修 士 論 文

楕円曲線暗号におけるスカラー倍算の効率化に関する研究

北陸先端科学技術大学院大学
情報科学研究科

河面　祥男

2013 年 3 月

修 士 論 文

楕円曲線暗号におけるスカラー倍算の効率化に関する研究

指導教員　宮地充子　教授

審査委員主査　宮地充子 教授
審査委員　平石邦彦 教授
審査委員　面和成 准教授

北陸先端科学技術大学院大学
情報科学研究科

s1010901 河面　祥男

提出年月: 2013 年 2 月

目 次

1 はじめに

1.1 研究の背景

近年，使用されている暗号は大きく共通鍵暗号と公開鍵暗号に大別される．共通鍵暗号が暗号化と復号に同一の鍵を使用するのに対し，公開鍵暗号は暗号化と復号に異なる鍵（公開鍵と秘密鍵という）を使用する方式である．共通鍵暗号は鍵の秘匿が絶対条件であり，暗号通信に先立って送信者と受信者の間で如何に安全に鍵の交換を行うかという課題がある．一方，公開鍵暗号に比べ短い鍵長で良く，処理時間も早くなる特徴がある．公開鍵暗号は公開鍵と秘密鍵から構成され，公開鍵は一般に開示可能であるため，暗号通信に先立って送信者と受信者の間での鍵の交換が不要となる長所がある．一方，共通鍵暗号に比べ長い鍵長を必要とするため，処理時間が長い課題がある．このため，公開鍵暗号は暗号通信前の共通鍵の交換やディジタル署名などに主に利用され，共通鍵暗号は主に暗号通信に利用されている [1].

具体的な公開鍵暗号としては，1976 年に Diffie と Hellman によって提唱された RSA 暗号が最初になる．RSA 暗号は素因数分解の困難性を安全性の根拠とした暗号で，ブラウザにおける HTTPS 通信など広く利用されている．しかしながら，近年のコンピュータの処理性能の向上に伴い，安全性の確保のためにより鍵長の長い暗号が必要になってきている．NIST（米国立標準技術研究所）では，主な米国政府標準暗号の運用終了の意向を固めており，RSA 暗号では，鍵長が 1024 ビット以上から 2048 ビット以上に拡大されることになっている．一方，楕円曲線暗号は，RSA 暗号と同じく公開鍵暗号の区分される暗号であり，1985 年に Koblitz と Miller によって提案された．楕円曲線暗号は，楕円曲線上で定義された加法群の離散対数問題の困難さを安全性の根拠とした暗号で RSA 暗号に比べ短い鍵長で同等の安全性を確保することが可能であり，NIST による鍵長の推奨値は今後も 224 ビット以上に留まる予定である [2][3]．以上のように楕円曲線暗号は RSA 暗号よりも後発な公開鍵暗号であるが，より小さい鍵長で同等の安全性を確保することが可能であり，RSA 暗号に比べ少ない処理能力で効率に演算可能な特徴をもつ．このため，楕円曲線暗号はスマートカードなど処理能力が限定された小型の組込みデバイスへの普及が期待されている．楕円曲線暗号の普及のためには，楕円曲線暗号の効率化，すなわちスカラー倍算 (kP) の効率化が重要となり，活発に研究が行われている．楕円曲線暗号の演算は，スカラー倍算と呼ばれる演算が主要な演算処理となり，スカラー倍算は楕円曲線上の加法演算と，加法演算を構成する有限体上の四則演算から構成される．楕円曲線暗号の効率化のための既存研究の手法は，楕円曲線上の加法演算数の削減，加法演算を構成する有限体上の四則演算数の削減に大別される．楕円曲線上の加法演算数の削減に関しては，window method, NAF，DBNS，加法演算を構成する有限体上の四則演算の削減に関しては mixed coordinates，四則演算の効果的な組み合わせなどが提案されている．また，それぞれの提案手法によるスカラー倍算の計算量を定義体上の乗算回数 [m] に換算すると，鍵長が 160 ビットの場合で，window method が 2511[m], NAF が 2214[m]，DBNS が 1926[m]，また，mixed coordinates が 1918.5[m] などを達成している [5][7]．一方で，スマートカードの用途としては，駅の改札や店での買い物など非常に短い時間で暗号処理を要求される場面が多く，このため，楕円曲線暗号の更なる効率化が引き続き重要な研究課題となっている．

1.2 研究の目的

スカラー倍算の効率化により，楕円曲線暗号の暗号化／復号処理の効率化を図ることで，スマートカードなど小型の組込みデバイスへの楕円曲線暗号の普及を促進することを目的とする．具体的には，既存研究の主な手法である加算連鎖，座標系の変換，四則演算における課題を洗い出し，その改善アルゴリズムの考案によって効率化を目指す.

1.3 研究の成果

スカラー倍算を構成する楕円曲線上の加法演算において，既存研究において最速と考えられる DBNS(Double-Base Number System) の問題点を明らかにすると共にその改善案を提案した．また，改善案と既存研究の手法を比較評価し，既存研究の手法に比べ最大で 6.7% の計算コスト削減が出来ることを示した．また，DBNS の導出処理に関して，既存研究の計算コスト $(\log k)$ に対し，より計算コストを削減した $(\log(\log k))$ の手法を提案した.

2　楕円曲線暗号

楕円曲線暗号は，前述のように楕円曲線上で定義された加法群の離散対数問題の困難さを安全性の根拠にした暗号方式である．楕円曲線暗号で利用する代表的な楕円曲線は Weierstrass 標準形と呼ばれ，標数 3 超の素体 F_p 上において下式により定義される．

$$E\colon y^2 = x^3 + ax + b \tag{1}$$

ここで，$a, b \in F_p, 4a^3 + 27b^2 \neq 0$ である．

2.1　楕円曲線上の加法演算

楕円曲線上の加法演算は，次のように定義される．楕円曲線上における有理点 P，Q を加法した点は，点 P，Q を結ぶ直線が楕円曲線と交わる点の y 座標の符号を反転した点 R で定義される．楕円曲線上の有理点は加法において群をなすことが分かっており，従って，点 R は必ず楕円曲線上の有理点として存在する．加法の特別な場合として，P=Q となる場合の加法は，点 P において楕円曲線に接する接線を引き，接線と楕円曲線が再び交わる点の y 座標の符号を反転した点 S で定義される．加法において，P≠Q の場合を加算，P=Q のときを 2 倍算と呼ぶ.

楕円曲線上の加法演算の定義に従い，先の Weierstrass 標準形における点 $\mathrm{P}(x_1, y_1)$，点 $\mathrm{Q}(x_2, y_2)$ の加算となる点 $\mathrm{R}(x_3, y_3)$ は以下の四則演算により求めることが出来る．[図 1]

$$x_3 = \lambda^2 + x_1 + x_2 \tag{2}$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \tag{3}$$

$$\lambda = (y_2 - y_1)/(x_2 - x_1) \tag{4}$$

また，点 $\mathrm{P}(x_1, y_1)$ の 2 倍算となる点 $\mathrm{S}(x_3, y_3)$ は，以下のように求められる．[図 1]

$$x_3 = \lambda^2 - 2x_1 \tag{5}$$

$$y3 = \lambda(x_1 - x_3) - y_1 \tag{6}$$

$$\lambda = (3(x_1)^2 - a)/2y_1 \tag{7}$$

楕円曲線上での加算，2 倍算の計算は，上式から四則演算の加算，減算，乗算，2 乗算及び逆元計算から構成されることが分かる．ここで，加算，減算は乗算，2 乗算及び逆元計算に比べ，無視出来るほどの計算量であるため，楕円曲線上での加算，2 倍算の計算量は，四則演算の乗算，2 乗算及び逆元計算の回数により表すことが出来る．上式から，乗算を M，2 乗算を S，逆元計算を I とおくと，加算の計算量は $2M + S + I$，2 倍算の計算量は $2M + 2S + I$ となることが分かる．なお，ここでの四則演算である乗算，2 乗算及び逆元計算は有限体上の演算であることに注意が必要である．

楕円曲線暗号の主要な演算処理であるスカラー倍算は，楕円曲線上に基本となる点 P（ベースポイント）を定め，スカラー k に対して $kP = P + P + \cdots + P$ を求める演算となる．逆に，kP と P からスカラー k

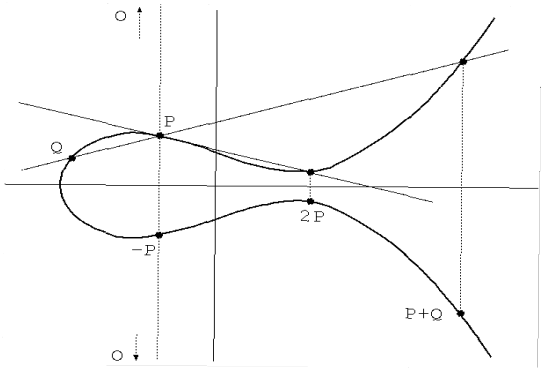

図 1: 楕円曲線上での加算と 2 倍算

を求めることを楕円曲線上の離散対数問題 (ECDLP) といい，現在までのところ，この問題を解く効率的なアルゴリズムは見つかっておらず，これが楕円曲線暗号の安全性の根拠となっている．

2.2 暗号化と復号

公開鍵暗号の用途は前述のとおり，主に暗号通信前の鍵交換とディジタル署名であり，公開鍵暗号の１つである楕円曲線暗号の用途も同様となる．以下に，楕円曲線暗号を利用した鍵交換及びディジタル署名について示す．[24] なお，以下では G を楕円曲線上のベースポイントとし，送信者 A の秘密鍵を d_A，公開鍵を $P_A = d_A G$ とする．また，同様に受信者 B の秘密鍵を d_B，公開鍵を $P_B = d_B G$ とする．

鍵交換　1. 送信者 A は，自身の秘密鍵 d_A と受信者 B の公開鍵 P_B から $d_A P_B = d_A d_B G = K_{A,B}$ を計算.
2. 受信者 B は，自身の秘密鍵 d_B と受信者 A の公開鍵 P_A から $d_B P_A = d_B d_A G = K_{A,B}$ を計算.
3. 送信者 A と送信者 B は $K_{A,B}$ を暗号通信用の鍵として共有する．

ディジタル署名　送信者 A が送信者 B に平文 m に署名して受信者 B に送る場合を考える．また，関数 H を平文 m を $Z_n^* = 1, 2, \ldots, n-1$ に圧縮するハッシュ関数とする．
（署名生成）
1.$e = H(m)$ を計算
2. 乱数 $k \quad \in \quad Z_n^*$ を選択し，$R_1 = kG$ を計算
3.$r_1' = x(R_1)(modq)$ を計算
$x(R_1)$:R_1 の x 座標
4.$s = (e + r_1' d_A)/k$ を計算
5.(m, r_1', s) を受信者 B に送信.

（署名検証）
1.$e = H(m)$ を計算
2.A の公開鍵 P_A を用いて $r_1' = x((H(m)G + r'1P_A)/s)(modq)$ を検証

3　既存研究

楕円曲線暗号のスカラー倍算を効率化する既存手法は，前述のとおり，楕円曲線上の加法演算の効率化と，加法演算を構成する有限体上の四則演算の効率化に大別される．以下に既存研究における代表的な手法を示す．

3.1　バイナリ法

楕円曲線上の加法演算を行う最も基本的な計算方法は，バイナリー法と呼ばれる計算手法である．バイナリ法では，スカラー k を下式のように n ビットのバイナリ表現で表す．

$$k = k_0 2^{n-1} + k_1 2^{n-2} + \cdots + k_{n-2} 2^1 + k_{n-1} 2^0 \tag{8}$$

$$\text{但し}, k_i \in 0, 1, k = (k_0 k_1 \ldots k_{n-1})_2 \tag{9}$$

バイナリー法によるスカラー倍算の計算では，k をバイナリ表現で表し，左端から右端に向かって，ビット 1 の個所では 2 倍算と加算を，ビット 0 の個所では 2 倍算を行う．

Algorithm Binary method for scalar multiplication

Input $P, k = (k_0 k_1 \ldots k_{n-1})_2, k \in 0, 1$
Output kP
1: $Q \leftarrow P$
2: for $0 \leqq i \leqq n-2$ do
2-1: $Q \leftarrow 2Q$
2-2: if $k_i = 1$ then $Q \leftarrow Q + P$
3: print Q

式 2 において，$k_i = 0$ となる項を消去して，変形すると，

$$\text{式 } 2 = 2^{b_0} + 2^{b_1} + \cdots + 2^{b_{l-2}} + 2^{b_{l-1}} = 2^{b_{l-1}}(2^{b_{l-2}-b_{l-1}}(\ldots 2^{b_1-b_2}(2^{b_0-b_1}+1)+1)\cdots+1) \tag{10}$$

となり，バイナリ法によるスカラー倍算の計算量は，2 倍算が b_0 回，加算が $(l-1)$ 回であることが分かる．ここで，$b_0 = n-1$ であること，また，k_i が 1 となる確率は 0.5 であることから，n ビットのスカラー k のバイナリ法による計算量を n で表すと，2 倍算が $O(n)$，加算が $O(2/n)$ となる．

3.2 NAF

スカラー k を NAF(Non Adjacent Form) と呼ばれる符号付きバイナリ表現で表す手法である．NAF ではスカラー k に対し, まず $3k$ の符号なしバイナリ表現 $(e_{n+1}e_n \dots e_0)_2$ と k のバイナリ表現 $(f_{n+1}f_n \dots f_0)_2$ を求める．次に $k = (3k - k)/2$ により，$3k$ のバイナリ表現から k のバイナリ表現を減算して 2 で割ることによって得ることができる．$(k_i = e_{i+1} - f_{i+1})$．NAF では，1 または-1 の値が現れる確率が 1/3 であり，このため，スカラー倍算における加算の数をバイナリ法に比べ減らすことが出来る．NAF における楕円曲線上の加算，2 倍算の回数はそれぞれ $O(n)$，$O(n/3)$ である．

$$k = k_0 2^{n-1} + k_1 2^{n-2} + \cdots + k_{n-2} 2^1 + k_{n-1} 2^0 \tag{11}$$

$$\text{但し, } k_i \in -1, 0, 1,\ k = (k_0 k_1 \dots k_{n-1})_2 \tag{12}$$

Algorithm NAF method for scalar multiplication

Input $P, k = (k_0 k_1 \dots k_{n-1})_2, k \in -1, 0, 1$
Output kP
1: $Q \leftarrow P$
2: for $0 \leqq i \leqq n-2$ do
2-1: $Q \leftarrow 2Q$
2-2: if $k_i = 1$ then $Q \leftarrow Q + P$
2-3: if $k_i = -1$ then $Q \leftarrow Q - P$
3: print Q

3.3 DBNS

DBNS(Double Base Number System) とは，スカラー k の値を 2 及び 3 のべき乗の和とする表現方法である．DBNS では，スカラー k は下式で表される．

$$k = \sum_{i=1}^{n} s_i 2^{b_i} 3^{t_i}$$
$$s_i \in -1, 1 \quad b_i, t_i \geq 0$$

DBNS では，一般にバイナリー法に比べ項の数が少なくなる特性があるため，加算の回数を減らすことが可能である．一方で，各項を構成する 2 のべき乗や 3 のべき乗の数が増え，2 倍算や 3 倍算の回数が大幅に増えてしまっては，計算コストの削減にはつながらない．そこで，2 倍算や 3 倍算の回数を一定に抑える仕組みが提案されており，代表的な手法が DIMITROV ら [7] と，Meloni ら [10] によって提案されている．

3.3.1 Dimitrov らの手法

Dimitrov らの手法では，DBNS 表現に 2 のべき乗，3 のべき乗の回数が漸減する制約を付与し，そのような DBNS 表現を DBchains と呼ぶ．これによって，2 のべき乗，3 のべき乗の回数が小さい時の計算結果を 2 のべき乗，3 のべき乗の回数がより大きい時の計算に再利用することが可能であり，これによって，実際に計算が必要な 2 のべき乗，3 のべき乗の回数を，それらの値が最大の項（すなわち，下式の b_1, t_1）の値に抑えている．

$$k = \sum_{i=1}^{n} s_i 2^{b_i} 3^{t_i}$$
$$s_i \in -1, 1 \quad b_1 \geq b_2 \geq \cdots \geq b_n \geq 0, t_1 \geq t_2 \geq \cdots \geq t_n \geq 0$$

なお，上式を満たすような DBNS 表現の導出については，greedy algorithm による手法が提案されている．

Algorithm Greedy algorithm for DBchains

Input k, a n-bit positive integer; $b_{max}, t_{max} > 0$, the largest allowed binary and ternary exponents
Output The sequence $(s_i, b_i, t_i)_{i \geq 0}$ such that
$k = \sum_{i=1}^{m} s_i 2^{b_i} 3^{t_i}$,with $b_0 \geq \cdots \geq b_m \geq 0$ and $t_0 \geq \cdots \geq t_m \geq 0$
1: $s \leftarrow 0$
2: while $k > 0$ do
3: define $z = 2^b 3^t$, the best approximation of k with $0 \leq b \leq b_{max} and 0 \leq t \leq t_{max}$
4: print (s, b, t)
5: $b_{max} \leftarrow b, t_{max} \leftarrow t$
6: if $k < z$ then
7: $s \leftarrow -s$
8: $k \leftarrow |k - z|$

また，Double Base chainsr を利用したスカラー倍算のアルゴリズムが提案されている [7]．下記に素体上（Jacobian 座標）のスカラー倍算のアルゴリズムを示す．

Algorithm Scalar multiplication on prime field using Jacabian coodinates

---

Input An integer $k = \sum s_i 2^{b_i} 3^{t_i} s_i \in -1, 1$
$b_1 \geq b_2 \geq \cdots \geq b_n \geq 0, t_1 \geq t_2 \geq \cdots \geq t_n \geq 0, point P \in E(K)$
Output the point $kP \in E(K)$
1: $z \leftarrow s_1 P$
2: **for** $i = 1, \cdots, n-1$ **do**
3: $u \leftarrow b_i - b_{i-1}$
4: $v \leftarrow t_i - t_{i-1}$
5: $z \leftarrow 3^v z$
6: $z \leftarrow 2^u z$
7: $z \leftarrow z + s_{i+1} P$
8:**Return** z

また，下記に拡大体上（Affine 座標）のスカラー倍算のアルゴリズムを示す.

Algorithm Scalar multiplication on binary field using Affine coodinates

---

Input An integer $k = \sum s_i 2^{b_i} 3^{t_i} s_i \in -1, 1$
$b_1 \geq b_2 \geq \cdots \geq b_n \geq 0, t_1 \geq t_2 \geq \cdots \geq t_n \geq 0, pointP \in E(K)$
Output the point $kP \in E(K)$
1: $z \leftarrow s_1 P$
2: **for** $i = 1, \cdots, n-1$ **do**
3: $u \leftarrow b_i - b_{i-1}$
4: $v \leftarrow t_i - t_{i-1}$
5: **if** $u = 0$ **then**
6: $z \leftarrow 3(3^{v-1}z) + s_{i+1}P$
7: **else**
8: $z \leftarrow 3^v z$
9: $z \leftarrow 4^{(u-1)/2}z$
10: **if** $u \equiv 0(mod 2)$ **then**
11: $z \leftarrow 4z + s_{i+1}P$
12: **else**
13: $z \leftarrow 2z + s_{i+1}P$
14:**Return** z

また，DBNS では，2 倍算の繰り返しや，3 倍算，4 倍算を効率的に行うことでスカラー倍算の高速化が可能であるため，これらをを効率的に行う手法が提案されている.

表 1 に標数 3 超の素体上において，itoh らによって提案されている 2 倍算の繰り返しの計算コスト及び DIMITROV らによって提案 [7] されている 3 倍算，4 倍算の計算コストを示す.

表 1: 標数 3 超の素体上での計算コスト

operation	computation time
w-DBL^J	4w [m] +(4w+2)[s]
TPL^J	6[s]+10[m]
w-TPL^J	(4w+2)[s]+(11w-1)[m]
w-TPL^J/w'-DBL^J	(11w+4w'-1)[s]+(4w+4w'+3)[m]

また，表 2 に拡大体上において，Eisentrager ら，Ciet らによって提案されている 2 倍算＋加算，3 倍算，3 倍算+加算の計算コスト及び DIMITROV らによって提案 [7] された 3 倍算，4 倍算の計算コストを示す．

表 2: 拡大体上での計算コスト

operation	computation time				
	Eisentrager ら	Ciet ら	Guajardo ら	DIMITROV ら	BreakEvenPoint
DA^A	2[i]+2[s]+3[m]	1[i]+2[s]+9[m]			[i]/[m]=6
TPL^A	2[i]+2[s]+3[m]	1[i]+4[s]+7[m]		1[i]+3[s]+6[m]	[i]/[m]=3
TPL^A	3[i]+3[s]+4[m]	2[i]+3[s]+9[m]			[i]/[m]=5
QPL^A			1[i]+5[s]+8[m]	2[i]+3[s]+3[m]	[i]/[m]=5
QA^A			2[i]+6[s]+10[m]	3[i]+3[s]+5[m]	[i]/[m]=5

unsigned な DBNS 表現の個数に関する定理　$f(n)$ を DBNS の表現個数とした時，以下の式が成り立つ．

$f(n) = f(n-1) + f(n/3)$ if $n \equiv 0 \pmod 3$,

$\quad = f(n-1) \quad otherwise$

以下に上式が成り立つ理由を示す．

整数 n を以下の形式で表現し，

$n = h_0 + 3h_1 + 9h_2 + \cdots + 3^k h_k$

上式で $k = \log_3 n$

n を表現可能な $h_0, h_1, \cdots, h_k$ の組の個数を $g(n)$ と定義する．ここで $h_i (i = 0, 1, \cdots, k)$ をバイナリ表現に置き換えると，n は $2^a 3^b$ の和で表され，$h_0, h_1, \cdots, h_k$ の組は，DBNS 表現と 1 対 1 に対応する．

（例）$h_i = 101100$ の場合，$h_i 3^i = 2^5 3^i + 2^3 3^i + 2^2 3^i$

従って，整数 n の unsigned な DBNS 表現の個数は $g(n)$ と等しいため，$g(n)$ において先の式が成り立つことを示せばよい．証明方法としては，先の式を満たすような $f(n)$ の生成関数が $g(n)$ の生成関数と等しいことを示す．これを示せば，$g(n) = f(n)$ となり，$g(n)$ が先の $f(n)$ の式を満たすことが言える．

ここで，$g(n)$ の生成関数を $G(z)$ とおくと，

$$G(z) = (1 + z + z^2 + \cdots)(1 + z^3 + z^6 + \cdots) \cdots (1 + z^{3^k} + \cdots) = \frac{1}{(1-z)(1-z^3) \cdots (1-z^{3^k})}$$

一方，$f(n)$ の生成関数を $F(z)$ とおくと，

$$\begin{aligned} F(z) = \sum_{n=1}^{\infty} z^{3n} f(3n) + \sum_{n=1_{3\nmid n}}^{\infty} z^n f(n) &= \sum_{n=1}^{\infty} z^{3n} (f(3n-1) + f(n)) + \sum_{n=2_{3\nmid n}}^{\infty} z^n f(n-1) + z f(1) \\ &= z + \sum_{n=2}^{\infty} z^n f(n-1) + \sum_{n=1}^{\infty} z^{3n} f(n) \\ &= z + zF(z) + F(z^3) \end{aligned}$$

よって,

$$F(z) = \frac{z}{1-z} + \frac{z^3}{(1-z)(1-z^3)} + \frac{z^9}{(1-z)(1-z^3)(1-z^9)} + \cdots\cdots$$

$$\because F(z^3) = \frac{z^3}{1-z^3} + \frac{z^9}{(1-z^3)(1-z^9)} + \frac{z^{27}}{(1-z^3)(1-z^9)(1-z^{27})} + \cdots$$

$$= (1-z)(\frac{z^3}{(1-z)(1-z^3)} + \frac{z^9}{(1-z)(1-z^3)(1-z^9)} + \frac{z^{27}}{(1-z)(1-z^3)(1-z^9)(1-z^{27})} + \cdots)$$

$$= (1-z)(\frac{z}{1-z} + \frac{z^3}{(1-z)(1-z^3)} + \frac{z^9}{(1-z)(1-z^3)(1-z^9)} + \frac{z^{27}}{(1-z)(1-z^3)(1-z^9)(1-z^{27})} + \cdots) - z$$

$$= (1-z)F(z) - z$$

次に，$F(z)$ の z^n の係数を $[z^n]F(z)$ とすると,

$$[z^n]F(z) = [z^n](\frac{z}{1-z} + \frac{z^3}{(1-z)(1-z^3)} + \frac{z^9}{(1-z)(1-z^3)(1-z^9)} + \cdots$$

$$+ \frac{z^{3^k}}{(1-z)(1-z^3)\cdots(1-z^{3^k})} + \frac{z^{3^{k+1}}}{(1-z)(1-z^3)\cdots(1-z^{3^k})(1-z^{3^{k+1}})} \cdots$$

$$= [z^n](\frac{z}{1-z} + \frac{z^3}{(1-z)(1-z^3)} + \frac{z^9}{(1-z)(1-z^3)(1-z^9)} + \cdots + \frac{z^{3^k}}{(1-z)(1-z^3)\cdots(1-z^{3^k})}$$

$$= [z^n](\frac{1}{(1-z)(1-z^3)\cdots(1-z^{3^k})} - 1)$$

$$= [z^n](G(z) - 1) = [z^n](G(z)) \text{ (if } n \geq 1)$$

$$\because \frac{1}{(1-z)(1-z^3)\cdots(1-z^{3^k})}$$

$$= (1+z+\cdots)(1+z^3+\cdots)\cdots(1+z^{3^k}+\cdots)$$

$$= (1+z+\cdots)(1+z^3+\cdots)\cdots(1+z^{3^{k-1}}+\cdots)(z^{3^k}+\cdots) + (1+z+\cdots)(1+z^3+\cdots)\cdots(1+z^{3^{k-1}}+\cdots)$$

$$= \frac{z^{3^k}}{(1-z)(1-z^3)\cdots(1-z^{3^k})} + (1+z+\cdots)(1+z^3+\cdots)\cdots(1+z^{3^{k-1}}+\cdots)$$

$$= \frac{z^{3^k}}{(1-z)(1-z^3)\cdots(1-z^{3^k})} + \frac{z^{3^{k-1}}}{(1-z)(1-z^3)\cdots(1-z^{3^{k-1}})} + \cdots + \frac{z^3}{(1-z)(1-z^3)} + (1+z+\cdots)$$

$$= \frac{z^{3^k}}{(1-z)(1-z^3)\cdots(1-z^{3^k})} + \frac{z^{3^{k-1}}}{(1-z)(1-z^3)\cdots(1-z^{3^{k-1}})} + \cdots + \frac{z^3}{(1-z)(1-z^3)} + \frac{z}{1-z} + 1$$

3.3.2 Meloni らの手法

前述の DBChains では，2 及び 3 のべき乗の値が降順になるような DBNS 表現に限定するため，選択可能な DBNS 表現の数が大幅に削減される．このため，必ずしも加算の数を最小化するような DBNS 表現の選択が出来るとは限らない．この制約を緩和する手法として Yao's algorithm を利用した手法を Meloni らが提案している [10]．Meloni らの手法では，DBNS 表現を 3 のべき乗が同一の項をまとめ，以下のように変形する．

$k = d(0) + 3d(1) + 3^2 d(2) + \cdots + 3^{max(t_i)} d(max(t_i))$
$= (3(\cdots 3(3d(max(t_i)) + d(max(t_i) - 1)) + \cdots + d(1)) + d(0)$
但し，$d(j) = \sum_{t_i = j} 2^i$

ここで，$2^0, 2^1, 2^2, \cdots\cdots, 2^{max(b_i)}$ を事前に計算しておくことで，スカラー倍算の計算コストは，2 倍算が $max(b_i)$， 3 倍算が $max(t_i)$ となり，DBchains と同一の計算コストとなる．一方，加算の計算コストは DBNS の項数 -1 となり DBchains と同一に見えるが，DBchains に比べ，選択可能な DBNS 表現の数が増えるため，DBNS の項数もより小さいものを選択可能であり，DBchains よりも加算の計算コストが下がる可能性が高い手法である．

しかしながら，多くの 2 のべき乗の事前計算が必要になるため，メモリ容量が小さいデバイスでは利用が困難となる問題がある．

表 3 に Dimitrov らの手法と Meloni らの手法の比較を示す．

表 3: DBNS の既存研究

	Dimitrov ら (2005 年) [7]	Meloni ら (2009 年) [10]
DBNS 表現	2 及び 3 の降べき順 $(b_0 \geq b_1 \geq \cdots \geq b_m \geq 0,$ $t_0 \geq t_1 \geq \cdots \geq t_m \geq 0)$	2 と 3 のべき乗の最大値 の積がスカラー K 以内 $(k \geq 2^{max(b_i)}3^{max(t_i)})$
DBNS 表現 導出方法	Greedy algorithm	n/a
スカラー倍算 の計算方法	Left to Right	3 のべきが同一の項 をまとめて Left to Right
計算コスト (2 倍算)	2 のべきの最大値 （=初項の 2 のべき b0）	2 のべきの最大値
計算コスト (3 倍算)	3 のべきの最大値 （=初項の 3 のべき t0）	3 のべきの最大値
計算コスト (加算)	DBNS 表現の項数-1 ※ Meloni らの手法 に比べ，DBNS 表現の 項数は増加傾向	DBNS 表現の項数-1 ※ Dimitrov らの手法 に比べ制限が緩やかで, DBNS 表現の項数は 減少傾向
プレ計算	不要	2 のべきの最大値までの 全ての 2 のべき乗のプレ計算 が必要. $(2^0P, 2^1P, \cdots, 2^{max(b_i)})$

3.4 座標変換

3.4.1 単一座標

座標の変換を行うことで，有限体上の四則演算の回数を減らしたり，計算コストの高い演算（逆数計算等）を他の演算に置換する手法である．座標としては，座標の変換を行わない Affine 座標，座標の変換を行う Projective 座標，Jacobian 座標，Chudnovsky Jacobian 座標，Jacobian 座標がある．以下にそれぞれの概要を示す．

Affine 座標　座標表現の変換を行わない唯一の座標である．2 倍算，加算共に体上の逆数計算を含むため，その計算コストは逆数計算の処理時間に大きく依存する．仮に逆数計算の処理時間は乗算の 4〜10 倍程度とすると，Affine 座標を用いた加算は，すべての座標で最速となる．一方，逆数計算の処理時間がそれ以上の場合は，以下に示す座標の変換によって逆数計算を他の演算に置き換えることで処理時間の短縮を図ることが可能である．

Projective 座標　x=X/Z, y=Y/Z の座標変換を行うことで，体上の逆数計算を不要にする座標である．

Jacobian 座標　Projective 座標と同じく逆数計算を不要にする座標で，$x = X/Z^2, y = Y/Z^3$ の座標表現の変換を行う．Projective 座標に比べ，2 倍算は早いが，加算が遅くなる．

Chudnovsky Jacobian 座標　Jacobian 座標と同じく $x = X/Z^2, y = Y/Z^3$ の座標表現の変換を行うもので，Jacobian 座標が (x, y, z) の 3 つの値を保持するのに対し，Chudnovsky Jacobian 座標は (X, Y, Z, Z^2, Z^3) の 5 つの値を保持する．Jacobian 座標に対し，加算の処理速度を改善する一方で 2 倍算は遅くなる．

Modified Jacobian 座標　Jacobian 座標と同じく $x = X/Z^2, y = Y/Z^3$ の座標表現の変換を行うもので，Jacobian 座標が (x, y, z) の 3 つの値を保持するのに対し，Modified Jacobian 座標は (X, Y, Z, aZ^4) の 4 つの値を保持する．Jacobian 座標に対し，2 倍算の処理速度を大幅に向上させる一方，加算は遅い．

表 4 に各座標系での 2 倍算，加算の計算コストを示す．なお，() 内は 2 乗算，逆数計算をそれぞれ乗算の 0.8 倍，4〜10 倍とした場合の計算コストである．

表 4: 各座標での計算コスト

座標系	変換方法	座標構成要素	計算コスト	
			DBL	ADD
Affine	変換なし	2 つ (x, y)	2M+2S+I (=7.6M〜13.6M)	2M+S+I (=6.8M〜12.8M)
Projective	$x = X/Z, y = Y/Z$	3 つ (X, Y, Z)	7M+5S (=11M)	12M+2S (=13.6M)
Jacobian	$x = X/Z^2, y = Y/Z^3$	3 つ (X, Y, Z)	4M+6S (=8.8M)	12M+4S (=15.2M)
Chudnovsky Jacobian	$x = X/Z^2, y = Y/Z^3$	5 つ (X, Y, Z, Z^2, Z^3)	5M+6S (=9.8M)	11M+3S (=13.4M)
Modified Jacobian	$x = X/Z^2, y = Y/Z^3$	4 つ (X, Y, Z, aZ^4)	4M+4S (=7.2M)	13M+6S (=17.8M)

3.4.2 混合座標

異なる座標系同士で加算を行ったり，同一座標系の加算や 2 倍算の計算結果を別の座標で表すことでスカラー倍算の計算時間を早める手法である．表 5 に計算コストに有用と考えられる混合座標の計算コストを示す.

表 5: 混合座標での計算コスト

DBL		ADD	
operation	computation time	operation	computation time
$2P$	$7M + 5S$	$J^m + J^m$	$13M + 6S$
$2J^c$	$5M + 6S$	$J^m + J^c = J^m$	$12M + 5S$
$2J$	$4M + 6S$	$J + J^c = J^m$	$12M + 5S$
$2J^m$=J^c	$4M + 5S$	$J + J$	$12M + 4S$
$2J^m$	$4M + 4S$	$P + P$	$12M + 2S$
$2A$=J^c	$3M + 5S$	$J^c + J^c = J^m$	$11M + 4S$
$2J^m$=J	$3M + 4S$	$J^c + J^c$	$11M + 3S$
$2A$=J^m	$3M + 4S$	$J^c + J = J$	$11M + 3S$
$2A$=J	$2M + 4S$	$J^c + J^c = J$	$10M + 2S$
		$J^c + A = J^m$	$9M + 5S$
		$J^m + A = J^m$	$9M + 5S$
		$J^c + A = J^m$	$8M + 4S$
		$J^c + A = J^c$	$8M + 3S$
		$J + A = J$	$8M + 3S$
		$J^m + A = J$	$8M + 3S$
		$A + A = J^m$	$5M + 4S$
		$A + A = J^c$	$5M + 3S$
$2A$	$2M + 2S + I$	$A + A$	$2M + S + I$

なお，[5] の論文では，スカラー倍算を

1. 2 倍算 (doubling) の繰り返し
2. 2 倍算 (doubling) の繰り返しにおける最後の 2 倍算
3. pre-computed point の計算

に分割し，1. は 2 倍算が最も高速な Modified Jacobian，3. は加算が高速な Affine または Chudnovsky Jacobian，2. は 3. との加算をした結果を Modified Jacobian で表現するのに最も高速な Jacobian を使用することで，従来手法に比べての大幅な高速化を実現している．

3.5 サイドチャネル攻撃への耐性

近年，暗号に対する現実的な脅威としてサイドチャネル攻撃が指摘されている．サイドチャネル攻撃は，暗号化処理中の処理時間や電力といった情報を取得し，それらの情報を元に暗号化のための鍵を解読する攻撃である．例えば，体上での乗算と 2 乗算の計算において，処理時間や電力の違いが生じることから，乗算と 2 乗算がどのような順序で何回行われたかの情報を取得し，それたの情報を元に鍵を解読するといった攻撃手法である．
サイドチャネル攻撃に対して提案されている既存研究の手法について，以下に述べる．

3.5.1 side channel atomicity

side channel atomicity では暗号化（復号化）処理を，攻撃者にとって同一の処理にしか見えない処理単位 (side channel atomic block) に分割する．これによって，暗号化処理をサイドチャネル攻撃者にとって同一の処理単位の繰り返しに見せる手法である．以下に side channel atomicity の楕円曲線暗号への適用例について示す．なお，本研究では楕円曲線暗号に絞って記載を行うが，side channel atomicity は，楕円曲線暗号だけでなく，あらゆる暗号に適用可能な手法である．

表 6 は，素体上の楕円曲線暗号のスカラー倍算における 2 倍算，加算について side channel atomicity に分割する例である．[11]

(素体上での加算の計算式)
$P = (X_1, Y_1, Z_1), Q = (X_2, Y_2, Z_2), P + Q = (X_3, Y_3, Z_3)$
$X_3 = W^3 - 2U_1W^2 + R^2, Y_3 = -S_1W^3 + R(U_1W^2 - X_3), Z_3 = Z_1Z_2W, U_1 = X_1Z_2^2, U_2 = X_2Z_1^2, S_1 = Y_1Z_2^3, S_2 = Y_2Z_1^3, W = U_1 - U_2, R = S_1 - S_2$
$T_1 \leftarrow X_1, T_2 \leftarrow Y_1, T_3 \leftarrow Z_1, T_7 \leftarrow X_2, T_8 \leftarrow Y_2, T_9 \leftarrow Z_2$

(素体上での 2 倍算の計算式)
$P = (X_1, Y_1, Z_1), 2P = (X_3, Y_3, Z_3)$
$X_3 = M^2 - 2S, Y_3 = -M(S - X_3) - T, Z_3 = 2Y_1Z_1, S = 4X_1Y_1^2, M = 3X_1^2 + aZ_1^4, T = 8Y_1^2$
$T_0 \leftarrow a, T_1 \leftarrow X_1, T_2 \leftarrow Y_1, T_3 \leftarrow Z_1$

表 6: 素体上での 2 倍算，加算の side channel atomicity

	2 倍算	加算		2 倍算	加算
1	$T_4 \leftarrow T_1 \times T_1 (= X_1^2)$	$T_4 \leftarrow T_9 \times T_9 (= Z_2^2)$	9	$T_2 \leftarrow T_2 \times T_2 (= 4Y_1^4)$	$T_3 \leftarrow T_3 \times T_9 (= Z_1 Z_2)$
	$T_5 \leftarrow T_4 + T_4 (= 2X_1^2)$	(dummy)		$T_2 \leftarrow T_2 + T_2 (= T)$	(dummy)
	(dummy)	(dummy)		(dummy)	(dummy)
	$T_4 \leftarrow T_4 + T_5 (= 3X_1^2)$	(dummy)		$T_5 \leftarrow T_1 + T_5 (= X_2 - S)$	(dummy)
2	$T_5 \leftarrow T_3 \times T_3 (= Z_1^2)$	$T_1 \leftarrow T_1 \times T_4 (= U_1)$	10	$T_4 \leftarrow T_4 \times T_5 (= -Y_2 - T)$	$T_3 \leftarrow T_3 \times T_5 (= Z_3)$
	$T_1 \leftarrow T_1 + T_1 (= 2X_1)$	(dummy)		$T_2 \leftarrow T_2 + T_4 (= -Y_2)$	(dummy)
	(dummy)	(dummy)		$T_2 \leftarrow (-T_2) (= Y_2)$	(dummy)
	(dummy)	(dummy)		(dummy)	(dummy)
3	$T_5 \leftarrow T_5 \times T_5 (= Z_1^4)$	$T_4 \leftarrow T_4 \times T_9 (= Z_2^3)$	11		$T_6 \leftarrow T_5 \times T_5 (= W^2)$
	(dummy)	(dummy)			(dummy)
	(dummy)	(dummy)			(dummy)
	(dummy)	(dummy)			(dummy)
4	$T_5 \leftarrow T_0 \times T_5 (= aZ_1^4)$	$T_2 \leftarrow T_2 \times T_4 (= S_1)$	12		$T_1 \leftarrow T_1 \times T_6 (= U_1 W^2)$
	$T_4 \leftarrow T_4 + T_5 (= M)$	(dummy)			(dummy)
	(dummy)	(dummy)			$T_4 \leftarrow (-T_4) (= -R)$
	$T_5 \leftarrow T_2 + T_2 (= 2Y_1)$	(dummy)			(dummy)
5	$T_3 \leftarrow T_3 \times T_5 (= Z_2)$	$T_4 \leftarrow T_3 \times T_3 (= Z_1^2)$	13		$T_5 \leftarrow T_5 \times T_6 (= W^3)$
	(dummy)	(dummy)			$T_6 \leftarrow T_1 + T_2 (= S_1 + U_1 W^2)$
	(dummy)	(dummy)			$T_2 \leftarrow (-T_2) (= -S_1)$
	(dummy)	(dummy)			$T_6 \leftarrow T_2 + T_6 (= U_1 W^2)$
6	$T_2 \leftarrow T_2 \times T_2 (= Y_1^2)$	$T_5 \leftarrow T_4 \times T_7 (= U_2)$	14		$T_1 \leftarrow T_4 \times T_4 (= R_2)$
	$T_2 \leftarrow T_2 + T_2 (= 2Y_1^2)$	(dummy)			$T_1 \leftarrow T_1 + T_5 (= R_2 + W^3)$
	(dummy)	$T_5 \leftarrow (-T_5) (= -U_2)$			$T_6 \leftarrow (-T_6) (= -U_1 W^2)$
	(dummy)	$T_5 \leftarrow T_1 + T_5 (= W)$			$T_1 \leftarrow T_1 + T_6 (= X_3)$
7	$T_5 \leftarrow T_1 \times T_2 (= S)$	$T_4 \leftarrow T_3 \times T_4 (= Z_1^3)$	15		$T_2 \leftarrow T_2 \times T_5 (= S_1 W^3)$
	(dummy)	(dummy)			$T_1 \leftarrow T_1 + T_6 (= X_3)$
	$T_5 \leftarrow (-T_5) (= -S)$	(dummy)			(dummy)
	(dummy)	(dummy)			$T_6 \leftarrow T_1 + T_6 (= X_3 - U_1 W^2)$
8	$T_1 \leftarrow T_4 \times T_4 (= M^2)$	$T_4 \leftarrow T_4 \times T_8 (= S_2)$	16		$T_4 \leftarrow T_4 \times T_6 (= Y_3 + S_1 W^3)$
	$T_1 \leftarrow T_1 + T_5$	(dummy)			$T_2 \leftarrow T_2 + T_4 (= Y_3)$
	$(= M^2 - S)$				
	(dummy)	$T_4 \leftarrow (-T_4) (= -S_2)$			(dummy)
	$T_1 \leftarrow T_1 + T_5 (= X_2)$	$T_4 \leftarrow T_2 + T_4 (= R)$			(dummy)

また，表 7 は，拡大体上の楕円曲線暗号のスカラー倍算における 2 倍算，加算について side channel atomicity に分割する例である．[11]

(拡大体上での加算の計算式)
$P = (X_1, Y_1), Q = (X_2, Y_2), P + Q = (X_3, Y_3)$
$X_3 = a + \lambda^2 + \lambda + X_1 + X_2, Y_3 = (X_1 + X_3)\lambda + X_3 + Y_1, \lambda = (Y_1 + Y_2)/(X_1 + X_2)$
$T_1 \leftarrow X_1, T_2 \leftarrow Y_1, T_3 \leftarrow X_2, T_4 \leftarrow Y_2$

(拡大体上での 2 倍算の計算式)
$P = (X_1, Y_1), 2P = (X_3, Y_3)$
$X_3 = a + \lambda^2 + \lambda, Y_3 = (X_1 + X_3)\lambda + X_3 + Y_1, \lambda = X_1 + (Y_1/X_1)$
$T_1 \leftarrow X_1, T_2 \leftarrow Y_1$

表 7: 拡大体上での 2 倍算，加算の side channel atomicity

No.	2 倍算	加算
1	$T_1 \leftarrow T_1 + T_3 (= X_1 + X_2)$	(dummy)
	$T_2 \leftarrow T_2 + T_4 (= Y_1 + Y_2)$	$T_6 \leftarrow T_3 + T_6 (= X_1)$
	$T_5 \leftarrow T_2/T_1 (= \lambda)$	$T_5 \leftarrow T_2/T_1 (= Y_1/X_1)$
	$T_1 \leftarrow T_1 + T_5$	$T_5 \leftarrow T_1 + T_5 (= \lambda)$
	$T_6 \leftarrow T_5^2 (= \lambda^2)$	$T_1 \leftarrow T_5^2 (= \lambda^2)$
	$T_6 \leftarrow T_6 + a (= \lambda^2 + a)$	$T_1 \leftarrow T_1 + a (= \lambda^2 + a)$
	$T_1 \leftarrow T_1 + T_6 (= X_3)$	$T_1 \leftarrow T_1 + T_6 (= X_3)$
	$T_2 \leftarrow T_1 + T_4 (= X_3 + Y_2)$	$T_2 \leftarrow T_1 + T_2 (= X_3 + Y_1)$
	$T_6 \leftarrow T_1 + T_3 (= X_2 + X_3)$	$T_6 \leftarrow T_1 + T_6 (= X_1 + X_3)$
	$T_5 \leftarrow T_5 \times T_6$	$T_5 \leftarrow T_5 \times T_6$
	$T_2 \leftarrow T_2 + T_5 (= Y_3)$	$T_2 \leftarrow T_2 + T_5 (= Y_3)$

4　スカラー倍算の効率化検討

前述の通り，楕円曲線暗号の効率化は，楕円曲線上の加法演算の効率化及び加法演算を構成する有限体上の四則演算の効率化の２つが必要になる．本研究では，楕円曲線上の加法演算の更なる効率化を目指し，既存手法の中で最速と考えられる DBNS 表現を利用した方法の改良に取り組む．なお，DBNS 表現を利用した既存手法には，Dimitrov らの手法と Meloni らの手法があるが，本研究では，スマートカードなど CPU やメモリ等の処理能力が限定されたデバイスへの適用を考慮するため，大量のプレ計算とメモリを必要とする Meloni らの手法を検討対象外とし，Dimitrov らの手法をベースに検討を行う．なお，加算，2 倍算，3 倍算の計算公式については，Dimitrov らの手法と条件を合わせるため，Dimitrov らの手法 [6] で用いられている以下の計算式を用いる．

・加算（Cohen, Miyaji 等．$Jacobian + Affine \rightarrow Jacobian$）
$(X_1, Y_1, Z_1) + (x_2, y_2, 1) \rightarrow (X_3, Y_3, Z_3)$
$X_3 = -H^3 - 2X_1H^2 + r^2$
$Y_3 = -Y_1H^3 + r(X_1H^2 - X_3)$
$Z_3 = Z_1H$
但し，$U = x_2Z_1^2, S = y_2Z_1^3 - M^2, H = U - X_1, r = S - Y_1$.

・2 倍算（$Jacobian \rightarrow Jacobian$）
$2(X_1, Y_1, Z_1) \rightarrow (X_2, Y_2, Z_2)$
$X_2 = T$
$Y_2 = -8Y_1^4 + M(S - T)$
$Z_2 = 2Y_1Z_1$
但し，$S = 4X_1Y_1^2, M = 3X_1^2 - aZ_1^4, T = -2S + M^2$.

・3 倍算（Dimitrov 等 [7]．$Jacobian \rightarrow Jacobian$）
$3(X_1, Y_1, Z_1) \rightarrow (X_3, Y_3, Z_3)$
$X_3 = 8Y_1^2(T - ME) + X_1E^2$
$Y_3 = Y_1(4(ME - T)(2T - ME) - E^3)$
$Z_3 = Z_1E$
但し，$M = 3X_1^2 + aZ_1^4, E = 12X_1Y_1^2 - M^2, T = 8Y_1^4$.

4.1　既存研究の課題

Dimitrov らの手法における DBNS 表現 (DBchains) は，下式のように変形することができる．

$$k = \sum_{i=1}^{n} s_i 2^{b_i} 3^{t_i}$$

$$k = 2^{b_0}3^{t_0} + 2^{b_1}3^{t_1} + \cdots + 2^{b_{n-1}}3^{t_{n-1}} + 2^{b_n}3^{t_n} = 2^{b_n}3^{t_n}(2^{b_{n-1}-b_n}3^{t_{n-1}-t_n}(\cdots(2^{b_0-b_1}3^{t_0-t_1}+1)+1)\cdots+1)$$

表 8: 加算，2 倍算，3 倍算の計算コスト

	座標系	計算量	計算量 ($S = 0.8M$)	備考
加算	$J \leftarrow J + A$	$8M + 3S$	$10.4M$	cohen, Miyaji 等 [5]
2 倍算	$J \leftarrow 2J$	$4M + 6S$	$8.8M$	$a = -3$ の時は, $4M + 4S$
3 倍算	$J \rightarrow 3J$	$10M + 6S$	$14.8M$	Dimitrov 等 [?]

上式から，DBchains のスカラー倍算の計算量は，2 倍算が b_0 回，3 倍算が t_0 回，加算が n 回であることが分かる．すなわち，2 倍算，3 倍算の回数は，それぞれ DBchains の初項の 2 のべき，3 のべきとなり，加算の回数は (DBchains の項数-1) となる．ここで，2 倍算に比べ 3 倍算の計算コストは高くなるため，3 倍算の回数は極力小さくすることが望ましい．しかしながら，スカラー k から DBchains を導出するための手法として Dimitrov らが提案している Greedy algorithm では，初項 $(2^{b_0}3^{t_0})$ をスカラー k に最も近い値になるという条件のみで設定するため，初項の 3 のべきが大きい値になってしまう可能性もあり，スカラー倍算の計算コストを最適化しているとは言えない．

4.2 研究の方向性

DBchains のスカラー倍算の計算コストを表 8 を使用して求めると，

(2 倍算の計算コスト×回数)+(3 倍算の計算コスト×回数)+(加算の計算コスト×回数)
$= 8.8 \times b_0 + 14.8 \times t_0 + 10.4 \times n[m]$
$= -0.54 \times b_0 + 14.8 \times \log_3 k + 10.4 \times n[m]$　　※ $2^{b_0}3^{t_0} \fallingdotseq k$

となる．上式から，スカラー倍算の計算コストを小さくするには，初項の 2 のべきの値 (b_0) が大きく，かつ，項数 (n) が小さくなるような DBchains を導出すれば良いことが分かる．ここで，DBchains の項数は Greedy algorithm の収束の早さ（ループ処理の回数）に相当するが，初項の 2 のべきの値が非常に大きい場合（初項の 3 のべきの値が小さいことを意味する）や，逆に初項の 2 のべきの値が非常に小さい場合は，DBcahins の制約条件から，2 項目以降の 2 のべき，3 のべきはそれぞれ b_0，t_0 以下の値となるため，選択の幅が狭まり，Greedy Algorithm の収束は遅くなることが想定される．従って，スカラー倍算の計算コストを小さくするには，Greedy algorithm の収束の早さへの影響を最小限にしつつ，初項の 2 のべきの値を極力大きく設定することが良いと考えられる．このような初項の 2 のべきの最適値を求めることが重要であるが，理論的にそのような最適値を算出することは非常に困難であるため，本研究では実験により初項の 2 のべきの最適値を求める．

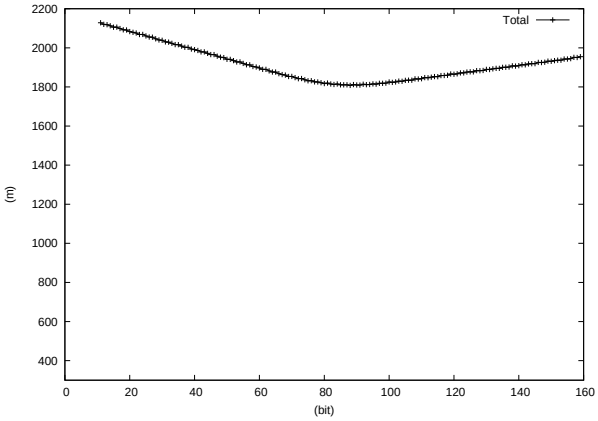

図 2: b_0 毎のスカラー倍算の計算コスト（160 ビットの場合）

4.3 実験

実験では，ランダムに発生した 160 ビットの整数に対し，初項の b_0 を 0 から 160 の範囲で変化させ，第 2 項以降は Greedy algorithm を用いて DBchains を導出し，そのスカラー倍算の計算コストが b_0 によってどのように変化するかを確認した．実験結果を図 2 に示す．図中，横軸は b_0 のビット数，縦軸にはスカラー倍算の計算コストを示し，スカラー倍算の計算コストの単位は，乗算回数 (m) で示す．なお，計算コストは，10,000 個のランダムに発生した 160 ビットの整数に対し，それらのスカラー倍算の計算コストの平均値をとっている．結果として，トータルの計算コストは b0 が 88 の時に最小になることを確認した．

次に，スカラー k のビット数が 32,64,96,128,160,176,192,208,224 の各ビットについて，ランダムに発生した 10,000 個の整数に対してスカラー倍算の平均値を最小にする b_0 を実験により算出した．実験結果を図 3 に示す．図中，横軸はスカラー k のビット数，縦軸はスカラー倍算の計算コストを最小にする b_0 のビット数を示す．結果として，最適な b_0 は，スカラー k のビット数に対して比例関係に近くなること，また，その傾きは最小二乗法により，約 0.55 となることを確認した．

なお，b_1 以降のビット数について，本アルゴリズムを繰り返し適用することで更なる計算コストの削減が可能か実験を実施してみたが，計算コストはむしろ増加する傾向を確認した．
これは，以下の理由によると考えられる．
・b_1 及び t_1 以降のビット数については，それぞれ b_0,t_0 以下の値という条件が付くが，b_0 を前述の最適値で固定することで t_0 も適切な値となり（例えば鍵長が 160 ビットの場合，b_0 は 94 ビット，t_0 は 41 ビット），b_1，t_1 のどちらかが極端に大きい，または極端に小さい値をとる可能性が低くなる．
・b_0，t_0 はそれぞれ 2 倍算，3 倍算の計算コストに影響するため，t_0 の値が大きい値をとった場合，計算コストの割高な 3 倍算の回数が増え，計算コストの増加要因となる．一方，b_1,t_1 以降に関しては，2 倍算，3 倍算の計算コストには影響しない．
・一方で，greedy algorithm と異なり，z の値に対し，最も近い値をとるとは限らない．

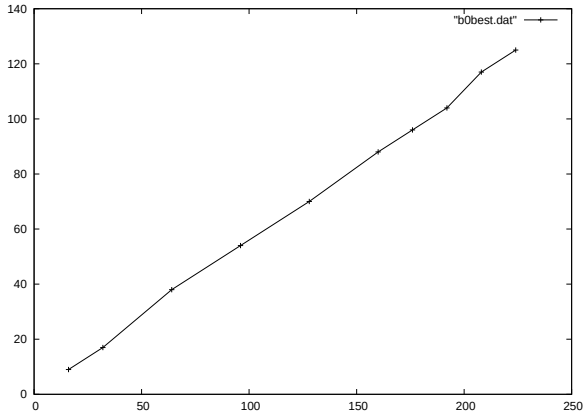

図 3: スカラー k に対する最適な b_0(32-224 ビット)

4.4 提案手法

実験により，スカラー倍算の計算コストを最小化する b_0 を算出することが出来た．本報告では，Greedy algorithm の初項の導出にはこの b_0 を利用し，(例えば，160 ビットの楕円曲線暗号の場合，b_0 は固定的に 88 を設定）第 2 項以降は，従来の Greedy algorithm を用いる Modified Greedy algorithm を提案する．

Algorithm 2. Modified Greedy algorithm for DBchains

Input k, a n-bit positive integer; b_0,t_0
Output The sequence $(s_i, b_i, t_i)_{i\geq 0}$ such that
$k = \sum_{i=1}^{m} s_i 2^{b_i} 3^{t_i}$ with $b_0 \geq \cdots \geq b_m \geq 0$ and $t_0 \geq \cdots \geq t_m \geq 0$
1: $b_{max} \leftarrow b_0, t_{max} \leftarrow t_0$
2: $s \leftarrow 1$
3: $k \leftarrow |k - 2^{b_0} 3^{t_0}|$
4: while $k > 0$ do
5: define $z = 2^b 3^t$, the best approximation of k with $0 \leq b \leq b_{max}$ and $0 \leq t \leq t_{max}$
6: print (s, b, t)
7: $b_{max} \leftarrow b, t_{max} \leftarrow t$
8: if $k < z$ then
9: $s \leftarrow -s$
10:$k \leftarrow |k - z|$

4.5 比較評価

先に求めたスカラー倍算の平均値を最小にする b0 を使用し，DBNS 表現の第 2 項以降は Greedy algorithm により導出した DBNS 表現と，b0 を含めすべて Greedy algorithm によって導出した DBNS 表現について，スカラー倍算の計算コストの比較を実施した．なお，スカラー倍算の計算コストは，スカラー k のビット数 (32,64,96,128,160,176,192,208,224）に対し，それぞれ 10,000 個のランダムに発生させた整数の平均値を使用する．結果として，鍵長がいずれのビット数でも既存研究に比べ，本研究の手法が高速となること，また，ビット数が上がるほど，その効果が高まることを確認した．

表 9: 既存研究との計算コスト比較評価

スカラー k (bit)	スカラー倍算の計算コスト [m]				削減率 対 DBNS (Dimitrov)
	バイナリ法	NAF	DBNS (Dimitrov)	DBNS (本研究)	
32	448	393	366	360	1.64
64	896	785	755	724	4.11
96	1344	1178	1128	1085	3.81
128	1792	1570	1509	1447	4.11
160	2240	1963	1908	1808	5.24
176	2492	2183	2106	1989	5.56
192	2688	2355	2308	2170	5.98
208	2912	2551	2509	2350	6.34
224	3136	2748	2712	2530	6.71

5 DBchains導出アルゴリズムの効率化検討

Greedy Algorithm では，整数 z に対する $|z - 2^b 3^t|$ の値を最小にする b, t の値を総当たり方式で求めるため，時間を要する．この時間を短縮するための手法について，検討した．

前節にて，スカラー倍算の計算量を最小化する DBchians を導出する手法である Modified Greedy algorithm を提案した．ここで，本手法により DBchians を導出する処理自体が時間を要するものであるため，本節では，本処理を効率化することについて検討する．

5.1 既存手法

Greedy Algorithm を単純に計算する方法としては，各ループ処理において，$b_{i-1} \geq b_i \geq 0, t_{i-1} \geq t_i \geq 0$ を満たす全ての b_i, t_i について $2^{b_i}3^{t_i}$ を計算し，k に最も近いものを選択する方法が考えられる．

この手法では $(\log k)^2$ の計算量が必要となるが，Yu ら [21] は，これに対して，$b_{i-1} \geq b_i \geq 0, t_{i-1} \geq t_i \geq 0$ を満たす b_i, t_i のうち，$2^{b_i}3^{t_i} = k$ を満たす直線の整数 b_i, t_i に対し $2^{b_i}3^{t_i}$ を計算し，k に最も近いものを選択する Line Algorithm を提案している．

表 10: Greedy Algorithm の計算方法に関する既存研究

	全探索	Line Algorithm (Yu ら)
概要	$b_{i-1} \geq b_i \geq 0$, $t_{i-1} \geq t_i \geq 0$ を満たす全ての b_i, t_i について $2^{b_i}3^{t_i}$ を計算し， k に最も近いものを選択する	$b_{i-1} \geq b_i \geq 0$, $t_{i-1} \geq t_i \geq 0$ を満たす b_i, t_i の内， $2^{b_i}3^{t_i} = k$ の直線の 前後の $2^{b_i}3^{t_i}$ を計算し，k に最も近いもの を選択する
計算量	$(\log k)^2$	$\log k$

なお，Imbert ら [20] による Ostrowski 's number を利用した計算量 $\log(\log k)$ の手法もあるが，一般的な DBNS 表現の導出手法であり，DBchains の条件である $b_{i-1} \geq b_i$,$t_{i-1} \geq t_i$ は考慮していないため，本研究では比較対象から除外する．

5.2 提案手法

本研究では，DBchains の条件である $b_{i-1} \geq b_i$,$t_{i-1} \geq t_i$ を満たしつつ，Imbert らと同等の計算量 $\log(\log k)$ を実現する手法を提案する．$2^{b_i}3^{t_i}$ が k にもっとも近いものを求めたいことから，$2^{b_i}3^{t_i} = k$ の両辺の対数を

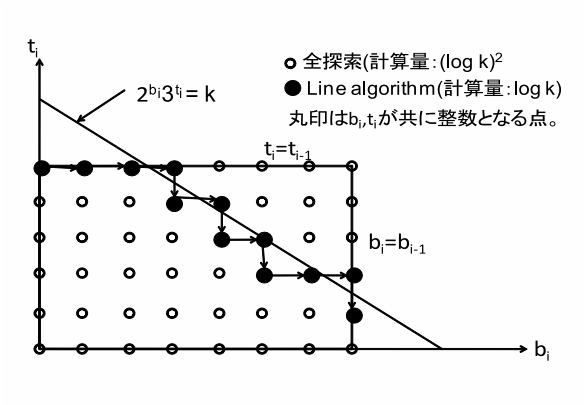

図 4: Greedy Algorithm の計算方法の既存研究

とって変形する.

$$
\begin{aligned}
& 2^{b_i}3^{t_i} = k \\
\rightarrow\ & b_i + t_i \log_2 3 = \log_2 k \\
\rightarrow\ & b_i + \lfloor t_i \log_2 3 \rfloor + (t_i \log_2 3 - \lfloor t_i \log_2 3 \rfloor) \\
=\ & \lfloor \log_2 k \rfloor + (\log_2 k - \lfloor \log_2 k \rfloor)
\end{aligned}
$$

提案手法では，上式を利用して k にもっとも近い $2^{b_i}3^{t_i}$ を求める.

[提案手法]

(1)t_i の範囲の決定

左辺と右辺で整数部を等しくする $b_i(b_{i-1} \geq b_i)$ を選択可能な t_i の範囲を求める.

(2)t_i の決定

(1) で求めた範囲の t_i から $t_i \log_2 3$ の小数部が $\log_2 k$ の小数部と最も近い t_i を選択.

(3)b_i の決定

(2) で求めた t_i に対し，左辺と右辺で整数部を等しくする $b_i(b_{i-1} \geq b_i)$ を求める.

以下，(1)-(3) の処理について，詳細を示す．

(1)t_i の範囲の決定

左辺と右辺で整数部を等しくする $b_i(b_{i-1} \geq b_i)$ を選択可能な t_i の範囲を求める．

$$b_i + \lfloor t_i \log_2 3 \rfloor = \lfloor \log_2 k \rfloor \tag{13}$$

$$b_{i-1} \geq b_i \geq 0 \tag{14}$$

$$t_{i-1} \geq t_i \geq 0 \tag{15}$$

式 (13) を式 (14) に代入して，
$b_{i-1} \geqq \lfloor \log_2 k \rfloor - \lfloor t_i \log_2 3 \rfloor \geqq 0$
$\lfloor \log_2 k \rfloor \geqq \lfloor t_i \log_2 3 \rfloor \geqq \lfloor \log_2 k \rfloor - b_{i-1}$
$\lfloor \log_2 k \rfloor + 1 \geqq t_i \log_2 3 \geqq \lfloor \log_2 k \rfloor - b_{i-1}$
$(\lfloor \log_2 k \rfloor + 1)/\log_2 3 \geqq t_i \geqq (\lfloor \log_2 k \rfloor - b_{i-1})/\log_2 3$

従って，求める t_i の範囲は，式 (15) と併せて，
$\mathrm{MIN}[t_{i-1}, \lfloor \log_2 k \rfloor + 1/\log_2 3] \geqq t_i$
$\geqq \mathrm{MAX}[0, (\lfloor \log_2 k \rfloor - b_{i-1})/\log_2 3]$
となる．ここで，計算量は，k の大きさに依存しないので，定数時間である．

(2) t_i の決定

(1) で求めた範囲の t_i の内，$t_i \log_2 3$ の小数部が $\log_2 k$ の小数部と最も近い t_i を選択する．選択方法は，$t_i \log_2 3$ をあらかじめ計算して小さい順に並べておき，二分探索法でその中から $\log_2 k$ の小数部が最も近い値を選択する．計算量は $\log k$ の要素から，二分探索法で探索を行うため，その計算量は $\log(\log k)$ である．

(3) b_i の決定

(2) で求めた t_i に対し，整数部を等しくする $b_i(b_{i-1} \geq b_i \geqq 0)$ を求める．
$b_i + \lfloor t_i \log_2 3 \rfloor = \lfloor \log_2 k \rfloor$
より，
$b_i = \lfloor \log_2 k \rfloor - \lfloor t_i \log_2 3 \rfloor$

本処理の計算量は，k の大きさに依存しないため定数時間となる．

5.3 比較評価

前節から提案手法のの各処理の計算量はスカラー k のビット数にのみ依存し，$\log(\log k)$ となり，既存研究における全探索（計算量：$(\log k)^2$），Line Algorithm（計算量：$\log k$）に比べ，少ない計算量を実現出来る．

6　結論

DBNS 表現の導出に関して，Dimitrov らの Greedy algorithm に対し，DBNS 表現の初項をスカラー k のビット数に応じて固定的に設定することで，スカラー倍算の計算コストを削減可能であることを示した．DBNS 表現の初項における 2 のべき乗の最適値は，スカラー k のビット数に対してほぼ比例関係に近くなり，スカラー k のビット数に対し，0.55 を乗じた値となることを示した．また，DBNS 表現の導出の主要処理となる Greedy Algorithm の実装方法として，既存研究（計算量：$\log k$）に比べ計算量の小さい方式（計算量：$\log(\log k)$）を提案した．

本論文で提案したスカラー倍算の効率化手法，並びに DBNS 導出処理の効率化手法を用いることによって，高速に楕円曲線暗号の暗号化及び復号を行うことが可能となる．

7　謝辞

本論文を執筆するにあたり数々の貴重な御指導，御助言を賜りました本学情報科学研究科，宮地充子教授に深く感謝致します．本論文の執筆にあたり様々な御指導，御助言を頂きました本学情報科学研究科，平石邦彦教授，丹康雄教授，及び面和成准教授に深く御礼申しあげます．また，ゼミやサマースクールにて様々な御意見，御助言を頂きました宮地研究室の皆様に感謝致します

参考文献

[1] 情報セキュリティ(アイテック)

[2] 現代暗号（岡本龍明他）

[3] 最新暗号技術 (NTT 情報流通プラットフォーム研究所)

[4] A Booth,A Signed Binary Multiplication Technique (1951)

[5] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In K. Ohta and D. Pei, editors, Advances in Cryptology – ASIACRYPT ’98, volume 1514 of Lecture Notes in Computer Science, pages 51–65. Springer-Verlag, 1998.

[6] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. An algorithm for modular exponentiation. Information Processing Letters, 66(3):155–159, May 1998.

[7] Dimitrov, V.S., Imbert, L., Mishra, P.K.: Efficient and secure elliptic curve point multiplication using double-base chains. In: Advances in Cryptology – Asiacrypt 2005. Volume 3788 of Lecture Notes in Comput. Sci., Springer (2005) 59–78

[8] VASSIL DIMITROV, LAURENT IMBERT, AND PRADEEP K. MISHRA:THE DOUBLE-BASE NUMBER SYSTEM AND ITS APPLICATION TO ELLIPTIC CURVE CRYPTOGRAPHY. MATHEMATICS OF COMPUTATION Volume 77, Number 262, April 2008, Pages 1075–1104

[9] VASSIL S. DIMITROV AND EVERETT W. HOWE:LOWER BOUNDS ON THE LENGTHS OF DOUBLE-BASE REPRESENTATIONS. PROCEEDINGS OF THE AMERICAN MATHEMATICAL SOCIETY Volume 139, Number 10, October 2011, Pages 3423–3430

[10] Meloni, N., Hasan, M.A.: Elliptic Curve Scalar Multiplication Combining Yao ’s Algorithm and Double Bases. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 304–316. Springer, Heidelberg (2009)

[11] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye:Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. IEEE TRANSACTIONS ON COMPUTERS, VOL. 53, NO. 6, JUNE 2004.

[12] Christophe Giraud and Vincent Verneuil:Atomicity Improvement for Elliptic Curve Scalar Multiplication. CARDIS 2010, LNCS 6035, pp. 80–101, 2010.

[13] Sung-Ming Yen and Marc Joye:Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. IEEE TRANSACTIONS ON COMPUTERS, VOL. 49, NO. 9, SEPTEMBER

[14] Guillaume Gilbert and J. M. Pierre Langlois:Multipath Greedy Algorithm for Canonical Representation of Numbers in the Double Base Number System. 2005 IEEE

[15] Patrick Longa and Catherine Gebotys:Efficient Techniques for High-Speed Elliptic Curve Cryptography. CHES 2010, LNCS 6225, pp. 80–94, 2010.

[16] Erik Woodward Knudsen:Elliptic Scalar Multiplication Using Point Halving. ASIACRYPT '99, LNCS 1716, pp. 135–149, 1999.

[17] K.W. Wong, Edward C.W. Lee, L.M. Cheng1, and Xiaofeng Liao:Fast Elliptic Scalar Multiplication using New Double-base Chain and Point Halving.

[18] Herbert S. Wilf:generatingfunctionology

[19] Jithra Adikari, Student Member, IEEE, Vassil S. Dimitrov, and Laurent Imbert: Hybrid Binary-Ternary Number System for Elliptic Curve Cryptosystems. IEEE TRANSACTIONS ON COMPUTERS, VOL. 60, NO. 2, FEBRUARY 2011 Herbert S. Wilf:generatingfunctionology

[20] Valérie Berthéa, and Laurent Imbert: On Converting Numbers to the Double-Base Number System.

[21] W. Yu, K. Wang and B. Li, Fast Algorithm Converting integer to Double Base Chain. Information Security and Cryptology, Inscrypt 2010 short papers. pp.44–54, 2011.

[22] 本間尚文，青木孝文，佐藤証:暗号モジュールへのサイドチャネル攻撃とその安全性評価の動向電子情報通信学会論文誌　A　Vol.J93-A No.2 pp.42–51

[23] 宮地充子: 代数学から学ぶ暗号理論（日本評論社）

[24] 小山謙二，宮地充子他: 楕円暗号の数理，電子情報通信学会論文誌 A Vol.J82-A No.8 pp.1212-1222 1999 年 8 月

A　外部投稿（2012.11月 ISEC)

A.1　はじめに

楕円曲線暗号は，小さい鍵長で RSA 暗号と同等の安全性を確保できるため，処理能力の小さいスマートカードなどの小型の組み込みデバイスを中心に普及が期待されている．楕円曲線暗号の更なる普及のためには，その暗号化処理の高速化が重要となり，活発に研究が行われている．

楕円曲線暗号の暗号化処理は，主にスカラー倍算と呼ばれる処理，すなわち，楕円曲線上のベースポイント P に対して，$kP = P + \cdots + P(k$ 回) を計算する処理により構成される．また，スカラー倍算は，楕円曲線上における演算（加算，2 倍算等），有限体上の四則演算から構成され，これらの演算数を減らすことが必要になる．加算，2 倍算等を構成する有限体上の四則演算を減らす手法としては，座標系を変換 [1] したり，同一演算の再利用等の手法が提案されている．一方，前者の楕円曲線上における演算（加算，2 倍算等）数を減らすには，non-zero digit の密度を減らすことが重要になり，バイナリ法，NAF，w-NAF などの手法が提案されている．

DBNS(Double-Base Number System) は，スカラー k を 2 つの整数（2 及び 3) のべき乗の和で表す表現手法で，non-zero digit の密度をバイナリ法，NAF，w-NAF に比べ，大幅に削減可能な特徴がある．例えば，160 ビットのスカラー k に対し，バイナリ法，NAF がそれぞれ平均で 80 個，53 個の non-zero digit が出現するの対し，DBNS では 22 個の non-zero digit で表現可能である [3]．本研究では，DBNS を利用した既存研究に対し，DBNS 表現の導出方法に課題があることを示し，その解決手法の提案を行う．

本報告では，2 章に本研究の背景となる楕円曲線暗号のスカラー倍算について，3 章で DBNS を利用した既存研究とその課題，4 章，5 章に提案手法，6 章に既存研究との比較評価，7 章に結論，8 章に今後の予定を記述する．

A.2　研究の背景

A.2.1　楕円曲線暗号

標数 5 以上の素体 F_p 上の Weierstrass 標準形の楕円曲線は，下式により定義される．

$$E\colon y^2 = x^3 + ax + b \tag{16}$$

ここで，$a, b \in F_p, 4a^3 + 27b^2 \neq 0$ である．楕円曲線上における加算 $(P + Q)$，2 倍算 $(2P)$ の定義は，[19] を参照されたい．

また，楕円曲線上の加算，2 倍算，3 倍算等は，有限体（素体）上の四則演算で構成され，座標系の工夫や同一の四則演算の再利用等によって，四則演算数を減らす研究が行われている．

以下に既存研究における加算，2 倍算，3 倍算の計算公式と計算量を示す．詳細は [19] を参照されたい．

・加算（Cohen, Miyaji 等．$Jacobian + Affine \rightarrow Jacobian$)
$(X_1, Y_1, Z_1) + (x_2, y_2, 1) \rightarrow (X_3, Y_3, Z_3)$
$X_3 = -H^3 - 2X_1H^2 + r^2$

$Y_3 = -Y_1H^3 + r(X_1H^2 - X_3)$
$Z_3 = Z_1H$
但し，$U = x_2Z_1^2, S = y_2Z_1^3 - M^2, H = U - X_1, r = S - Y_1$.

・2 倍算（$Jacobian \rightarrow Jacobian$）
$2(X_1, Y_1, Z_1) \rightarrow (X_2, Y_2, Z_2)$
$X_2 = T$
$Y_2 = -8Y_1^4 + M(S - T)$
$Z_2 = 2Y_1Z_1$
但し，$S = 4X_1Y_1^2, M = 3X_1^2 - aZ_1^4, T = -2S + M^2$.

・3 倍算（Dimitrov 等 [3]．$Jacobian \rightarrow Jacobian$）
$3(X_1, Y_1, Z_1) \rightarrow (X_3, Y_3, Z_3)$
$X_3 = 8Y_1^2(T - ME) + X_1E^2$
$Y_3 = Y_1(4(ME - T)(2T - ME) - E^3)$
$Z_3 = Z_1E$
但し，$M = 3X_1^2 + aZ_1^4, E = 12X_1Y_1^2 - M^2, T = 8Y_1^4$.

表 11: 加算，2 倍算，3 倍算の計算コスト

	座標系	計算量	計算量 ($S = 0.8M$)	備考
加算	$J \leftarrow J + A$	$8M + 3S$	$10.4M$	cohen, Miyaji 等 [1]
2 倍算	$J \leftarrow 2J$	$4M + 6S$	$8.8M$	$a = -3$ の時は, $4M + 4S$
3 倍算	$J \rightarrow 3J$	$10M + 6S$	$14.8M$	Dimitrov 等 [?]

A.3 既存研究とその課題

A.3.1 Double-Base Number System

DBNS（Double-Base Number System）は，正の整数 k の表現形式の 1 つであり，整数 k を下式のように 2 及び 3 のべき乗の和で表すものである．

$$\sum_{i=0}^{m} s_i 2^{b_i} 3^{t_i} s_i \in \{-1, 1\} \quad b_i, t_i \geqq 0 \tag{17}$$

DBNS は，1 つの正の整数が多くの別表現をもつ表現形式である．例えば，10 は 5 個の DBNS 表現を，また，100 は 402 個，1,000 は 1,295,579 個の DBNS 表現をもつ．一般には正の整数 k の表現形式の個数 n に関して，最大で以下の個数になることが Dimitrov らによって示されている [2].

$$O\left(\frac{\log k}{\log\log k}\right) \tag{18}$$

DBNS 表現を使用すると，バイナリ表現に比べ，加算の数が減らせ，これによってスカラー倍算における加算の数が減少するメリットがある．ランダムに選択された 160 ビットの整数に対するスカラー倍算の加算数の平均値は，バイナリ表現の場合で 80，NAF で 53，DBNS で 22 である．具体な例をとると，例えばスカラー k が 127 の時，バイナリ表現の場合，

$$\begin{aligned} 127 &= 64+32+16+8+4+2+1 \\ &= 2^6+2^5+2^4+2^3+2^2+2^1+2^0\text{(加算 6 回)} \end{aligned}$$

また，DBNS 表現の場合，

$$\begin{aligned} 127 &= 96+27+4 \\ &= 2^5 3^1+2^0 3^3+2^2 3^0\text{(加算 2 回)} \end{aligned}$$

となり，実際に DBNS 表現の方が加算の数が少ないことが確認できる．

一方で，バイナリ表現と異なり，2 及び 3 のべき乗が降べき順とは限らず，降べき順でない場合，スカラー倍算における 2 倍算及び 3 倍算が増えるという課題がある．例えば，先の整数 127 の場合，バイナリ表現では，

$$\begin{aligned} 127 &= 64+32+16+8+4+2+1 \\ &= 2^1(2^1(2^1(2^1(2^1(2^1+2^0)+2^0)+2^0)+2^0)+2^0)+2^0) \end{aligned}$$

で，2 倍算が 6 回（加算は 6 回）となるが，DBNS 表現では，

$$\begin{aligned} 127 &= 96+27+4 = 2^5 3^1+2^0 3^3+2^2 3^0 \\ &= 2^2 3^0(2^3 3^1+2^0 3^0)+2^0 3^2 \end{aligned}$$

であり，2 倍算，3 倍算がそれぞれ 5 回，3 回（加算は 2 回）となり，2 倍算，3 倍算だけで $2^5 3^3 = 864 \gg 127$ となり，明らかに不要な計算をすることになり，非効率である．

A.3.2 DBNS に関わる既存研究

前節で述べたスカラー倍算に置いて DBNS を使用する際の課題を解決する手法が Dimitrov らによって提案された DBchains である [3].

DBChians は DBNS 表現の 1 種で，下記のように DBNS 表現を 2 及び 3 の降べき順のものに限定したものである．

$$k = \sum_{i=0}^{n} s_i 2^{b_i} 3^{t_i}$$
$$s_i \in \{-1, 1\} b_0 \geq b_1 \geq \cdots \geq b_n \geq 0, t_0 \geq t_1 \geq \cdots \geq t_n \geq 0$$

DBChains では，下式のように b_i, t_i の値が小さいときの計算結果を，b_i, t_i の値が大きい時の計算に利用出来るため，スカラー倍算の 2 倍算及び 3 倍算が増える問題を回避可能である．

$$\begin{aligned} k &= \sum_{i=0}^{n} s_i 2^{b_i} 3^{t_i} \\ &= 2^{b_0} 3^{t_0} (2^{b_1 - b_0} 3^{t_1 - t_0} (2^{b_2 - b_1} 3^{t_2 - t_1} + 1) \cdots\cdots + 1) \end{aligned}$$

例えば，k=127 の場合，前節のような DBchains でない DBNS 表現の場合，

$$\begin{aligned} 127 &= 96 + 27 + 4 = 2^5 3^1 + 2^0 3^3 + 2^2 3^0 \\ &= 2^2 3^0 (2^3 3^1 + 1) + 2^0 3^2 \end{aligned}$$

となり，2 倍算，3 倍算がそれぞれ 5 回と 3 回であったが，DBchains となる DBNS 表現を使用すると，

$$\begin{aligned} 127 &= 108 + 18 + 1 = 2^2 3^3 + 2^1 3^2 + 2^0 3^0 \\ &= 2^0 3^0 (2^1 3^2 (2^1 3^1 + 1) + 1) \end{aligned}$$

となり，2 倍算，3 倍算はそれぞれ 2 回と 3 回に減ることが確認できる．また，$2^2 3^3 = 108 < 127$ であり，不要な 2 倍算，3 倍算は行われていない．

一方，DBChains では，2 及び 3 のべき乗の値が降順になるような DBNS 表現に限定するため，選択可能な DBNS 表現の数が大幅に削減される．このため，必ずしも加算の数を最小化するような DBNS 表現の選択が出来るとは限らない．この制約を緩和する手法として Yao's algorithm を利用した手法を Meloni らが提案している [6]．Meloni らの手法は，DBNS 表現を 3 のべき乗が同一の項をまとめ，以下のように変形する．

$$k = d(0) + 3d(1) + 3^2 d(2) + \cdots + 3^{max(t_i)} d(max(t_i))$$
$$= (3(\cdots 3(3d(max(t_i)) + d(max(t_i) - 1)) + \cdots + d(1)) + d(0)$$
但し，$d(j) = \sum_{t_i = j} 2^i$

ここで，$2^0, 2^1, 2^2, \cdots\cdots, 2^{max(b_i)}$ を事前に計算しておくことで，スカラー倍算の計算コストは，2 倍算が $max(b_i)$， 3 倍算が $max(t_i)$ となり，DBchains と同一の計算コストとなる．一方，加算の計算コストは DBNS の項数 -1 となり DBchains と同一に見えるが，DBchains に比べ，選択可能な DBNS 表現の数が増えるため，DBNS の項数もより小さいものを選択可能であり，DBchains よりも加算の計算コストが下がる可能性が高い手法である．

しかしながら，多くの 2 のべき乗の事前計算が必要になるため，メモリ容量が小さいデバイスでは利用が困難である．そこで，本研究では，Dimitrov らの DBchains の手法をベースに検討を行うこととする．

表 12: DBNS の既存研究

	Dimitrov ら (2005 年) [3]	Meloni ら (2009 年) [6]
DBNS 表現	2 及び 3 の降べき順 ($b_0 \geq b_1 \geq \cdots \geq b_m \geq 0$, $t_0 \geq t_1 \geq \cdots \geq t_m \geq 0$)	2 と 3 のべき乗の最大値の積がスカラー K 以内 ($k \geq 2^{max(b_i)}3^{max(t_i)}$)
DBNS 表現導出方法	Greedy algorithm	n/a
スカラー倍算の計算方法	Left to Right	3 のべきが同一の項をまとめて Left to Right
計算コスト (2 倍算)	2 のべきの最大値 （=初項の 2 のべき b0）	2 のべきの最大値
計算コスト (3 倍算)	3 のべきの最大値 （=初項の 3 のべき t0）	3 のべきの最大値
計算コスト (加算)	DBNS 表現の項数-1 ※ Meloni らの手法に比べ，DBNS 表現の項数は増加傾向	DBNS 表現の項数-1 ※ Dimitrov らの手法に比べ制限が緩やかで，DBNS 表現の項数は減少傾向
プレ計算	不要	2 のべきの最大値までの全ての 2 のべき乗のプレ計算が必要. ($2^0P, 2^1P, \cdots, 2^{max(b_i)}$)

A.3.3 既存研究の課題

Dimitrov らの手法である DBchains では，スカラー k から DBNS 表現を導出するために Greedy algorithm を用いている．Greedy algorithm は，スカラー k に対して，$|k - 2^b3^t|$ を最小にするような b, t を求め，そのような b, t に対して，$|k - 2^b3^t| \quad \rightarrow k$ と置いて，同様に $|k - 2^b3^t|$ を最小にするような b, t を求める処理を，k の値が 0 になるまで繰り返すというものである．

Algorithm 1. Greedy algorithm for DBchains

Input k, a n-bit positive integer; $b_{max}, t_{max} > 0$, the largest allowed binary and ternary exponents
Output The sequence $(s_i, b_i, t_i)_{i \geq 0}$ such that $k = \sum_{i=1}^{m} s_i 2^{b_i} 3^{t_i}$, with $b_0 \geq \cdots \geq b_m \geq 0$ and $t_0 \geq \cdots \geq t_m \geq 0$
1: $s \leftarrow 0$
2: while $k > 0$ do

3: define $z = 2^b 3^t$, the best approximation of k with $0 \leq b \leq b_{max} and 0 \leq t \leq t_{max}$
4: print (s, b, t)
5: $b_{max} \leftarrow b, t_{max} \leftarrow t$
6: if $k < z$ then
7: $s \leftarrow -s$
8: $k \leftarrow |k - z|$

ここで DBchains の条件を満たす DBNS 表現は，下式のように変形することができる．

$$k = 2^{b_0}3^{t_0} + 2^{b_1}3^{t_1} + \cdots + 2^{b_{m-1}}3^{t_{m-1}} + 2^{b_m}3^{t_m} = 2^{b_m}3^{t_m}(2^{b_{m-1}-b_m}3^{t_{m-1}-t_m}(\cdots(2^{b_0-b_1}3^{t_0-t_1}+1)+1)\cdots+1)$$

上式から，DBchains のスカラー倍算の計算量は，2 倍算が b_0 回，3 倍算が t_0 回，加算が m 回であることが分かる．すなわち，2 倍算，3 倍算の回数は，それぞれ DBchains の初項の 2 のべき，3 のべきとなる．

一方で，2 倍算に比べ 3 倍算の計算コストは高く (表 11 参照) なるため，3 倍算の回数は極力小さくすることが望ましい．しかしながら，Greedy algorithm では，初項 $(2^{b_0}3^{t_0})$ をスカラー k に最も近い値になるという条件のみで設定するため，初項の 3 のべきは小さい値になるとは限らず，初項の算出方法に問題があると考えられる．

A.4 提案手法

DBchains におけるスカラー倍算の計算量は， 前節の議論及び表 11 から，

$$\begin{aligned} &8.8[m] \times (2\text{ 倍算の回数}) + 14.8[m] \times (3\text{ 倍算の回数}) \\ &+10.4[m] \times (\text{加算の回数}) \\ &= 8.8 \times b_0[m] + 14.8 \times t_0 + 10.4 \times [m] \end{aligned} \tag{19}$$

となる．ここで，$2^{b_0}3^{t_0} \fallingdotseq k$ から，t_0 を b_0 とスカラー k で表すと，式 19 は，

$$\begin{aligned} &8.8 \times b_0 + 14.8 \times (\log_3 k - b_0 \log_3 2) + 10.4 \times m[m] \\ &= -0.54 \times b_0 + 10.4 \times m + 14.8 \times \log_3 k[m] \end{aligned} \tag{20}$$

となる．

式 20 の第 3 項は定数なので，スカラー倍算の計算量を減らすには第 1 項と第 2 項の和を小さくする必要がある．まず，第 1 項は，明らかに b_0 が大きくなるほど小さくなる．第 2 項は，DBchains の項数である m が小さくなるほど小さくなる．ここで，b_0 と m の関係を考えると，b_0 がかなり小さい場合，またはかなり大きい場合（すなわち，t_0 がかなり小さい場合）に Greedy algorithm の収束が遅くなり，m の値は大きくなる方向と考えられる．

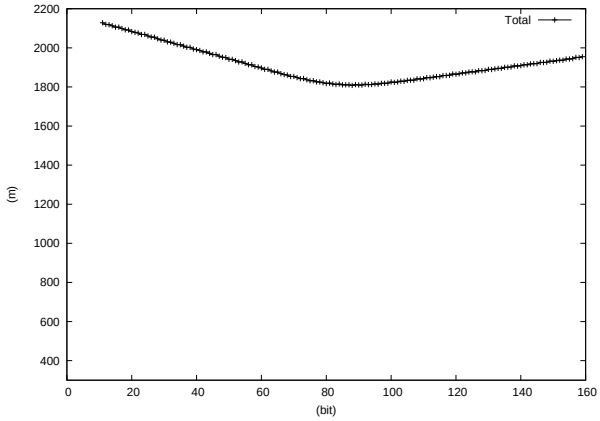

図 5: b_0 毎のスカラー倍算の計算コスト（160 ビットの場合）

従って，上式を b_0 の関数と見ると，スカラー倍算の計算コストを最小にするには，b_0 が大きくなり過ぎない範囲で，大きい値を設定するのが良いと考えられる．そこで，以降では最適な b_0 の選定方法について考える．

A.4.1 スカラー倍算の計算コストを最小化する b_0 について

DBchains の項数を理論的に算出することは非常に困難であるため，ここでは，最適な b_0 を実験により求めることを考える．

実験では，ランダムに発生した 160 ビットの整数に対し，初項の b_0 を 0 から 160 の範囲で変化させ，第 2 項以降は Greedy algorithm を用いて DBchains を導出し，そのスカラー倍算の計算コストが b_0 によってどのように変化するかを確認した．実験結果を図 5 に示す．図中，横軸は b_0 のビット数，縦軸にはスカラー倍算の計算コストを示し，スカラー倍算の計算コストの単位は，乗算回数 (m) で示す．なお，計算コストは，10,000 個のランダムに発生した 160 ビットの整数に対し，それらのスカラー倍算の計算コストの平均値をとっている．結果として，トータルの計算コストは b0 が 88 の時に最小になることを確認した．

次に，スカラー k のビット数が 32,64,96,128,160,176,192,208,224 の各ビットについて，ランダムに発生した 10,000 個の整数に対してスカラー倍算の平均値を最小にする b_0 を実験により算出した．実験結果を図 6 に示す．図中，横軸はスカラー k のビット数，縦軸はスカラー倍算の計算コストを最小にする b_0 のビット数を示す．結果として，最適な b_0 は，スカラー k のビット数に対して比例関係に近くなること，また，その傾きは最小二乗法により，約 0.55 となることを確認した．

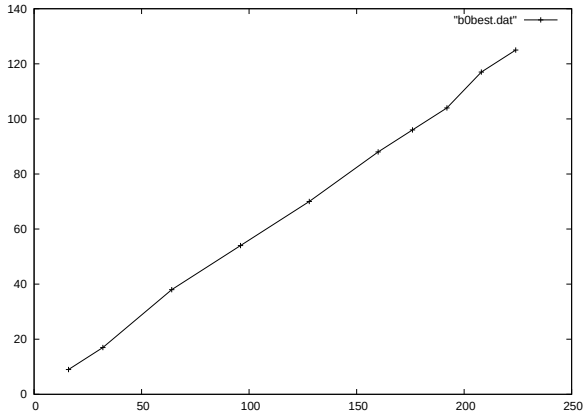

図 6: スカラー k に対する最適な b_0(32-224 ビット)

A.4.2 提案手法について

実験により，スカラー倍算の計算コストを最小化する b_0 を算出することが出来た．本報告では，Greedy algorithm の初項の導出にはこの b_0 を利用し，(例えば，160 ビットの楕円曲線暗号の場合，b_0 は固定的に 88 を設定）第 2 項以降は，従来の Greedy algorithm を用いる Modified Greedy algorithm を提案する．

Algorithm 2. Modified Greedy algorithm for DBchains

Input k, a n-bit positive integer; b_0,t_0
Output The sequence $(s_i, b_i, t_i)_{i\geq 0}$ such that $k = \sum_{i=1}^{m} s_i 2^{b_i} 3^{t_i}$ with $b_0 \geq \cdots \geq b_m \geq 0$ and $t_0 \geq \cdots \geq t_m \geq 0$
1: $b_{max} \leftarrow b_0, t_{max} \leftarrow t_0$
2: $s \leftarrow 1$
3: $k \leftarrow |k - 2^{b_0} 3^{t_0}|$
4: while $k > 0$ do
5: define $z = 2^b 3^t$, the best approximation of k with $0 \leq b \leq b_{max}$ and $0 \leq t \leq t_{max}$
6: print (s, b, t)
7: $b_{max} \leftarrow b, t_{max} \leftarrow t$
8: if $k < z$ then
9: $s \leftarrow -s$
10:$k \leftarrow |k - z|$

A.5 Greedy Algorithm の具体的な計算方法について

前節にて，スカラー倍算の計算量を最小化する DBchians を導出する手法である Modified Greedy algorithm を提案した．ここで，本手法により DBchians を導出する処理自体が時間を要するものであるため，本節では，本処理を効率化することについて検討する．

Greedy Algorithm を単純に計算する方法としては，各ループ処理において，$b_{i-1} \geq b_i \geq 0, t_{i-1} \geq t_i \geq 0$ を満たす全ての b_i, t_i について $2^{b_i}3^{t_i}$ を計算し，k に最も近いものを選択する方法が考えられる．

この手法では $(\log k)^2$ の計算量が必要となるが，Yu ら [17] は，これに対して，$b_{i-1} \geq b_i \geq 0, t_{i-1} \geq t_i \geq 0$ を満たす b_i, t_i のうち，$2^{b_i}3^{t_i} = k$ を満たす直線の整数 b_i, t_i に対し $2^{b_i}3^{t_i}$ を計算し，k に最も近いものを選択する Line Algorithm を提案している．

表 13: Greedy Algorithm の計算方法に関する既存研究

	全探索	Line Algorithm (Yu ら)
概要	$b_{i-1} \geq b_i \geq 0$, $t_{i-1} \geq t_i \geq 0$ を満たす全ての b_i, t_i について $2^{b_i}3^{t_i}$ を計算し， k に最も近いものを選択する	$b_{i-1} \geq b_i \geq 0$, $t_{i-1} \geq t_i \geq 0$ を満たす b_i, t_i の内， $2^{b_i}3^{t_i} = k$ の直線の 前後の $2^{b_i}3^{t_i}$ を計算し，k に最も近いもの を選択する
計算量	$(\log k)^2$	$\log k$

なお，Imbert ら [16] による Ostrowski ’s number を利用した計算量 $\log(\log k)$ の手法もあるが，一般的な DBNS 表現の導出手法であり，DBchains の条件である $b_{i-1} \geq b_i$,$t_{i-1} \geq t_i$ は考慮していないため，本研究では比較対象から除外する．

本研究では，DBchains の条件である $b_{i-1} \geq b_i$,$t_{i-1} \geq t_i$ を満たしつつ，Imbert らと同等の計算量 $\log(\log k)$ を実現する手法を提案する．$2^{b_i}3^{t_i}$ が k にもっとも近いものを求めたいことから，$2^{b_i}3^{t_i} = k$ の両辺の対数をとって変形する．

$2^{b_i}3^{t_i} = k$
$\rightarrow b_i + t_i \log_2 3 = \log_2 k$
$\rightarrow b_i + \lfloor t_i \log_2 3 \rfloor + (t_i \log_2 3 - \lfloor t_i \log_2 3 \rfloor)$
$= \lfloor \log_2 k \rfloor + (\log_2 k - \lfloor \log_2 k \rfloor)$
提案手法では，上式を利用して k にもっとも近い $2^{b_i}3^{t_i}$ を求める．

[提案手法]

(1)t_i の範囲の決定

左辺と右辺で整数部を等しくする $b_i(b_{i-1} \geq b_i)$ を選択可能な t_i の範囲を求める．

(2)t_i の決定

(1) で求めた範囲の t_i から $t_i \log_2 3$ の小数部が $\log_2 k$ の小数部と最も近い t_i を選択.

(3)b_i の決定

(2) で求めた t_i に対し，左辺と右辺で整数部を等しくする $b_i(b_{i-1} \geq b_i)$ を求める．

以下，(1)-(3) の処理について，詳細を示す.

(1)t_i の範囲の決定

左辺と右辺で整数部を等しくする $b_i(b_{i-1} \geq b_i)$ を選択可能な t_i の範囲を求める．

$$b_i + \lfloor t_i \log_2 3 \rfloor = \lfloor \log_2 k \rfloor \tag{21}$$

$$b_{i-1} \geq b_i \geq 0 \tag{22}$$

$$t_{i-1} \geq t_i \geq 0 \tag{23}$$

式 (21) を式 (22) に代入して，

$b_{i-1} \geqq \lfloor \log_2 k \rfloor - \lfloor t_i \log_2 3 \rfloor \geqq 0$

$\lfloor \log_2 k \rfloor \geqq \lfloor t_i \log_2 3 \rfloor \geqq \lfloor \log_2 k \rfloor - b_{i-1}$

$\lfloor \log_2 k \rfloor + 1 \geqq t_i \log_2 3 \geqq \lfloor \log_2 k \rfloor - b_{i-1}$

$(\lfloor \log_2 k \rfloor + 1)/\log_2 3 \geqq t_i \geqq (\lfloor \log_2 k \rfloor - b_{i-1})/\log_2 3$

従って，求める t_i の範囲は，式 (23) と併せて，

$\mathrm{MIN}[t_{i-1}, \lfloor \log_2 k \rfloor + 1/\log_2 3] \geqq t_i$

$\geqq \mathrm{MAX}[0, (\lfloor \log_2 k \rfloor - b_{i-1})/\log_2 3]$

となる．ここで，計算量は，k の大きさに依存しないので，定数時間である．

(2) t_i の決定

(1) で求めた範囲の t_i の内，$t_i \log_2 3$ の小数部が $\log_2 k$ の小数部と最も近い t_i を選択する．選択方法は，$t_i \log_2 3$ をあらかじめ計算して小さい順に並べておき，二分探索法でその中から $\log_2 k$ の小数部が最も近い値を選択する．計算量は $\log k$ の要素から，二分探索法で探索を行うため，その計算量は $\log(\log k)$ である．

(3) b_i の決定

(2) で求めた t_i に対し，整数部を等しくする $b_i(b_{i-1} \geq b_i \geqq 0)$ を求める．

$b_i + \lfloor t_i \log_2 3 \rfloor = \lfloor \log_2 k \rfloor$

より，

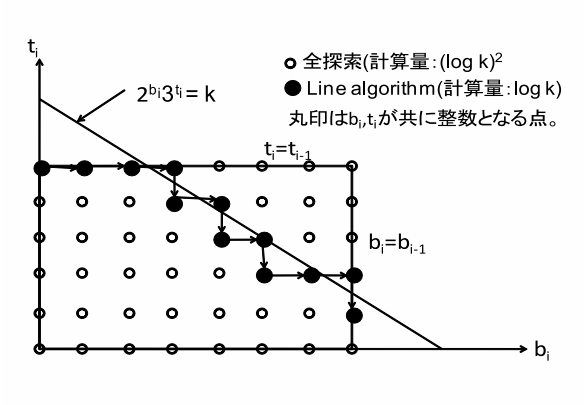

図 7: Greedy Algorithm の計算方法の既存研究

$b_i = \lfloor \log_2 k \rfloor - \lfloor t_i \log_2 3 \rfloor$

本処理の計算量は，k の大きさに依存しないため定数時間となる．以上から，(1)-(3) の各処理の計算量はスカラー k のビット数にのみ依存し，$\log(\log k)$ となり，既存研究における全探索（計算量：$(\log k)^2$），Line Algorithm（計算量：$\log k$）に比べ，少ない計算量を実現出来る．

A.6 比較評価

先に求めたスカラー倍算の平均値を最小にする b0 を使用し，DBNS 表現の第 2 項以降は Greedy algorithm により導出した DBNS 表現と，b0 を含めすべて Greedy algorithm によって導出した DBNS 表現について，スカラー倍算の計算コストの比較を実施した．なお，スカラー倍算の計算コストは，スカラー k のビット数 (32,64,96,128,160,176,192,208,224）に対し，それぞれ 10,000 個のランダムに発生させた整数の平均値を使用する．結果として，鍵長がいずれのビット数でも既存研究に比べ，本研究の手法が高速となること，また，ビット数が上がるほど，その効果が高まることを確認した．

A.7 結論

DBNS 表現の導出に関して，Dimitrov らの Greedy algorithm に対し，DBNS 表現の初項をスカラー k のビット数に応じて固定的に設定することで，スカラー倍算の計算コストを削減可能であることを示した．DBNS 表現の初項における 2 のべき乗の最適値は，スカラー k のビット数に対してほぼ比例関係に近くなり，スカラー k のビット数に対し，0.55 を乗じた値となることを示した．また，DBNS 表現の導出の主要処理となる Greedy Algorithm の実装方法として，既存研究（計算量：$\log k$）に比べ計算量の小さい方式（計算量：$\log(\log k)$）を提案した．

表 14: 既存研究との計算コスト比較評価

スカラー k (bit)	スカラー倍算の計算コスト [m]				削減率 対 DBNS (Dimitrov)
	バイナリ法	NAF	DBNS (Dimitrov)	DBNS (本研究)	
32	448	393	366	360	1.64
64	896	785	755	724	4.11
96	1344	1178	1128	1085	3.81
128	1792	1570	1509	1447	4.11
160	2240	1963	1908	1808	5.24
176	2492	2183	2106	1989	5.56
192	2688	2355	2308	2170	5.98
208	2912	2551	2509	2350	6.34
224	3136	2748	2712	2530	6.71

A.8 今後の予定

本研究では，DBNS 表現の導出アルゴリズムとして DBNS 表現の初項を固定的に設定する以外は Greedy algorithm を使用することとし，また，Greedy algorithm の具体的な計算方法として既存研究に比べ効率的な手法を提案した．今後は提案手法を実装し，既存研究の手法との実装ベースでの比較評価を実施すると共に，Greedy algorithm に代わるより効率的な DBNS 表現の導出アルゴリズムについて検討していく．

参考文献

[1] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In K. Ohta and D. Pei, editors, Advances in Cryptology – ASIACRYPT ’98, volume 1514 of Lecture Notes in Computer Science, pages 51–65. Springer-Verlag, 1998.

[2] V. S. Dimitrov, G. A. Jullien, and W. C. Miller. An algorithm for modular exponentiation. Information Processing Letters, 66(3):155–159, May 1998.

[3] Dimitrov, V.S., Imbert, L., Mishra, P.K.: Efficient and secure elliptic curve point multiplication using double-base chains. In: Advances in Cryptology – Asiacrypt 2005. Volume 3788 of Lecture Notes in Comput. Sci., Springer (2005) 59–78

[4] VASSIL DIMITROV, LAURENT IMBERT, AND PRADEEP K. MISHRA:THE DOUBLE-BASE NUMBER SYSTEM AND ITS APPLICATION TO ELLIPTIC CURVE CRYPTOGRAPHY. MATHEMATICS OF COMPUTATION Volume 77, Number 262, April 2008, Pages 1075–1104

[5] VASSIL S. DIMITROV AND EVERETT W. HOWE:LOWER BOUNDS ON THE LENGTHS OF DOUBLE-BASE REPRESENTATIONS. PROCEEDINGS OF THE AMERICAN MATHEMATICAL SOCIETY Volume 139, Number 10, October 2011, Pages 3423–3430

[6] Meloni, N., Hasan, M.A.: Elliptic Curve Scalar Multiplication Combining Yao 's Algorithm and Double Bases. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 304–316. Springer, Heidelberg (2009)

[7] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye:Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. IEEE TRANSACTIONS ON COMPUTERS, VOL. 53, NO. 6, JUNE 2004.

[8] Christophe Giraud and Vincent Verneuil:Atomicity Improvement for Elliptic Curve Scalar Multiplication. CARDIS 2010, LNCS 6035, pp. 80–101, 2010.

[9] Sung-Ming Yen and Marc Joye:Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis. IEEE TRANSACTIONS ON COMPUTERS, VOL. 49, NO. 9, SEPTEMBER

[10] Guillaume Gilbert and J. M. Pierre Langlois:Multipath Greedy Algorithm for Canonical Representation of Numbers in the Double Base Number System. 2005 IEEE

[11] Patrick Longa and Catherine Gebotys:Efficient Techniques for High-Speed Elliptic Curve Cryptography. CHES 2010, LNCS 6225, pp. 80–94, 2010.

[12] Erik Woodward Knudsen:Elliptic Scalar Multiplication Using Point Halving. ASIACRYPT '99, LNCS 1716, pp. 135–149, 1999.

[13] K.W. Wong, Edward C.W. Lee, L.M. Cheng1, and Xiaofeng Liao:Fast Elliptic Scalar Multiplication using New Double-base Chain and Point Halving.

[14] Herbert S. Wilf:generatingfunctionology

[15] Jithra Adikari, Student Member, IEEE, Vassil S. Dimitrov, and Laurent Imbert: Hybrid Binary-Ternary Number System for Elliptic Curve Cryptosystems. IEEE TRANSACTIONS ON COMPUTERS, VOL. 60, NO. 2, FEBRUARY 2011 Herbert S. Wilf:generatingfunctionology

[16] Valérie Berthéa, and Laurent Imbert: On Converting Numbers to the Double-Base Number System.

[17] W. Yu, K. Wang and B. Li, Fast Algorithm Converting integer to Double Base Chain. Information Security and Cryptology, Inscrypt 2010 short papers. pp.44–54, 2011.

[18] 本間尚文，青木孝文，佐藤証:暗号モジュールへのサイドチャネル攻撃とその安全性評価の動向電子情報通信学会論文誌　A　Vol.J93-A No.2 pp.42–51

[19] 宮地充子: 代数学から学ぶ暗号理論（日本評論社）

B 本研究で使用したプログラム

B.1 提案手法のスカラー倍算計算コストを求めるプログラム (k=160 ビット)

```
-------------------------------------------------------------------------
#include <stdio.h>
#include<stdlib.h>
#include <math.h>
#include <windows.h>
#include <longint.h>

main()
{
LINT N, z[100], z_tmp[100], b_beki[100], t_beki[100], n[8], z_min[1];
int b_sisuu[300], t_sisuu[300], b_sisuu_tmp[300], t_sisuu_tmp[300], k_sum[300];
int b_sisuu_z_min[1], b_sisuu_cost_min;
double cal_cost, cal_cost_sum[256], cal_cost_ave[256], k_ave[300];
int i, j, k, l, m, p, q, bit, byte;
char *s;

scanf("%d",&bit);
byte = bit/8;
s="1000000000000000000000000000000000000000000000000";


for( b_sisuu[0] = 1; b_sisuu[0] <= (bit -1); b_sisuu[0]++){
cal_cost_sum[b_sisuu[0]] = 0;
}

for (m=0; m <= 9999; m++){

for(;;){

HCRYPTPROV hProv;
BYTE buf[20];

/* デフォルト鍵コンテナの取得 */
CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0);
```

```
/* 乱数生成 */
 CryptGenRandom(hProv, 20, buf);

/* 生成した乱数を表示 */

/* 鍵コンテナの開放 */
CryptReleaseContext(hProv, 0);

if( buf[(byte-1)] >= 128){

for(p=0; p <= (byte-1); p++){
n[p] = lset(buf[p]);
for(q=1; q <= p*8; q++){
n[p] = smul(n[p],2);
}
}

N=lset(0L);
for(p=0; p <= (byte-1); p++){
N = ladd(N,n[p]);
}
break;
}
}

/* N = lread(s);
*/
lwrite("N", N);
z_min[0] = N;

for (b_sisuu[0]=(bit-1); b_sisuu[0] >= 1; b_sisuu[0]--){

b_beki[0] = lset(1L);
for (i = 1; i <= b_sisuu[0]; i++){
b_beki[0] = smul(b_beki[0],2);
}

t_beki[0] = lset(1L);
for(t_sisuu[0] = 0; lcmp(N,smul(lmul(b_beki[0],t_beki[0]),3))==1; t_sisuu[0]++){
```

```
t_beki[0] = smul(t_beki[0],3);
}

if(lcmp(lsub(N,lmul(b_beki[0],t_beki[0])),lsub(lmul(b_beki[0],smul(t_beki[0],3)),N))==1){
t_sisuu[0]++;
t_beki[0] = smul(t_beki[0],3);
}

z[0] = lsub(N,lmul(b_beki[0],t_beki[0]));

/* if (labs(z[0]) < labs(z_min[0])){
z_min[0] = z[0];
b_sisuu_z_min[0] = b_sisuu[0];
}
*/

for(k=0; lcmp(z[k],lset(0L)) != 0; k++){
b_sisuu_tmp[k+1] = b_sisuu[k];
b_sisuu[k+1] = b_sisuu[k];
t_sisuu[k+1] = t_sisuu[k];
z[k+1] = z[k];

for(; b_sisuu_tmp[k+1] >= 0; b_sisuu_tmp[k+1]--){
b_beki[k+1] =lset(1L);
for(i = 1; i <= b_sisuu_tmp[k+1] ; i++){
b_beki[k+1] = smul(b_beki[k+1],2);
}
t_sisuu_tmp[k+1] = t_sisuu[k];
for(; t_sisuu_tmp[k+1] >= 0; t_sisuu_tmp[k+1]--){
t_beki[k+1] = lset(1L);
for(j = 1; j <= t_sisuu_tmp[k+1]; j++){
t_beki[k+1] = smul(t_beki[k+1],3);
}

if(z[k].sign == 0)
z_tmp[k+1] = lsub(z[k],lmul(b_beki[k+1], t_beki[k+1]));
else
z_tmp[k+1] = ladd(z[k], lmul(b_beki[k+1], t_beki[k+1]));
```

```
if(lcmp(z[k+1], z_tmp[k+1])==1){
z[k+1] = z_tmp[k+1];
b_sisuu[k+1] = b_sisuu_tmp[k+1];
t_sisuu[k+1] = t_sisuu_tmp[k+1];
}
}

}

}

/* printf("b_sisuu[0]=%d\n", b_sisuu[0]);
printf("t_sisuu[0]=%d\n", t_sisuu[0]);
printf("k=%d\n", k);
*/
cal_cost = 8.8*b_sisuu[0] + 14.8*t_sisuu[0] + 10.4*k;
cal_cost_sum[b_sisuu[0]] = cal_cost_sum[b_sisuu[0]] + cal_cost;
k_sum[b_sisuu[0]] = k_sum[b_sisuu[0]] + k;
}
}

for (b_sisuu[0]=1; b_sisuu[0] <= (bit-1); b_sisuu[0]++){
cal_cost_ave[b_sisuu[0]] = cal_cost_sum[b_sisuu[0]]/m;
k_ave[b_sisuu[0]] = k_sum[b_sisuu[0]]/m;
printf("cal_cost_ave[%ld]=%f\n", b_sisuu[0], cal_cost_ave[b_sisuu[0]]);
printf("k_ave[%ld]=%f\n", b_sisuu[0], k_ave[b_sisuu[0]]);
}
}
```

B.2 Greedy Algorithmのスカラー倍算計算コストを求めるプログラム(k=160ビット)

```
#include <stdio.h>
#include<stdlib.h>
#include <math.h>
#include <windows.h>
#include <longint.h>
```

```
main()
{
LINT N, z[100], z_tmp[100], b_beki[100], t_beki[100], n[8], z_min[1];
int b_sisuu[300], t_sisuu[300], b_sisuu_tmp[300], t_sisuu_tmp[300], k_sum;
int b_sisuu_z_min[1], b_sisuu_cost_min, b_sisuu_sum, t_sisuu_sum;
double cal_cost, cal_cost_sum[256], cal_cost_ave[256], k_ave, b_sisuu_ave, t_sisuu_ave;
int i, j, k, l, m, p, q, r, bit, byte;
char *s;

scanf("%d",&bit);
byte = bit/8;
s="10000000000000005230000000000000000000087500000551";

for( b_sisuu[0] = 1; b_sisuu[0] <= (bit -1); b_sisuu[0]++){
cal_cost_sum[b_sisuu[0]] = 0;
}

b_sisuu_sum =0;
t_sisuu_sum =0;
b_sisuu_ave=0;
t_sisuu_ave=0;

for (m=0; m <= 10000; m++){

for(;;){

HCRYPTPROV hProv;
BYTE buf[20];

/* デフォルト鍵コンテナの取得 */
CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_FULL, 0);

/* 乱数生成 */
 CryptGenRandom(hProv, 20, buf);

/* 生成した乱数を表示 */

/* 鍵コンテナの開放 */
```

```
CryptReleaseContext(hProv, 0);

if( buf[(byte-1)] >= 128){

for(p=0; p <= (byte-1); p++){
n[p] = lset(buf[p]);
for(q=1; q <= p*8; q++){
n[p] = smul(n[p],2);
}
}

N=lset(0L);
for(p=0; p <= (byte-1); p++){
N = ladd(N,n[p]);
}
break;
}
}

/* N = lread(s);
*/
lwrite("N", N);

z_min[0] = N;

/***** b0-Kotei algorithm only(begin) *****/

/*
b_sisuu[0]=94;
b_beki[0] = lset(1L);

for (i = 1; i <= (b_sisuu[0]); i++){
b_beki[0] = smul(b_beki[0], 2);
}

t_beki[0] = lset(1L);
for(t_sisuu[0] = 0; lcmp(N,smul(lmul(b_beki[0],t_beki[0]),3))==1; t_sisuu[0]++){
t_beki[0] = smul(t_beki[0],3);
}
```

```
if(lcmp(lsub(N,lmul(b_beki[0],t_beki[0])),lsub(lmul(b_beki[0],smul(t_beki[0],3)),N))==1){
t_sisuu[0]++;
t_beki[0] = smul(t_beki[0],3);
}

z[0] = lsub(N,lmul(b_beki[0],t_beki[0]));
lwrite("z[0]", z[0]);
*/


/* printf("z[0]=%ld\n", z[0]);
*/
/* printf("b_sisuu[0]=%ld\n", b_sisuu[0]);
printf("t_sisuu[0]=%ld\n", t_sisuu[0]);
*/


/***** b0-Kotei algorithm only(end) *****/

for(k=0; k<0; k++){

b_beki[k+1] = lset(1L);
for(b_sisuu[k+1] = 0; lcmp(z[k],b_beki[k+1])==1; b_sisuu[k+1]++){
b_beki[k+1] = smul(b_beki[k+1], 2);
}

/* printf("b_sisuu[%d] =%ld\n", k+1, b_sisuu[k+1]);
*/
b_sisuu[k+1] = b_sisuu[k+1]*57/100;
b_beki[k+1] = lset(1L);
for(i = 1; i <= b_sisuu[k+1]; i++){
b_beki[k+1] = smul(b_beki[k+1], 2);
}

t_beki[k+1] = lset(1L);
for(t_sisuu[k+1] = 0; lcmp(z[k], smul(lmul(b_beki[k+1],t_beki[k+1]),3))==1 && (t_sisuu[k+1] <= t_sis
t_beki[k+1] = smul(t_beki[k+1], 3);
}
```

```
if(z[k].sign == 0){
if (lcmp(lsub(z[k],lmul(b_beki[k+1],t_beki[k+1])), lsub(smul(lmul(b_beki[k+1],t_beki[k+1]),3),z[k]))
t_sisuu[k+1]++;
t_beki[k+1] = smul(t_beki[k+1], 3);
}
}
else{
if (lcmp(ladd(z[k],lmul(b_beki[k+1],t_beki[k+1])), ladd(smul(lmul(b_beki[k+1],t_beki[k+1]),3),z[k]))
t_sisuu[k+1]++;
t_beki[k+1] = smul(t_beki[k+1], 3);
}
}

if(z[k].sign == 0)
z[k+1] = lsub(z[k], lmul(b_beki[k+1],t_beki[k+1]));
else
z[k+1] = ladd(z[k], lmul(b_beki[k+1],t_beki[k+1]));

/* printf("b_sisuu[%ld]=%ld\n", k+1,b_sisuu[k+1]);
printf("t_sisuu[%ld]=%ld\n", k+1, t_sisuu[k+1]);
*/
}



/* if (labs(z[0]) < labs(z_min[0])){
z_min[0] = z[0];
b_sisuu_z_min[0] = b_sisuu[0];
}
*/

/*****  for greedy algorithm only(begin)  *****/

z[0] = N;
b_sisuu_tmp[0] = bit;
for(; b_sisuu_tmp[0] >= 0; b_sisuu_tmp[0]--){
b_beki[0] =lset(1L);
for(i = 1; i <= b_sisuu_tmp[0] ; i++){
```

```
b_beki[0] = smul(b_beki[0],2);
}
t_sisuu_tmp[0] = bit/1.584963;
for(; t_sisuu_tmp[0] >= 0; t_sisuu_tmp[0]--){
t_beki[0] = lset(1L);
for(j = 1; j <= t_sisuu_tmp[0]; j++){
t_beki[0] = smul(t_beki[0],3);
}

z_tmp[0] = lsub(N,lmul(b_beki[0], t_beki[0]));

if(lcmp(z[0], z_tmp[0])==1){
z[0] = z_tmp[0];
b_sisuu[0] = b_sisuu_tmp[0];
t_sisuu[0] = t_sisuu_tmp[0];
}
}
}

/*****  for greedy algorithm only(end)  *****/

for(; lcmp(z[k],lset(0L)) != 0; k++){
b_sisuu_tmp[k+1] = b_sisuu[k];
b_sisuu[k+1] = b_sisuu[k];
t_sisuu[k+1] = t_sisuu[k];
z[k+1] = z[k];

for(; b_sisuu_tmp[k+1] >= 0; b_sisuu_tmp[k+1]--){
b_beki[k+1] =lset(1L);
for(i = 1; i <= b_sisuu_tmp[k+1] ; i++){
b_beki[k+1] = smul(b_beki[k+1],2);
}
t_sisuu_tmp[k+1] = t_sisuu[k];
for(; t_sisuu_tmp[k+1] >= 0; t_sisuu_tmp[k+1]--){
t_beki[k+1] = lset(1L);
for(j = 1; j <= t_sisuu_tmp[k+1]; j++){
t_beki[k+1] = smul(t_beki[k+1],3);
}
```

```
if(z[k].sign == 0)
z_tmp[k+1] = lsub(z[k],lmul(b_beki[k+1], t_beki[k+1]));
else
z_tmp[k+1] = ladd(z[k], lmul(b_beki[k+1], t_beki[k+1]));

if(lcmp(z[k+1], z_tmp[k+1])==1){
z[k+1] = z_tmp[k+1];
b_sisuu[k+1] = b_sisuu_tmp[k+1];
t_sisuu[k+1] = t_sisuu_tmp[k+1];

/* printf("b_sisuu[%ld]=%ld\n", k+1, b_sisuu[k+1]);
printf("t_sisuu[%ld]=%ld\n", k+1, t_sisuu[k+1]);
*/
}
}

}
}

/* printf("b_sisuu[0]=%d\n", b_sisuu[0]);
printf("t_sisuu[0]=%d\n", t_sisuu[0]);
printf("k=%d\n", k);
*/
cal_cost = 8.8*b_sisuu[0] + 14.8*t_sisuu[0] + 10.4*k;
/* printf("cal_cost=%f\n", cal_cost);
*/
printf("m=%d\n", m);
cal_cost_sum[0] = cal_cost_sum[0] + cal_cost;
k_sum = k_sum + k;
b_sisuu_sum = b_sisuu_sum + b_sisuu[0];
t_sisuu_sum = t_sisuu_sum + t_sisuu[0];
}

/* for (r=0; r <= k; r++){
printf("b_sisuu[%d]=%d\n", r, b_sisuu[r]);
printf("t_sisuu[%d]=%d\n", r, t_sisuu[r]);
lwrite("z[1]", z[r]);
}
*/
```

```
cal_cost_ave[0] = cal_cost_sum[0]/m;
printf("m=%d\n", m);
printf("cal_cost_ave[0]=%f\n", cal_cost_ave[0]);
b_sisuu_ave = b_sisuu_sum/m;
/* printf("b_sisuu_ave=%f\n", b_sisuu_ave);
*/
printf("b_sisuu_sum=%d\n", b_sisuu_sum);
t_sisuu_ave = t_sisuu_sum/m;
printf("t_sisuu_sum=%d\n", t_sisuu_sum);
k_ave = k_sum/m;
printf("k_sum=%d\n", k_sum);
}
```
