Efficient Exponentiation

In the previous chapter we have used an exponentiation as a basic operation, i.e., we did not detail on how (modular) exponentiations with long numbers are realized in practice. In the following, we introduce and compare the most common exponentiation algorithms and their complexity. This chapter is based in parts on the algorithms in [MvOV97, Chapter 14.6].

Exponentiation with Signed-Digit Representation

All exponentiation algorithms considered so far have been using the standard binary representation of the exponent. In this section we will show that it can be advantageous to represent the exponent not only with the values $\{0, 1\}$ but with the set $\{-1, 0, 1\}$. The general strategy is that the Hamming weight, i.e., the number of non-zero digits, is decreased which will result in fewer multiplications if the square-and-multiply algorithm is being used.
In practice this re-coding yields faster point multiplication algorithms (which are equivalent to exponentiation algorithms) in the case of elliptic curve cryptosystems but not for RSA or classical discrete logarithm schemes.

Preliminary Observations

So far we restricted ourselves to purely binary representations of the exponent, e.g.,

$$e = 31 = 11111.$$

If we extend the set of digits of the exponents from $\{0, 1\}$ to $\{-1, 0, 1\}$, the exponents can often be represented by a lower Hamming weight, e.g.,

$$e = 31 = 1, 0, 0, 0, 0, -1.$$

This holds since $e = 1 \cdot 2^5 + (-1) \cdot 2^0 = 32 - 1 = 31$. In the following we'll use $\overline{1}$ to denote -1, e.g., $31 = 10000\overline{1}$. We know that in many exponentiation algorithms, for instance the square-and-multiply algorithm, the number of operations is lowered if the exponent's Hamming weight is reduced. The question now is whether there are crypto

schemes where we can take advantage of the signed digit representation.

Let's start with modular exponentiation which is needed for RSA and discrete logarithm schemes: As an example, we want to compute A^{31} using the signed-digit representation of 31 from above:

$$A^{31} = A^{10000\bar{1}} \bmod m$$

We recall that computing $A^{31} = A^{11111}$ using the square-and-multiply algorithm requires 4 squarings and 4 multiplications.

We now compute $A^{10000\bar{1}}$ using the square-and-multiply algorithm as shown below:

Step	Exponentiation	Operation
Step 1	$A^2 = A^{10} \bmod m$	SQ
Step 2	$A^4 = A^{100} \bmod m$	SQ
Step 3	$A^8 = A^{1000} \bmod m$	SQ
Step 4	$A^{16} = A^{10000} \bmod m$	SQ
Step 5	$A^{32} = A^{100000} \bmod m$	SQ
Step 6	$A^{31} = A^{100000} \cdot A^{-1} \bmod m$	MUL + INV

Even though we reduced the number of multiplications and squarings from 8 to 6, the modular inversion in Step 6 is extremely costly so that the reduced number of multiplications does *not* pay off. However, the situation is different for elliptic curves cryptosystems (ECC). The core operation is:

$$e \cdot P$$

where e is an integer and $P = (x_P, y_P)$ a point on the curve. A property of elliptic curves is that it is trivial to compute the inverse $-P$ of a point:

$$-P = (x_P, -y_P)$$

i.e., we simply take the negative of the y-coordinate.

Let's try to compute $31 \cdot P$ using the double-and-add algorithm which is completely analogues to the square-and-multiply algorithm, with squarings replaced by point doubling and multiplications by point additions. Again, we represent the scalar 31 with signed digits:

$$31 \cdot P = (10000\bar{1}) \cdot P$$

Step	Point Multiplication	Operation
Step 1	$P + P = 2P = (10)P$	DOUBLE
Step 2	$2P + 2P = 4P = (100)P$	DOUBLE
Step 3	$4P + 4P = 8P = (1000)P$	DOUBLE
Step 4	$8P + 8P = 16P = (10000)P$	DOUBLE
Step 5	$16P + 16P = 32P = (100000)P$	DOUBLE
Step 6	$32P + (-P) = 31P = (1000\overline{1})P$	ADD + INV

We note that Step 6 simply adds $(x_P, -y_P)$ to the previous result $32P$, where $P = (x_P, y_P)$. The signed-digit point multiplication has an arithmetic complexity of 5 point doublings and 1 point addition, where the straight double-and-add algorithm would require 4 doublings and 4 additions for $31 \cdot P$.

Non-Adjacent Form (NAF) Exponentiation

Interestingly, this trick can be generalized. Using the digit set $\{\overline{1}, 0, 1\}$ it is possible to represent any integer such that there are no two non-zero digits (i.e., $\overline{1}$ or 1) next to each other, i.e., a $\overline{1}$ or 1 is always followed by a 0. We define this representation now:

Definition 0.0.1 NAF
The non-adjacent form *(NAF) of a positive integer e is the representation:*

$$e = \sum_{i=0}^{l-1} e_i \cdot 2^i \ , \quad \textit{with } e_i \ \epsilon \ \{-1, 0, 1\}$$

and $e_{l-1} \neq 0$*, such that no two neighbouring digits* e_i *are non-zero.*

Let's first look at an example for a NAF representation.

Example 0.1 Given is the integer $e = 157$. Its regular binary representation is:

$$157 = 128 + 16 + 8 + 4 + 1 = 10011101_2$$

If we convert it to the NAF representation we obtain:

$$157 = 128 + 32 - 4 + 1 = 10100\overline{1}01$$

Note that the Hamming weight, i.e., the number of non-zero digits, is five in the regular representation but only four in the NAF form.

△

The NAF has a number of interesting properties (adopted from [HMV04]):

(i) e has a unique NAF denoted NAF(e).

(ii) NAF(e) has the fewest nonzero digits of any signed-digit representation of e.

(iii) The length of NAF(e) is at most one more than the length of the binary representation of e.

(iv) If the length of NAF(e) is l, then $\frac{2^l}{3} < e < \frac{2^{l+1}}{3}$.

(v) The average density of nonzero digits among all NAFs of length l is approximately $\frac{1}{3}$.

The NAF is based on the observation that the bit pattern "11" at any point in the binary representation of an integer can be replaced by "$10\overline{1}$". Let's look at a few simple examples:

Example 0.2

(i) $e = 3 = 11_2$
$e = 11_2 = 10\overline{1}$

(ii) $e = 6 = 110_2$
$e = 110_2 = 10\overline{1}0$

(iii) $e = 12 = 1100_2$
$e = 1100_2 = 10\overline{1}00$

△

The trick is not restricted to the special cases shown above where $e = 1100\ldots0$. In general, we consider integers with two adjacent ones:

$$e = e_{l-2}, \ldots, e_{i+2}, 1, 1, e_{i-1}, \ldots, e_0$$

where $e_{i+1} = e_i = 1$ and all other $e_j \; \epsilon \; \{0, 1\}$. These two bits ($e_{i+1}$, e_i) can now be replaced by

$$e = e_{l-2}, \ldots, e_{i+2} + 1, 0, \overline{1}, e_{i-1}, \ldots, e_0$$

This simple transformation holds since e_{i+1} represents 2^{i+1} and e_i represents 2^i and we have the equality:

$$2^{i+1} + 2^i = 2^{i+2} - 2^i$$

We recall that your goal is to reduce the Hamming weight of the exponent e, where the Hamming weight is defined here as the number of values 1 and -1 in the representation

of e. The re-coding actually reduces the Hamming weight if at least three 1 are adjacent in the original representation of e, i.e., the number looks like this in the standard binary representation:

$$e = e_{l-2}, \ldots, 0, 1, 1, 1, \ldots, e_0$$

Performing the recoding yields:

$$e = e_{l-2}, \ldots, 1, 0, 0, \overline{1}, \ldots, e_0$$

As one can easily see, the three 1s in the original representation are replace by only two non-zero values, 1 and $\overline{1}$.

With this observation we can now introduce the re-coding algorithm which computes the NAF representation of a positive integer e:

Algorithm 0.1 ***Computing the NAF of a positive integer (adopted from [HMV04, Alg. 3.30])***

Input: A positive integer e
Output: NAF(e)

1. $X \leftarrow e \quad , \quad i \leftarrow 0$
2. WHILE $X \geq 1$ DO
   - 2.1 IF X is odd THEN
     $e_i \leftarrow 2 - (X \bmod 4)$
     $X \leftarrow X - e_i$
   - 2.2 ELSE $e_i \leftarrow 0$
   - 2.3 $X \leftarrow X/2$
     $i \leftarrow i + 1$
3. RETURN $(e_{i-1}, e_{i-2}, \ldots, e_1, e_0)$

This is a right-to-left algorithm. The crucial part is Step 2.1. "$X \bmod 4$" yields the value of the two rightmost bits of the current X. Since this expression is only reached if X is odd, it has one of the two values:

$$X \bmod 4 = \begin{cases} 01_2 = 1 \\ 11_2 = 3 \end{cases}$$

If $X \bmod 4 = 01$, the bit e_i is set to $2 - 1 = 1$, i.e., no re-coding takes place. If $X \bmod 4 = 11$, the bit e_i is set to $2 - 3 = -1 = \overline{1}$ and the expression $X \leftarrow X - e_i$

computes

$$X - e_i = X - (-1) = X + 1 = (x, \ldots, x, x, 1, 1) + 1 = x, \ldots, x, x + 1, 0, 0$$

where "x" denotes any value of 0 or 1. This expression is shifted to the right in Step 2.3 of the current iteration.

Example 0.3 The goal is the conversion of the integer $e = 157 = 128 + 16 + 8 + 4 + 1 = 10011101_2$ into the NAF representation. All internal steps of the algorithm are shown in Table 1.

Step	Operation
1	$i = 0$
2	WHILE $X \geq 1$ DO
2.1	$X = 157 = odd$
	$e_0 = 2 - (X \bmod 4) = 2 - (157 \bmod 4) = 2 - 1 = 1$
	$X = X - e_0 = 157 - 1 = 156$
2.3	$X = X/2 = 156/2 = 78 = 1001110_2$
	$i = 1$
2.1	$X = 78 \neq odd$
2.2	$e_1 = 0$
2.3	$X = X/2 = 78/2 = 39 = 100111_2$
	$i = 2$
2.1	$X = 39 = odd$
	$e_2 = 2 - (X \bmod 4) = 2 - (39 \bmod 4) = 2 - 3 = -1$
	$X = X - e_2 = 39 - (-1) = 40$
2.3	$X = X/2 = 40/2 = 20 = 10100_2$
	$i = 3$
2.1	$X = 20 \neq odd$
2.2	$e_3 = 0$
2.3	$X = X/2 = 20/2 = 10 = 1010_2$
	$i = 4$
2.1	$X = 10 \neq odd$
2.2	$e_4 = 0$
2.3	$X = X/2 = 10/2 = 5 = 101_2$
	$i = 5$
2.1	$X = 5 = odd$
	$e_5 = 2 - (X \bmod 4) = 2 - (5 \bmod 4) = 2 - 1 = 1$
	$X = X - e_5 = 5 - 1 = 4$
2.3	$X = X/2 = 4/2 = 2 = 10_2$
	$i = 6$
2.1	$X = 2 \neq odd$
2.2	$e_6 = 0$
2.3	$X = X/2 = 2/2 = 1$
	$i = 7$
2.1	$X = 1 = odd$
	$e_7 = 2 - (X \bmod 4) = 2 - (1 \bmod 4) = 2 - 1 = 1$
	$X = X - e_7 = 1 - 1 = 0$ ✓
3	RETURN $(e_7, e_6, e_5, e_4, e_3, e_2, e_1, e_0)$

Table 1: Internal steps of the NAF computation of the integer 95

We note that the Hamming weight of the binary representation is 5

whereas the NAF representation has a Hamming weight of 4. We can verify the result $e = 10011101_2 = 10100\bar{1}01$ as follows:

$$\begin{array}{rcccccccccc} e & = & e_7 & e_6 & e_5 & e_4 & e_3 & e_2 & e_1 & e_0 \\ & = & 1 & 0 & 1 & 0 & 0 & -1 & 0 & 1 \\ & = & 2^7 & & +2^5 & & & -2^2 & & +1 \\ & = & 128 & & +32 & & & -4 & & +1 \\ & = & 157 & & & & & & & \end{array}$$

$\triangle$

Using the NAF representation of an integer e it is straightforward to create a modified version of the double-and-add-algorithm:

Algorithm 0.2 ***Binary NAF method for point multiplication (adopted from [HMV04, Alg. 3.31])***

Input: NAF representation of positive integer $e = \sum_{i=0}^{l-1} e_i \cdot 2^i$, $e_i \in \{-1, 0, 1\}$
Point P of an elliptic curve
Output: $e \cdot P$

1. $Q \leftarrow \mathcal{O}$
2. FOR i FROM $l-1$ DOWN TO 0
   - 2.1 $Q \leftarrow 2Q$
   - 2.2 IF $e_i = 1$ THEN $Q \leftarrow Q + P$
   - 2.3 IF $e_i = -1$ THEN $Q \leftarrow Q - P$
3. RETURN Q

Since the Hamming weight of an l-bit integer in NAF representation is on average $\frac{1}{3}$, the algorithm has a complexity of approximately:

$$\begin{array}{rcl} \#\text{Add} & = & \dfrac{l+1}{3} \\ \#\text{Double} & = & l \end{array}$$

for a random integer e. We note that the standard double-and-add algorithm requires

$$\begin{array}{rcl} \#\text{Add} & = & \dfrac{l-1}{2} \\ \#\text{Double} & = & l-1 \end{array}$$

Note that the bit length of the original scalar e is often $l-1$ rather than l.

It is straightforward to extend the NAF-based-double-and-add algorithm to a windows-based algorithm, cf. [HMV04].
We conclude this section with two remarks about addition chains.

1. Finding the shortest possible, i.e., optimum, addition chain is a very mathematical problem (certain generalizations of this problem are known to be NP-hard).

2. However, there are efficient algorithms known which generate sub-optimum addition chains for e. A good introduction to the general area of addition chains can be found in [Knu98].

References

[BB03] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, pages 1–14, 2003.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer Berlin Heidelberg, 2004.

[BDL96] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the Importance of Checking Computations, 1996.

[BGMW92] Ernest F. Brickell, Daniel M. Gordon, Kevin S. McCurley, and David B. Wilson. Fast Exponentiation with Precomputation: Algorithms and Lower Bounds. In *EUROCRYPT*, 1992.

[BS97] Eli Biham and Adi Shamir. Differential Fault Analysis of Secret Key Cryptosystems, 1997.

[GP02] Jorge Guajardo and Christof Paar. Itoh-Tsujii Inversion in Standard Basis and Its Application in Cryptography and Codes. *Design, Codes and Cryptography*, 25:207–216, February 2002.

[HMV04] Darrel Hankerson, Alfred Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[Knu98] Donald E. Knuth. *The Art of Computer Programming Vol. 2 / Seminumerical Algorithms*. Addision-Wesley, 1998.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[Koç08] Cetin Kaya Koç. *Cryptographic Engineering*. Springer, 2008.

[Mon83] Peter Montgomery. Mathematics of Computations, 1983.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer, 2007.

[MvOV97] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997. Full text version available at www.cacr.math.uwaterloo.ca/hac.

[PP10] Christof Paar and Jan Pelzl. *Understanding Cryptography - A Textbook for Students and Practitioners*. Springer, 2010.