A fast multiplier over GF (2^n)

Potgieter, M.J.; Dyk, van, B.J.; Tjalkens, T.J.

Published in:
23rd symposium on information theory in the Benelux

Published: 01/01/2002

Document Version
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the author's version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

Citation for published version (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
A Fast Multiplier over $GF(2^n)$

M.J. Potgieter¹, B.J. van Dyk¹ and Tj.J. Tjalkens²

¹ University of Pretoria, Pretoria, South Africa
² Eindhoven University of Technology, Eindhoven, the Netherlands
e-mail: t.j.tjalkens@tue.nl

Abstract. In this paper we will present a hardware implementation of a $GF(2^n)$ polynomial basis multiplier that is twice as fast as the classical multiplier while requiring about 50% more chip area. We implement a flexible scalar (or point) multiplier for elliptic curve cryptosystems using this multiplier and find that the flexible system performs almost twice as fast as compared with the classical multiplier.

1 Introduction

An elliptic curve cryptosystem (ECC) can be used to exchange keys over an insecure channel. ECC belongs to the class of public key cryptosystems. The famous Diffie-Hellman system [1] relies on the prohibitive complexity of solving $s$ from

$$g^s \equiv k \mod p,$$

where $g$, $k$, and $p$ (a large prime) are known. The RSA system [2] is a widely used variant on this theme.

In an ECC another complex operation is used, namely the point addition, see [3,4] or the more recent [5]. Here solving $m$ from $mP$, where $P$ is a known point on a known elliptic curve in $GF(2^n)$ is the intractable operation. The operation $mP$ is known as the elliptic curve scalar multiplication.

For the Diffie-Hellman and RSA systems that use a multiplicative group, a sub-exponential, w.r.t. log $p$, running time algorithm, the index-calculus method, exists, while all known algorithms for solving the EC scalar multiplication problem are still exponential in $n$. Thus, much smaller keys can be used in an ECC (about 160 bit keys) than in RSA systems (about 1000 bit keys). Even though the basic operation in an ECC, point addition, might be more complex than the multiplication in RSA, the difference in field sizes needed, make the ECC system an attractive choice for low power/low complexity applications.

While implementing a flexible, programmable, hardware EC scalar multiplier we found a novel field multiplier that is twice as fast as the standard multiplier, while requiring 50% more chip area.
2 Field multipliers

Because we are interested in a flexible system a polynomial basis for $GF(2^n)$ seems more appropriate than a normal basis. Multiplication in a normal basis representation is only efficient if an optimal normal basis exists, and in the range of $n$ that we were interested in, 160...200, only a few optimal bases exists.

So, the field elements are expressed as binary vectors $(a_0, a_1, \ldots, a_{n-1})$ of dimension $n$, relative to the base $\{1, \alpha, \alpha^2, \ldots, \alpha^{n-1}\}$, where $\alpha$ is a root of the irreducible polynomial $f(x)$ of degree $n$ over $\mathbb{F}_2$.

The classical multiplier implements the multiplication operation together with the modular reduction. The following pseudo-code describes this multiplier and figure 1 gives a hardware implementation hereof. It is clear that the running time, or number of clock cycles, is equal to $n$.

Note that the multiplication by $x$ in the code is a simple left shift of the register $r$.

\begin{verbatim}
Inputs: a, b and the polynomial f(x).
Output: r, where $r \equiv a \cdot b \mod f(x)$.

r <- 0
for i from n downto 0 do
    msb <- $r_{n-1}$
    r <- (msb and f(x)) \oplus (r \cdot x)
    r <- r \oplus (a_i and b)
endfor
return r

Listing 1: Pseudo code for the classical multiplier
\end{verbatim}

The running time is again equal to $n$ clock cycles. Note also that the division by $x$ in the code is a simple right shift of the register $r$.

\begin{verbatim}
Inputs: a, b and the polynomial f(x).
Output: r, where $r \equiv a \cdot b \cdot x^{-n} \mod f(x)$.

r <- 0
for i from 0 to n - 1 do
    r <- r \oplus (a_i and b)
    r <- (r_0 and f(x)) \oplus r
    r <- r/x
endfor
return r

Listing 2: Pseudo code for the Montgomery multiplier
\end{verbatim}

For odd characteristic fields, a more efficient multiplier exists, the Montgomery multiplier [6]. Applying this idea to $GF(2^n)$ we end up with a multiplier that is very similar to the classical multiplier, see Listing 2. The main difference is that it operates on the least significant bits of $a$.

Instead of computing $r \equiv a \cdot b \mod f(x)$ the Montgomery multiplier $(MM)$ computes $MM[a, b] \triangleq r \equiv a \cdot b \cdot x^{-n} \mod f(x)$. Therefor we represent every element $a \in GF(2^n)$ by $M(a) \triangleq a \cdot x^n \mod f(x)$. So,
if we wish to compute $a \cdot b \mod f(x)$ we compute $MM[M(a), M(b)] \equiv a \cdot b \cdot x^n \mod f(x)$. We observe that $MM[M(a), M(b)] = M(ab)$. Converting a value $a$ to its Montgomery representation $M(a)$ is easily obtained using the Montgomery multiplier as $M(a) = MM[a, x^{2n}]$. The conversion from $M(a)$ to $a$ is similarly performed by $a = MM[M(a), 1]$.

3 A modified field multiplier

If we briefly ignore the modular reduction we see that the classical multiplier and the Montgomery multiplier actually compute the same product. Say $a(x) = a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ and $b(x) = b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$. Then the classical multiplier results in

$$
(a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0)(b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0) = 
$$

$$
c_8x^8 + c_7x^7 + c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \quad (2)
$$

It is easy to see that the result for the Montgomery multiplier is $c(x) \cdot x^{-n}$ because the intermediate results are shifted to the right.

Assume that $n$ is even, then multiplying $c(x)$ by $x^{n/2}$ can be seen as a combination of left and right shifts on the partial results. However these results can be performed in parallel and the two partial end results can then be added together. It is still possible to perform the modular reduction using the original algorithms from listing 1 and listing 2.

Inputs: $a$, $b$ and the polynomial $f(x)$.
Output: $r$, where $r \equiv a \cdot b \cdot x^{-n/2} \mod f(x)$.

$$
rc \leftarrow 0 
\text{for } i \text{ from } 0 \text{ to } n/2 - 1 \text{ do}
\msb \leftarrow rc_{n-1}
rc \leftarrow (\msb \text{ and } f(x)) \oplus (rc \ast x)
rc \leftarrow rc \oplus (a_0 \text{ and } b)
rm \leftarrow rm \oplus (a_0 \text{ and } b)
rm \leftarrow (rm_0 \text{ and } f(x)) \oplus rm
\text{return } rc \oplus rm
$$

Listing 3: Pseudo code for the modified multiplier

The main increase in chip area is caused by the extra (temporary) result register, $rm$ and $rc$ in stead of $r$ in the original multipliers.

It is easy to extend this method to the case where $n$ is odd. In that case the Montgomery part works on the $(n-1)/2$ least significant coordinates of $a$ and the representation is $M(a) = a \cdot x^{(n-1)/2} \mod f(x)$. The number of clock cycles needed is $(n+1)/2$. Odd, or preferably prime, values for $n$ are often used in cryptographic systems.

4 Comparing the classical and modified multipliers

The multipliers were implemented in a field programmable gate array FPGA. The following two tables give an indication of the speed-up and chip area cost of the modified multiplier relative to the classical multiplier. We show the timing and chip area for the complete ECC scalar multiplier unit in table 1 and table 2 respectively.

5 Conclusion

The results of the ECC scalar multiplier comparison indicate that the field multiplier is the predominant factor in the speed-up of the design.
<table>
<thead>
<tr>
<th>Field size n</th>
<th>$t_{\text{classical}}$</th>
<th>$t_{\text{modified}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>163</td>
<td>6.619</td>
<td>3.776</td>
</tr>
<tr>
<td>233</td>
<td>13.316</td>
<td>7.158</td>
</tr>
<tr>
<td>283</td>
<td>19.518</td>
<td>10.299</td>
</tr>
</tbody>
</table>

Table 1. Timing comparison in milliseconds

<table>
<thead>
<tr>
<th>Field size n</th>
<th>% slices (modified)</th>
<th>% slices (classical)</th>
<th>% modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>96</td>
<td>35</td>
<td>26</td>
<td>1.346</td>
</tr>
<tr>
<td>192</td>
<td>56</td>
<td>40</td>
<td>1.400</td>
</tr>
<tr>
<td>304</td>
<td>81</td>
<td>56</td>
<td>1.446</td>
</tr>
<tr>
<td>384</td>
<td>99</td>
<td>67</td>
<td>1.478</td>
</tr>
</tbody>
</table>

Table 2. Chip area in FPGA slices

The increase in chip area is caused by the field multiplier only. This and a more detailed comparison of the designs show that the modified multiplier costs about 50% more chip area. Because we designed a flexible and programmable ECC unit, many optimizations that are possible with fixed and clever choice of parameters were not possible in this implementation. This might influence the speed and chip area cost of a design enormously. Still the speed-up factor will be more or less the same because in many designs the field multiplier will determine the speed of the overall circuit.

References