Partially Fixed Point Multiplication

Majid Khabbazian [*], T. Aaron Gulliver [†] and Vijay K. Bhargava [‡]

Abstract

A new technique is proposed in which bandwidth and memory are together used to reduce both the number of point additions and doublings required in computing random point multiplication. Using the proposed technique, we show that a significant speed-up can be obtained at the cost of slightly increased bandwidth. In addition, we show that the proposed technique is well-suited for parallel processing.

Keywords: cryptography, elliptic curves, point multiplication, fast computation, parallel processing.

1 Introduction

Point multiplication, kP, is a fundamental operation which dominates the execution time of elliptic curve cryptosystems. Therefore, significant study has been done to reduce the time needed to perform this operation. There are several approaches to speed up this operation, such as: (1) reducing the number of elliptic curve operations required in computing point multiplication, (2) speeding up elliptic curve operations such as point doubling, and (3) using faster operations available in certain special elliptic curves such as Koblitz curves [1]. Our contribution deals with the first approach. In this approach, memory is typically used by point multiplication algorithms to reduce the required number of elliptic curve operations. For example, when the point P is fixed, the number of point additions and doublings required in computing kP can be significantly reduced using a lookup table [2], leading to a fast implementation of the operation. However, lookup tables cannot be used to speed up random point multiplication because the point P is variable. In this case, a typical technique is to use low average Hamming weight integer representations such as w-NAF (see [3]) to reduce the required number of point additions.

In this paper, we propose a new technique which uses memory and bandwidth together to reduce both the number of point additions and point doublings required in computing random point multiplication. In the proposed technique, we add redundant information to the public key. This information can then be used by any entity, particularly those with constrained computational power or servers overloaded with EC-DSA verification requests, to speed up the operation. We also show that a substantial performance improvement can be obtained when multiple processors are available.

The rest of this paper is organized as follows. In Section 2, we briefly summarize elliptic curve operations. In Section 3, we explain random point multiplication. We then propose our new technique and explain its impact on computing random point multiplication in Sections 4 and 5, respectively. In Section 6, we show that the proposed technique is appropriate for parallel processing. Finally, we present some conclusions in Section 7.

[*] Dept. of Electrical and Computer Engineering, University of British Columbia, Canada, majidk@ece.ubc.ca

[†] Dept. of Electrical and Computer Engineering, University of Victoria, Canada

[‡] Dept. of Electrical and Computer Engineering, University of British Columbia, Canada

2 Elliptic Curves: Definition and Operations

An elliptic curve E over the field $\mathbb{F}$ is a smooth curve in the so called “long weirestraß form”

$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1}$$

where $a_1, \ldots, a_6 \in \mathbb{F}$. Equation (1) may be simplified to

$$E : y^2 = x^3 + ax + b, \tag{2}$$

and

$$E : y^2 + xy = x^3 + ax^2 + b, \tag{3}$$

when $Char(\mathbb{F}) \neq 2, 3$ and $Char(\mathbb{F}) = 2$, respectively.

2.1 Point Addition, Doubling, Subtraction and Multiplication

Let $E(\mathbb{F})$ be the set of points $P = (x, y) \in \mathbb{F}^2$ that satisfy the elliptic curve equation (along with a “point at infinity” denoted $\mathcal{O}$). It is well known that $E(\mathbb{F})$ together with the point addition operation given in Table 1 form an abelian group. When two operands are equal, i.e. $P_1 = P_2$, the operation is called point doubling. As shown in Table 2, point inversion in this group can be computed easily. Thus, point subtraction has the same cost as point addition. Note that in Tables 1 and 2, the points $P_1 = (x_1, y_1), P_2 = (x_2, y_2)$ and $P_3 = (x_3, y_3)$ are represented by Affine coordinates. Alternative coordinates such as Projective coordinates or Jacobian coordinates can be used to avoid field inversion when field inversion is much more expensive than field multiplication. Point multiplication is defined as repeated addition

$$kP = \underbrace{P + P + \ldots + P}_{k \text{ times}},$$

where P is an elliptic curve point, $k \in [1, N - 1]$ is an integer, and N is the order of point P.

$Char(\mathbb{F}) = 2$	$P_1 \neq \pm P_2$	$\lambda = \frac{y_2+y_1}{x_2+x_1}$ $x_3 = \lambda^2 + \lambda + x_1 + x_2 + a$ $y_3 = (x_1 + x_3)\lambda + y_1 + x_3$
	$P_1 = P_2$	$\lambda = \frac{y_1}{x_1} + x_1$ $x_3 = \lambda^2 + \lambda + a$ $y_3 = (x_1 + x_3)\lambda + y_1 + x_3$
$Char(\mathbb{F}) \neq 2, 3$	$P_1 \neq \pm P_2$	$\lambda = \frac{y_2-y_1}{x_2-x_1}$ $x_3 = \lambda^2 - x_1 - x_2$ $y_3 = (x_1 - x_3)\lambda - y_1$
	$P_1 = P_2$	$\lambda = \frac{3x_1^2+a}{2y_1}$ $x_3 = \lambda^2 - 2x_1$ $y_3 = (x_1 - x_3)\lambda - y_1$

Table 1: Elliptic curve point addition, $P_3 = P_1 + P_2$.

$Char(\mathbb{F}) = 2$	$-P_1 = (x_1, x_1 + y_1)$
$Char(\mathbb{F}) \neq 2, 3$	$-P_1 = (x_1, -y_1)$

Table 2: Elliptic curve point inversion.

3 Random Point Multiplication

There are many algorithms to accelerate computation of random point multiplication by reducing the required number of point additions [4, 5]. These algorithms typically consist of two stages, i.e. a precomputation stage and an evaluation stage. Algorithm 1 is an example of such algorithms. The precomputation stage of this algorithm consists of two steps. In Step 1, the multiplier k is recoded to a B-representation as

$$k = \sum_{0\leq i<l} k_i 2^i,$$

where $k_i \in B \cup \{0\}$ and B is a set of nonzero integers including 1. In Step 2, point multiplications dP for the integers $d \in B, d > 1$ are computed and stored. In the evaluation stage, the information obtained from these two steps is used to compute kP. Note that, in some algorithms such as window algorithm [4], the recoding process is done in the evaluation stage. This is because the recoding and evaluating processes scan the multiplier digits in the same direction, i.e. left-to-right or right-to-left. Therefore, these processes can be carried out simultaneously.

```
Algorithm 1: Random point multiplication
Input: An integer $k$ and a point $P \in E(\mathbb{F})$.
Output: $kP$.

Precomputation Stage:
1.  Compute the $B$-representation of the multiplier
    $k = \sum_{0\leq i<l} k_i 2^i$, $k_i \in B \cup \{0\}$.
2.  Compute and store $dP$ for all integers $d \in B$, $d > 1$.
Evaluation Stage:
1.  $R \leftarrow \mathcal{O}$.
2.  for $i$ from $l-1$ down to 0 do
3.      $R \leftarrow 2R$.
4.      if $k_i > 0$ then $R \leftarrow R + k_i P$.
5.      else if $k_i < 0$ then $R \leftarrow R - (-k_i)P$.
6.  return $R$.
```

The integer representation of the multiplier plays an important role in the performance of these algorithms. Some useful integer representations are binary, NAF, and w-NAF for which $B = \{1\}$, $B = \{\pm 1\}$, and $B = \{\pm 1, \pm 3, \ldots, \pm(2^{w-1}-1)\}$, respectively. If a binary representation is used in Algorithm 1, Steps 1 and 2 of the precomputation stage are not required. However, the evaluation stage would need $(\frac{l}{2}-1)$ point additions on average. Using the NAF representation, only Step 2 of the precomputation stage is required. In this case, the average number of point additions required in the evaluation stage is reduced to $(\frac{l}{3}-1)$. This is because the average Hamming weight (the number of nonzero digits in the representation) of NAF is $(\frac{l}{3})$. In fact, NAF has the minimal average Hamming weight among all B-representations with $B = \{\pm 1\}$. The NAF representation can be generalized to w-NAF. The w-NAF representation has the minimal average Hamming weight of $(\frac{l}{w+1})$ among all B-representations with $B = \{\pm 1, \pm 3, \ldots, \pm(2^{w-1}-1)\}$. Consequently, if w-NAF is used in Algorithm 1, the evaluation stage would require on average $(\frac{l}{w+1}-1)$ point additions. However, Step 2 of the precomputation stage requires $(2^{w-2}-1)$ point additions and one point doubling. In all these cases, the evaluation stage requires about $(l-1)$ point doublings. Note that $l = \lceil \log_2 N \rceil$ for the binary representation and $l = \lceil \log_2 N \rceil + 1$ for the NAF and w-NAF representations.

4 The Proposed Technique

Memory is commonly used in many point multiplication algorithms to speed up the operation. When the point P is known a priori, precomputing and storing some points in a lookup table can significantly speed up the operation by reducing both the required number of point additions and doublings. Random point multiplication algorithms cannot reduce both of these, even if there is a large amount of memory. This is due to the fact that the point P is variable. However, bandwidth can be used to speed up elliptic curve operations. It can for example be used to eliminate the square root operation when a point, Q, is sent in uncompressed format, i.e. both the x-coordinate and y-coordinate of Q are sent [6].

In our new technique, we use memory and bandwidth together to reduce both the required number of point additions and doublings. This can be done by adding redundant information to the public key (P). In Section 5, we explain how we can reduce the required number of point doublings for random point multiplication by half when we are provided with the point $2^{\lceil \frac{l}{2} \rceil}P$. In general, we can reduce the required number of point doublings by about $\frac{1}{n}$ when we are provided with the points $2^{\lceil \frac{l}{n} \rceil \times i}P$ for $i \in \{1, 2, \ldots n-1\}$. Therefore, having $P_2 = (P, 2^{\lceil \frac{l}{2} \rceil}P)$ or in general $P_n = (P, 2^{\lceil \frac{l}{n} \rceil \times 1}P, \ldots, 2^{\lceil \frac{l}{n} \rceil \times (n-1)}P)$ as the new public key allows other parties to significantly speed up the computation of random point multiplication. The redundant information in P_n need be precomputed and stored only once. However, this information is often used by fixed point multiplication algorithms such as the Lim and Lee algorithm [2], in which case extra storage is not required as the information is already available.

When public key certificates are used, we require a certificate on P_n. Including P_n instead of P in the certificate does not significantly increase the certificate size. For example, a typical size for an $X.509$ certificate is about $1K$ bytes [7]. For a 192-bit elliptic curve (as an example), the public key P requires $\lceil \frac{192 \times 2}{8} \rceil = 48$ bytes. However, using a compressed representation, the point P could be represented using one 192-bit value and one additional bit. It then requires $\lceil \frac{192+1}{8} \rceil = 25$ bytes. Therefore, changing the public key from P to $P_2 = (P, 2^{\lceil \frac{l}{2} \rceil}P)$ would increase the certificate size by less than 5% and 2.5% when the points are represented in an uncompressed and a compressed format, respectively. Note that the additional data can be stored as an extension of the certificate as $X.509$ version 3 [8] supports extensions.

5 Fixed-base-like Point Multiplication Using The Proposed Technique

In this section we determine the impact of using the proposed technique on the speed of random point multiplication algorithms. To give a detailed explanation, we present an improvement of Algorithm 1 (Algorithm 2), which employs P_n. We then compare the performance of these two algorithms in terms of the required number of point additions and point doublings. A similar approach can be used to show the impact of using the proposed technique on the performance of other random point multiplication algorithms.

Algorithm 2 is a generalization of Algorithm 1 because it is the same as Algorithm 1 for $n = 1$. In Algorithm 2, the B-representation of the multiplier k is divided into n subblocks of length $s = \lceil \frac{l}{n} \rceil$. In the evaluation stage of this algorithm, the point multiplication kP is computed as

$$kP = (\sum_{0 \le i < l} k_i 2^i)P = \sum_{0 \le i < s} \left(\sum_{0 \le j < n} (k_{sj+i} 2^i)(2^{sj}P) \right) .$$

Therefore, the required number of point doublings at this stage is reduced to $(\lceil \frac{l}{n} \rceil - 1)$ while the required number of point additions remains the same as that with Algorithm 1. Note that the evaluation stage of Algorithm 2 is the same as Moller's window w-NAF splitting [9]. However, Moller's algorithm does not require a precomputation stage as it is used for fixed point multiplication. In the precomputation stage of Algorithm 2, we need to compute the points $d(2^{sj}P)$, where $d \in B$, $d > 1$ and $0 \leq j < n$. For example, if the w-NAF representation is used, Algorithm 2 would require $n(2^{w-2}-1)$ point additions and n point doublings in its precomputation stage. Therefore, the cost of this stage is approximately $n(2^{w-2}-1)A + nD$, where A denotes the cost of a point addition, and D the cost of a point doubling.

Algorithm 2: Random Point Multiplication using P_n

Input: An integer k and $P_n = (P, 2^{\lceil \frac{l}{n} \rceil \times 1}P, \ldots, 2^{\lceil \frac{l}{n} \rceil \times (n-1)}P)$.
Output: kP.

```
Precomputation Stage:
Compute the $B$-representation of the multiplier
    $k = \sum_{0 \leq i < l} k_i 2^i$, $k_i \in B \cup \{0\}$.
Compute and store $d(2^{\lceil \frac{l}{n} \rceil \times j}P)$ for all integers $d \in B$,
    $d > 1$ and $0 \leq j < n$.
Evaluation Stage:
$s \leftarrow \lceil \frac{l}{n} \rceil$, $R \leftarrow \mathcal{O}$.
for $i$ from $(s-1)$ down to 0 do
    $R \leftarrow 2R$.
    for $j$ from $(n-1)$ down to 0 do
        if $(k_{sj+i} > 0)$ then $R \leftarrow R + k_{sj+i}(2^{sj}P)$
        else if $(k_{sj+i} < 0)$ then $R \leftarrow R - (-k_{sj+i})(2^{sj}P)$
return $R$.
```

The total cost of Algorithm 1 or Algorithm 2 depends on some properties of the B-representation employed, such as Hamming weight. Therefore, for an accurate performance comparison of these two algorithms we must know which B-representation is used. If a binary representation is used, the cost of Algorithm 1 and Algorithm 2 would be approximately $(\frac{l}{2}A + lD)$ and $(\frac{l}{2}A + \frac{l}{n}D)$, respectively. Therefore, we have

$$R \approx \frac{\frac{l}{2}A + lD}{\frac{l}{2}A + \frac{l}{n}D} = \frac{\frac{1}{2}+t}{\frac{1}{2}+\frac{t}{n}},$$

where $R = \dfrac{\text{cost of Algorithm 1}}{\text{cost of Algorithm 2}}$ and $t = \frac{D}{A}$. Similarly, when the NAF representation is used we have

$$R \approx \frac{\frac{l}{3}A + lD}{\frac{l}{3}A + \frac{l}{n}D} = \frac{\frac{1}{3}+t}{\frac{1}{3}+\frac{t}{n}}.$$

Figure 1 compares the performance of Algorithm 1 with that of Algorithm 2 for the cases when binary and NAF representations are used. As shown in this figure, the higher the values of t and n, the better the performance improvement achieved by replacing Algorithm 1 with Algorithm 2. The value of t depends on the finite field and coordinate system employed. For example, if the binary finite field and Affine coordinates system are used, we would have $t = 1$ (see Table 1). In this case, for $n = 3$ (as an example), Algorithm 2 is twice as fast as Algorithm 1 if they both use the NAF representation.

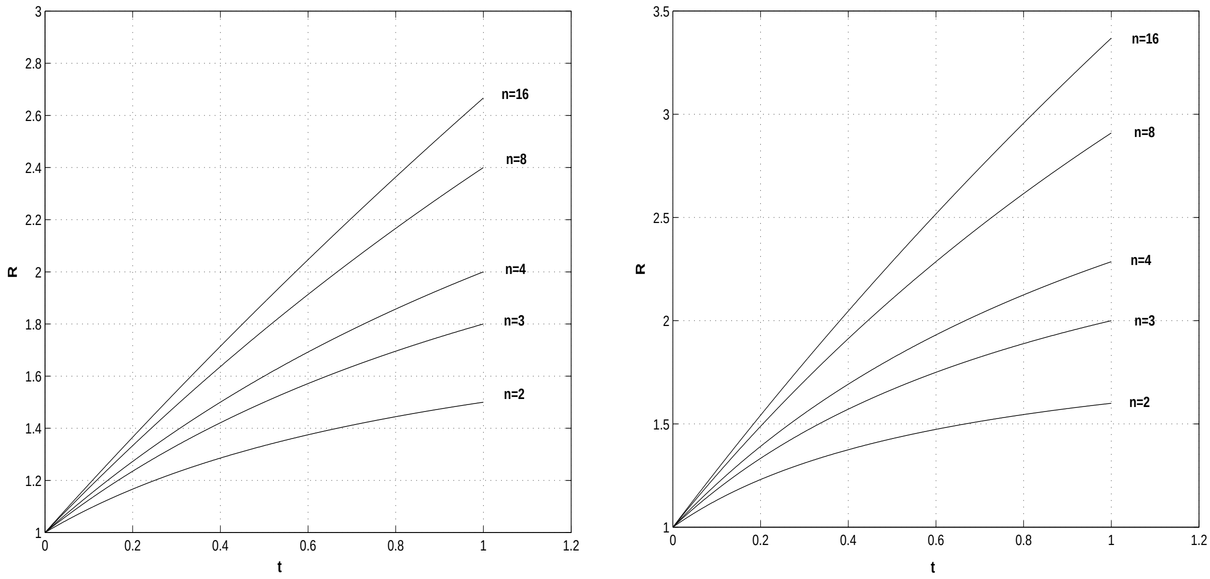

Figure 1: Performance comparison of Algorithms 1 and 2 for the cases when binary (left) and NAF (right) representations are used.

Now, consider the case that w-NAF ($w > 2$) is used by both Algorithms 1 and 2. The costs of Algorithms 1 and 2 are then $((\frac{l}{w+1} - 1) + (2^{w-2} - 1))A + lD$ and $((\frac{l}{w+1} - 1) + n(2^{w-2} - 1))A + (\lceil \frac{l}{n} \rceil + n - 1)D$, respectively. Therefore, we have

$$R \approx \frac{((\frac{l}{w+1} - 1) + (2^{w-2} - 1)) + lt}{((\frac{l}{w+1} - 1) + n(2^{w-2} - 1)) + (\frac{l}{n} + n - 1)t} = \frac{F(l, w)}{G(l, w, n)},$$

where

$$F(l, w) = ((\frac{l}{w+1} - 1) + (2^{w-2} - 1)) + lt,$$

and

$$G(l, w, n) = (\frac{l}{w+1} - 1) + n(2^{w-2} - 1)) + (\frac{l}{n} + n - 1)t.$$

One way to compare the performance of Algorithm 1 with that of Algorithm 2 is to compare their minimum costs. In order to do this, we need to determine the integers w_1, w_2 and n_2 for which $F(l, w_1)$ and $G(l, w_2, n_2)$ are minimal. Let

$$F_w(l) = arg \min_w (F(l, w)), \qquad l, w \in \mathbb{N}.$$

It is easy to verify that $w_1 = F_w(l)$, $w_2 \approx F_w([\frac{l}{n_2}])$ and $n_2 \approx \left\lfloor \sqrt{\frac{lt}{2^{w_2-2}+t-1}} \right\rfloor$. However, we set $w = \max\{3, w_2\}$ to avoid using ($w_2 = 2$) which was considered before (note that the NAF representation is the same as the w-NAF representation for $w = 2$). Figure 2 presents $F_w(l)$ for $100 \leq l \leq 1000$. As shown in this figure, $w_1 = 5, 6$ for typical values of l (i.e. $160 \leq l \leq 600$). Figure 3 compares the maximum achievable speed of Algorithm 2 with that of Algorithm 1. For example, for $t = 0.8$ and $l = 192$ we have $w_1 = 5$, $w_2 = 3$ and $n_2 = 9$. As shown in Figure 3, $R' = \frac{F(l,w_1)}{G(l,w_2,n_2)} = 2.4$. Therefore, the maximum achievable speed of Algorithm 2 is 2.4 times higher than that of Algorithm 1.

The cost of Algorithm 2 can be further reduced using Montgomery's trick [10]. As mentioned before, in the precomputation stage of Algorithm 2, we require that $d(2^{sj}P)$ be computed for $d \in B$, $d > 1$ and $0 \leq j < n$. When w-NAF is employed, this computation can be accomplished in n steps by computing the points $P_{i,j} = P_{i-1,j} + P_{0,j}$ for $0 \leq j < n$ in the i-th step, where $P_{0,j} = 2^{sj}P$ and $1 \leq i \leq n$. In the precomputation stage, it is preferable to represent the points $P_{i,j}$ using Affine coordinates in order to reduce the amount of memory required to store them, and to speed up the evaluation stage of the algorithm using mixed coordinates systems [11]. Using Affine coordinates, we require n simultaneous field inversions in each step. Montgomery's trick enables us to do this using one field inversion and $3(n-1)$ field multiplications [10]. Thus, for large n, one field inversion is replaced by approximately 3 field multiplications, which is a significant cost saving as a field inversion typically costs more than 3 field multiplications in most useful fields.

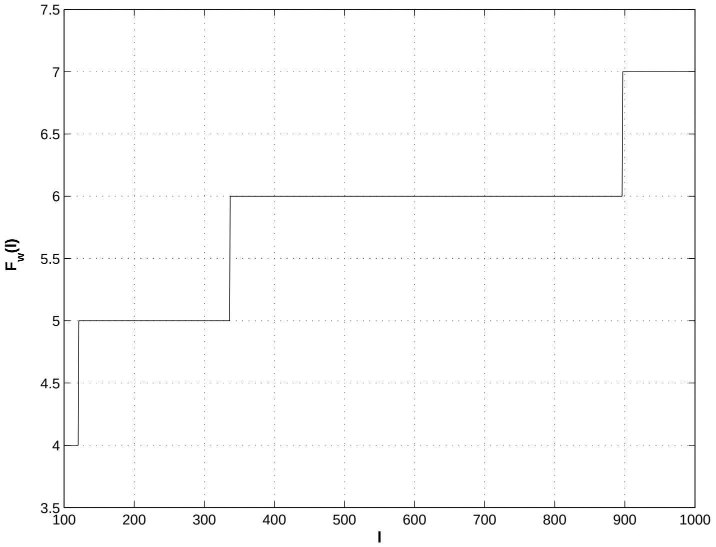

Figure 2: $F_w(l)$ for $100 \leq l \leq 1000$.

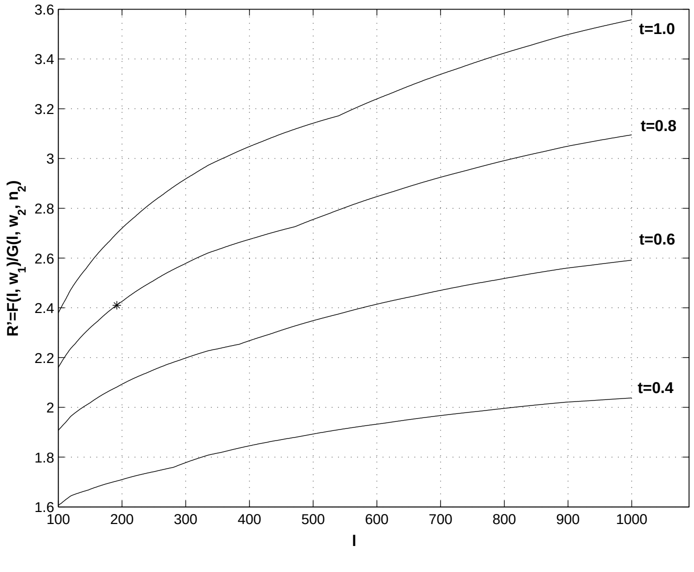

Figure 3: Comparison between the maximum speeds of Algorithms 1 and 2 when w-NAF is used.

6 Parallel Processing

Using the proposed technique, random point multiplication can be computed much faster with multiple processors. For example, assume that n processors are available and we have P_n. Then kP can be computed as follows

$$kP = (\sum_{i=0}^{l-1} k_i 2^i)P = \sum_{j=0}^{n-1} \left((\sum_{i=0}^{s-1} k_{sj+i} 2^i)(2^{sj}P) \right) = \sum_{j=0}^{n-1} R_j,$$

where $R_j = (\sum_{i=0}^{s-1} k_{sj+i}2^i)(2^{sj}P)$. The j-th processor can then compute R_{j-1} using Algorithm 1. Hence, each processor requires s point doublings and on average $((\frac{s}{w+1} - 1) + (2^{w-2} - 1))$ point additions when the w-NAF representation is used. Then $\lceil \log_2 n \rceil$ point additions are required to compute the final result. Therefore the total cost of parallel computation of kP is $((\frac{s}{w+1} - 1) + (2^{w-2} - 1) + \lceil \log_2 n \rceil)A + sD$. Consequently, using the proposed technique and parallel processing, random point multiplication can be computed about n times faster than Algorithm 1 when $(n \ll l)$ because

$$\frac{((\frac{l}{w+1} - 1) + (2^{w-2} - 1))A + lD}{((\frac{s}{w+1} - 1) + (2^{w-2} - 1) + \lceil \log_2 n \rceil)A + sD} \approx \frac{\frac{l}{w+1}A + lD}{\frac{1}{n}(\frac{l}{w+1}A + lD)} = n. \tag{4}$$

Note that for the algorithm used in the parallel processing, the optimal w is $F_w(\lceil \frac{l}{n} \rceil)$ which is generally smaller than $F_w(l)$ used in Algorithm 1. However, we used the same w in (4) for computational convenience. In fact, using the optimal w for parallel processing results in a value closer to n than in (4).

Consider the general case where we have m processors $(m < n)$ and we know P_n. Random point multiplication can then be computed in the following (similar) way

$$kP = (\sum_{i=0}^{l-1} k_i 2^i)P = \sum_{j=0}^{m-1} \left((\sum_{i=0}^{s'-1} k_{s'j+i} 2^i)(2^{s'j}P) \right) = \sum_{j=0}^{m-1} R_j,$$

where $s' = \lceil \frac{l}{n} \rceil \times \lceil \frac{n}{m} \rceil$ and $R_j = (\sum_{i=0}^{s'-1} k_{s'j+i}2^i)(2^{s'j}P)$. Note that in this case, each processor uses Algorithm 2 instead of Algorithm 1 to compute R_j.

7 Conclusions

In this paper, we proposed a new technique to use both bandwidth and memory to speed up the computation of random point multiplication. In the proposed technique, redundant information of the form $2^{\lceil \frac{l}{n} \rceil \times i}P$ is added to the public key P. It was shown that using these points significantly reduces the cost of computing random point multiplication via reducing the required number of point doublings. In addition, these points may already be available in a lookup table as they are often used in elliptic curve digital signature generation algorithms. We also showed that further cost savings can be obtained by using Montgomery's trick. Finally, the proposed technique was shown to be suitable for parallel processing. This technique can also be used to speed up exponentiation in Hyperelliptic Curve Cryptography (HECC) [12] and RSA (when a large public exponent is used).

References

[1] N. Koblitz, “CM-curves with good cryptographic properties,” *Advances in Cryptology - Proc. Crypto ’92*, vol. 576, pp. 279–287, 1992.

[2] C. H. Lim and P. J. Lee, “More flexible exponentiation with precomputation,” *Springer-Verlag Lecture Notes in Computer Science*, vol. 839, pp. 95–107, 1994.

[3] J. A. Muir and D. R. Stinson, “Minimality and other properties of the with-w nonadjacent form,” *Preprint*, 2004, Available from http://www.cacr.math.uwaterloo.ca/tech_reports.html.

[4] D. Gordon, “A survey of fast exponentiation methods,” *J. Algorithms*, vol. 27, pp. 129–146, 1998.

[5] I. Blake, G. Seroussy, and N. Smart, *Elliptic Curves in Cryptography.* Cambridge: Cambridge University Press, 1999.

[6] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography.* CRC Press, 1997.

[7] R. Zuccherato, “Elliptic curve cryptography support in entrust,” May 9, 2000.

[8] ITU-T, *ITU-T Recommendation X.509 version 3 (1997)*, Information Technology - Open Systems Interconnection - The Directory Authentication Framewordk, ISO/IEC 9594-8, 1997.

[9] B. Möller, “Improved techniques for fast exponentiation,” *Springer-Verlag Lecture Notes in Computer Science*, vol. 2587, pp. 298–312, 2003.

[10] H. Cohen, *A Course in Computational Number Theory, Graduate Texts in Math. 138.* Springer-Verlag, 1993, third Corrected Printing, 1996.

[11] H. Cohen, A. Miyaji, and T. Ono, “Efficient elliptic curve exponentiation using mixed coordinates,” *Springer-Verlag Lecture Notes in Computer Science*, vol. 1514, pp. 51–65, 1998.

[12] N. Koblitz, “Hyperelliptic cryptosystems,” *J. Cryptology*, vol. 1, pp. 139–150, 1989.