A New Algorithm for Inversion mod p^k

Çetin Kaya Koç
koc@cs.ucsb.edu
Department of Computer Science
University of California Santa Barbara

May 11, 2017

Abstract

A new algorithm for computing $x = a^{-1} \pmod{p^k}$ is introduced. It is based on the exact solution of linear equations using p-adic expansions. It starts with the initial value $c = a^{-1} \pmod{p}$ and iteratively computes the digits of the inverse $x = a^{-1} \pmod{p^k}$ in base p. The mod 2 version of the algorithm is significantly more efficient than the algorithm given by Dussé and Kaliski for computing $a^{-1} \pmod{2^k}$.

1 Introduction

Hardware and software realizations of public-key cryptographic algorithms require implementations the multiplicative inverse mod p (prime) or n (composite). When the modulus is prime, we can compute the multiplicative inverse using Fermat's method as $a^{-1} = a^{p-2} \pmod{p}$. When it is composite, we can use Euler's method to compute the multiplicative inverse as $a^{-1} = a^{\phi(n)-1} \pmod{n}$, provided that we know $\phi(n)$. On the other hand, the extended Euclidean algorithm works for both prime and composite modulus

$$\begin{array}{rcl} (u,v) & \leftarrow & \mathrm{EEA}(a,n) \\ u\cdot a - v\cdot n & = & 1 \\ a^{-1} & = & u \pmod{n} \end{array}$$

The classical extended Euclidean algorithm requires division operations, which is not preferred for fast implementations. On the other hand, the binary extended Euclidean and Kaliski's algorithms use shift, addition and subtraction operations [3, 5, 6]. We must note however that all of these inversion algorithms are variants of the classical Euclidean algorithm for computing the greatest common divisor of two integers $g = \gcd(a,n)$.

2 Inversion mod 2^k

The Montgomery multiplication algorithm is introduced by Peter Montgomery [4] in 1985. It computes the product $c = a \cdot b \cdot r^{-1} \pmod{n}$ for an arbitrary modulus n, without actually performing any mod n reductions. Interestingly, the algorithm does not directly need $r^{-1} \pmod{n}$, but it requires another quantity n' which is related to it. The steps of the classical Montgomery multiplication algorithm are given below.

```
function Montgomery($a, b$)
input: $a, b, n, r, n'$
output: $u = a \cdot b \cdot r^{-1} \bmod n$
1:  $t \leftarrow a \cdot b$
2:  $m \leftarrow t \cdot n' \pmod{r}$
3:  $u \leftarrow (t + m \cdot n)/r$
4:  if $u \geq n$ then $u \leftarrow u - n$
5:  return $u$
```

None of the steps of the Montgomery multiplication algorithm requires mod n calculations; instead they perform mod r reduction in Step 2 and division by r in Step 3. By selecting $r = 2^k$ where $k > \log_2(n)$, these calculations are trivially implemented in software or hardware. The selection of $r = 2^k$ requires that n be odd, which is often the case in cryptography.

The Montgomery multiplication algorithm makes use of a special quantity n' which is one of the numbers produced by the extended Euclidean algorithm with inputs 2^k and n:

$$\begin{aligned} (u, n') &\leftarrow \text{EEA}(2^k, n) \\ u \cdot 2^k - n' \cdot n &= 1 \\ n' &= -n^{-1} \pmod{2^k} \end{aligned}$$

In other words, the Montgomery multiplication algorithm requires the computation of $n^{-1} \pmod{2^k}$ rather than $r^{-1} \pmod{n}$. We can expect that inversion with respect to a special modulus such as 2^k might be easier than inversion with respect to an arbitrary modulus. Indeed this is the case. Dussé and Kaliski [2] gave an efficient algorithm for computing the inverse $x = a^{-1} \pmod{2^k}$ for an odd a, therefore, $\gcd(a, 2^k) = 1$. Their algorithm is based on a specialized version of the extended Euclidean algorithm for computing the inverse.

```
function DusseKaliski($a, 2^k$)
input: $a, k$
output: $x = a^{-1} \bmod 2^k$
1:  $x \leftarrow 1$
2:  for $i = 2$ to $k$
2a:     if $2^{i-1} < a \cdot x \pmod{2^i}$ then $x \leftarrow x + 2^{i-1}$
3:  return $x$
```

As an example, consider the computation of $23^{-1} \pmod{2^6}$. Here, we have $a = 23$ and $k = 6$, and we start with $x = 1$. At the end of the algorithm we find $x = 39$, implying $23^{-1} = 39 \pmod{64}$; this is correct since $23 \cdot 39 = 897 = 1 \pmod{64}$.

i	2^{i-1}	2^i	x	$a \cdot x \pmod{2^i}$		x
2	2	4	1	$(23 \cdot 1 \bmod 4) \to 3$	$3 > 2$	$1 + 2 = 3$
3	4	8	3	$(23 \cdot 3 \bmod 8) \to 5$	$5 > 4$	$3 + 4 = 7$
4	8	16	7	$(23 \cdot 7 \bmod 16) \to 1$	$7 \not> 8$	7
5	16	32	7	$(23 \cdot 7 \bmod 32) \to 1$	$7 \not> 14$	7
6	32	64	7	$(23 \cdot 7 \bmod 64) \to 33$	$33 > 32$	$7 + 32 = 39$

It is not clear how the Dussé-Kaliski algorithm can be generalized to inversion mod p^k for a prime $p > 2$. However, it is instructive to give a complexity analysis of it. The algorithm runs $k-1$ steps where at each step a multiplication $\pmod{2^i}$, a comparison (subtraction), and an addition is performed with operands whose sizes are as much as k bits.

3 A New Algorithm for Inversion mod p^k

We introduce a new algorithm for computing $x = a^{-1} \pmod{p^k}$ for an arbitrary prime p, requiring s bits in its representation. The size of the input operand a and the inverse x will be sk bits. Our algorithm relies on Dixon's algorithm [1] for exact solution linear equations using p-adix expansions. Dixon's algorithm aims to exactly solve a linear system of equations with integer coefficients, such as $\mathsf{A} \cdot \mathsf{x} = \mathsf{b}$ in the sense that the solutions are obtained as rational numbers rather than approximate values using floating-point arithmetic.

Similar to Dixon's approach, we formulate the inversion problem as the exact solution of the linear equation

$$a \cdot x = 1 \pmod{p^k}$$

for a prime p, integer $k > 1$ and $\gcd(a, p) = 1$ or $1 < a < p$. By solving this equation, we compute the inverse $x = a^{-1} \pmod{p^k}$. The algorithm starts with the computation of

$$c = a^{-1} \pmod{p}$$

using the extended Euclidean algorithm. It is more often the case that the prime p is small, thus, this computation will not constitute a bottleneck. In fact, the case of $p = 2$ is trivial, since $c = 1$ for any odd a. The algorithm then iteratively finds the digits of x expressed in *base* p such that $x = a^{-1} \pmod{p^k}$. In other words, the algorithm computes the vector $(x_{k-1} \cdots x_1 x_0)_p$ with $x_i \in [0, p-1]$ such that

$$x = \sum_{i=0}^{k-1} x_i p^i = x_0 + x_1 \cdot p + x_2 \cdot p^2 + \cdots + x_{k-1} \cdot p^{k-1}$$

```
function ModInverse($a, p^k$)
input: $a, p, k$
output: $x = a^{-1} \bmod p^k$
1:   $c \leftarrow a^{-1} \pmod p$
2:   $b_0 \leftarrow 1$
3:   for $i = 0$ to $k-1$
3a:      $x_i \leftarrow c \cdot b_i \pmod p$
3b:      $b_{i+1} \leftarrow (b_i - a \cdot x_i)/p$
4:   return $x = (x_{k-1} \cdots x_1 x_0)_p$
```

4 Correctness of ModInverse

First of all, the term $(b_i - a \cdot x_i)$ in Step 3b is divisible by p for every i since

$$b_i - a \cdot x_i = b_i - a \cdot c \cdot b_i = b_i - b_i = 0 \pmod p$$

due to the fact that $a \cdot c = 1 \pmod p$. Therefore, b_i is integer for every $i \in [0, k-1]$. It also follows that when $i = 0$, the term $(b_0 - a \cdot x_0) = (1 - a \cdot c)$ is divisible by p. Furthermore, the terms b_i and x_i are found as

$$\begin{aligned} b_i &= (1 - a \cdot c)^i / p^i \\ b_i \cdot p^i &= (1 - a \cdot c)^i \\ x_i &= c \cdot b_i \pmod p \end{aligned}$$

for $i = 0, 1, \ldots, k-1$. The identity for b_i can be proven by induction on i.

The Basis Step: For $i = 0$, we have

$$\begin{aligned} b_0 &= 1 \\ x_0 &= c \cdot b_0 = c \pmod p \end{aligned}$$

These follow from Step 2 and Step 3a of the algorithm for $i = 0$.

The Inductive Step: Assume the formulas for b_i and x_i are correct for i. Due to Step 3b, we can write $b_{i+1} \cdot p = b_i - a \cdot x_i$, and thus

$$\begin{aligned} b_{i+1} \cdot p &= b_i - a \cdot x_i \\ &= (1 - a \cdot c)^i / p^i - a \cdot c \cdot (1 - a \cdot c)^i / p^i \\ &= (1 - a \cdot c)^i \cdot (1 - a \cdot c) / p^i \\ &= (1 - a \cdot c)^{i+1} / p^i \\ b_{i+1} \cdot p^{i+1} &= (1 - a \cdot c)^{i+1} \end{aligned}$$

Once b_{i+1} is available, we can write from Step 3a as $x_{i+1} = c \cdot b_{i+1} \pmod p$.

To prove that the algorithm indeed computes $x = a^{-1} \pmod{p^k}$, we note that $a \cdot x$ can be written as

$$\begin{aligned}
a \cdot \sum_{i=0}^{k-1} x_i \cdot p^i &= a \cdot \sum_{i=0}^{k-1} c \cdot b_i \cdot p^i \\
&= a \cdot \sum_{i=0}^{k-1} c \cdot (1 - a \cdot c)^i \\
&= a \cdot c \cdot \frac{(1 - a \cdot c)^k - 1}{1 - a \cdot c - 1} \\
&= 1 - (1 - a \cdot c)^k
\end{aligned}$$

Thus, we find $a \cdot x = 1 - (1 - a \cdot c)^k$. We have already determined that $(1 - a \cdot c)$ is a multiple of p, thus, we conclude that $(1 - a \cdot c)^k$ is a multiple of p^k. Therefore, we have $a \cdot x = 1 \pmod{p^k}$.

5 Example Computation and Complexity

Consider the computation of $12^{-1} \pmod{5^4}$. We have $a = 12$, $p = 5$, and $k = 4$. First we compute $c = a^{-1} \pmod{p}$, which is found as $c = 12^{-1} = 2^{-1} = 3 \pmod{5}$. Starting with the initial value $b_0 = 1$, the algorithm proceeds for $i = 0, 1, 2, 3$ as follows. The algorithm computes x expressed in base 5 as $x = (x_3 x_2 x_1 x_0)_5 = (4243)_5$. In decimal, this is equal to $4 \cdot 5^3 + 2 \cdot 5^2 + 4 \cdot 5 + 3 = 573$. Indeed $12^{-1} = 573 \pmod{5^4}$.

i	b_i	$x_i = c \cdot b_i \bmod p$	$b_{i+1} = (b_i - a \cdot x_i)/p$
0	1	$(3 \cdot 1 \bmod 5) \rightarrow 3$	$(1 - 12 \cdot 3)/5 \rightarrow -7$
1	-7	$(3 \cdot (-7) \bmod 5) \rightarrow 4$	$(-7 - 12 \cdot 4)/5 \rightarrow -11$
2	-11	$(3 \cdot (-11) \bmod 5) \rightarrow 2$	$(-11 - 12 \cdot 2)/5 \rightarrow -7$
3	-7	$(3 \cdot (-7) \bmod 5) \rightarrow 4$	$\ldots$

In the analysis of the algorithm, a more likely scenario is that the prime number p fits into the word size of the computer. Assume, $s = \log_2(p)$ which is also the word size of the processor. We can count the addition, subtraction, and multiplications mod p (size s-bit) single-precision arithmetic operations.

In Step 1, the EEA algorithm can be used to compute $c = a^{-1} \pmod{p}$ which requires $O(\log s)$ arithmetic operations. Note that the input parameter a and the output x may occupy up to ks-bit of space however, the inverse c is only s bits.

In Step 3a we perform one s-bit modular multiplication requiring one arithmetic operation. In Step 3b we multiply an s-bit number x_i with the ks-bit number a, subtract the result from ks-bit number b_i, and divide by p in order to compute b_{i+1}. Therefore, the computations in Step 3b require $O(k)$ arithmetic operations. We conclude that the inversion algorithm requires $O(k^2 + \log s)$ arithmetic operations in order to compute $x = a^{-1} \pmod{p^k}$, where a and x are ks-bit numbers.

6 Inversion mod 2^k

The proposed algorithm significantly simplifies when $p = 2$, and it constitutes a more efficient alternative to the Dussé-Kaliski algorithm. First of all, for $x = a^{-1} \pmod{2^k}$ to exist, $\gcd(a, 2^k)$ must be 1, which implies that a is odd. Given an odd a, the value of $c = a^{-1} \pmod 2$ is trivially found: $c = 1$. The modified algorithm is given below.

```
function ModInverse($a, 2^k$)
input: $a, k$
output: $x = a^{-1} \bmod 2^k$
1:   $b_0 \leftarrow 1$
2:   for $i = 0$ to $k - 1$
2a:        $x_i \leftarrow b_i \pmod 2$
2b:        $b_{i+1} \leftarrow (b_i - a \cdot x_i)/2$
3:   return $x = (x_{k-1} \cdots x_1 x_0)_2$
```

The mod 2 operation in Step 2a is computed by checking the LSB. Obviously we have $x_i \in \{0, 1\}$, and the inverse x is produced in base 2, that is $x = (x_{k-1} \cdots x_1 x_0)_2$. On the other hand, the division by 2 in Step 2b is performed by right shift. Below, we illustrate the computation of $a = 23$ and $k = 6$, in order to compare to the Dussé-Kaliski algorithm. The algorithm produces the binary result $x = (100111)_2 = 39$; this is indeed correct, since $23^{-1} = 39 \pmod{2^6}$.

i	b_i	$x_i = b_i \pmod 2$	$b_{i+1} = (b_i - a \cdot x_i)/2$
0	1	$1 \pmod 2 \rightarrow 1$	$(1 - 23 \cdot 1)/2 \rightarrow -11$
1	-11	$-11 \pmod 2 \rightarrow 1$	$(-11 - 23 \cdot 1)/2 \rightarrow -17$
2	-17	$-17 \pmod 2 \rightarrow 1$	$(-17 - 23 \cdot 1)/2 \rightarrow -20$
3	-20	$-20 \pmod 2 \rightarrow 0$	$(-20 - 23 \cdot 0)/2 \rightarrow -10$
4	-10	$-10 \pmod 2 \rightarrow 0$	$(-10 - 23 \cdot 0)/2 \rightarrow -5$
5	-5	$-5 \pmod 2 \rightarrow 1$	$(-5 - 23 \cdot 1)/2 \rightarrow -14$

Realistic analysis of the algorithm would require that we count of number of bit operations. In Step 2a we perform a "check the LSB" operation. The division by 2 in Step 2b is usually implemented by a "right shift of the operand". We will not involve these computations in the complexity calculations since they are of $O(1)$. Depending on whether $x_i = 1$ (or $x_i = 0$), in Step 2b we subtract (or not subtract) a from b_i, each of which is k bits. If we assume uniform probability that half of the x_i bits would be nonzero, our algorithm requires only $k/2$ k-bit subtraction operations in the average in order to compute $x = a^{-1} \pmod{2^k}$, where a and x are k-bit numbers.

7 Conclusions

We have introduced a new algorithm for computing the inverse $a^{-1} \pmod{p^k}$ given a prime p and $a \in [1, p-1]$. The algorithm is based on the exact solution of linear equations using p-adic expansions, due to Dixon [1]. The new algorithm starts with the initial value $c = a^{-1} \pmod p$ and iteratively computes the inverse $x = a^{-1} \pmod{p^k}$.

The mod 2 version of the algorithm (that is, when $p = 2$) is significantly more efficient than the algorithm given by Dussé and Kaliski in [2] for computing $a^{-1} \pmod{2^k}$ for an odd integer a.

The new algorithm requires $k/2$ subtractions in the average with (up to) k-bit operands. The Dussé-Kaliski algorithm requires $k-1$ multiplications, $k-1$ comparisons (subtractions), and $k/2$ additions in the average with (up to) k-bit operands.

References

[1] J. D. Dixon. Exact solution of linear equations using p-adic expansions. *Numerische Mathematik*, 40(1):137–141, 1982.

[2] S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology - EUROCRYPT 90*, pages 230–244. Springer, LNCS Nr. 473, 1990.

[3] B. S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, 1995.

[4] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[5] E. Savaş and Ç. K. Koç. The Montgomery modular inverse - revisited. *IEEE Transactions on Computers*, 49(7):763–766, July 2000.

[6] E. Savaş and Ç. K. Koç. Montgomery inversion. *Journal of Cryptographic Engineering*, to appear, 2017.