

# On inversion modulo pseudo-Mersenne primes

Michael Scott

Cryptographic Researcher  
MIRACL Labs  
mscott@indigo.ie

**Abstract.** It is well established that the method of choice for implementing a side-channel secure modular inversion, is to use Fermat's little theorem. So  $1/x = x^{p-2} \pmod p$ . This can be calculated using any square-and-multiply method safe in the knowledge that no branching or indexing with potentially secret data (such as  $x$ ) will be required. However in the case where the modulus  $p$  is a pseudo-Mersenne, or Mersenne, prime of the form  $p = 2^n - c$ , where  $c$  is small, this process can be optimized to greatly reduce the number of multiplications required. Unfortunately an optimal solution must it appears be tailored specifically depending on  $n$  and  $c$ . What appears to be missing from the literature is a near-optimal heuristic method that works well in all cases.

**Keywords:** Elliptic Curves

## 1 Introduction

In elliptic curve cryptography (ECC), a pseudo-Mersenne modulus is often proposed, as it introduces no known weaknesses, and allows a much faster modular reduction algorithm, independent of the underlying computer architecture. Less appreciated is the fact that modular inversions can also benefit from such a choice. For maximum efficiency projective coordinates are most often used for ECC, which greatly diminishes the significance of the cost of modular inversion. Nevertheless in a paper that popularised such moduli, Bernstein [1] pointed out that modular inversion still absorbed 7% of the time for a curve computation, and in a recent paper Nath and Sarkar [7] point out that as a consequence of the vigorous optimization of the other aspects of implementation, this may rise as high as 9%.

Unfortunately working out the optimal strategy in the general case where  $p = 2^n - c$  is not that simple, as it depends on both  $n$  and  $c$ . Since the binary expansion of such a  $p$  mostly consists of 1 bits, a simple square and multiply algorithm will be particularly inefficient, requiring nearly as many multiplications as squarings. In fact it is impossible to avoid  $n - 1$  squarings. So any attempt at optimisation will focus on reducing the number of multiplications.

To get an idea of what is possible, let us deconstruct Bernsteins approach for his prime  $2^{255} - 19$ , as described by Bos [2]. The value after the # indicates the exponent of  $x$  at that stage in the calculation. In this case the inverse will be calculated as  $1/x = x^{255} - 21$

$$\begin{aligned}
x_2 &\leftarrow x^2 & \# & 2 \\
x_4 &\leftarrow x_2^2 & \# & 4 \\
x_8 &\leftarrow x_4^2 & \# & 8 \\
x_9 &\leftarrow x.x_8 & \# & 9 \\
x_{11} &\leftarrow x_2.x_9 & \# & 11 \\
x_{22} &\leftarrow x_{11}^2 & \# & 22 \\
t_0 &\leftarrow x_9.x_{22} & \# & 2^5 - 1 \\
\\
t_1 &\leftarrow t_0^{2^5}.t_0 & \# & 2^{10} - 1 \\
t_2 &\leftarrow t_1^{2^{10}}.t_1 & \# & 2^{20} - 1 \\
t_3 &\leftarrow t_2^{2^{20}}.t_2 & \# & 2^{40} - 1 \\
t_4 &\leftarrow t_3^{2^{10}}.t_2 & \# & 2^{50} - 1 \\
t_5 &\leftarrow t_4^{2^{50}}.t_4 & \# & 2^{100} - 1 \\
t_6 &\leftarrow t_5^{2^{100}}.t_5 & \# & 2^{200} - 1 \\
t_7 &\leftarrow t_6^{2^{50}}.t_4 & \# & 2^{250} - 1 \\
\\
x &\leftarrow t_6^{2^5}.x_{11} & \# & 2^{255} - 21
\end{aligned}$$

On closer examination the process consists of 3 phases. It makes extensive use of the identity

$$x^{2^{n+m}-1} = (x^{2^n-1})^{2^m}.x^{2^m-1} \quad (1)$$

By the end of the second phase we have calculated  $x^{2^{250}-1}$ . Note that  $c + 2 = 21$  and  $2^5 - 21 = 11$ . The third phase then calculates  $x^{2^{255}-21} = (x^{2^{250}-1})^{2^5}.x^{2^5-21}$ . It is easily confirmed that the whole process requires 254 squarings and just 11 multiplications.

Clearly the final phase requires only one extra multiplication. In the first phase the important work is the calculation of  $k = x^{11}$  and  $x^{31} = x^{2^5-1}$ . The former to provide the “key” value needed by the final phase, and the latter is used to kick-start the second phase with some value of the form  $x^{2^m-1}$ . Therefore there are two addition chain calculations involved, the first to determine  $x^{31}$ , with the constraint that the chain pass through the value of  $x^{11}$ . The second applies to the second phase where the point is to calculate in as few steps as possible  $x^{2^{250}-1}$  using identity (1) above. The first chain requires a multiplication for each addition step in the chain (but not for a doubling). The second addition chain requires a multiplication for each step, so the shortest possible chain is desired.

To generalise this approach, first choose a value  $w$  such that  $2^w > c + 2$ . Calculate the key  $k = x^{2^w-c-2}$  and  $x^{2^w-1}$  in phase 1. Raise this value up to  $x^{2^{n-w}-1}$  in phase 2, and finally calculate the inverse of  $x$  as  $x^{2^n-c-2}$  in phase 3. Clearly what makes this process rather awkward is the involvement of addition chains. As is well known calculating shortest addition chains is an NP-complete problem, and therefore not suitable to be calculated on-the-fly. So it appears

that an optimal solution must be tailored by hand to each pseudo-Mersenne prime of interest.

The alternative is to come up with a heuristic approach which can quickly produce a good (if sub-optimal) solution given only  $n$  and  $c$ . That is the contribution of this paper.

## 2 A heuristic approach

Choose  $w$  to be the smallest number such that  $2^w > c + 2$ . Next calculate and store powers of  $x$  using this fixed addition chain of exponents

$$[1\ 2\ 3\ 6\ 12\ 15\ 30\ 60\ 120\ 240\ 255]$$

This will require three multiplications and seven squarings. Next evaluate the key value  $k = x^{2^w - c - 2}$  by multiplying together appropriate powers, which will probably require a few more multiplications. It is easily confirmed that this is possible for any values of  $2^w - c - 2 < 745$ , which will cover most cases of interest. Note that these stored values include all of  $x, x^{2^2-1}, x^{2^4-1}, x^{2^8-1}$ . Extract these values, and use them to initialise another addition chain in the exponents, where the entry  $i$  represents the power  $x^{2^i-1}$ .

$$[1\ 2\ 4\ 8\ 16\ \dots\ 2^m\ \dots\ n - w]$$

which is just the powers of 2 while  $n - w < 2^m$ . Finally complete the addition chain using a simple binary method. This chain dictates how to use the identity (1) to ramp up  $x^{2^8-1}$  to  $x^{2^{n-w}-1}$ . Finally calculate the result by squaring this final value  $w$  times, and multiplying it by the key  $k$ .

For the modulus  $2^{255} - 19$ ,  $w = 5$  and the key value will be  $k = x^{11} = x^2 \cdot x^3 \cdot x^6$ , which would require two extra multiplications. The addition chain in the exponents will be

$$[1\ 2\ 4\ 8\ 16\ 32\ 64\ 128\ 192\ 224\ 240\ 248\ 250]$$

which will require 9 further multiplications (recall that the first 4 entries are already available), plus one for phase 3. The total number of multiplications is 15, somewhat inferior to the optimal value of 11.

When it comes to implementing this method, note that the same array used to store the powers of  $x$  in phase 1, can be re-used in phase 2. Specifically the 11 element array used in phase 1 can be re-used in phase 2 for values of  $n$  less than 2048. Therefore the method as described will work for all primes of the form  $2^n - c$  for  $n < 2048$  and  $c < 1024$ . See algorithm 1.

## 3 Performance

To get an idea of just how suboptimal this general approach will be, we compare it with some implementations that are already “out there” in the wild, in existing

---

**Algorithm 1** Modular inversion with respect to a pseudo-Mersenne prime

---

INPUT: An element  $x \in \mathbb{F}_p$ ,  $n$  and  $c$ , where prime  $p = 2^n - c$ ,  $n < 2048$ ,  $c < 1024$ INPUT: An array  $a=[1,2,3,6,12,15,30,60,120,240,255]$ OUTPUT:  $1/x \bmod p$ 

```
1: function FPINV( $x, n, c$ )
2:    $h[0] \leftarrow x$  ▷ Phase 1
3:    $h[1] \leftarrow x^2$ 
4:    $h[2] \leftarrow x \cdot h[1]$ 
5:    $h[3] \leftarrow h[2]^2$ 
6:    $h[4] \leftarrow h[3]^2$ 
7:    $h[5] \leftarrow h[4] \cdot h[2]$ 
8:    $h[6] \leftarrow h[5]^2$ 
9:    $h[7] \leftarrow h[6]^2$ 
10:   $h[8] \leftarrow h[7]^2$ 
11:   $h[9] \leftarrow h[8]^2$ 
12:   $h[10] \leftarrow h[9] \cdot h[5]$ 
13:   $b \leftarrow 0$ 
14:   $w \leftarrow 1$ 
15:  while  $w < c + 2$  do
16:     $w \leftarrow 2w$ 
17:     $b \leftarrow b + 1$ 
18:     $j \leftarrow w - c - 2$ 
19:    if  $j \neq 0$  then
20:       $i \leftarrow 10$ 
21:      while  $a[i] > j$  do
22:         $i \leftarrow i - 1$ 
23:         $k \leftarrow h[i]$  ▷ Calculate Key
24:         $j \leftarrow j - a[i]$ 
25:      while  $j \neq 0$  do
26:         $i \leftarrow i - 1$ 
27:        if  $k \geq a[i]$  then
28:           $k \leftarrow k \cdot h[i]$ 
29:           $j \leftarrow j - a[i]$ 
▷ Phase 2
30:   $h[1] \leftarrow h[2]$  ▷ Re-use the array
31:   $h[2] \leftarrow h[5]$ 
32:   $h[3] \leftarrow h[10]$ 
33:   $j \leftarrow 3$ 
34:   $m \leftarrow 8$ 
35:   $n \leftarrow n - b$ 
36:  while  $2m < n$  do ▷ Double up
37:     $t \leftarrow h[j]$ 
38:     $j \leftarrow j + 1$ 
39:    for  $i \leftarrow 0; i < m; i \leftarrow i + 1$  do
40:       $t \leftarrow t^2$ 
41:       $h[j] \leftarrow t \cdot h[j - 1]$ 
42:       $m \leftarrow 2m$ 
43:   $l \leftarrow n - m$ 
44:   $r \leftarrow h[j]$ 
45:  while  $l \neq 0$  do ▷ Complete addition chain
46:     $m \leftarrow m/2$ 
47:     $j \leftarrow j - 1$ 
48:    if  $l \geq m$  then
49:       $l \leftarrow l - m$ 
50:       $t \leftarrow r$ 
51:      for  $i \leftarrow 0; i < m; i \leftarrow i + 1$  do
52:         $t \leftarrow t^2$ 
53:         $r \leftarrow t \cdot h[j]$  ▷ Phase 3
54:  for  $i \leftarrow 0; i < b; i \leftarrow i + 1$  do
55:     $r \leftarrow r^2$ 
56:  if  $w - c - 2 \neq 0$  then
57:     $r \leftarrow r \cdot k$ 
58:  return  $r$ 
```

---

code and libraries. Specifically we looked at the code associated with references [1], [3], [4], [7] and [8] in October 2018. In Table 1 we show the number of multiplications as implemented, and as required by the method described here. In cases where the number of squarings exceeds  $n - 1$ , these are recorded as well. As expected our method is suboptimal, but not by much. In fact it may be considered as better than expected. For example Bos et al. [3] found that they could always calculate the modular inversion using at most  $1.11 \lceil \log_2(p) \rceil$  squarings and multiplications. With our much larger sample we find that it can be done using at most  $1.09 \lceil \log_2(p) \rceil$ . As can be seen we also find that on occasion this method improves on a tailored solution.

Prime	As Implemented	This Method
$2^{127} - 1$	10 [7]	12
$2^{221} - 3$	12 [7]	12
$2^{222} - 117$	12 [7]	14
$2^{251} - 9$	14 [7]	15
$2^{255} - 19$	11 [1]	15
$2^{256} - 189$	21(260) [3]	14
$2^{266} - 3$	12 [7]	12
$2^{336} - 3$	13 [8]	13
$2^{382} - 105$	14 [7]	16
$2^{383} - 187$	15 [7]	17
$2^{384} - 317$	15 [3]	18
$2^{414} - 17$	14 [7]	14
$2^{511} - 187$	15 [7]	18
$2^{512} - 569$	16(512) [3], 18 [7]	19
$2^{521} - 1$	13 [4]	13
$2^{607} - 1$	14 [7]	15
$2^{751} - 165$	19 [7]	19
$2^{832} - 143$	16 [7]	17
$2^{896} - 213$	18 [7]	18
$2^{960} - 167$	19 [7]	17
$2^{1024} - 105$	20 [7]	18
$2^{1088} - 89$	16 [7]	17

Table 1: Number of Multiplications (Squarings)

## 4 Two birds, one stone

The same idea can be used to efficiently calculate modular square roots, as often required in ECC implementation for point decompression. If the prime  $p = 3 \pmod 4$ , then the square root  $y$  of a quadratic residue  $x$  can be calculated from  $y = x^{(p-3)/4}$  as  $\sqrt{x} = xy \pmod p$ , and if  $p = 5 \pmod 8$ , the square root can be calculated from  $y = x^{(p-5)/8}$ , with a small amount of extra work – see algorithm 3.37 in chapter 3 of [6] for details. By modifying our algorithm in an obvious

way to calculate these  $y$  values instead, and then re-using the same function, the inverse  $x^{p-2} \bmod p$  can be found in the former case as  $xy^4$ , and in the latter case as  $x^3y^8$ . For an example of the deployment of this idea, see [5].

## 5 Conclusion

When implementing an easy-to-maintain general purpose cryptographic library, it helps to avoid duplication and special case implementation where possible, while still obtaining respectable performance. In the case of side-channel resistant modular inversion with respect to pseudo-Mersenne primes, the method of choice is to use Fermat's Little Theorem. Optimal performance seems to require a tailored solution for each modulus of interest, a process prone to error. Here we have presented a slightly suboptimal algorithm which provides acceptable performance, but using a single function which works well for all cases likely to be of interest to cryptographers.

## References

1. D. Bernstein. Curve25519: New Diffie-Hellman speed records. In *PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Berlin Heidelberg, 2006.
2. J. Bos. Constant time modular inversion. *Journal of Cryptographic Engineering*, 4(4):275–281, 2014.
3. J. Bos, C. Costello, P. Longa, and M. Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. Cryptology ePrint Archive, Report 2014/130, 2014. <http://eprint.iacr.org/2014/130>.
4. R. Granger and M. Scott. Faster ECC over  $\mathbb{F}_{2^{521}-1}$ . In *Public-Key Cryptography – PKC 2015*, volume 9020 of *Lecture Notes in Computer Science*, pages 539–553. Springer Berlin Heidelberg, 2015.
5. M. Hamburg. <https://github.com/otrv4/little-ed448-Goldilocks>.
6. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography*. CRC Press, Boca Raton, Florida, 1996. URL: <http://cacr.math.uwaterloo.ca/hac>.
7. K. Nath and P. Sarkar. Efficient inversion in (pseudo-)Mersenne prime order fields. Cryptology ePrint Archive, Report 2018/985, 2018. <http://eprint.iacr.org/2018/985>.
8. M. Scott. Ed3363 (highfive) – an alternative elliptic curve. Cryptology ePrint Archive, Report 2015/991, 2015. <http://eprint.iacr.org/2015/991>.