Fast Scalar Multiplication for ECC over GF(p) using Division Chains

Colin D. Walter

Royal Holloway, UK

Colin.Walter@rhul.ac.uk

Supported by European Commission grant ICT-2007-216676 ECRYPT II

Outline

- Motivation & History
- Standard Exponentiation Algorithms
- $\mathcal{OP}$-Addition Chains
- Division Chains
- Base/Digit Selection
- Implementation
- Examples
- Conclusion

Motivation

- Faster Exponentiation
- Better understanding of recoding choices
- More widely applicable methods
- Pairings with small characteristic, e.g. 3
  - The Frobenius AM means the usual weighting of squares & multiplies is inappropriate

History

- Division Chains / Double Base Rep^n – Arith 13 (1997)
  - Resource constrained environments:
  - Division chains save execution space (CDW)
  - DBNS saves storage space (Dimitrov)
- Composite ECC operations $dP+Q$ (Montgomery *et al*)
  - Reduced field operation count from shared values
- Gebotys & Longa (PKC 2009)
  - Fixed algorithm for using $2P+Q$, $3P$ and $5P$.

Standard Methods

For resource-constrained environment:

- Binary Square and Multiply

  ~3/2 $\log_2 n$ $\times^{ve}$ operations for exponent n.

- Sliding Window

  ~4/3 $\log_2 n$ $\times^{ve}$ operations for 2-bit window, digits ±1.

- NAF (non-adjacent form)

  Same as for 2-bit sliding window.

- Division chains (case of no negative digits)

  ~5/4 $\log_2 n$ with expensive pre-processing of exponent.

  ~7/5 $\log_2 n$ without effort

$\mathcal{OP}$-Addition Chains

- Wider range of operations than just adding.

Set $\mathcal{OP}$ of binary operators (λ,μ), representing $\lambda P+\mu Q$.

An $\mathcal{OP}$-addition chain is a sequence of quadruples

(a_i, b_i, k_i, p_i) where

$p_i = (\lambda_i,\mu_i) \in \mathcal{OP}$ and $k_i = \lambda_i a_i + \mu_i b_i$

$a_i = k_s$, $b_i = k_t$ for some $s, t < i$

$(a_0,b_0,k_0,p_0) = (1,0,1,(1,0))$

The standard addition chain has $a_i+b_i = k_i$ and starts (1,0,1)

Division Chains

- Location aware chains – two locations.

Restricted to previous value and initial (table) value:

$(k_{i-1}, 1, k_i, p_i)$ where

$p_i = (\lambda_i, \mu_i) \in \mathcal{OP}$ and $k_i = \lambda_i k_{i-1} + \mu_i$

These are generated in reverse order:

From $k = k_n$, choose $p_i = (\lambda_i, \mu_i)$ where $k_i \equiv \mu_i \bmod \lambda_i$ and calculate $k_{i-1} = (k_i - \mu_i) / \lambda_i$.

- Hence the name “division” chain.
- If all $\lambda_i = r$ are the same, this is the change a base algorithm and μ_i are the digits of k base r.

Change of Basis

- The rule $k_{i-1} = (k_i - \mu_i)/\lambda_i$ produces

$$k = (((\mu_1\lambda_2 + \mu_2)\lambda_3 + \ldots + \mu_{n-2})\lambda_{n-1} + \mu_{n-1})\lambda_n + \mu_n$$

- Rewrite this using **bases** r_i and **digits** d_i :

$$k = (((d_{n-1}r_{n-2} + d_{n-2})r_{n-3} + \ldots + d_2)r_1 + d_1)r_0 + d_0$$

- This recoding gives a left-to-right algorithm with table values m_d and iterative step

$$m \leftarrow m^{r_i} \times m_{d_i}$$

- When possible choose $d_i = 0$ to save a multiplication.

Example

$$235_{10} = (((((1)3 + 0)2 + 1)5 + 4)2 + 0)3 + 1$$

- Pair (3,1) $(235 - 1)/3 = 78$
- Pair (2,0) $(78 - 0)/2 = 39$
- Pair (5,4) $(39 - 4)/5 = 7$
- Pair (2,1) $(7 - 1)/2 = 3$
- Pair (3,0) $(3 - 0)/3 = 1$
- Pair (2,1) $(1 - 1)/2 = 0$

There are usually several alternatives at each point.

- Set of possible bases is usually $\mathcal{B} = \{2,3\}$ or $\mathcal{B} = \{2,3,5\}$.

Choosing the Chain

- Assign a cost $c_{d,r}$ to each operation $m \leftarrow m^r \times m_d$.
  - e.g. clock cycles if implementation is known,
  - else native word operations,
  - or ... field mult^{ns} when in ECC, perhaps.
- Simplest cost is min^{mum} length of addition chain for r, plus 1 if $d \neq 0$ (i.e. the count of $\times^{ve}$ ops.)
- Each digit/base choice affects remaining digits; the effect on cost diminishes with distance from the choice.
- Build search tree of next λ digits, say, and find cost, *including* average cost c for remainder of k: for each digit,

$$c_{d,r} - c \log r$$

- Pick first digit of cheapest choice, and repeat for rest of k.

Digit Choice (1)

- Let $\pi_{\mathcal{B}} = \mathrm{lcm}\ \{r \in \mathcal{B}\}$ for $\mathcal{B}$ = set of possible bases.

- If $k \equiv k' \bmod \pi_{\mathcal{B}}^{\lambda}$ then k, k' generate the same costs for each of next λ base/digit choices.
- So next digit is determined by $k \bmod \pi_{\mathcal{B}}^{\lambda}$ & cost function c

- Ideally maximize λ. In practice consider $k \bmod \pi$ for one of the largest practical factors π of $\pi_{\mathcal{B}}^{\lambda}$.
  - If $r = 2$, say, is particularly cheap, preferentially increase the power of 2 in π so choice of π reflects greater likelihood of 2.
- For each set of λ choices $(r_1,d_1),\ldots,(r_\lambda,d_\lambda)$ and $\rho = r_1 r_2 \ldots r_\lambda$,

$$(\ldots((k - r_1)/d_1 - r_2)/d_2 \ldots - r_\lambda)/d_\lambda \bmod \pi/\rho$$

still contains some info^{n} which should be included in cost.

Digit Choice (2)

- For cheapest $(r_1,d_1),\ldots,(r_\lambda,d_\lambda)$ for k mod π, choose (r_1,d_1) as the next digit/base pair for k. This gives a recoding table mod π.

- The recoding is a Markov process. The states are residues mod π. So asymptotic cost per key bit can be calculated. (Monte Carlo simulation.)

- During recoding, the residues k_i mod π are not distributed uniformly for random keys k. So costs for digit choices may have been slightly inaccurate.
  - Make local changes to the table, calculate new cost per bit, and update table if new average cost is cheaper.

Implementation

- The table generally has good structure and can be easily translated into a simple set of rules, e.g.

  if $k \equiv 0 \bmod 2$ **then** $r = 2$, $d = 0$

  else if $k \equiv 0 \bmod 5$ **then** $r = 5$, $d = 0$

  else ...

- There may be a few deeply nested, rarely occurring rules which can be safely deleted without much effect.

- The result is a space and time efficient recoding scheme, tailored to any required constrained environment.

- Including a base 3 or 5, say, as well as 2
  makes it faster than binary algorithms
  if the recoding process is cheap enough.

Example 1

Digits $\mathcal{D} = \{0,\pm1,\pm3,...,\pm15\}$, bases $\mathcal{B} = \{2,3\}$, $\mathcal{OP} = \mathcal{B}\times\mathcal{D}$, $\pi = 2^6 3^2$

If $k = 0 \bmod 9$ and $k \neq 0 \bmod 4$ then

$\quad r \leftarrow 3,\ d \leftarrow 0$

else if $k = 0 \bmod 2$ then

$\quad r \leftarrow 2,\ d \leftarrow 0$

else if $k = 0 \bmod 3$ and $18 < (k \bmod 64) < 46$

and $((k \bmod 64) - 32) \neq 0 \bmod 3$ then

$\quad r \leftarrow 3,\ d \leftarrow 0$

else $r \leftarrow 2,\ d \leftarrow ((k+16) \bmod 32) - 16$

- This is faster than the "record" algorithm in PKC 2009 (using Jacobi Quartic coordinates) but rather space hungry.
- About 1200 field multiplications for 160-bit key (~7.5 per bit).

Example 2

Digits $\mathcal{D} = \{0,\pm1,\pm3,\pm5,\pm7\}$, bases $\mathcal{B} = \{2,3\}$, $\mathcal{OP} = \mathcal{B}\times\mathcal{D}$, $\pi = 2^8 3^2$

If $k = 0 \bmod 9$ and $k \neq 0 \bmod 4$
  and $(16 < (k \bmod 256) < 240)$ then
    $r \leftarrow 3,\ d \leftarrow 0$
else if $k = 0 \bmod 2$ then
    $r \leftarrow 2,\ d \leftarrow 0$
else if $k = 0 \bmod 3$ and $8 < (k \bmod 32) < 24$
  and $((k \bmod 32) - 16) \neq 0 \bmod 3$ then
    $r \leftarrow 3,\ d \leftarrow 0$
else $r \leftarrow 2,\ d \leftarrow ((k+8) \bmod 16) - 8$

- The pre-computed table has effectively just 4 elements.
- This is only ½% slower than Example 1
- 2% faster than $\mathcal{B} = \{2\}$; easily enough to cover the recoding.

Results & Conclusions

- A technique for generating fast algorithms for scalar multiplication in a wide variety of environments.
- Uses a multibase representation and can make use of efficient composite elliptic curve operations.
- Faster than binary-based methods, but small recoding overhead.
- Can benefit from cheap Frobenius operation.
- Takes advantage of the available space resources.
- **Unbeatable?**