An efficient and scalable postcomputation-based generic-point parallel scalar multiplication method

Turki F. Al-Somani¹ᵃ), Ayman G. Fayoumi², and Mohammed K. Ibrahim¹

¹ Umm Al-Qura University, Computer Engineering Department, P.O. Box: 715, Makkah 21955, Saudi Arabia
² King Abdulaziz University, Department of Information Systems, P.O. Box 80220 Jeddah 21589, Saudi Arabia
a) tfsomani@uqu.edu.sa

Abstract: An efficient generic-point parallel scalar multiplication method is presented here where a new mapping technique is used with a modified version of the postcomputation-based method [6]. The results show that the proposed method outperforms that of the work in [6] when the number of consecutive requests is two or more. Furthermore, the results show that the proposed method is scalable for any number of parallel processors and performs better as the number of consecutive requests increases. This method consequently is very attractive for use in high-performance end servers that employ parallel elliptic curve processors.

Keywords: elliptic curves cryptosystems, parallel scalar multiplication

Classification: Electron devices, circuits, and systems

References

1 Introduction

Elliptic curve cryptosystems (ECCs)-initially proposed by Niel Koblitz and Victor Miller in 1985 [1] are perceived as serious alternatives to the RSA system with a much shorter word length. An ECC with a key size of 128–256 bits was proven to offer an equal security to that of an RSA system with a key size of 1–2 Kbits [2]. To date, no significant breakthroughs have been made in determining the weaknesses of ECCs, which are based on a discrete logarithm problem over points on an elliptic curve. The fact that the problem appears so difficult to tackle means that key sizes can considerably be reduced [3]. This advantage of ECCs has recently gained remarkable recognition as to be incorporated in many standards such as IEEE, ANSI, NIST, SEC and WTLS.

Scalar multiplication is the basic operation for ECCs. Scalar multiplication of a group of points on an elliptic curve is analogous to the exponentiation of a multiplicative group of integers modulo a fixed integer $m$. The scalar multiplication operation, denoted as $kP$, where $k$ is an integer and $P$ is a point on the elliptic curve, represents the addition of $k$ copies of point $P$. Scalar multiplication is then computed by a series of point doubling and point addition operations of the point $P$ that depends on the bit sequence representing the scalar multiplier $k$. Several scalar multiplication methods have been proposed [4].

However, for high-performance end servers, the current sequential scalar multiplication methods are too slow to meet the demands of the increasing number of customers. Identifying efficient scalar multiplication methods for such servers has thus become very crucial. Scalar multiplication methods that can be parallelized are often used for high-speed implementations. Precomputations [5] have been applied to speed up scalar multiplication, but require sequential steps that cannot be parallelized, and are primarily advantageous when the elliptic curve point is fixed. However, during secure communication sessions that use public keys, the elliptic curve point changes, as it depends on the public key of the communicating entity, and hence it is session dependant. This is also the case when digital signatures are used. Therefore, the computation of scalar multiplications is generally performed with a generic elliptic curve point. Because the elliptic curve point is likely to differ in each session, the overheads resulting from the necessary precomputations must be considered when estimating the total computational time required.

Postcomputations have recently been proposed [6] as an alternative method to speedup scalar multiplications. In [6], the precomputation overheads are replaced by postcomputations that can be parallelized. The multiplier $k$ is partitioned into $u$ partitions that can be processed in parallel by $u$ processors using the binary method. Postcomputations are then distributed on $u - 1$ processors to be performed in parallel. The points that result from processing these key partitions with the postcomputations are finally assimilated to produce $kP$. However, the performance of the postcomputations-based method has been analyzed in [7] and the results show that the best performance is achieved when eight cryptoprocessors are used with $128 \leq m \leq 256$, which limits the performance when more parallel cryptoprocessors are available.
2 Proposed method

In [6], the multiplier \( k \) is partitioned into \( u \) partitions as

\[
k = (k^{(u-1)}\|k^{(u-2)}\| \ldots \|k^{(0)})
\]  

(1)

where \( k = (k_{m-1}, \ldots, k_0) \) is the binary representation of \( k \) and \( k_{m-1} \) is the most significant bit of \( k \). Each partition is mapped to a specific cryptoprocessor as

\[
(k^{(i)}, \text{Cryptoprocessor}_{(i)})
\]

(2)

Scalar multiplication product \( kP \) can then be computed as

\[
kP = \sum_{0 \leq i \leq u} s_i,
\]

(3)

where \( s_i \) is defined as

\[
s_i = (2^v)[2(2k_{iv+1}P + k_{iv+2}P) + \ldots + k_{iv+1}P + k_{iv+0}P].
\]

(4)

Eq. (4) implies that each partition requires \( iv \), where \( v = \left\lceil \frac{m}{u} \right\rceil \), point doublings to produce the correct partial product. In [6], the multiplier \( k \) has been partitioned into \( u \) partitions of different sizes to balance the number of point operations in terms of the total number of field multiplications, which was the main reason limiting the performance of the method in [6]. A key observation is that the mapping of equation (2) can be rescheduled whenever a new request for computing \( kP \) for a particular \( P \) and \( k \) takes place. Accordingly, there is no need to make the key partitions with different sizes. Each partition size will be equal to \( \left\lceil \frac{m}{u} \right\rceil \) bits. Accordingly, equation (2) can be rewritten as

\[
(k^{(i)}, \text{Cryptoprocessor}_{(j)})
\]

(5)

where

\[
j = \begin{cases} 
    i & \text{for request number (x)} \\
    (u - 1) + i & \text{for request number (x + 1)} 
\end{cases}
\]

(6)

The computation of \( kP \) in parallel with the proposed method can be performed efficiently using the following algorithm.

Algorithm 1: Efficient Postcomputations-Based Method.
1. **Inputs:** \( P, k \)
2. By padding \( k \) with zeros if necessary and writing \( k = (k^{(u-1)}\|k^{(u-2)}\| \ldots \|k^{(0)}) \), where \( k^{(i)} \) is a key partition of length \( \left\lceil \frac{m}{u} \right\rceil \) bits.
3. **Initialisation:** \( Q \leftarrow P, R \leftarrow O. \)
4. **Key partitions association with cryptoprocessors:**
   4.1. for \( i = 0 \) to \( u - 1 \) do
   4.1.1. \( (k^{(i)}, \text{Cryptoprocessor}_{(j)}) \), where \( j \) is defined in equation (6)
5. **Parallel Scalar Multiplication:**
   5.1. For \( i = 0 \) to \( u - 1 \) do in parallel
   5.1.1. \( Q \leftarrow \text{Binary method } (k^{(i)}, P) \)
   5.1.2. If \( (i > 0) \), then
The pseudo code of the proposed method is given in Algorithm 1. The multiplier $k$ is partitioned into $u$ partitions with equal sizes. The partitioning step is performed at Step 2. For a particular $k$ and $P$, each key partition is mapped to a certain cryptoprocessor according to equations (5) and (6) in Step 4. Parallel scalar multiplications start at Step 5. Each partition is processed independently in parallel by an individual cryptoprocessor. Only partition $k^{(0)}$ does not require any postcomputations. The remaining partitions need postcomputations after executing the binary algorithm (Step 5.1.1). Finally, the resulting points of each partition are accumulated in the accumulation point $R$ (Step 5.1.3) which requires $u - 1$ extra point additions.

**Example:** Let $k = (1000\, 0101\, 1100\, 0011)_2 = (34243)_{10}$, $m = 16$, $u = 4$. The key partitions are $k^{(0)} = 0011$, $k^{(1)} = 1100$, $k^{(2)} = 0101$, and $k^{(3)} = 1000$.

(a) The scalar multiplication of these partitions is then computed in parallel for a single request, each by an individual processor, as

\[
\begin{align*}
 s_0 &= [2(2(2(0)P_1 + (0)P_1) + (1)P_1) + (1)P_1] = 3P_1, \\
 s_1 &= (2^4)[2(2(1)P_1 + (1)P_1) + (0)P_1] + (0)P_1] = 192P_1, \\
 s_2 &= (2^8)[2(2(0)P_1 + (1)P_1) + (0)P_1] + (1)P_1] = 1280P_1 \text{ and} \\
 s_3 &= (2^{12})[2(2(1)P_1 + (0)P_1) + (0)P_1] + (0)P_1] = 32768P_1.
\end{align*}
\]

Finally, $kP_1$ is computed as

\[ kP_1 = s_0 + s_1 + s_2 + s_3 = 3P_1 + 192P_1 + 1280P_1 + 32768P_1 = 34243P_1. \]

(b) The scalar multiplication of these partitions is then computed in parallel for two consecutive requests, using the same key and two different points for simplicity, as

**Processor**\(_{(0)}\):

\[
\begin{align*}
 s_{0,0} &= [2(2(0)P_1 + (0)P_1) + (1)P_1] + (1)P_1] = 3P_1, \\
 s_{0,1} &= (2^4)[2(2(1)P_2 + (0)P_2) + (0)P_2] + (0)P_2] = 32768P_2
\end{align*}
\]

**Processor**\(_{(1)}\):

\[
\begin{align*}
 s_{1,0} &= (2^4)[2(2(1)P_1 + (1)P_1) + (0)P_1] + (0)P_1] = 192P_1, \\
 s_{1,1} &= (2^8)[2(2(0)P_2 + (1)P_2) + (0)P_2] + (1)P_2] = 1280P_2
\end{align*}
\]

**Processor**\(_{(2)}\):

\[
\begin{align*}
 s_{2,0} &= (2^8)[2(2(0)P_1 + (1)P_1) + (0)P_1] + (1)P_1] = 1280P_1, \\
 s_{2,1} &= (2^4)[2(2(1)P_2 + (1)P_2) + (0)P_2] + (0)P_2] = 192P_2
\end{align*}
\]

**Processor**\(_{(3)}\):

\[
\begin{align*}
 s_{3,0} &= (2^{12})[2(2(1)P_1 + (0)P_1) + (0)P_1] + (0)P_1] = 32768P_1, \\
 s_{3,1} &= [2(2(0)P_2 + (0)P_2) + (1)P_2] + (1)P_2] = 3P_2
\end{align*}
\]
Finally, $kP_1$ and $kP_2$ are computed concurrently as

$$kP_1 = s_{0,P_1} + s_{1,P_1} + s_{2,P_1} + s_{3,P_1} = 3P_1 + 192P_1 + 1280P_1 + 32768P_1 = 34243P_1,$$

$$kP_2 = s_{3,P_2} + s_{2,P_2} + s_{1,P_2} + s_{0,P_2} = 32768P_2 + 1280P_2 + 192P_2 + 3P_2 = 34243P_2.$$ 

### 3 Performance analysis

The time complexity of the proposed method, depending on the number of consecutive number of requests, denoted here by $r$, to compute $kPs$ equal:

$$time\ complexity = \frac{r}{2} \left( (u + 1)(v)DBL + \left( \frac{v}{2} + u - 1 \right)ADD \right)$$

The space complexity of the proposed method, in terms of number of stored points, on the other hand, depends on the number of partitions $u$ that will be processed by the $u$ processors using the binary method. Each processor requires the storage of two points to perform scalar multiplications of two consecutive requests using the binary method as shown in the aforementioned example. No precomputations are required and accordingly only the base points will be stored and shared between the parallel processors. Finally, the accumulation point will be required for the accumulation process at the end. Accordingly, the space complexity of the proposed method is equal to $(2u + 4)$ points.

### 4 Results

The performance analysis results in [7] show that the best performance is achieved when eight processors are used. Accordingly, we compare the method in [6] with the proposed method in this work when eight processors ($u = 8$) are deployed using different values of $m$. For simplicity, it is assumed here that the required computation time for point addition is twice that required for point doubling. Table I shows the results of the method in [6] and the results of the proposed method here with up to 32 consecutive requests. Furthermore, Fig. 1 depicts the results when $m = 256$ bits.

Clearly, the results show that the proposed method in this work outperforms that of the method in [6] when the number of consecutive requests is two or more. The results also show that the proposed method performs better as the number of consecutive requests increases. Moreover, the proposed method is scalable for any number of parallel processors and is not limited to only eight parallel processors as in [6]. Accordingly, the proposed method here will be very attractive for use in high-performance end servers that employ parallel elliptic curve processors.
5 Conclusion

Sequential scalar multiplication methods are too slow for high-performance end servers because of the demand resulting from increasing numbers of customers. Existing parallel methods, however, require precomputations or postcomputations for each new session. Postcomputations have recently been proposed [6] where the precomputation overheads are replaced by postcomputations that can be parallelized. However, the performance of the postcomputations-based method [6] has been analyzed and the results show that it is limited to only eight parallel processors. For these reasons, we have here proposed the first efficient generic-point parallel scalar multiplication method, which is scalable and outperforms the postcomputations method in [6].

Acknowledgments

The first and the third authors would like to acknowledge the support of Umm Al-Qura University and the second author would like to acknowledge the support of King Abdulaziz University.

Table I. Results of the method in [6] and the new method proposed here with $m = 128, 160, 200, 256$ and $u = 8$.

<table>
<thead>
<tr>
<th>M</th>
<th>Total (DBLs)</th>
<th>( r = 2 )</th>
<th>( r = 4 )</th>
<th>( r = 8 )</th>
<th>( r = 16 )</th>
<th>( r = 32 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>268 190</td>
<td>536 380</td>
<td>1072 760</td>
<td>2144 1520</td>
<td>4288 3040</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>332 230</td>
<td>664 460</td>
<td>1328 920</td>
<td>2656 1840</td>
<td>5312 3680</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>412 280</td>
<td>824 560</td>
<td>1648 1120</td>
<td>3296 2240</td>
<td>6592 4480</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>530 350</td>
<td>1060 700</td>
<td>2120 1400</td>
<td>4240 2800</td>
<td>8480 5600</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 1. Results of the method in [6] and the new method proposed here with $m = 256$ and $u = 8$.