A New Multiplication Technique for GF(2^m) with Cryptographic Significance

Athar Mahboob and Nassar Ikram

National University of Sciences & Technology, Pakistan

Presented at WISA 2004

August 23-25, 2004

Jeju Island, South Korea

Multiplication

- Multiplication is a fundamental arithmetic operation in many algebraic structures such as GF(2^m)
- Other fundamental operations include
  - Addition
  - Subtraction

  } Addition and Subtraction are same in GF(2^m). Addition is not a computational bottleneck Complexity is $O(m)$

  - Squaring

  Squaring can be done as multiplication but there are very fast methods for squaring in GF(2^m)

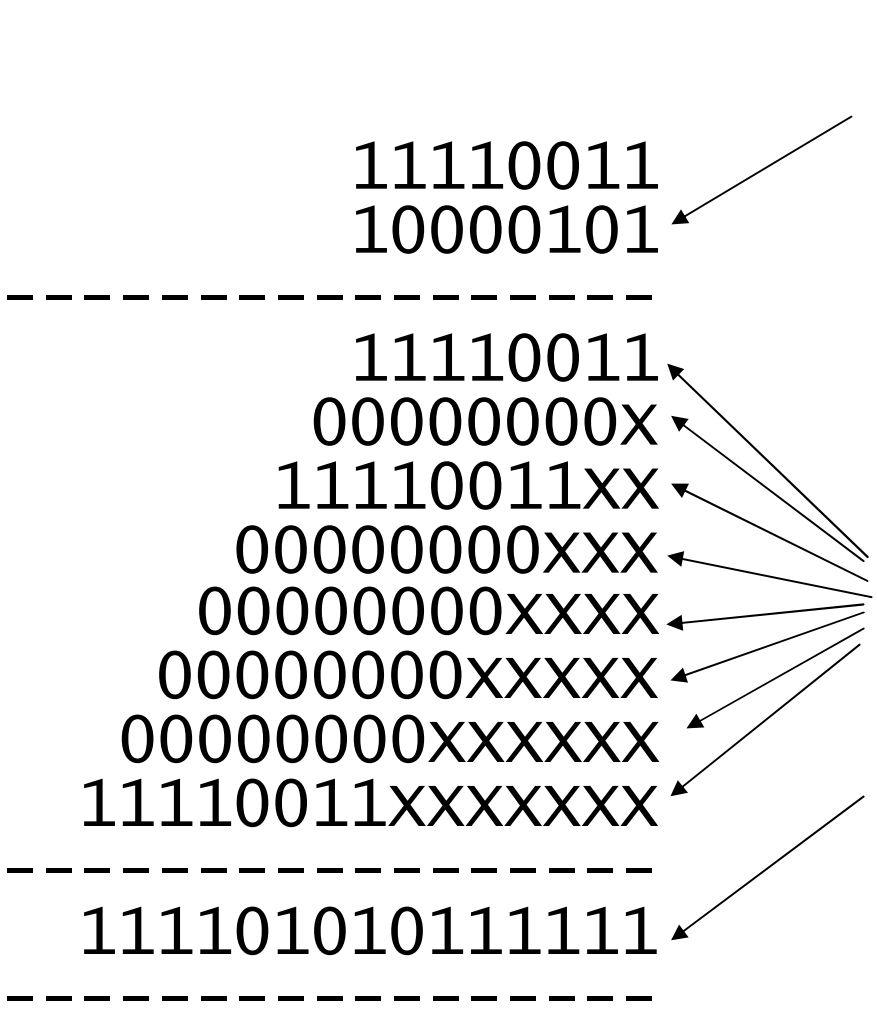
  - Division
  - Inversion

  } Division can be replaced by inversion followed by multiplication. Inversion can be done using multiplication as per Fermat's Theorem

  - Exponentiation

  Exponentiation (repeated multiplication) can be efficiently done using binary method and improved versions of it such as ITI
- Hence multiplication is the most important arithmetic op but ...
- Multiplication more time consuming than addition, subtraction and squaring

Schoolbook Method



- Examines bits of the **multiplier** and adds shifted versions of the **multiplicand** based on the bit
- Maps easily into a **Shift-and-Add** Scheme
- **Partial Products** are first created
- Partial Products are added or accumulated to generate the **Final Product**
- Number of bit operations are $O(m^2)$

Our Research

- GF(2^m) Multiplication performance is critical for ECC cryptosystems defined using Elliptic Curves over GF(2^m)
- EC Point Addition and Doubling require 6 additions, 1 squaring, 2 multiplications and 1 inversion
- Inversions can be replaced by multiplications using **projective coordinates**
- Inversions can also be done using multiplications and squaring – **Fermat's Little Theorem**
- Typical ECC Field sizes are $112 < m < 600$
- Standard fields use m which is **prime (for presumed security)**
- Standard fields support **fast polynomial modular reduction**

Issues with GF(2^m) Multiplication in Software

- Lack of machine level GF(2^m) word-level multiply instruction
- Bit level operations are required such as examining single bit at a time, shifts, ANDs, XORs
- Modular Reduction may be a computational bottleneck for a general field polynomial
- Decide on doing inter-leaved modular reduction or fast modular reduction at the end
- We do fast modular reduction using trinomial or pentanomial at the end like most typical ECC implementations

Multitude of Multiplication Techniques

- ***Multiplication***
  - ***Classical Multiplication***
  - ***Modular Multiplication***
    - ***GF(p)***
      - ***Multiplication followed by classical long division***
      - ***Multiplication followed by Fast Reduction Techniques***
        - ***Barret Reduction***
      - ***Montgomery Multiplication dispenses with modular reduction step***
    - ***GF(2^m) – Polynomial Basis***
      - ***Multiplication followed by classical long division***
      - ***Multiplication followed by Fast Reduction Techniques***
        - ***Trinomial/Pentanomial Modular Reduction***
      - ***Montgomery Multiplication dispenses with modular reduction step***
      - ***LookUp Tables based Multiplication***
        - ***Any m***
        - ***Composite m***
    - ***GF(2^m) – Normal Basis***
      - ***Massey Omura Multiplier***

This list is not exhaustive by any means ...

Classical Multiplication Techniques

- **Schoolbook** – if nothing else makes sense ...
- **Karatsuba** – recursive binary subdivision technique
- **Toom-Cook** – recursive three-way subdivision technique
- **FFT** – asymptotically fastest technique, not used for sizes typical in PKC
- **Comba** – something similar to what we have proposed, little known in cryptographic circles, we do not have access to the source, never has been proposed for use in GF(2^m)
- There are typical cutoff values where one technique starts to be more efficient than others due to inherent overheads of each technique

Classical Shift and Add

Algorithm 1 Shift and Add Polynomial Basis $GF(2^m)$ Multiplication

Input: $a(x)$ and $b(x) \in GF(2^m)$ stored as array of machine words $a(x) = A_{s-1}A_{s-2}\ldots A_0$ and $b(x) = B_{s-1}B_{s-2}\ldots B_0$

Output: $c(x) = a(x) \cdot b(x) \bmod p(x)$

1: $c(x) \leftarrow 0$
2: $a'(x) \leftarrow a(x)$
3: **for** i from 0 to $s-1$ **do**
4: **for** j from 0 to $w-1$ **do**
5: **if** bit j of B_i is 1 **then**
6: $c(x) \leftarrow c(x) + a'(x)$
7: **end if**
8: $a'(x) \leftarrow a'(x) \ll 1$
9: **end for**
10: **end for**
11: **return** $c(x) \bmod p(x), c(x) = C_{s-1}C_{s-2}\ldots C_0$

Shift and Add with Precomputation

Algorithm 2 Window-Based Shift and Add Polynomial Basis $GF(2^m)$ Multiplication

Input: $a(x), b(x) \in GF(2^m)$ stored as array of machine words $a(x) = A_{s-1}A_{s-2}\ldots A_0$ and $b(x) = B_{s-1}B_{s-2}\ldots B_0$

Output: $c(x) = a(x) \cdot b(x) \bmod p(x)$

1: Precomputation: For all 2^u possible u bit values $(t_{u-1}\ldots t_0)$ calculate $t(x) \cdot a(x)$ and store the entry in location $(t_{u-1}\ldots t_0)$ in the table $multable[2^u]$
2: $c(x) \leftarrow 0$
3: **for** i from $s-1$ down to 0 **do**
4: **for** j from $\frac{w}{u} - 1$ down to 0 **do**
5: Examine u bits $b_{uj+(u-1)}\ldots b_0$ of B_i and $c(x) \leftarrow c(x) + multable[b_{uj+(u-1)}\ldots b_0]$
6: $c(x) \leftarrow c(x) \ll u$
7: **end for**
8: **end for**
9: **return** $c(x) \bmod p(x), c(x) = C_{s-1}C_{s-2}\ldots C_0$

Other LUT Techniques

- Composite *m*
  - Generally use composite extension fields $m = ab$ in GF(2^m)
  - Perform arithmetic using LUT in the subfield where LUT is small enough to fit in memory
  - Use log and antilogs to convert from multiplication to addition (which are just XORs) resulting processing *a* bits at a time instead of one bit for Shift and Add
  - Not applicable to standardized fields for ECC
- Any *m*
  - One general LUT technique creates LUT on fly for each different multiplier, it is more of a windowing technique
  - Our LUT is calculated once and for all for all *m and for all multipliers*