Generic **τ**-*adic NAF* key generator

Elliptic curve cryptography (ECC) is more efficient than public key schemes such as RSA due to its superior security strength per bit [1]. The EC defined over GF(2^m) represented by equation (1) is called a non-supersingular Weierstrass equation:

$$y^2 + xy = x^3 + ax + b \quad (1)$$

where a, b $\in$ GF(2^m) and b $\neq$ 0.

In ECC the main operation is the scalar point multiplication:

$$Q = k\,P \quad (2)$$

where P, Q $\in$ E and *k* is an integer. The value *k* represents the private key and Q the public key. This point scalar multiplication in equation (2) can be computed by repeated doubling and addition with the average calculation of ((m/2) point addition + m point doubling). In ECC over the GF(2^m) two classes are defined: binary curves and Koblitz curves. Both curves are recommended for prime numbers m = 113, 131, 163, 233, 283, 409, 571 etc, in order to avoid the cryptanalysis attack named “Weil decent attack” [2].

Koblitz curve is the special equation of the Weierstrass non-supersingular form defined by $y^2 + xy = x^3 + ax + b$, where its parameter b is equal to 1 and $a \in \{0,1\}$.

The main advantage of Koblitz curves is that their special form allows the doubling operation in scalar multiplication to be replaced by the so called Frobenius mapping of the point P in equation (2), namely τP. For a point P = (x,y) $\in$ Koblitz curve E_a, then $\tau P = (x^2, y^2)$ also belongs to the same curve. The Frobenius map can be effectively computed through the field squaring operation of the coordinates of a point which is relatively inexpensive over GF(2^m).

In order to be able to apply properties of the Koblitz curve, the integer *k* in equation (2) has to be represented in *τ-adic* non-adjacent form (*τNAF*).

$$k \cdot P = \tau NAF(k) \cdot P = \sum_{i-0}^{n-1} u_i \cdot \tau^i = u_0 P + \tau u_1 P + ... + \tau^{n-1} u_{n-1} P \qquad (3)$$

where u_i is an integer $\in \{0, \pm 1\}$; for i=0,1,2,...,(n-1) with $u_{n-1} \neq 0$ and the sum of the absolute values of two consecutive u_i is less or equal to 1 and *n* is the *τNAF* length. The average density of non-zero elements in the *τNAF* is n/3. On average, this calculation requires ((n/3)(point additions) + n*τ*(mapping(field squaring))) to be performed.

In [3] Solinas suggests an algorithm for generating *τNAF* by repeatedly dividing *k* by τ and the remainder is the digit u_i in equation (3) which is similar to the approach introduced in [4]. However, the result of Solinas' algorithm creates the key with approximately double size of the input key length (this in average doubles the calculation time compared to the time mentioned at the end of the previous paragraph) [[[The overall time of calculating a key of length xxx has an elapsed time of ...]]]. To bypass this problem he further proposed an additional algorithm which was used for reduction of the generated key to the size approximately equal of the input key length. Another method introduced in [5] simplifies the second algorithm of Solinas to two separate algorithms B and C. All three algorithms A, B and C are presented below:

Algorithm A: Generation of *τNAF* from binary key

$INPUT : k = r_0 + r_1\tau, \in Z[\tau]$

$for(i = 0; ((r_0 \neq 0)OR(r_1 \neq 0)); i++)$

{

$if\ ((r_0 \equiv 1(\mathrm{mod}\ 2))$

{

$u_i \leftarrow 2 - ((r_0 - 2r_1)(\mathrm{mod}\ 4));$

$r_0 \leftarrow r_0 - u_i;$

}

else

{

$u_i \leftarrow 0;$

}

$t \leftarrow r_0; r_0 \leftarrow r_1 + \mu(r_0 / 2); r_1 \leftarrow -t / 2;$

}

$OUTPUT : u = \tau NAF(k), where(u_{i-1}, u_{i-2}, ..., u_1, u_0)$

Algorithm B: Reduction of the key to original size

$INPUT : u = \Sigma_{i=0}^{l_u - 1} u_i \tau^i, u_i \in \{0, 1, -1\}$

$t = \{0_{m-1}, ..., 0_0\};$

$if\ (l_u > 2m)$

{

$t \leftarrow (0_{3m-1}, ..., 0, u_{l_u - 1}, ..., u_{2m})$

}

$if\ (l_u > m)$

{

$t \leftarrow t + (u_{2m-1}, ..., u_m)$

}

$u \leftarrow t + (u_{m-1}, ..., u_0)$

Algorithm C: Regeneration of the key to *τNAF* with reuse of algorithm A for calculation of $u \leftarrow u + \tau NAF(r_0 + r_1\tau)$

$i = 0;$
$while((u_j \neq 0) AND (j > i))$
{
$if\ (u_i == 0)$
$\{i++;\}$
$else$
{
$(r_1, r_0) \leftarrow (u_{i+1}, u_i); (u_{i+1}, u_i) \leftarrow (0,0);$
$u \leftarrow u + \tau NAF(r_0 + r_1\tau);$
$w_i \leftarrow u_i; i++;$
}
}
$OUTPUT : w = \Sigma_{i=0}^{lw-1} w_i \tau^i, w_i \in \{0,1,-1\}$

This tool is used for the generation of the τ-adic NAF key. The sample calculation number for generation of *τNAF* was confirmed with reference in [6], which used the number 9 equivalent to 1001 in binary and has its τ-adic NAF equal to {1,0,-1,0,0,1}. (In the literature I didn't find another sample with concrete numbers).

The program *t-adicNAF.exe* is a command line tool which also has an external GUI. This tool is responsible for reading and writing data in text files and also for checking that the data, read in that input file, has a suitable binary key. Then it executes 3 algorithms which are:

- A) generates the τ-adic NAF key but increase the length size,
- B) makes the reduction of that length key but loses it τ-adic NAF form, and
- C) finally recreates the length of the key size to the original binary key length size and brings the key stream back to τ-adic NAF form.

When the program reads the binary key from the text file it accepts only continuous binary streams of data, so only 0 and 1 are allowed. If any other digit or character will be read, then the tool will stop immediately, informing where the first wrong data was detected.

To execute the tool you will need to specify two parameters: The first is the *'a'* which is the curve parameter and may have values of 0 or 1 for *Koblitz curves*, and the second parameter is the text file name containing your binary key. After completion of all algorithms the τ-adic NAF key will be appended to another text file called key-out.txt.

The generated key, as proposed in [1], can be used in two separated memory banks of a device such as MCU, FPGA, DSP, PC or etc, where the one of it represents 0 and 1, and the other represents with 1 the -1.

Note from the author:

To my knowledge there is no similar freeware or commercial software available which allows the evaluation of the performance and the generation of a τ-adic NAF key.

My source code is based only on the standard C library. I developed it from scratch in order to easily port it to microcontroller (MCU) and keep the size of the code to the minimum possible. The source code will be published when we completed the according publications.

Author:

PhD Anastasios Kanakis [electronix79@yahoo.com],

Acknowledgements:

PhD M.N. Hassan [m.nabil@sheffield.ac.uk], Dr M. Benaissa [m.benaissa@sheffield.ac.uk] and Prof B. Esslinger [bernhard.esslinger@cryptool.org]

This project was done in the University of Sheffield, UK, Department EEE, in 2009 to 2012.

From 2012 this tool is a part of CrypTool.

Reference:

[1] “Flexible Hardware/Software Co-design for Scalable Elliptic Curve Cryptography for Low end Applications”, M.N. Hassan, Dr M. Benaissa and A. Kanakis, ASAP 2010, IEEE

[2] Elliptic curve cryptography Wikipedia, http://en.wikipedia.org/wiki/Elliptic_curve_cryptography

[3] A. Solinas, “Efficient arithmetic on Koblitz curves”, Designs, Codes and Cryptography, vol. 19, no. 2–3, pp. 195–249, Mar. 2000

[4] “Guide to Elliptic Curve Cryptography”, Darrel Hankerson, Alfred Menezes, Scott Vanstone, Springer 2004

[5] Lutz, “High performance elliptic curve cryptographic co-processor”, Master’s thesis, University of Waterloo, Canada, 2003.

[6] Jerome A. Solinas, “An Improved Algorithm for Arithmetic on a Family of Elliptic Curves”, National Security Agency, Ft. Meade, MD 20755, USA

[7] NIST, “Recommended elliptic curves for federal government use”, Available at http://csrc.nist.gov/encryption/.2000