A Note on Schoenmakers Algorithm for Multi Exponentiation

Srinivasa Rao Subramanya Rao
Mathematical Sciences Institute
The Australian National University, Union Lane, Canberra ACT 2601, Australia

Keywords: Elliptic Curve Cryptography, Montgomery Curves, Multiexponentiation, Schoenmakers' Algorithm, Double Scalar Multiplication, Triple Scalar Multiplication, Smart Cards.

Abstract: In this paper, we provide a triple scalar multiplication analogue of the simultaneous double scalar Schoenmakers' algorithm for multiexponentiation. We analyse this algorithm to show that on the average, the triple scalar Schoenmakers' algorithm is more expensive than the straight forward method of computing the individual exponents and then computing the requisite product, thus making it undesirable for use in resource constrained environments. We also show the derivation of the Schoenmakers' algorithm for simultaneous double scalar multiplication and this is then used to construct the triple scalar multiplication analogue.

1 INTRODUCTION

Elliptic Curve Cryptography(ECC) is seen as an increasingly attractive alternative to conventional public key cryptography such as RSA, as it is able to match RSA-level security, but using smaller parameter sizes and thus is ideal for implementation in constrained environments such as smart cards and mobile devices where computational resources such as computational capacity, network bandwidth, silicon real estate and memory are at a premium.

In recent years, amongst other things, research in the area of ECC has focused on efficient implementations. Point multiplication is at the core of most ECC applications and dominates ECC. Further, multi-exponentiation is often an important ingredient in the implementation of contemporary public key cryptographic algorithms, including ECC. The abstract setting is usually an Abelian group G and it is required to compute a product of the form

$$\prod_{1\leq j\leq n} g_i^{e_i}$$

where the g_i are elements of G and the e_i are integer exponents. In a group with an additive notation for the group operation such as Elliptic Curve groups over a finite field, multi-exponentiation is appropriately termed multi-scalar multiplication. Clearly multi-exponentiation can be achieved by first computing the individual exponents and then multiplying the individual products. However, one can do better than this by utilizing Strauss' method (Bellman and Straus, 1964) which was constructed in the context of addition chains as a solution to a problem posed by Bellman.

A finite sequence of integers $a_0, a_1, \ldots a_r$ is called an addition chain (section 4.63 in (Knuth, 1998)) for a_r if for each element a_i, there exists a_j and a_k in the sequence such that $a_0 = 1$ and for all $i = 1, 2, \ldots r$

$$a_i = a_j + a_k, \quad \text{for some } k \leq \ j < i \qquad (1)$$

Addition chains can be used to efficiently compute either a single exponentiation or a multi-exponentiation, by using Strauss' method. Addition chains are applicable both in the context of multiplicative groups and additive groups such as Elliptic curve groups over a finite field.

The author in (Montgomery, 1987) proposed a special type of an elliptic curve, now known as Montgomery form of elliptic curve or simply Montgomery curve. The arithmetic on a Montgomery curve relies on x-coordinate only arithmetic and also requires the 'difference' of two group elements (points) to be known prior to the computation of addition of these two elements. Thus ordinary addition chains and improvements of these chains cannot be directly utilized for scalar multiplication on Montgomery curves. A special form of addition chain called a Lucas chain is useful in this context. A Lucas chain is a restricted variant of an addition chain where the indices in equation (1) above are such that either $j = k$ or the difference $a_k - a_j$ is already part of the chain. A special case of Lucas chains occur when either $j = k$ or $a_k - a_j = a_0 = 1$ and these are called

Rao Subramanya Rao S..
A Note on Schoenmakers Algorithm for Multi Exponentiation.
DOI: 10.5220/0005566903840391
In *Proceedings of the 12th International Conference on Security and Cryptography* (SECRYPT-2015), pages 384-391
ISBN: 978-989-758-117-5

binary chains. The Montgomery ladder in Section 2 of this paper is an example of a binary chain. A good reference for Lucas chains is (Montgomery, 1992).

The first algorithm towards double scalar multiplication for Montgomery curves was constructed by Schoenmakers in 2000 and published in (Stam, 2003). Akishitha's algorithm for double scalar multiplication was published in (Akishita, 2001). Shoenmakers' and Akishita's algorithm for double scalar multiplication produce binary chains. The author in (Bernstein, 2006b) proposed algorithms for double scalar multiplication which included binary chain algorithms as well as euclidean chain algorithms. In (Azarderakhsh and Karabina, 2012), another double scalar multiplication algorithm was proposed.

A natural question to ask is, if one could construct triple scalar multiplication analogues of the double scalar multiplication algorithms listed above. A practical motivation to construct such multi scalar multiplication algorithms arises in the implementation of some digital signature and identification schemes and their elliptic curve analogues. Chapter 11 in (Menezes et al., 1997) covers some of these signature schemes. The Okamoto Identification scheme (Refer to Sec 9.3 in (Stinson, 2006)) requires a triple scalar multiplication operation to be performed by the signature verifier. Triple scalar multiplication can also be utilized in the accelerated verification of ElGamal like Signatures(Antipa et al., 2005). The need for higher order analogues can be seen in the batch verification of multiple signatures (Cheon and Yi, 2007). Whilst proposing double scalar multiplication algorithms in (Bernstein, 2006b), the author motivated future researchers to construct triple scalar multiplication versions of ideas in his paper. This exploration can be extended to other double scalar multiplication algorithms too, such as Akishita's and Schoenmakers'. In this paper we focus on the Schoenmakers' algorithm - we first motivate the derivation of this algorithm for double scalar multiplication and then use this to construct a triple scalar extension. On analyzing this extension, we find that on the average, this algorithm cannot even be made as efficient as the straight forward method of computing the three individual scalar multiplications first and then adding up the individual sums. The lesson learnt here is that other algorithms (possibly Akishita's or Bernstein's algorithm for double scalar multiplication), should be researched/extended towards constructing triple and higher order scalar multiplication algorithms, while Schoenmakers' can be avoided. This paper shows that the Schoenmakers' algorithm for triple scalar multiplication is not suitable for implementation, especially in constrained environments. An efficient extension of an other double scalar algorithm is the subject matter of a sequel paper to be published by this author in due course(Rao, 2015).

This paper is structured as follows: Section 2 is an introduction to Montgomery curve arithmetic and also presents a Montgomery binary ladder for single exponentiation. Section 3 presents the easy double scalar multiplication algorithm followed by a derivation of Schoenmaker's algorithm for double scalar multiplication. The extension of Schoenmakers' algorithm for triple scalar multiplication is then provided in section 4 with an analysis for best, average and worst case conditions. We conclude in section 5 with a motivation for further work in this area.

2 PRELIMINARIES

The Montgomery curve defined over a finite field F_p is given by

$$E_m : By^2 = x^3 + Ax^2 + x$$

Let $P = (x_1, y_1)$ be a point on the above curve. In projective coordinates, P can be written as $P = (X_1, Y_1, Z_1)$ and let $[n]P = (X_n : Y_n : Z_n)$. Here the scalar multiplication by n on E_m is denoted by $[n]$ and

$$[n]p = \underbrace{P + P + \cdots + P}_{\text{n times}}$$

The sum $[n+m]P = [n]P + [m]P$ can be computed using the formulae below:

Addition: $n \neq m$

$$X_{m+n} = Z_{m-n}((X_m - Z_m)(X_n + Z_n) + (X_m + Z_m)(X_n - Z_n))^2$$

$$Z_{m+n} = X_{m-n}((X_m - Z_m)(X_n + Z_n) - (X_m + Z_m)(X_n - Z_n))^2$$

Doubling: $n = m$

$$4X_nZ_n = (X_n + Z_n)^2 - (X_n - Z_n)^2$$

$$X_{2n} = (X_n + Z_n)^2 (X_n - Z_n)^2$$

$$Z_{2n} = 4X_nZ_n((X_n - Z_n)^2 + ((A+2)/4)(4X_nZ_n))$$

The addition formulae above costs $(4M + 2S)$ and the doubling formulae costs $(3M + 2S)$ respectively, where M and S are the costs of a field multiplication and a squaring respectively and we follow this convention in the rest of this paper. If coordinates of $(X_{m-n}, Y_{m-n}, Z_{m-n})$ can be scaled such that $Z_{m-n} = 1$, the addition formula above costs $(3M+2S)$.

The well-known Montgomery ladder for scalar multiplication on a Montgomery curve is:

Algorithm 1: L-R Binary Montgomery ladder.

```
INPUT: A point $P$ on $E_m$ and a positive integer
        $n=(n_t\ldots n_0)_2$
OUTPUT: The point $[n]P$

[Initialize]
$P_1 \leftarrow P$ and $P_2 \leftarrow [2]P$
[Loop through the scalar bits]
for $i=t-1$ down to 0 do
    if $n_i=0$ then
        $P_2 \leftarrow P_2+P_1$    (P);  $P_1 \leftarrow 2P_1$
    else
        $P_1 \leftarrow P_2+P_1$    (P);  $P_2 \leftarrow 2P_2$
    end if
end for

[Finalise]
return $P_1$
```

Algorithm 1 is a left-to-right algorithm as it processes the scalar bits from left to right. In all algorithms in this paper, whenever the difference between two points is required to compute the sum of those points, that difference is indicated in brackets immediately after the addition formula. For example, in the above algorithm we have

$$P_1 \leftarrow P_2+P_1 \qquad (P) \tag{2}$$

The notation in equation (2) means that when P_2 is added to P_1 and the result is stored in P_1 while the difference required between these two points is P i.e., $P_2-P_1=P$. From the above algorithm we can see that, to compute $[n]P$ (where $(n_t\ldots n_1n_0)_2$ is the binary representation of n and the most significant bit $n_t=1$), we hold $\{m_iP,(m_i+1)P\}$ for $m_i=(n_t\ldots n_i)_2$. If $n_i=0, m_iP=2m_{i+1}P$ and $(m_i+1)P=(m_{i+1}+1)P+m_{i+1}P$ else $m_iP=(m_{i+1}+1)P+m_{i+1}P$ and $(m_i+1)P=2(m_{i+1}+1)P$. Beginning from $\{P,2P\}$, the above algorithm computes $\{[n]P,[n+1]P\}$. A good reference to Montgomery ladders is (Joye and Yen, 2002).

3 SCHOENMAKERS' ALGORITHM FOR DOUBLE SCALAR MULTIPLICATION

Now we motivate the construction of the Schoenmakers' algorithm. Towards this end we define a set of four points $G_i=\{m_iP+n_iQ, m_iP+(n_i+1)Q,(m_i+1)P+n_iQ,(m_i+1)P+(n_i+1)Q\}$, for $m_i=(k_t\ldots k_i)_2$, $n_i=(l_t\ldots l_i)_2$. Below we show the construction of elements of G_i from G_{i+1} for all four combinations of (k_i,l_i):

1. $(k_i,l_i)=(0,0)$:
Here $m_i=2m_{i+1}$ and $n_i=2n_{i+1}$.

$$\begin{aligned}
&m_iP+n_iQ=2(m_{i+1}P+n_{i+1}Q);\\
&(m_i+1)P+n_iQ\\
&\ =((m_{i+1}+1)P+n_{i+1}Q)+(m_{i+1}P+n_{i+1}Q);\\
&m_iP+(n_i+1)Q\\
&\ =(m_{i+1}P+(n_{i+1}+1)Q)+(m_{i+1}P+n_{i+1}Q);\\
&(m_i+1)P+(n_i+1)Q\\
&\ =((m_{i+1}+1)P+n_{i+1}Q)+(m_{i+1}P+(n_{i+1}+1)Q)
\end{aligned}$$

2. $(k_i,l_i)=(1,0)$:
Here $m_i=2m_{i+1}+1$ and $n_i=2n_{i+1}$.

$$\begin{aligned}
&m_iP+n_iQ\\
&\ =((m_{i+1}+1)P+n_{i+1}Q)+(m_{i+1}P+n_{i+1}Q);\\
&(m_i+1)P+n_iQ=2((m_{i+1}+1)P+n_{i+1}Q);\\
&m_iP+(n_i+1)Q\\
&\ =((m_{i+1}+1)P+n_{i+1}Q)\\
&\qquad +(m_{i+1}P+(n_{i+1}+1)Q);\\
&((m_i+1)P+(n_i+1)Q\\
&\ =((m_{i+1}+1)P+(n_{i+1}+1)Q)\\
&\qquad +((m_{i+1}+1)P+n_{i+1}Q);
\end{aligned}$$

3. $(k_i,l_i)=(0,1)$:
Here $m_i=2m_{i+1}$ and $n_i=2n_{i+1}+1$.

$$\begin{aligned}
&m_iP+n_iQ\\
&\ =(m_{i+1}P+(n_{i+1}+1)Q)+(m_{i+1}P+n_{i+1}Q);\\
&(m_i+1)P+n_iQ\\
&\ =((m_{i+1}+1)P+n_{i+1}Q)\\
&\qquad +(m_{i+1}P+(n_{i+1}+1)Q);\\
&m_iP+(n_i+1)Q=2(m_{i+1}P+(n_{i+1}+1)Q);\\
&((m_i+1)P+(n_i+1)Q\\
&\ =((m_{i+1}+1)P+(n_{i+1}+1)Q)\\
&\qquad +(m_{i+1}P+(n_{i+1}+1)Q);
\end{aligned}$$

4. $(k_i,l_i)=(1,1)$:
Here $m_i=2m_{i+1}+1$ and $n_i=2n_{i+1}+1$.

$$\begin{aligned}
&m_iP+n_iQ=((m_{i+1}+1)P+n_{i+1}Q)\\
&\qquad +(m_{i+1}P+(n_{i+1}+1)Q);\\
&(m_i+1)P+n_iQ=((m_{i+1}+1)P+(n_{i+1}+1)Q)\\
&\qquad +((m_{i+1}+1)P+n_{i+1}Q);\\
&m_iP+(n_i+1)Q=((m_{i+1}+1)P+(n_{i+1}+1)Q)\\
&\qquad +(m_{i+1}P+(n_{i+1}+1)Q);
\end{aligned}$$

$$
\begin{aligned}
&(m_i+1)P+(n_i+1)Q\\
&\qquad = 2((m_{i+1}+1)P+(n_{i+1}+1)Q);
\end{aligned}
$$

The four elements in G_i, $0 \le i \le t$ give rise to the easy double scalar binary chain. Denoting $m_iP + n_iQ, m_iP+(n_i+1)Q, (m_i+1)P+n_iQ$ and $(m_i+1)P+(n_i+1)Q$ as P_1, P_2, P_3 and P_4 respectively, the easy double scalar algorithm is as follows:

Algorithm 2: L-R Easy Double scalar algorithm.

```
INPUT: Points $P$ and $Q$ on $E_m$; positive integers
        $k = (k_t \dots k_0)_2$ and $l = (l_t \dots l_0)_2$;
        Precompute $(P-Q)$; Identity is $\mathcal{O}$
OUTPUT: The point $[k]P+[l]Q$

[Initialize]
$P_1 \leftarrow \mathcal{O}$; $P_2 \leftarrow P$; $P_3 \leftarrow Q$; $P_4 \leftarrow P+Q$
[Loop through the two scalar bits simultaneously]
for $i = t$ down to 0 do
   if $(k_i, l_i) = (0,0)$ then
      $P_2 \leftarrow P_2 + P_1$   (P);   $P_3 \leftarrow P_3 + P_1$   (Q);
      $P_4 \leftarrow P_4 + P_1$   (P+Q);   $P_1 \leftarrow 2 * P_1$
   else if $(k_i, l_i) = (1,0)$ then
      $P_1 \leftarrow P_2 + P_1$   (P);   $P_3 \leftarrow P_2 + P_3$   (P-Q);
      $P_4 \leftarrow P_4 + P_2$   (Q);   $P_2 \leftarrow 2 * P_2$
   else if $(k_i, l_i) = (0,1)$ then
      $P_1 \leftarrow P_3 + P_1$   (Q);   $P_2 \leftarrow P_2 + P_3$   (P-Q);
      $P_4 \leftarrow P_4 + P_3$   (P);   $P_3 \leftarrow 2 * P_3$
   else if $(k_i, l_i) = (1,1)$ then
      $P_1 \leftarrow P_4 + P_1$   (P+Q);   $P_2 \leftarrow P_4 + P_2$   (Q);
      $P_3 \leftarrow P_4 + P_3$   (P);   $P_4 \leftarrow 2 * P_4$
end for
[$m_0P + n_0Q$ is in $P_1$]
return $P_1$
```

However, to compute m_0P+n_0Q, it is not necessary to use all the four elements of G_i for this purpose. If we omit $(m_i+1)P+(n_i+1)Q$ in G_i, it would still be possible to compute the rest of the elements in all of the G_i, $0 \le i \le t$. Amongst the above set of formulae, for the cases where $(k_i, l_i) = (0,0)$ or $(1,0)$ or $(0,1)$, no change is required, except omitting $(m_i+1)P+(n_i+1)Q$. However, when $(k_i,l_i) = (1,1)$, $(m_i+1)P+n_iQ$ and $m_iP+(n_i+1)Q$ may have to be computed differently from the formula above, as they depend on $(m_{i+1}+1)P+(n_{i+1}+1)Q$ in G_{i+1}. Therefore when $(k_i,l_i) = (1,1)$,
$(m_i+1)P+n_iQ$

$$
\begin{aligned}
&= ((m_{i+1}+1)P+(n_{i+1}+1)Q)\\
&\qquad + ((m_{i+1}+1)P+n_{i+1}Q)\\
&= ((m_{i+1}+1)P+n_{i+1}Q)\\
&\qquad + (m_{i+1}P+(n_{i+1}+1)Q)+P
\end{aligned}
$$

The difference between the first two terms in the above rewritten equation is $(P-Q)$. The difference between the first two terms taken together which is $((m_{i+1}+1)P+n_{i+1}Q+m_{i+1}P+(n_{i+1}+1)Q)$ and P can also be expressed in terms of elements in G_{i+1} i.e.,

$$
\begin{aligned}
&((m_{i+1}+1)P+n_{i+1}Q+m_{i+1}P+(n_{i+1}+1)Q)-P\\
&\qquad = (m_{i+1}P+(n_{i+1}+1)Q)+(m_{i+1}P+n_{i+1}Q);
\end{aligned}
$$

Similarly,
$m_iP+(n_i+1)Q$

$$
\begin{aligned}
&= (m_{i+1}+1)P+(n_{i+1}+1)Q)\\
&\qquad + (m_{i+1}P+(n_{i+1}+1)Q);\\
&= ((m_{i+1}+1)P+n_{i+1}Q)\\
&\qquad + (m_{i+1}P+(n_{i+1}+1)Q)+Q;
\end{aligned}
$$

As before, the difference between the first two terms in the above rewritten equation is $(P-Q)$. The difference between the first two terms taken together is $((m_{i+1}+1)P+n_{i+1}Q+m_{i+1}P+(n_{i+1}+1)Q)$and Q can also be expressed in terms of elements in G_{i+1} i.e.,

$$
\begin{aligned}
&((m_{i+1}+1)P+n_{i+1}Q+m_{i+1}P+(n_{i+1}+1)Q)-Q\\
&\qquad = ((m_{i+1}+1)P+n_{i+1}Q)+(m_{i+1}P+n_{i+1}Q);
\end{aligned}
$$

If we denote the reduced G_i i.e., G_i without $(m_i+1)P+(n_i+1)Q$ as G'_i and the elements (m_iP+n_iQ), $((m_i+1)P+n_iQ)$ and $(m_iP+(n_i+1)Q)$ as $P_1[i]$, $P_2[i]$ and $P_3[i]$ respectively, then for the case $(k_i,l_i) = (1,1)$, the elements of G'_i can be computed as follows:

$$
\left.
\begin{array}{lll}
P_1[i] \leftarrow P_2[i+1]+P_3[i+1] & & (P-Q)\\
P_2[i] \leftarrow P_1[i]+P & & (P_3[i+1]+P_1[i+1])\\
P_3[i] \leftarrow P_1[i]+Q & & (P_2[i+1]+P_1[i+1])
\end{array}
\right\} \quad (3)
$$

Thus in order to compute $P_2[i]$ and $P_3[i]$, we need to first compute $P_3[i+1]+P_1[i+1]$ and $P_2[i+1]+P_1[i+1]$. The difference between $P_3[i+1]$ and $P_1[i+1]$ is Q and the difference between $P_2[i+1]$ and $P_1[i+1]$ is P and thus it is possible to compute both $P_3[i+1]+P_1[i+1]$ and $P_2[i+1]+P_1[i+1]$. Rules for the other cases can be constructed from the formulae above. For example when $(k_i,l_i) = (0,0)$,

$$
\begin{array}{lll}
P_1[i] \leftarrow 2P_1[i+1] & & \\
P_2[i] \leftarrow P_2[i+1]+P_1[i+1] & & (P)\\
P_3[i] \leftarrow P_3[i+1]+P_1[i+1] & & (Q)
\end{array}
$$

As stated previously, Schoenmakers' algorithm was designed in 2000 and published in (Stam, 2003) in 2003. The derivation is not available in (Stam, 2003). In [5, Sec 4], Bernstein specifies which one of the four elements of G_i is eliminated. We fill this gap by motivating the construction here and this helps us in constructing the three dimensional analogue of Schoenmakers' algorithm in the next section. Doing away with the array notation for P_1, P_2 and P_3 above, we can present Schoenmakers' algorithm for double scalar multiplication as follows:

Algorithm 3: L-R Schoenmakers' Double scalar multiplication algorithm.

```
INPUT: Points $P$ and $Q$ on $E_m$; positive integers
        $k = (k_t \dots k_0)_2$ and $l = (l_t \dots l_0)_2$;
        Precompute $(P - Q)$
OUTPUT: The point $[k]P + [l]Q$

[Initialize]
 $P_1 \leftarrow \mathcal{O}$; $P_2 \leftarrow P$; $P_3 \leftarrow Q$

[Loop through the scalar bits simultaneously]
for $i = t$ down to 0 do
    $P_1 \leftarrow P_1$Store; $P_2 \leftarrow P_2$Store; $P_3 \leftarrow P_3$Store;
    if $(k_i, l_i) = (0,0)$ then
        $P_1 \leftarrow 2 * P_1$Store;$P_2 \leftarrow P_2$Store $+ P_1$Store    (P);
$P_3 \leftarrow P_3$Store $+ P_1$Store    (Q)
    else if $(k_i, l_i) = (1,0)$ then
        $P_1 \leftarrow P_2$Store $+ P_1$Store    (P);
        $P_2 \leftarrow 2 * P_2$Store;
        $P_3 \leftarrow P_2$Store$+ P_3$Store    (P-Q)
    else if $(k_i, l_i) = (0,1)$ then
        $P_1 \leftarrow P_3$Store $+ P_1$Store    (Q);
        $P_2 \leftarrow P_2$Store$+ P_3$Store    (P-Q);
        $P_3 \leftarrow 2 * P_3$Store
    else if $(k_i, l_i) = (1,1)$ then
        $P_1 \leftarrow P_2$Store $+ P_3$Store    (P-Q);
        $P_2$Partial $\leftarrow P_3$Store $+ P_1$Store    (Q);
        $P_3$Partial $\leftarrow P_2$Store $+ P_1$Store    (P);
        $P_2 \leftarrow P_1 + P$    ($P_2$Partial);
        $P_3 \leftarrow P_1 + Q$    ($P_3$Partial)
    end if
 end for

[$m_0 P + n_0 Q$ is in $P_1$]
return $P_1$
```

We note here that $P_2 Partial$ and $P_3 Partial$ are required as a result of Equation (3) above. We now compare Schoenmakers' algorithm for double scalar multiplication (Algorithm 3) with the straight forward method of achieving the same. Since in the For loop in Algorithm 3, (k_i, l_i) can take on any of the four values of $(0,0), (1,0), (1,0)$ and $(1,1)$ with equal probability, the average cost per bit of the two scalars taken simultaneously can be computed as:
when $(k_i, l_i) \neq (1,1)$:
three point additions are required.
i.e, $3(3M + 2S) = (9M + 6S)$ operations.
when $(k_i, l_i) = (1,1)$:
five point additions are required.
out of these five, three require $(3M + 2S)$ operations.
the other two require $4M + 2S$ operations (as Z-coordinate of $P_2 Partial, P_3 Partial \neq 1$).
resulting in a total of $3(3M + 2S) + 2(4M + 2S) = (17M + 10S)$ operations
Thus on the average

$$\frac{(3(9M+6S)+3(3M+2S)+2(4M+2S))}{4}$$

= $(11M + 7S)$ operations would be required to run Algorithm 3 for every bit of the two scalars taken together.

The straight forward method of computing $[k]P + [l]Q$ constitutes computing $[k]P$ and $[l]Q$ separately, recovering the Y-coordinates of $[k]P$ and $[l]Q$ and adding up $[k]P$ and $[l]Q$ in projective coordinates (Section 2.1 in (Akishita, 2001)). If we take the bit lengths of scalars k and l to be the same and equal to $|k|$, then this method requires $(12|k| + 28)M + (8|k|S)$. If one ignored the complexity of recovering the Y-coordinates and the complexity of adding up $[k]P$ and $[l]Q$, then the complexity of the straight forward method per bit of the scalar multiple is $((12|k|)M + (8|k|)S)/|k| = 12M + 8S$ operations. This can also be inferred from the fact that the total complexity to compute $[n]P$ using the binary ladder is $(6M + 4S)(|n|_2 - 1)$ for Montgomery curves (Refer to Remarks 13.36, page 288 in (Cohen and Frey, 2006)). Thus on the average, Schoenmakers' algorithm performs better than the straight forward method for double scalar multiplication.

Best case per bit cost of running Schoenmakers' double scalar multiplication algorithm is $(9M + 6S)$ and this occurs when none of the (k_i, l_i) is $(1,1)$ and under these circumstances Schoenmakers' algorithm will perform better compared to the straight forward algorithm.

Worst case per bit cost to run Algorithm 3 is $(17M + 10S)$ and this occurs when all of the $(k_i, l_i) = (1,1)$. $(17M + 10S) > 12M + 8S$ and thus under worst case conditions, the straight forward algorithm would be better than Schoenmakers' algorithm for double scalar multiplication.

In comparing the costs here, we have not taken into consideration the costs of the precomputation in Schoenmakers' algorithm and at the same time we have not taken into consideration the cost of recovering the Y-Coordinates and adding up $[k]P$ and $[l]Q$ in the straight forward method as these are small constant time costs and do not impact the result of the analysis presented here.

4 SCHOENMAKERS' ALGORITHM FOR TRIPLE SCALAR MULTIPLICATION

We now extend Schoenmakers' ideas for triple scalar multiplication. We do not explicitly derive the algorithm as the derivation is very similar to that of the double scalar multiplication algorithm. However, we do note that in every G_i', where the G_i' is analogous to that used in the double scalar multiplication case, $G_i' = \{m_iP + n_iQ + S_iR$, $(m_i+1)P + n_iQ + S_iR$, $m_iP + (n_i+1)Q + S_iR$, $m_iP + n_iQ + (S_i+1)R$, $(m_i+1)P + (n_i+1)Q + S_iR\}$. Taking $(m_iP + n_iQ + S_iR)$, $((m_i+1)P + n_iQ + S_iR)$, $(m_iP + (n_i+1)Q + S_iR)$, $(m_iP + n_iQ + (S_i+1)R)$ and $((m_i+1)P + (n_i+1)Q + S_iR)$ to be P_1 P_2, P_3, P_4 and P_5 respectively, the algorithm for triple scalar multiplication is as follows:

Algorithm 4: L-R Schoenmakers' triple scalar multiplication algorithm.

```
INPUT: Points $P$, $Q$ and $R$ on $E_m$;
      Positive integers $k = (k_t \ldots k_0)_2$, $l = (l_t \ldots l_0)_2$,
      $s = (s_t \ldots s_0)_2$
     Precompute $(P+Q)$, $(P-Q)$, $(P-R)$,
      $(Q-R)$, $(P+Q-R)$
OUTPUT: The point $[k]P + [l]Q + [s]R$

[Initialize]
 $P_1 \leftarrow \mathcal{O}$; $P_2 \leftarrow P$; $P_3 \leftarrow Q$; $P_4 \leftarrow R$; $P_5 \leftarrow P+Q$
[Loop through the 3 scalar bits simultaneously]
for $i = t$ down to 0 do
   $P_1 \leftarrow P_1$Store; $P_2 \leftarrow P_2$Store; $P_3 \leftarrow P_3$Store;
   $P_4 \leftarrow P_4$Store; $P_5 \leftarrow P_5$Store;
   if $(k_i, l_i, s_i) = (0,0,0)$ then
        $P_1 \leftarrow 2 * P_1$Store ;
        $P_2 \leftarrow P_2$Store $+P_1$Store     (P);
        $P_3 \leftarrow P_3$Store $+P_1$Store     (Q) ;
        $P_4 \leftarrow P_4$Store $+P_1$Store     (R);
        $P_5 \leftarrow P_5$Store $+P_1$Store     (P+Q)
   else if $(k_i, l_i, s_i) = (0,0,1)$ then
        $P_1 \leftarrow P_4$Store $+P_1$Store     (R);
        $P_2 \leftarrow P_2$Store $+P_4$Store     (P-R);
        $P_3 \leftarrow P_3$Store$+P_4$Store      (Q-R);
        $P_4 \leftarrow 2 * P_4$Store;
        $P_5 \leftarrow P_5$Store $+P_4$Store     (P+Q-R)
   else if $(k_i, l_i, s_i) = (0,1,0)$ then
        $P_1 \leftarrow P_3$Store $+P_1$Store     (Q);
        $P_2 \leftarrow P_2$Store $+P_3$Store     (P-Q);
        $P_3 \leftarrow 2 * P_3$Store;
        $P_4 \leftarrow P_3$Store $+P_4$Store     (Q-R);
        $P_5 \leftarrow P_5$Store $+P_3$Store     (P)
   else if $(k_i, l_i, s_i) = (0,1,1)$ then
        $P_1 \leftarrow P_3$Store $+P_4$Store     (Q-R) ;
        $P_2 \leftarrow P_5$Store $+P_4$Store     (P+Q-R) ;
        $P_3$Partial $\leftarrow P_4$Store $+P_1$Store     (R);
        $P_4$Partial $\leftarrow P_3$Store $+P_1$Store     (Q) ;
        $P_5$Partial $\leftarrow P_2$Store $+P_4$Store     (P-R) ;
        $P_3 \leftarrow P_1 + Q$     ($P_3$Partial) ;
        $P_4 \leftarrow P_1 + R$     ($P_4$Partial) ;
        $P_5 \leftarrow P_2 + Q$     ($P_5$Partial)
   else if $(k_i, l_i, s_i) = (1,0,0)$ then
        $P_1 \leftarrow P_2$Store $+P_1$Store     (P) ;
        $P_2 \leftarrow 2 * P_2$Store ;
        $P_3 \leftarrow P_2$Store $+P_3$Store     (P-Q) ;
        $P_4 \leftarrow P_2$Store $+P_4$Store     (P-R) ;
        $P_5 \leftarrow P_5$Store $+P_2$Store     (Q)
   else if $(k_i, l_i, s_i) = (1,0,1)$ then
        $P_1 \leftarrow P_2$Store $+P_4$Store     (P-R);
        $P_3 \leftarrow P_5$Store $+P_4$Store     (P+Q-R) ;
        $P_2$Partial $\leftarrow P_4$Store $+P_1$Store     (R);
        $P_4$Partial $\leftarrow P_2$Store $+P_1$Store     (P) ;
        $P_5$Partial $\leftarrow P_3$Store$+P_4$Store     (Q-R) ;
        $P_2 \leftarrow P_1 + P$     ($P_2$Partial) ;
        $P_4 \leftarrow P_1 + R$     ($P_4$Partial) ;
        $P_5 \leftarrow P_3 + P$     ($P_5$Partial)
   else if $(k_i, l_i, s_i) = (1,1,0)$ then
        $P_1 \leftarrow P_5$Store $+P_1$Store     (P+Q) ;
        $P_2 \leftarrow P_5$Store $+P_2$Store     (Q) ;
        $P_3 \leftarrow P_5$Store $+P_3$Store     (P) ;
        $P_4 \leftarrow P_5$Store $+P_4$Store     (P+Q-R);
        $P_5 \leftarrow 2 * P_5$Store
   else if $(k_i, l_i, s_i) = (1,1,1)$ then
        $P_1 \leftarrow P_5$Store $+P_4$Store     (P+Q-R) ;
        $P_2$Partial $\leftarrow P_3$Store $+P_4$Store     (Q-R) ;
        $P_3$Partial $\leftarrow P_2$Store $+P_4$Store     (P-R) ;
        $P_4$Partial $\leftarrow P_2$Store $+P_3$Store     (P-Q) ;
        $P_5$Partial $\leftarrow P_4$Store $+P_1$Store     (R) ;
        $P_2 \leftarrow P_1 + P$     ($P_2$Partial)
        $P_3 \leftarrow P_1 + Q$     ($P_3$Partial);
        $P_4 \leftarrow P_1 + R$     ($P_4$Partial);
        $P_5 \leftarrow P_1 + (P+Q)$     ($P_5$Partial)
   end if
end for
return $P_1$
```

We now compute the per bit average cost of the above algorithm.

when $(k_i, l_i, s_i) = (0,0,0)$ or $(0,0,1)$ or $(0,1,0)$ or $(1,0,0)$ or $(1,1,0)$:

five point additions are required.

i.e, $5(3M+2S) = (15M+10S)$ operations.

when $(k_i, l_i, s_i) = (0,1,1)$ or $(1,0,1)$:

eight point additions are required.

out of these eight, five require $(3M+2S)$ operations each i.e.,

$$5(3M+2S) = 15M+10S.$$

the other three require $4M+2S$ operations each i.e.,

$$3(4M+2S) = 12M+6S.$$

resulting in a total of $(27M+16S)$ operations.

when $(k_i, l_i, s_i) = (1,1,1)$:
nine point additions are required.
out of these nine, five require $(3M + 2S)$ operations each i.e.,

$$5(3M + 2S) = 15M + 10S.$$

the other four require $4M + 2S$ operations each i.e.,

$$4(4M + 2S) = 16M + 8S.$$

resulting in a total of $(31M + 18S)$ operations.

Thus on the average

$$\frac{(5(15M + 10S) + 2(27M + 16S) + (31M + 18S))}{8}$$

$= (20M + 12.5S)$ operations would be required to run Algorithm 4 for every bit of the exponent.
To compute x-coordinate of $kP + lQ + uR$ using the straight forward method, we require the following steps:

1. Compute kP using Algorithm 1(Binary ladder).
2. Recover Y-coordinate of kP.
3. Compute lQ using Algorithm 1(Binary ladder).
4. Recover Y-coordinate of lQ.
5. Compute uR using Algorithm 1(Binary ladder).
6. Recover Y-coordinate of uR.
7. Compute $kP + lQ + uR$ in projective coordinates.
8. Compute x-coordinate of $kP + lQ + uR$ as $x = X/Z$

We will assume the bit length of all the three scalars k, l and u to be the same. The algorithm for recovery of the Y-coordinate is described in (Okeya and Sakurai, 2001) and this costs $(12M + S)$ operations. Then, the computational cost of step 1, 3 and 5 together is $3\left[(6|k| - 3)M + (4|k| - 2)S\right]$. Steps 2, 4 and 6 together costs $3(12M + S)$. Step 7 costs $2(10M + 2S)$ while step 8 costs $M + I$ where I denotes a field inversion. Thus the cost of computing the x-coordinate of $kP + lQ + sR$ is $(18|k| + 48)M + (12|k| + 3)S + I$. Ignoring the costs of recovering the individual Y-coordinates and adding the resulting sums, the approximate cost per bit when the straight forward algorithm is used to compute the triple exponentiation is $(18M + 12S)$ operations. Then on the average, the above straight forward method performs better than Schoenmakers' algorithm for triple scalar multiplication.

In the best case, the per bit cost of running Algorithm 4 is $5(3M + 2S) = (15M + 10S)$ and this occurs when $(k_i, l_i, s_i) = (0,0,0)$ or $(0,0,1)$ or $(0,1,0)$ or $(1,0,0)$ or $(1,1,0)$. Under these circumstances Schoenmakers' algorithm performs better than the straight forward algorithm.

Worst case per bit cost of Schoenmakers' algorithm is $31M + 18S$ and this occurs when all of $(k_i, l_i, s_i) = (1,1,1)$. Thus under worst case conditions, the straight forward algorithm would perform better than Schoenmakers' algorithm for triple scalar multiplication.

The best case, average and worst case comparisons between Schoenmakers' algorithm and the straight forward method can be summarized as in the table below. The table lists the better option between Schoenmakers' algorithm and the straight forward method under best case, average and worst case conditions.

Table 1: Straight Forward Vs Schoenmakers'.

	Double scalar Schoenmaker Vs Straight Forward	Triple scalar Schoenmaker Vs Straight Forward
Best Case	Schoenmaker	Schoenmaker
Average	Schoenmaker	Straight Forward
Worst	Straight Forward	Straight Forward

5 CONCLUSION

In this paper, we showed the derivation of Schoenmakers' algorithm for double scalar multiplication and used this to construct the analogue of Schoenmakers' algorithm for triple scalar multiplication. We further showed that Schoenmakers' algorithm for triple scalar multiplication is more expensive than the straight forward method except under best case conditions. Thus Schoenmakers' algorithm for triple exponentiation may not be the best option for implementation in smart cards and mobile devices. In (Brown, 2014), the author constructed higher degree analogues of Bernstein's binary double scalar multiplication algorithm, but this algorithm is currently patented. Moreover, the x-coordinate only formulae that was initially developed for Montgomery curves have been generalized for other types of elliptic curves as well. For instance, the authors in (Lopez and Dahab, 1999) generalized this idea to Weierstrass form binary curves and the authors in (Brier and Joye, 2002) generalized it to Weierstrass Curves defined over $GF(p)$. In addition, Montgomery curves themselves have been the focus of recent research (Bernstein, 2006a) and there is potential for the Montgomery curve Curve25519 to be standardized. These reasons motivate the need to construct triple scalar multiplication analogues of other double scalar multiplication algorithms such as

Akishita's and Euclidean chain algorithms. One such efficient triple scalar multiplication algorithm is the subject matter of a sequel publication by this author (Rao, 2015) in due course.

ACKNOWLEDGEMENTS

The author would like to thank the anonymous reviewers for their useful comments and suggestions.

REFERENCES

Akishita, T. (2001). Fast simultaneous scalar multiplication on elliptic curve with montgomery form. In *Proceedings of Selected Areas in Cryptography, 2001*, LNCS Vol 2259.

Antipa, A., Brown, D., Gallant, R., Lambert, R., Struik, R., and Vanstone, S. (2005). Accelerated verification of ecdsa signatures. Technical report, http://www.cacr.math.uwaterloo.ca/techreports/2005/tech_reports2005.html.

Azarderakhsh, R. and Karabina, K. (2012). A new double point multiplication method and its implementation on binary elliptic curves with endomorphisms. Technical report, http://cacr.uwaterloo.ca/techreports/2012/cacr2012-24.pdf.

Bellman, R. and Straus, E. (1964). Addition chains of vectors (problem 5125). *The American Mathematical Monthly*, 71.

Bernstein, D. J. (2006a). Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006*, LNCS Vol 3958.

Bernstein, D. J. (2006b). Differential addition chains. Technical report, http://cr.yp.to/ecdh/diffchain-20060219.pdf.

Brier, E. and Joye, M. (2002). Weierstrass elliptic curves and side channel attacks. In *Public Key Cryptography - PKC 2002*, LNCS Vol 2274.

Brown, D. (2014). Multi-dimensional montgomery ladders for elliptic curves, patent no. us8750500 b2. Technical report, http://www.google.com/patents/US8750500.

Cheon, J. H. and Yi, J. H. (2007). Fast batch verification of multiple signatures. In *10th International Conference on Practice and Theory in Public Key Cryptography 2007*, LNCS Vol 4450.

Cohen, H. and Frey, G. (2006). *Handbook of Elliptic and HyperElliptic Curve Cryptography*. Chapman and Hall/CRC.

Joye, M. and Yen, S. (2002). The montgomery powering ladder. In *Proccedings of Cryptographic hardware and embedded systems (CHES) 2002*, LNCS Vol 2523.

Knuth, D. (1998). *The Art of Computer Programming, Vol2, Third Edition*. Pearson.

Lopez, J. and Dahab, R. (1999). Fast multiplication on elliptic curves over gf(2^m) without precomputation. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES) 1999*, LNCS Vol 1717.

Menezes, A., van Oorschot, P., and Vanstone, S. (1997). *Handbook of Applied Cryptography*. Taylor and Francis, 1997.

Montgomery, P. L. (1987). *Speeding the Pollard and elliptic curve methods of factorization, Mathematics of Computation 48*.

Montgomery, P. L. (1992). Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via lucas chains. Technical report, ftp://ftp.cwi.nl/pub/pmontgom/Lucas.ps.gz.

Okeya, K. and Sakurai, K. (2001). Efficient elliptic curve cryptosystems from a scalar multiplication algorithm with recovery of the y-coordinate on a montgomery form elliptic curve. In *Proceedings of Cryptographic Hardware and Embedded Systems (CHES) 2001*, LNCS Vol 2162.

Rao, S. R. S. (2015). Three dimensional montgomery ladder for elliptic curves, awaiting publication. Technical report, to be published.

Stam, M. (2003). *Speeding up subgroup cryptosystems*. PhD thesis, Technische Universiteit Eindhoven.

Stinson, D. (2006). *Cryptography - Theory and Practice, 3rd Edition*. CRC Press.