Algorithms for Inversion Mod p^k

Çetin Kaya Koç, *Fellow, IEEE*

Abstract—This article describes and analyzes all existing algorithms for computing $x = a^{-1} \pmod{p^k}$ for a prime p, and also introduces a new algorithm based on the exact solution of linear equations using p-adic expansions. The algorithm 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 more efficient than all existing algorithms for small values of k. Moreover, it stands out as being the only one that works for any p, any k, and digit-by-digit. While the new algorithm is asymptotically worse off, it requires the minimal number of arithmetic operations (just a single addition) per step, as compared to all existing algorithms.

Index Terms—Number-theoretic algorithms, computer arithmetic, multiplicative inverse

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 or can compute $\phi(n)$.

On the other hand, the extended euclidean algorithm (EEA) works for both prime and composite modulus, and does not require the knowledge of ϕ. The classical EEA requires division operations at each step, which is costly. On the other hand, variations of the binary extended euclidean algorithms use shift, addition and subtraction operations [9], [14], [15]. We must note however that most 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 [13] 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 one of the numbers produced by the extended euclidean algorithm with inputs 2^k and n:

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

In other words, the Montgomery multiplication algorithm requires the computation of $n^{-1} \pmod{2^k}$ rather than $r^{-1} \pmod{n}$. We may 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. Several algorithms for computing multiplicative inverse mod 2^k appeared in the literature some of which are significantly simpler than the classical EEA algorithm.

• *The author is with İstinye University, 34010 İstanbul, Turkey, and the Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu 210016, China, and also with the University of California Santa Barbara, Santa Barbara, CA 93106. E-mail: cetinkoc@ucsb.edu.*

Manuscript received 28 June 2017; revised 20 Apr. 2018; accepted 24 Apr. 2018. Date of publication 30 Jan. 2020; date of current version 8 May 2020.
(Corresponding author: Çetin Kaya Koç.)
Recommended for acceptance by W. Liu.
Digital Object Identifier no. 10.1109/TC.2020.2970411

3 Suffix Property of Inverse Mod 2^k and p^k

Given $x = a^{-1} \pmod{2^k}$, we can compute $y = a^{-1} \pmod{2^j}$ for $1 \le j < k$ by reduction: $y = x \pmod{2^j}$. We can easily prove that y is the inverse of a mod 2^j for some $j \in [1, k)$, by noting that $a \cdot x = 1 \pmod{2^k}$ implies $a \cdot x = 1 + N \cdot 2^k$ for some integer N; when we reduce both sides mod 2^j, we obtain:

$$\begin{aligned} a \cdot \sum_{i=0}^{k-1} X_i \cdot 2^i &= 1 + N \cdot 2^k \pmod{2^j} \\ a \cdot \sum_{j=0}^{j-1} X_i \cdot 2^i &= 1 \pmod{2^j}. \end{aligned}$$

Therefore, we conclude that $y = a^{-1} \pmod{2^j}$. Moreover if $x = a^{-1} \pmod{2^k}$ is expressed as a k-bit binary number $x = (X_{k-1} \cdots X_1 X_0)$, then the suffixes (the least significant bits) of x are actually the inverses mod 2^j for $j = 1, 2, \ldots, k-2$. That is, (X_0) is the inverse of a mod 2, and $(X_1 X_0)$ is the inverse of a mod 2^2, and so on, up to $k-1$.

For the case of p^k, we note that $a \cdot x = 1 \pmod{p^k}$ implies $a \cdot x = 1 + N \cdot p^k$ for some integer N, and therefore, when we reduce both sides mod p^j, we obtain:

$$\begin{aligned} a \cdot \sum_{i=0}^{k-1} X_i \cdot p^i &= 1 + N \cdot p^k \pmod{p^j} \\ a \cdot \sum_{i=0}^{i-1} X_i \cdot p^i &= 1 \pmod{p^j}. \end{aligned}$$

If the inverse x is expressed in base p, we have $X_i \in [0, p-1]$ and $x = (X_{k-1} \cdots X_1 X_0)$, and thus, the inverse mod p^j is equal to $(X_{j-1} \cdots X_1 X_0)$. In other words, the suffix property also holds for the inverse mod p^k, provided that the inverse x mod p^k is expressed in base p.

To summarize: if $x = a^{-1} \pmod{2^k}$ is available, we can reduce it mod 2^j to obtain $a^{-1} \pmod{2^j}$ for any $j \in [1, k-1]$. If x is expressed in binary as $x = (X_{k-1} \cdots X_1 X_0)$, then the inverse mod 2^j is simply the j-bit suffix of x as $(X_{j-1} \cdots X_1 X_0)$. Similarly, if $x = a^{-1} \pmod{p^k}$ is available, we can reduce it mod p^j to obtain $a^{-1} \pmod{p^j}$ for any $j \in [1, k-1]$. If x is expressed in base p as $x = (X_{k-1} \cdots X_1 X_0)$, then the inverse mod p^j is simply the j-digit suffix of x as $(X_{j-1} \cdots X_1 X_0)$.

4 Existing Inversion Algorithms

There are several algorithms in the literature. Dussé and Kaliski [5] gave an efficient algorithm for computing the inverse $x = a^{-1} \pmod{2^k}$ for an odd a, therefore, $\gcd(a, 2^k) = 1$. Arazi and Qi [1] review 3 known algorithms (as of 2008), and introduce a new algorithm (Algorithm 4) for computing $a^{-1} \pmod{2^k}$, where $k = 2^s$. Furthermore, Dumas proved [3], [4] that Algorithm 4 in [1] is a specific case of Hensel lifting [12], and introduced an iterative formula for computing $x = a^{-1} \pmod{p^k}$, where $k = 2^s$. In this section, we describe these algorithms.

4.1 Dussé and Kaliski Algorithm

Dussé and Kaliski algorithm [5] is based on a specialized version of the extended euclidean algorithm for computing the inverse. The pseudocode is given below [5], [10].

TABLE 1
Dussé and Kaliski Algorithm for Computing $23^{-1} \pmod{2^6}$

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

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

As an example, consider the computation of $23^{-1} \pmod{2^6}$ illustrated in Table 1. 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 $23^{-1} = 39 \pmod{2^6}$; this is indeed correct since $23 \cdot 39 = 1 \pmod{2^6}$. On the other hand, the inverses mod 2^j for $j = 1, 2, 3, 4, 5$ can be obtained by reduction $39 \pmod{2^j}$. We can also compute them using the suffix property, by expressing 39 in binary as $(100111)_2$, and taking its suffixes. However, we notice that the Dussé and Kaliski algorithm already computes consecutive inverses $23^{-1} \pmod{2^i}$ for $i = 1, 2, 3, 4, 5, 6$ as $(1)_2 = 1$, $(11)_2 = 3$, $(111)_3 = 7$, $(0111)_3 = 7$, $(00111)_3 = 7$, and $(100111)_3 = 39$.

These consecutive inverses are computed *in whole* at each step (rather than bit-by-bit, as we will see some other algorithms do). The j-bit inverse $23^{-1} \pmod{2^j}$ is computed at the jth step. This property affects the performance, since the entire j-bit number is computed (rather than a single bit).

4.2 Algorithm 2 in Arazi and Qi Paper

Arazi and Qi review three existing algorithms, and introduce a new algorithm in their paper [1]. All 4 algorithms in [1] compute $x = a^{-1} \pmod{2^k}$. First of all, Algorithm 1 is Dussé and Kaliski algorithm which we have already covered.

Algorithm 2 is described in the narrative of the article [1] without explicitly giving its steps. We find it useful to describe this algorithm and give its pseudocode. Assume a and x are k-bit binary numbers. Since a and x are both odd, i.e., $A_0 = X_0 = 1$, they can be written as

$$a = (A_{k-1}A_{k-2} \cdots A_1A_0) = (A_{k-1}A_{k-2} \cdots A_1 1)$$
$$x = (X_{k-1}X_{k-2} \cdots X_1X_0) = (X_{k-1}X_{k-2} \cdots X_1 1)$$

The main idea of Algorithm 2 is that the equality

$$a \cdot x = 1 = (00 \cdots 01)_2 \pmod{2^k},$$

implies that the least significant k bits of $y = a \cdot x$ is equal to $(00 \cdots 01)_2$, and y can be written as

$$y = a \cdot x = (\overbrace{Z_{k-1} \cdots Z_1 Z_0}^{k \text{ bits}} \; \overbrace{00 \cdots 01}^{k \text{ bits}})_2. \tag{1}$$

Our aim is to compute the remaining bits of x, i.e., X_i for $i = 1, 2, \ldots, k-1$, making sure that as y is iteratively computed, its least significant k bits become equal to $(00 \cdots 01)_2$ according to Equation (1).

Notice that the LSB of a is 1, and thus, the ith bit of $2^i \cdot a$ is equal to 1 for any $i \in [1, k-1]$. Iterative computation of y is accomplished by starting with $y = a$, adding $2^i \cdot a$ to y if $Y_i = 1$, since this would make the resulting Y_i zero. By proceeding to the left, we make all $Y_i = 0$ for $i = 1, 2, \ldots, k-1$, except $Y_0 = 1$. The steps of Algorithm2 are given below. It computes the bits of the inverse x from the least significant to the most significant bit, at the ith step either adding $2^i \cdot a$ to y or not, and determining X_i as 1 or zero.

TABLE 2
Algorithm 2 for Computing $23^{-1} \pmod{2^6}$

i	y	Y_i	$y = y + 2^i \cdot a$	X_i
0	$23 = (000000\ 01011\mathbf{1})$	1	$y = 23$	1
1	$23 = (000000\ 0101\mathbf{1}1)$	1	$y = 23 + 2 \cdot 23 \rightarrow 69$	1
2	$69 = (000001\ 000\mathbf{1}01)$	1	$y = 69 + 2^2 \cdot 23 \rightarrow 161$	1
3	$161 = (000010\ 10\mathbf{0}001)$	0	$y = 161$	0
4	$161 = (000010\ 1\mathbf{0}0001)$	0	$y = 161$	0
5	$161 = (000010\ \mathbf{1}00001)$	1	$y = 161 + 2^5 \cdot 23 \rightarrow 897$	1
	$897 = (001110\ 000001)$			

```
function Algorithm2($a, 2^k$)
input: $a, k$ where $a$ is odd and $a < 2^k$
output: $x = a^{-1} \bmod 2^k$
1:    $y \leftarrow a$
2:    $X_0 \leftarrow 1$
3:    for $i = 1$ to $k - 1$
3a:       if $Y_i = 1$
3aa:          $y \leftarrow y + 2^i \cdot a$
3ab:          $X_i \leftarrow 1$
3b:       else
3ba:          $X_i \leftarrow 0$
4:    return $x = (X_{k-1} \cdots X_1X_0)_2$
```

The computation of $23^{-1} \pmod{2^6}$ using Algorithm 2 is illustrated in Table 2. The initial value of y is $a = 23$, and at each step Y_i is checked; if $Y_i = 1$, then $2^i \cdot a$ is added to y. As the progress of the algorithm shows, the lower $k = 6$ bits of y eventually becomes (000001). The inverse is computed as $x = (100111)_2 = 39$. This is indeed correct since $23 \cdot 39 = 1 \pmod{2^6}$.

Algorithm 2 computes the inverse $x = a^{-1} \pmod{2^k}$ *bit by bit*. At the jth step, the jth bit of x is computed. Hence, the inverse mod 2^j becomes available at the jth step: $(X_{j-1} \cdots X_1X_0)$ is the inverse mod 2^j.

4.3 Algorithm 3 in Arazi and Qi Paper

Arazi and Qi describe Algorithm 3 in detail [1], and give pseudocode. This algorithm has two stages: in the first stage which is called Algorithm 3a, the quantity $-v = (2^k)^{-1} \pmod{a}$ is computed. In the second stage (Algorithm 3b), the quantity $-v$ is used to compute $x = a^{-1} \pmod{2^k}$. This algorithm is essentially the extended euclidean algorithm. Given $\gcd(a, 2^k) = 1$, the EEA computes

$$(x, v) \leftarrow \text{EEA}(a, 2^k)$$
$$x \cdot a - v \cdot 2^k = 1$$
$$a^{-1} = x \pmod{2^k}$$
$$(2^k)^{-1} = -v \pmod{a}$$

After $-v$ is available, we can compute x using the identity

$$x = \frac{1 + v \cdot 2^k}{a},$$

which requires a shift (the computation of $v \cdot 2^k$), an increment operation, and a division by a operation (which is very expensive). Algorithm 3 is the least efficient of all 4 algorithms in [1], since it requires a full division with k-bit integers in the second stage of the algorithm.

TABLE 3
Steps 1 and 2 of Algorithm 3 for Computing $23^{-1} \pmod{2^6}$

i	v	V_0	$v = v + a$	$v = v/2$
0	$1 = (000001)$	1	$v = 1 + 23 \to 24$	$v = 24/2 \to 12$
1	$12 = (001100)$	0	$v = 12$	$v = 12/2 \to 6$
2	$6 = (000110)$	0	$v = 6$	$v = 6/2 \to 3$
3	$3 = (000011)$	1	$v = 3 + 23 \to 26$	$v = 26/2 \to 13$
4	$13 = (001101)$	1	$v = 13 + 23 \to 36$	$v = 36/2 \to 18$
5	$18 = (010010)$	0	$v = 18$	$v = 18/2 \to 9$

The computation of $-v = (2^k)^{-1} \pmod{a}$ for an odd a is quite easy, due to the Montgomery reduction algorithm called the Coarsely Integrated Operand Scanning (CIOS) whose details are found in [11].

Writing it as $-v = 2^{-k} \pmod{a}$, we first compute this quantity $v = (V_{k-1} \cdots V_1 V_0)$ using the CIOS algorithm at the end of Step 2; we then compute the inverse x in Step 3.

```
function Algorithm3($a, 2^k$)
input: $a, k$ where $a$ is odd and $a < 2^k$
output: $x = a^{-1} \bmod 2^k$
1:    $v \leftarrow 1$
2:    for $i = 0$ to $k - 1$
2a:       if $V_0 = 1$
2aa:          $v \leftarrow v + a$
2b:       $v \leftarrow v/2$
3:    $x \leftarrow (1 + v \cdot 2^k)/a$
4:    return $x$
```

The correctness of Algorithm 3 depends on the fact that the quantity $(1 + v \cdot 2^k)$ is divisible by a. This is easily proved by noting that $-v = 2^{-k} \pmod{a}$ implies $-v \cdot 2^k = 1 \pmod{a}$, and thus, $-v \cdot 2^k = 1 + N \cdot a$ for some integer N. Therefore, $1 + v \cdot 2^k = -N \cdot a$.

Steps 1 and 2 of Algorithm 3 for computing $23^{-1} \pmod{2^6}$ is illustrated in Table 3. The initial value is $v = 1$, and at each step V_0 is checked; if $V_0 = 1$, then a is added to v, and v is shifted to left (i.e., divided by 2).

At the end of Step 2 for $i = 5$, we obtain $-v = 9$. In Step 3, we use the formula $(1 + v \cdot 2^k)/a$ and the value of $-v = 9$, to compute the inverse as $x = (1 + (-9) \cdot 2^6)/23 = -25$, which is equal to $39 \pmod{2^6}$. This inverse is computed *in whole* in a single step, using a shift, an addition and a division operation involving k-bit numbers. On the other hand, the inverses mod 2^i for $i \in [1, k-1]$ can be computed only after Step 3 is completed, by reducing $x \bmod 2^i$.

4.4 Algorithm 4 in Arazi and Qi Paper

Algorithm 4 is the last one described in [1], and it is presented as the authors' contribution. It is based on the idea that, given $a = (a_H a_L) = a_H \cdot 2^i + a_L$ where a_H and a_L are the upper and lower i bits of the $2i$-bit binary number a, the inverse $x = a^{-1} \pmod{2^{2i}}$ can be computed from the inverse of $a_L \bmod 2^i$. Algorithm 4 computes the inverse of $a \bmod 2^k$ where k is a power of 2, that is, it computes $x = a^{-1} \pmod{2^{2^s}}$, and it accomplishes this computation in $s = \log_2(k)$ steps. In other words, the number of steps of Algorithm 4 is logarithmic in k.

Given $a = (a_H a_L) = a_H \cdot 2^i + a_L$ and $x = (x_H x_L) = x_H \cdot 2^i + x_L$, we assume $x_L = a_L^{-1} \pmod{2^i}$ is already computed and available. Note that a_H, a_L, x_H, x_L are all i-bit integers. Algorithm 4 computes the upper part x_H of the inverse $x = a^{-1} \pmod{2^{2i}}$ in 3 steps:

1) Compute the product $a_L \cdot x_L = (b_H b_L) = b_H \cdot 2^i + b_L = b_H \cdot 2^i + 1$.
2) Compute the product $a_H \cdot x_L = (c_H c_L) = c_H \cdot 2^i + c_L$.
3) Compute the expression $x_H = -(b_H + c_L) \cdot x_L \pmod{2^i}$.
4) The inverse is given as $x = (x_H x_L) = x_H \cdot 2^i + x_L$.

An algebraic proof is given in [1]. Here we illustrate this method for the 32-bit number $a = 2583209455 = (99f8a5ef)_{16}$. This gives $a_H = 39416 = (99f8)_{16}$ and $a_L = 42479 = (a5ef)_{16}$. Furthermore, we assume the inverse of the lower part $a_L \bmod 2^{16}$ is already computed and available: $x_L = a_L^{-1} \pmod{2^{16}}$ as $x_L = 10511 = (290f)_{16}$. We then compute x_H using

1) $a_L \cdot x_L = 42479 \cdot 10511 = 446496769 = (1a9d0001)_{16} = (b_H b_L)$. This gives $b_H = 6813 = (1a9d)_{16}$ and $b_L = 1$.
2) $a_H \cdot x_L = 39416 \cdot 10511 = 414301576 = (18b1bd88)_{16} = (c_H c_L)$. This gives $c_H = (18b1)_{16} = 6321$ and $c_L = (bd88)_{16} = 48520$.
3) $x_H = -(6813 + 48520) \cdot 10511 \pmod{2^{16}}$. This gives $x_H = 26837 = (68d5)_{16}$.
4) The inverse: $x = (x_H x_L) = (68d5290f)_{16} = 1758800143$. This is indeed correct $2583209455 \cdot 1758800143 = 1 \pmod{2^{32}}$.

Algorithm 4 is a essentially a recursive algorithm. The inverse of $a \bmod 2^{32}$ invokes the computation of the inverse $a \bmod 2^{16}$, which the computation of the inverse $a \bmod 2^8$, and so on. However, it can also be made iterative by first computing the inverse mod 2^1, using this inverse to compute the inverse mod 2^2, and then mod 2^4, and so on. The authors describe Algorithm 4 in the narrative of the article [1], however they do not provide a pseudocode. Below we give the pseudocode for computing the inverse mod 2^k for $k = 2^s$. The binary expansion of a is expressed as $a = (A_{k-1} \cdots A_1 A_0)$ and $k = 2^s$ for some integer s.

```
function Algorithm4($a, 2^k$)
input: $a, k$ where $a$ is odd, $a < 2^k$, and $k = 2^s$
output: $x = a^{-1} \bmod 2^k$
1:    $a_L \leftarrow A_0$
2:    $a_H \leftarrow A_1$
3:    $x_L \leftarrow 1$
4:    for $i = 1$ to $s$
4a:       $(b_H b_L) \leftarrow a_L \cdot x_L$
4b:       $(c_H c_L) \leftarrow a_H \cdot x_L$
4c:       $x_H \leftarrow -(b_H + c_L) \cdot x_L \pmod{2^{2^{i-1}}}$
4d:       $a_L \leftarrow (A_{2^i - 1} \cdots A_0)_2$
4e:       $a_H \leftarrow (A_{2^{i+1} - 1} \cdots A_{2^i})_2$
4f:       $x_L \leftarrow (x_H x_L)$
4:    return $x = (x_H x_L)$
```

Table 4 illustrates the inverse computation $x = a^{-1} \pmod{2^{32}}$ for $a = (99f8a5ef)_{16}$, where $s = 5$. The algorithm computes the inverse $x = a^{-1} \pmod{2^{32}}$, by successively computing the inverse mod 2^i for $i = 1, 2, 4, 8, 16, 32$.

TABLE 4
Algorithm 4 for Computing $(99f8a5ef)_{16}^{-1} \pmod{2^{32}}$

s	$(a_H\ a_L)$	x_L	$(b_H\ b_L) \leftarrow a_L \cdot x_L$	$(c_H\ c_L) \leftarrow a_H \cdot x_L$	x_H	$(x_H\ x_L)$
1	$(1\ 1)_2$	$(1)_2$	$(0\ 1)_2$	$(0\ 1)_2$	$(1)_2$	$(1\ 1)_2$
2	$(11\ 11)_2$	$(11)_2$	$(10\ 01)_2$	$(10\ 01)_2$	$(11)_2$	$(11\ 11)_2$
3	$(e\ f)_{16}$	$(f)_{16}$	$(e\ 1)_{16}$	$(d\ 2)_{16}$	$(0)_{16}$	$(0\ f)_{16}$
4	$(a5\ ef)_{16}$	$(0f)_{16}$	$(0e\ 01)_{16}$	$(09\ ab)_{16}$	$(29)_{16}$	$(29\ 0f)_{16}$
5	$(99f8\ a5ef)_{16}$	$(290f)_{16}$	$(1a9d\ 0001)_{16}$	$(18b1\ bd88)_{16}$	$(68d5)_{16}$	$(68d5\ 290f)_{16}$

TABLE 5
Dumas Iteration for Computing $12^{-1} \pmod{5^{16}}$

i	x_{i-1}	p^{2^i}	$x_i = x_{i-1} \cdot (2 - a \cdot x_{i-1}) \bmod p^{2^i}$
1	$x_0 = 3$	5^2	$x_1 = 3 \cdot (2 - 12 \cdot 3) \rightarrow 23$
2	$x_1 = 23$	5^4	$x_2 = 23 \cdot (2 - 12 \cdot 23) \rightarrow 573$
3	$x_2 = 573$	5^8	$x_3 = 573 \cdot (2 - 12 \cdot 573) \rightarrow 358073$
4	$x_3 = 358073$	5^{16}	$x_4 = 358073 \cdot (2 - 12 \cdot 358073) \rightarrow 139872233073$

TABLE 6
Dumas Iteration for Computing $23^{-1} \pmod{2^{32}}$

i	x_{i-1}	2^{2^i}	$x_i = x_{i-1} \cdot (2 - a \cdot x_{i-1}) \bmod 2^{2^i}$
1	$x_0 = 1$	2^2	$x_1 = 1 \cdot (2 - 23 \cdot 1) \rightarrow 3$
2	$x_1 = 3$	2^4	$x_2 = 3 \cdot (2 - 23 \cdot 3) \rightarrow 7$
3	$x_2 = 7$	2^8	$x_3 = 7 \cdot (2 - 23 \cdot 7) \rightarrow 167$
4	$x_3 = 167$	2^{16}	$x_4 = 167 \cdot (2 - 23 \cdot 167) \rightarrow 14247$
5	$x_4 = 14247$	2^{32}	$x_5 = 14247 \cdot (2 - 23 \cdot 14247) \rightarrow 3921491879$

The result is indeed correct since $(99f8a5ef)_{16} \cdot (68d5290f)_{16} = 1 \pmod{2^{32}}$. Algorithm 4 also computes $a^{-1} \bmod 2^{2^i}$ for $i = 0, 1, 2, 3, 4, 5$ at every step:

$$\begin{aligned}
(99f8a5ef)_{16}^{-1} &= (1)_2 \pmod 2 \\
(99f8a5ef)_{16}^{-1} &= (11)_2 \pmod{2^2} \\
(99f8a5ef)_{16}^{-1} &= (f)_{16} \pmod{2^4} \\
(99f8a5ef)_{16}^{-1} &= (0f)_{16} \pmod{2^8} \\
(99f8a5ef)_{16}^{-1} &= (290f)_{16} \pmod{2^{16}} \\
(99f8a5ef)_{16}^{-1} &= (68d5290f)_{16} \pmod{2^{32}}.
\end{aligned}$$

It is not clear if Algorithm 4 *as formulated* can be generalized for an arbitrary k; it seems that it cannot be. There are s steps in the algorithm, and at step i the inverse mod 2^{2^i} computed for $i = 1, 2, \ldots, s$. The authors describe a method (without detail) in Section 2.2 of [1] for dealing with a composite k, but they do not give a method for computing the inverse for an arbitrary k. The inverse mod 2^k for an arbitrary (not a power of 2) is not directly computed by this algorithm. However, the inverse mod 2^k for an arbitrary k can be obtained by first computing the inverse mod 2^{2^s} for the nearest $2^s > k$, and then reducing the result mod 2^k. For example, if we need $a^{-1} \bmod 2^{29}$, then we will have to compute the inverse mod 2^{2^5} first, since $2^5 > 29$.

4.5 Newton-Raphson Iteration by Dumas

Dumas in [3], [4] shows that Algorithm 4 given by Arazi and Qi [1] is actually a specific case of Hensel lifting [12], and provides a proof of the derivation of it. Dumas also gives Hensel's lemma mod p^k and its proof from Newton-Raphson iteration. This results in several formulas for computing $a^{-1} \pmod{2^k}$ for $k = 2^s$, one of which is Algorithm 4. Dumas studies different implementation variants of this iteration and shows that the explicit formula works well for small exponent values but it is slower for large exponent, for example, more than 700 bits. An important contribution of Dumas is an iterative formula which computes $x_s = a^{-1} \pmod{p^{2^s}}$ for a prime p, by iterating over $i = 1, 2, \ldots, s$ as

$$\begin{aligned}
x_0 &= a^{-1} \pmod p \\
x_i &= x_{i-1} \cdot (2 - a \cdot x_{i-1}) \bmod p^{2^i}.
\end{aligned}$$

By selecting $p = 2$, the formula also specializes to the binary case. The number of steps of the iteration is $s = \log_2(k)$. Below we illustrate the computation of $x_s = a^{-1} \pmod{p^{2^s}}$ for $a = 12$, $p = 5$, and $s = 4$. The iteration starts with $x_0 = 12^{-1} \pmod 5$, which is found as $x_0 = 3$, and proceeds over $i = 1, 2, 3, 4$, as shown in Table 5.

The result $x_4 = 139872233073$ is indeed correct since $12 \cdot 139872233073 = 1 \pmod{5^{16}}$. We note that during its iteration the Dumas algorithm actually computes consecutive inverses $12^{-1} \pmod{5^{2^i}}$ for $i = 0, 1, 2, 3, 4$:

$$\begin{aligned}
12^{-1} &= 3 \pmod 5 \\
12^{-1} &= 23 \pmod{5^2} \\
12^{-1} &= 573 \pmod{5^4} \\
12^{-1} &= 358073 \pmod{5^8} \\
12^{-1} &= 139872233073 \pmod{5^{16}}.
\end{aligned}$$

However, inverses modulo other powers of 5 are not computed. While the algorithm takes $s = \log_2(k)$ steps, it also computes $s = \log_2(k)$ inverses. However, the inverse mod p^k for an arbitrary k can be obtained by first computing the inverse mod p^{2^s} for the nearest $2^s > k$, and then reducing the result mod 2^k. For example, if we need a^{-1} mod p^{29}, then we will have to compute the inverse mod p^{2^5} first, since $2^5 > 29$.

The binary version of the Dumas algorithm is similar, but it is more compact than Algorithm 4. It uses the same formula as for p, but taking $p = 2$ and assuming that a is odd. The starting value $x_0 = 1$ since $p = 2$ and a is odd. Below we illustrate the computation of $x_s = a^{-1} \pmod{p^{2^s}}$ for $a = 23, p = 2$, and $s = 5$. The iteration starts with $x_0 = 23^{-1} \pmod 2$, which is found as $x_0 = 1$, and proceeds over $i = 1, 2, 3, 4, 5$ by computing $x_i = x_{i-1} \cdot (2 - a \cdot x_{i-1}) \bmod 2^{2^i}$.

The result $x_5 = 3921491879$ is indeed correct since $23 \cdot 3921491879 = 1 \pmod{2^{16}}$. We note that during its iteration the Dumas algorithm actually computes $13^{-1} \pmod{2^{2^i}}$ for $i = 0, 1, 2, 3, 4, 5$, as shown in Table 6:

$$\begin{aligned}
23^{-1} &= 1 \pmod 2 \\
23^{-1} &= 3 \pmod{2^2} \\
23^{-1} &= 7 \pmod{2^4} \\
23^{-1} &= 167 \pmod{2^8} \\
23^{-1} &= 14247 \pmod{2^{16}} \\
23^{-1} &= 3921491879 \pmod{2^{32}},
\end{aligned}$$

However, inverses modulo other powers of 2 are not computed. Similarly, the inverse mod 2^k for an arbitrary k can be obtained by first computing the inverse mod 2^{2^s} for the nearest $2^s > k$, and then reducing the result mod 2^k.

TABLE 7
ModInverse Algorithm for Computing $12^{-1} \pmod{5^5}$

i	b_i	$X_i = c \cdot b_i \bmod p$	$b_{i+1} = (b_i - a \cdot X_i)/p$
0	$b_0 = 1$	$X_0 = (3 \cdot 1 \bmod 5) \to 3$	$b_1 = (1 - 12 \cdot 3)/5 \to -7$
1	$b_1 = -7$	$X_1 = (3 \cdot (-7) \bmod 5) \to 4$	$b_2 = (-7 - 12 \cdot 4)/5 \to -11$
2	$b_2 = -11$	$X_2 = (3 \cdot (-11) \bmod 5) \to 2$	$b_3 = (-11 - 12 \cdot 2)/5 \to -7$
3	$b_3 = -7$	$X_3 = (3 \cdot (-7) \bmod 5) \to 4$	$b_4 = (-7 - 12 \cdot 4)/5 \to -11$
4	$b_4 = -11$	$X_4 = (3 \cdot (-11) \bmod 5) \to 2$	$\ldots$

However, it turns we do not need to compute the inverses mod up to p^{2^s} for $2^s > k$ in order to obtain the inverse mod p^k for an arbitrary $k < 2^s$. As suggested by one of the Reviewers, we can compute the inverses up to mod $p^{2^{s-1}}$ for the nearest $2^{s-1} < k$ (rather than $2^s > k$), and then apply one additional iteration for $i = s$

$$x_s = x_{s-1} \cdot (2 - a \cdot x_{s-1}) \pmod{p^k},$$

which is computed mod p^k (rather than mod p^{2^s}).

5 A New Algorithm for Inversion Mod p^k

We introduce a new algorithm for computing $x = a^{-1} \pmod{p^k}$ for a prime p and arbitrary positive integer k. Our algorithm relies on Dixon's algorithm [2] for exact solution linear equations using p-adix expansions, whose general idea is credited to German mathematician Kurt Wilhelm Sebastian Hensel. 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, an arbitrary positive 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 may not constitute a bottleneck. In fact, the computation of c for 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

$$x = \sum_{i=0}^{k-1} X_i \cdot p^i = X_0 + X_1 \cdot p + X_2 \cdot p^2 + \cdots + X_{k-1} \cdot p^{k-1},$$

```
function ModInverse$(a, p^k)$
input: $a, p, k$ where $\gcd(a, p) = 1$ and $a < p^k$
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 $x = (X_{k-1} \cdots X_1 X_0)_p$
```

Consider the computation of $12^{-1} \pmod{5^5}$. We have $a = 12$, $p = 5$, and $k = 5$. 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 $i = 0, 1, 2, 3, 4$ as illustrated in Table 7. The algorithm computes x expressed in base 5 as $x = (X_4 X_3 X_2 X_1 X_0)_5 = (24243)_5$. In decimal, this is equal to $2 \cdot 5^4 + 4 \cdot 5^3 + 2 \cdot 5^2 + 4 \cdot 5 + 3 = 1823$. Indeed $12^{-1} = 1823 \pmod{5^5}$ since $12 \cdot 1823 = 1 \pmod{5^5}$.

Our algorithm actually computes $12^{-1} \pmod{5^j}$ for $j = 1, 2, 3, 4, 5$ at each step, since it generates the base 5 digits of the inverse as $x = (X_4 X_3 X_2 X_1 X_0)_5 = (24243)_5$. The inverses for 5^j are the suffixes of the inverse $x = (24243)_5$, given as

$$\begin{aligned}
12^{-1} &= (3)_5 &= 3 \pmod{5} \\
12^{-1} &= (43)_5 &= 23 \pmod{5^2} \\
12^{-1} &= (243)_5 &= 73 \pmod{5^3} \\
12^{-1} &= (4243)_5 &= 573 \pmod{5^4} \\
12^{-1} &= (24243)_5 &= 1823 \pmod{5^5}.
\end{aligned}$$

6 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 $x_{i+1} = c \cdot b_{i+1} \pmod{p}$. This concludes the induction.

TABLE 8
ModInverse Algorithm for Computing $23^{-1} \pmod{2^6}$

i	b_i	$X_i = b_i \pmod 2$	$b_{i+1} = (b_i - a \cdot X_i)/2$
0	$b_0 = 1$	$X_0 = 1 \pmod 2 \rightarrow 1$	$b_1 = (1 - 23 \cdot 1)/2 \rightarrow -11$
1	$b_1 = -11$	$X_1 = -11 \pmod 2 \rightarrow 1$	$b_2 = (-11 - 23 \cdot 1)/2 \rightarrow -17$
2	$b_2 = -17$	$X_2 = -17 \pmod 2 \rightarrow 1$	$b_3 = (-17 - 23 \cdot 1)/2 \rightarrow -20$
3	$b_3 = -20$	$x_3 = -20 \pmod 2 \rightarrow 0$	$b_4 = (-20 - 23 \cdot 0)/2 \rightarrow -10$
4	$b_4 = -10$	$X_4 = -10 \pmod 2 \rightarrow 0$	$b_5 = (-10 - 23 \cdot 0)/2 \rightarrow -5$
5	$b_5 = -5$	$X_5 = -5 \pmod 2 \rightarrow 1$	

TABLE 9
Complexity Analysis of the Modular Inversion Algorithms

Algorithm	Steps	Number of Operations	Operand Sizes	a^{-1} mod p^j	p	k	Output
DK [5]	k	$1M + 2A$	$1, .., k$	$j = 1, .., k$	2	any	whole
AQ [1] Alg 2	k	$1M + 1A$	$1, .., k$	only $j = k$	2	any	bits
AQ [1] Alg 3	k	$1M + 1A$	k	only $j = k$	2	any	whole
	1	$1D$	k				
AQ [1] Alg 4	s	$3M + 2A$	$2^1, .., 2^s$	$j = 2^0, .., 2^s$	2	2^s	bits
Dumas [3], [4] p^k	s	$2M + 1A$	$2^1, .., 2^s$	$j = 2^0, .., 2^s$	any	2^s	digits
Dumas [3], [4] 2^k	s	$2M + 1A$	$2^1, .., 2^s$	$j = 2^0, .., 2^s$	2	2^s	bits
ModInv p^k	k	$1M$	1	$j = 1, .., k$	any	any	digits
	k	$1M + 1A$	k				
ModInv 2^k	k	$1A$	k	$j = 1, .., k$	2	any	bits

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, $(1 - a \cdot c)^k$ is a multiple of p^k. This gives $a \cdot x = 1 \pmod{p^k}$.

7 Inversion Mod 2^k

The proposed algorithm significantly simplifies when $p = 2$, and it constitutes an efficient alternative to the existing algorithms. 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:** $a, k$ where $a$ is odd and $a < 2^k$
**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 presented algorithms.

The algorithm produces the binary result $x = (100111)_2 = 39$. This is indeed correct, since $23^{-1} = 39 \pmod{2^6}$. Moreover, our algorithm computes $23^{-1} \pmod{2^j}$ for $k = 1, \ldots, 6$, which are given in base 2 as: $(1)_2 = 1$, $(11)_2 = 3$, $(111)_3 = 7$, $(0111)_3 = 7$, $(00111)_3 = 7$, and $(100111)_3 = 39$, as shown in Table 8.

8 Complexity Analysis

For each algorithm presented in this paper, we analyze the number steps (within the for-loop), the number of arithmetic operations in each step, and the types and sizes of the operands involved, and what the algorithm actually computes. These algorithms differ from another in terms of the number of steps, the types of outputs (for example, the whole number at once or digit-by-digit) and whether or not the consecutive inverses are computed.

A realistic complexity analysis of the algorithms would require that we count of number of bit operations. However, operations requiring $O(1)$ bit operations per step can safely be ignored. These include *check the LSB* and *right or left shift of the operands*. Two important parameters are k (the size of a) and $s = \log_2(k)$. The symbols D, M, and A stand for the processing times for division, multiplication, and addition or subtraction operations. Table 9 summarizes our analysis.

There are four aspects of these modular inversion algorithms, and the interpretation of their complexity results should take them into account.

First of all, these algorithms can be divided into two categories in terms of their asymptotic complexity: linear versus logarithmic, i.e., those requiring k steps versus those requiring $s = \log_2(k)$ steps. There are 3 algorithms requiring logarithmic time which are Arazi and Qi Algorithm 3, and Dumas Algorithms for modulus p^k and 2^k. The remaining 5 algorithms require $O(k)$ steps. It is not automatically concluded that the logarithmic time algorithms are superior. First of all, this will depend on the size of k. As we have discussed in Section 2, the most common use of the modular inversion algorithm is for the implementation of the Montgomery multiplication algorithm. In regard to this application, we note. The classical Montgomery algorithm [13] requires k to be as large as the size of the RSA modulus n, thus, 512 to 2048. Here, the linear versus logarithmic complexity would be hugely different. However, the classical algorithm is hardly used in practice. The most deployed implementations use the CIOS algorithm [11] which chooses k to be the word size of the processor. If $k = 32$, then $s = \log_2(32) = 5$, and thus, the difference between linear versus logarithmic is not that great. For example, comparing Algorithm 4 to ModInverse algorithm, we see that the former requires $5 \cdot (3M + 2A)$ operations while the latter requires $32 \cdot A$ operations.

Algorithmically, a multiplication operation has at least logarithmic depth in gate delays compared to addition (in both cases of carry save and carry propagate adders) which is 4 or 5 if the operand size is 16 or 32 bits. Taking Pentium as a modern architecture example, we see that integer multiplication (in Pentium 2/3 and 4) Takes 5, 7 clock cycles of latency compared to integer addition which take 1 clock cycle, as seen in Table 5.2 of [7]. On the other hand, the latency is 1 cycle for an integer addition and 3 cycles for an integer multiplication in Intel Core Duo 2. One can find the latencies and throughput in Appendix C, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/TC.2020.2970411. of the "Intel 64 and IA-32 Architectures Optimization Reference Manual", which is located in [8].

We conclude that for $k = 32$, Algorithm 4 (Arazi and Qi) requires $5 \cdot (3M + 2A)$ operations, Dumas Algorithm requires $5 \cdot (2M + 1A)$ operations, while ModInverse requires only $32A$ operations. Assuming $M = 4A$, Algorithm 4 requires $70A$, Dumas Algorithm requires $45A$ and ModInverse requires only $32A$ operations. For $M = 3A$, the number of additions becomes $55A$, $35A$ and $32A$ for the Algorithm 4, Dumas Algorithm, and ModInverse algorithm.

The second point about comparing these 8 algorithms is that they can be divided into 2 categories: algorithms computing the inverse mod p^k (or 2^k) for any value of k versus algorithms that work only for specific values of k, here namely, for those k that is a power of 2. The modular inversion algorithms that work for any k are the Dussé and Kaliski Algorithm, Arazi and Qi Algorithms 2 and 3, and ModInverse Algorithms for p^k and 2^k. The remaining 3 algorithms compute the inverse mod p^k where $k = 2^s$. These algorithms will require an additional reduction to compute the inverse for an arbitrary k; for example, to compute the inverse mod p^{29}, we will first have the compute the inverse for mod p^{2^s} for an s such that $2^s > k$, and then obtain the inverse mod p^{29} by an additional reduction operation.

The third point about comparing these 8 algorithms is that they can be divided into 2 categories: algorithms that compute and output the consecutive inverses (for example, for $j = 1, 2, \ldots, k$ or $j = 1, 2, \ldots, s$) versus algorithms that compute the output for a single k only (however, consecutive inverses can still be obtained by reductions). Only Arazi and Qi Algorithm 2 and 3 compute the inverse for a single modulus; while the Dussé and Kaliski Algorithm and ModInverse Algorithms for p^k and 2^k compute and output the consecutive inverses for mod for $i = 1, 2, \ldots, k$. On the other hand, Arazi and Qi Algorithm 4 and Dumas Algorithms for p^k and 2^k) compute the inverse for consecutive s moduli, specifically for p^{2^j} for $j = 1, 2, \ldots, s$.

The fourth point about comparing these 8 algorithms is that they can be divided into 2 categories: algorithms that work only for $p = 2$ and algorithms that work for any prime p. The first category contains 6 algorithms; while only two algorithms, namely ModInverse and Dumas algorithms work for any p.

Finally, we note that the ModInverse algorithm is the only algorithm that produce the digits (base p or base 2) of the inverse directly, starting from the least significant digits proceeding to the most significant. These digit-by-digit arithmetic algorithms are also named as *on-line arithmetic*. Such algorithms introduce parallelism between sequential operations by overlapping these operations in a digit-pipelined fashion [6].

Furthermore, the ModInverse algorithm for mod 2^k requires the minimal number of arithmetic operation (just a single addition) among all 8 algorithms.

9 Conclusion

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 [2]. 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 binary version of the proposed algorithm (that is, when $p = 2$) is significantly more efficient than the existing algorithms for computing $a^{-1} \pmod{2^k}$ when k is small, which is the case for the CIOS Montgomery multiplication algorithm. Moreover, the proposed algorithm computes all inverses mod p^i or 2^i for $i = 1, 2, \ldots, k$ and work for an arbitrary k. We have also described and analyzed 6 existing algorithms, and provided an extensive comparison and interpretation o the proposed algorithm.

Our proposed algorithm stands out as being the only one that works for any p, any k, and digit-by-digit. Moreover it requires the minimal number of arithmetic operations (just a single addition) per step.

Acknowledgments

The author thanks to Francois Grieu for comments in [6], Watson Ladd for comments on Dixon's algorithm being actually due to Hensel, Markku-Juhani Olavi Saarinen for comments on Newton-Raphson algorithm, and Michael Scott for reminding the references [1], [3] and commenting on the first version of this article. The author also thanks reviewers for pointing out that the article [3] was also published in IEEE Transactions on Computers [4] and for greatly improving and correcting the article with their rigorous reviews.

References

[1] O. Arazi and H. Qi, "On calculating multiplicative inverses modulo 2^m," *IEEE Trans. Comput.*, vol. 57, no. 10, pp. 1435–1438, Oct. 2008.
[2] J. D. Dixon,"Exact solution of linear equations using p-adic expansions," *Numerische Mathematik*, vol. 40, no. 1, pp. 137–141, 1982.
[3] J. Dumas, "On Newton-Raphson iteration for multiplicative inverses modulo prime powers," *arXiv:1209.6626v3*, 2012. [Online]. Available: https://arxiv.org/abs/1209.6626v3
[4] J. Dumas, "On Newton-Raphson iteration for multiplicative inverses modulo prime powers," *IEEE Trans. Comput.*, vol. 63, no. 8, pp. 2106–2109, Aug. 2014.
[5] S. R. Dussé and B. S. Kaliski Jr, "A cryptographic library for the Motorola DSP56000," in *Proc. Workshop Theory Appl. Cryptographic Techn.*, 1990, pp. 230–244.
[6] F. Grieu, "Answer to 'How to determine the multiplicative inverse modulo 64 (or other power of two)?'," StackExchange Cryptography, 2017. [Online]. Available: https://crypto.stackexchange.com/questions/47493
[7] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, New York, NY, USA: Springer, 2004.
[8] Intel, "Intel 64 and IA-32 architectures software developer manuals," Jan. 18, 2018. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm
[9] B. S. Kaliski Jr., "The Montgomery inverse and its applications," *IEEE Trans. Comput.*, vol. 44, no. 8, pp. 1064–1065, Aug. 1995.
[10] Ç. K. Koç, "High-speed RSA implementation," RSA Lab., Hebron, CT, Tech. Rep. TR 201, Nov. 1994.
[11] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr., "Analyzing and comparing Montgomery multiplication algorithms," *IEEE Micro*, vol. 16, no. 3, pp. 26–33, Jun. 1996.
[12] E. V. Krishnamurthy and V. K. Murty, "Fast iterative division of p-adic numbers," *IEEE Trans. Comput.*, vol. 32, no. 4, pp. 396–398, Apr. 1983.
[13] P. L. Montgomery, "Modular multiplication without trial division," *Math. Comput.*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
[14] E. Savaş and Ç. K. Koç, "The montgomery modular inverse - revisited," *IEEE Trans. Comput.*, vol. 49, no. 7, pp. 763–766, Jul. 2000.
[15] E. Savaş and Ç. K. Koç, "Montgomery inversion," *J. Cryptographic Eng.*, vol. 8, pp. 201–210, 2017.