Using an RSA Accelerator for Modular Inversion

by Martin Seysen

CHES 2005

Coprocessors on Smart Cards

Coprocessors on smart cards have been designed to speed up RSA

Examples:

Infineon SLE66 ACE	**1024 Bit modular multiplication**
Hitachi/Renesas AE4	**1024 Bit modular Montgomery multiplication**
Philips P5/P8 FAME-X	**modular multiplication, max bit length depending on memory**

Coprocessors on Smart Cards

The focus of coprocessor design is speedup of modular multiplication.
This is very good for the RSA algorithm with the intended key size.

Efficient implementation becomes tricky in the following cases:

- **Operations with a greater modulus than supported by hardware (Paillier, 1999 ; Fischer , Seifert, CHES 2002)**

- **Elliptic curve Cryptography**
  Here the costs for other operations than modular multiplication, especially modular inversion, are non-negligible.

  This talk deals with modular inversion

Modular Inversions in ECC

Inversion needs much more CPU interaction than modular multiplication

- **Inversion can be up to 100 times slower than multiplication on a smart card with coprocessor.**

- **Less inversions are necessary when using projective co-ordinates (e.g. Jacobian or López-Dahab co-ordinates)**

- **Even with these optimisations, inversion may take about 20% of the run time of an ECDSA signing procedure with 192 bit keys.**

- **On the other hand, coprocessor registers have been designed for much longer keys (i.e moduli), as used in RSA cryptography.**

The Extended Euclidean Algorithm

The standard tool for inversion is the extended Euclidean algorithm

To invert u mod v, it computes numbers
$u_0=u, u_1, u_2, ..., u_n$ and $v_0=v, v_1, v_2, ..., v_n = \gcd(u,v)$,
with $u_i = v_{i-1}$ and $v_i = u_{i-1} \bmod v_{i-1}$.

It also computes numbers λ_i, λ'_i with $\lambda_i u = u_i$, $\lambda'_i u = v_i \pmod v$,
so that λ'_n is the requested inverse of u mod v in case $\gcd(u,v) = 1$.
(We can start with $\lambda_0 = 1$, $\lambda'_0 = 0$) .

→ Our modular inversion algorithm comes from the following idea :
Since coprocessors registers are much longer than needed for ECC, we may put each of the pairs (u_i, λ_i) and (v_i, λ'_i) into a single coprocessor register.

Modular inversion with a non-extended Euclidean Algorithm

The above idea leads to a very simple modular inversion algorithm

Algorithm NINV

Input: Integers $u, v > 0$, extension factor $f > 2v$.
Output: Modular inverse $x = u^{-1} \pmod{v}$ or error if u not invertible.

[1] **Put** $U = fu + 1$, $V = fv$.

[2] **While** $V \geq f + v$ **do**
$\{ T = V,\ V = U \bmod V,\ U = T \}$.

[3] **If** $V > f - v$ **then return** $V - f$ **and stop**,
else return "error" **and stop**.

Observations on Algorithm NINV

Step 2 of the algorithm performs the Euclidean process on $U = fu + 1$ and $V = fv$ for a suitable f, *(roughly)* until the current remainder V_i in step [2] has the same size as f.

U and V have about twice the bit length of u and v .

So for typical ECC bit lengths they fit well into 1024--bit registers.

This may save a lot of instructions on some coprocessors.

The values U, V in the i-th iteration of step [2] can be considered as representations of (u_i , λ_i) and (v_i , λ'_i) , respectively.

However, the Euclidean process on U and V might not lead to the same Euclidean quotients as the corresponding process on u and v .

→ The proof of correctness of Algorithm NINV is not trivial !

Correctness of Algorithm NINV

Let $V=V_0, V_1, ..., V_i, ...$ be the value of V after i iterations of step [2] .

The correctness of Algorithm NINV follows from

Theorem

In case $\gcd(u,v) = 1$ there is a V_i with $V_{i-1} > 2f - v$, $f + v > V_i > f - v$, and $(V_i - f)\, u = 1 \pmod{v}$.
Otherwise there is no V_i with $2f \geq V_i > v/2$.

The proof of the theorem uses continued fractions and is given in the paper.

Implementation

Algorithm NINV has been implemented on the Infineon SLE66CX322P. Its coprocessor "ACE" operates on 560 or 1120-bit integers.

Since the Euclidean quotients are usually quite small, it is best to use the fast shift and add/subtract operations on that coprocessor.

Run times obtained on this SLE66CX322P:

Algorithm	160 bit	192 bit	256 bit	320 bit
Extended Euclidean	4.80 ms	5.73 ms	7.46 ms	9.16 ms
NINV	2.09 ms	2.43 ms	3.16 ms	4.45 ms

So we have more than doubled the speed of the implementation
(compared to the standard extended Euclidean algorithm).
This leads to a significant speedup of the ECDSA signing procedure.

More on Implementation

The implementation is based on very simple coprocessor instructions, such as additions/subtractions and shifts.

These instructions are not much more expensive than some glue instructions such as register switching, loop control etc.
Using less variables in Algorithm NINV than in the standard algorithm saves a lot of glue instructions.

Comparison of V with $f + v$ in the main loop costs an extra subtraction.
Since the CPU has to keep track of the bit length of V anyway (due to the architecture of the coprocessor), we may check the bit length of V instead, if we choose $f = 3 \cdot 2^k$ for a k with $k \geq \log2(v) - 1$.
Then by Thm. 1, a (unique) value V with $2^{k+2} > v+f > V > f - v > 2^{k+1}$ exists if and only if u is invertible modulo v.

Comparison to other GCD implementations

Most papers on optimisation of the GCD calculation and modular inversion algorithm deal with different situations:

- **Improving the calculation of the GCD of long integers on a CPU with fixed size (e.g 32 bit)**
  Lehmer 1938, Jebelean 1993
- **Optimisation of a GCD hardware circuit**
  Lórencz, CHES 2002

In contrast to these objectives,
we try to make good use of an oversized arithmetic unit (designed for RSA cryptography) in EC cryptography, when computing a modular inverse.

Thank you for listening!

Martin Seysen

CHES 2005