所以大概需要 $k/2$ 次点加运算与 k 次倍加运算。

$$\frac{k}{2} \cdot A + k \cdot D$$

咱们现在就是要优化这个算法，使得点加运算的次数尽可能少。

背景介绍：In 1951, Booth proposed a new **scalar binary representation called signed binary method** and later Rietweisner proved that every integer could be uniquely represented in this format . More particularly, If $P = (x, y) \in E(\mathbb{F}_q)$ then $-P = (x, x + y)$ if \mathbb{F}_q is a binary field, and $-P = (x, -y)$ if \mathbb{F}_q has characteristic > 3 . Thus subtraction of points on an elliptic curve is just as efficient as addition.

优化方法就是使用 a particularly useful signed digit representation is the non-adjacent form (NAF)。NAF 就是将一个 k 的表示为， $k = \sum_{i=0}^{l-1} k_i \cdot 2^i$ ，其中 $k_i \in \{0, 1, -1\}, k_{l-1} \neq 0$ ，且**没有相邻的非 0**。NAF 的长度是 l 。

NAF 有如下属性：

- 1. k 有唯一的 NAF 表示： $NAF(k)$
- 2. $NAF(k)$ 在 k 的任何有符号数字表示中具有最少的非零数字。
- 3. $NAF(k)$ 的长度比 k 的二进制长度最多多1。
- 4. 如果 $NAF(k) = l$ ，则 $\frac{2^l}{3} < k < \frac{2^{l+1}}{3}$
- 5. 所有长度为 l 的 NAF 中非零数字的平均密度约为 1/3

$NAF(k)$ 可以用如下方法得到：

```
INPUT: A positive integer k
OUTPUT: NAF(k)
  i←0
  while k>=1 do
    if k is odd then
      k_i←2-(k mod 4)
      k←k-k_i
    else
      k_i←0
      k←k/2
      i←i+1
  return all k_i
```

C 语言实现：

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

void naf_conversion(int k) {
    int naf[100]; // 用于存储NAF表示形式
    int i = 0;

    while (k != 0) {
        if (k % 2 == 1) {
            naf[i] = 2 - (k % 4);

```

```

        k = k - naf[i];
    }
    else {
        naf[i] = 0;
    }
    k = k / 2;
    i++;
}

printf("The NAF representation of the input is: ");
for (int j = i - 1; j >= 0; j--) {
    printf("%d ", naf[j]);
}
printf("\n");
}

int main() {
    int k;
    printf("Enter an integer to convert to NAF format: ");
    scanf("%d", &k);

    naf_conversion(k);

    return 0;
}

```

重点来了! 咱们修改从左到右的标量乘法算法, 使得其可以使用 NAF 表示:

```

INPUT: P, k
OUTPUT: Q=k*P
Based on previous algorithm compute NAF(k)
Q ← ∞
for i from l-1 to 0 do
    Q ← 2Q
    if k_i=1 then Q ← Q+P
    if k_i=-1 then Q ← Q-P
return Q

```

基于 NAF 的第三个和第五个属性, NAF 标量乘法算法的预期运行时间大约为

$$\frac{k}{3} \cdot A + k \cdot D$$