SCIS 2013 The 30th Symposium on
Cryptography and Information Security
Kyoto, Japan, Jan. 22 - 25, 2013
The Institute of Electronics,
Information and Communication Engineers

NEON を利用した Android 上でのペアリング暗号の高速実装
Fast Implementation of Pairing-based Cryptosystems on Android with NEON

川原 祐人* 吉田 麗生* 星野 文学* 小林 鉄太郎*
Yuto KAWAHARA Reo YOSHIDA Fumitaka HOSHINO Tetsutaro KOBAYASHI

あらまし ペアリング暗号では，ID ベース暗号や関数型暗号をはじめとする従来の公開鍵暗号では実現できない高機能な暗号システムへの応用が可能である．現在，ペアリング暗号を利用したシステムの実用化に向けて，標準化やシステム実装報告が行われている．またペアリングの高速化も多数行われており，高速実装の例として，これまでに PC や FPGA，携帯電話，センサノード，IC カードなど様々な端末上での報告がされている．最近では，スマートフォンやタブレット端末が急速に普及していることから，Android や iOS 上での高速実装もされている．本稿では，高速なペアリング暗号の構成である BN 曲線上の Optimal Ate ペアリングに対して，NEON と呼ばれる ARM アーキテクチャでの SIMD 命令を利用した Android 端末上の高速実装の結果を示す．我々は，NEON を用いることで素体 $\mathbb{F}_p$ 上の高速な乗算アルゴリズムを構成し，ペアリングの高速化を図った．実験結果として，Android 端末上での 126-bit セキュリティ相当のペアリングの計算は約 6.77 ミリ秒となり，C による実装と比較し約 4.5 倍高速であった．また暗号システムへの応用として，本実装を関数型暗号へ適応し，Setup, Keygen, Encrypt, Decrypt の計算時間を評価した．

キーワード ペアリング暗号, Android, NEON, 高速実装

1 はじめに

ペアリング暗号では，双線型性を利用することにより，従来の公開鍵暗号において実現困難であった高機能な暗号プロトコルが構築可能である．実用的な ID ベース暗号が境ら [16]，Boneh ら [8] により提案されて以降，これまでに関数型暗号 [15, 19] などの様々な暗号プロトコルが提案されている．現在は，ペアリング暗号を利用したシステムの実用化へ向けて，標準化やシステム実装報告が行われている．一方，ペアリングの計算に関しては，Miller により初めて多項式時間で計算可能なアルゴリズムが提案されたが [12]，Miller アルゴリズムによる Tate ペアリングの計算は従来公開鍵暗号と比較して低速であった．近年では，η_T ペアリングや Ate ペアリング，Optimal Ate ペアリングなどの高速なアルゴリズムが提案されている．また実装例としては，PC をはじめとして FPGA や携帯電話，センサノード，IC カードなど，多岐に渡る端末で評価されている．現在，最も高速なペアリングである Barreto-Naehrig 曲線 [6] 上の Optimal Ate ペアリング [18] で，126-bit, 128-bit セキュリティの構成において，Beuchat ら [7]，Aranha ら [2]，光成ら [13] は PC 上で 1 ミリ秒以下という結果を示しており，PC 上でのペアリング暗号は十分実用的な計算時間で動作することが分かっている．またスマートフォンやタブレット端末が急速な普及から，これらの端末上でのペアリングの実装もされており，井山ら [10, 11] は Android 端末上で約 2 秒，また森ら [14] は iPhone, iPad 上で 140 ミリ秒，117 ミリ秒との結果を示している．

Android は，スマートフォンやタブレット端末向けのプラットフォームのひとつである．Android アプリケーションは Java ベースで実装され，Dalvik VM と呼ばれる Android 用の JVM 上で動作する．VM 上で動作する Java アプリケーションは C などと比較して低速であるため，Android では，C などで実装されたネイティブコードでも動作させることが可能である．また Android 端末で多く利用される ARM アーキテクチャでは，NEON と呼ばれる SIMD 命令がオプションでサポートされている．

本稿では，ARM アーキテクチャで NEON が利用可能な Android 端末を対象に，SIMD 命令を利用したペアリング暗号の高速実装を示す．本稿で利用する BN 曲線上の Optimal Ate ペアリングは，大きな素数 p として，素

* NTT セキュアプラットフォーム研究所，東京都武蔵野市緑町 3-9-11. NTT Secure Platform Laboratories, 3-9-11, Midori-cho Musashino-shi, Tokyo, Japan.

体 $\mathbb{F}_p$ 上の演算が必要であり，その中で支配的な計算コストであるのが素体 $\mathbb{F}_p$ 上の乗算である．我々は，NEON 命令を用いることで素体 $\mathbb{F}_p$ 上の高速な乗算アルゴリズムを構成し，ペアリングを高速化した．また暗号システム構築に利用する関数として，ペアリングの計算，楕円スカラー倍算および有限体上べき乗算の計算時間と，暗号システムへの応用として，関数型暗号の Setup, Keygen, Encrypt, Decrypt の計算時間を Android 端末上で評価した．実験結果として，126-bit セキュリティのパラメータに対して，Android 端末上でのペアリングの計算は約 6.77 ミリ秒であり，C による実装と比較して約 4.5 倍高速となった．

2 ペアリング暗号

本節では，Barreto-Naehrig 曲線上の Optimal Ate ペアリングに関して，楕円曲線，拡大体およびペアリングの構成を説明する．

2.1 Barreto-Naehrig 楕円曲線 [6]

素体 $\mathbb{F}_p$ 上の高速なペアリングの構成に用いる楕円曲線として Barreto-Naehrig 曲線がある．BN 曲線は埋め込み次数 $k = 12$ を持つ曲線であり，

$$E : y^2 = x^3 + b \ (b \neq 0)$$

で定義される．BN 曲線では，$z \in \mathbb{Z}$ に対して，素数 p, BN 曲線上の $\mathbb{F}_p$-有理点群の位数 r，フロベニウス写像のトレース t_r を以下のように表現できる．

$$\begin{aligned} p &= 36z^4 + 36z^3 + 24z^2 + 6z + 1 \\ r &= 36z^4 + 36z^3 + 18z^2 + 6z + 1 \\ t_r &= 6z^2 + 1 \end{aligned}$$

BN 曲線における，位数 r の $\mathbb{F}_p$-有理点群を，$\mathcal{O}$ を無限遠点として，次の通りとする．

$$E(\mathbb{F}_p) = \{(x, y) \in (\mathbb{F}_p)^2 \mid y^2 = x^3 + b\} \cup \{\mathcal{O}\}$$

次に，素体 $\mathbb{F}_p$ の拡大体 $\mathbb{F}_{p^{12}}$ を，逐次拡大により，下記の通りに構成する．

$$\begin{aligned} \mathbb{F}_{p^2} &= \mathbb{F}_p[u]/(u^2 - \beta) \\ \mathbb{F}_{p^6} &= \mathbb{F}_p[v]/(v^3 - \xi) \\ \mathbb{F}_{p^{12}} &= \mathbb{F}_p[w]/(w^2 - v) \end{aligned}$$

また $E(\mathbb{F}_{p^{12}})$ の 6 次ツイスト曲線の $\mathbb{F}_{p^2}$-有理点群を次の通りとする．

$$E'(\mathbb{F}_{p^2}) = \{(x, y) \in (\mathbb{F}_{p^2})^2 \mid y^2 = x^3 + b/\xi\} \cup \{\mathcal{O}\}$$

同型写像 $\phi : E'(\mathbb{F}_{p^2}) \to E(\mathbb{F}_{p^{12}})$ が存在するため，ペアリングの計算では $E'(\mathbb{F}_{p^2})$ 上の点を使用できる．

本稿の実装では，126-bit 安全なパラメータとして，Beuchat ら [7] と同様の $z = 2^{62} - 2^{54} + 2^{44} \in \mathbb{Z}$ を利用する．BN 曲線の定数 $b = -5$，また $\beta = -5$, $\xi = u$ とする．

2.2 BN 曲線上の Optimal Ate ペアリング [18]

$E[r]$ を E の r-ねじれ部分群，また $E(\mathbb{F}_{p^n})$ 上の点 $P = (x, y)$ に対して，フロベニウス写像 $\pi_p : (x, y) \mapsto (x^p, y^p)$ とする．ペアリングの入力となる群 $\mathbb{G}_1 = E[r] \cap \ker(\pi_p - [1]) = E(\mathbb{F}_p)[r]$, $\mathbb{G}_2 = E[r] \cap \ker(\pi_p - [p]) \subseteq E(\mathbb{F}_{p^{12}})[r]$, 出力となる群 $\mathbb{G}_3 = \mu^* \subset \mathbb{F}^*_{p^{12}}$ より，BN 曲線上の Optimal Ate ペアリング e_{opt} は以下のように定義される，

$$\begin{aligned} e_{opt} : \mathbb{G}_2 \times \mathbb{G}_1 &\to \mathbb{G}_3 \\ (Q, P) &\mapsto (f_{(6z+2),Q}(P) \cdot l_{(6z+2)Q,\pi_p(Q)}(P) \cdot \\ &\qquad l_{(6z+2)Q+\pi_p(Q),-\pi_p^2(Q)}(P))^{\frac{p^{12}-1}{r}} \end{aligned}$$

$f_{(6z+2),Q}(P)$ は Miller アルゴリズムを利用して，また $l_{(6z+2)Q,\pi_p(Q)}(P)$, $l_{(6z+2)Q+\pi_p(Q),-\pi_p^2(Q)}(P)$ は，点 Q_1, Q_2 を通る直線を l_{Q_1,Q_2} として計算する．最後に，最終べきである $(p^{12} - 1)/q$ 乗算によりペアリング e_{opt} を計算する．

3 Android

Android は，スマートフォンやタブレット端末などのモバイル端末向けのプラットフォームである．Android OS は，Linux ベースのオープンソース OS であり，アプリケーション開発や配布が容易に可能である．Android では，Java ベースでアプリケーションが実装され，基本的にはモバイル用に最適化された Java 仮想マシンである Dalvik VM 上で動作する．また高速処理が必要な場合は，Android NDK により，C, C++, アセンブリで実装されたネイティブコードの動作が可能である．Java との連携には JNI (Java Native Interface) を用いる．

Android 端末で利用可能な CPU アーキテクチャとして ARM, MIPS, x86 があるが，現在，スマートフォンなどに搭載される CPU のほとんどで ARM が用いられている．ARM アーキテクチャのひとつである ARMv7-A では，Advanced SIMD 拡張命令として NEON がオプションで利用可能であり，SIMD 命令を利用することでより高速な実装が可能となる．本稿の実装では，NEON がサポートされた ARMv7-A を対象とする．

以下に，基本的な ARM アセンブリと NEON の環境と命令を説明する．

3.1 ARM アセンブリ

ARM アセンブリでは，15 個の汎用 32-bit レジスタ (r-レジスタ) を備えており，各々のレジスタを r_0, ..., r_{12}, sp, lr と表現する．sp, lr はそれぞれスタックポインタ,

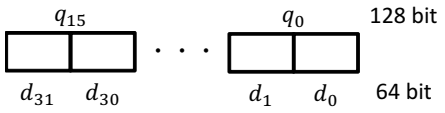

図 1: NEON でのレジスタ構成

リンクレジスタであり，スタックポインタの汎用レジスタとしての使用は推奨されていない．

ビット列 $a = \{0,1\}^{32}$ に対して，a を保持する r-レジスタの表現を $[a]_r$ とする．a, b をメモリ上の 32-bit の値とした場合，ARM アセンブリでは以下の命令を持つ．

キャリー無加算 adds：
$[a+b]_r, carry' \leftarrow [a]_r + [b]_r$

キャリー有加算 adcs：
$[a+b+carry]_r, carry' \leftarrow [a]_r + [b]_r + carry$

メモリロード ldr： $[a]_r = r_* \leftarrow a$

メモリストア str： $a \leftarrow r_* = [a]_r$

ここで $carry$ はキャリービットであり，32-bit 加算命令で発生するオーバーフロー $\{0,1\}$ が格納される．

3.2 NEON (Advanced SIMD 拡張命令)

NEON は，ARMv7-A に搭載される Advanced SIMD 拡張命令であり，128-bit レジスタを使用した整数演算，浮動小数点演算が可能である．NEON では，16 個の 128-bit レジスタ (q-レジスタ) を備えており，各々のレジスタを $q_0, ..., q_{15}$ と表現する．q-レジスタは 32 個の 64-bit レジスタ (d-レジスタ) としても利用可能であり，q-レジスタと d-レジスタの対応は図 1 の通りとなる．

$a_{ij} = \{0,1\}^{64}$，$a_i = \{0,1\}^{32}$ とし，ビット列 $a = a_{32}\|a_{10} = a_3\|a_2\|a_1\|a_0 = \{0,1\}^{128}$ としたとき，q-レジスタおよび d-レジスタを次の通りに表現する．

$$[a]_q = [[a_{32}]_d, [a_{10}]_d]_q = [[a_3, a_2]_d, [a_1, a_0]_d]_q$$

a, b をメモリ上の 128-bit の値とした場合，NEON 命令として以下の命令を持つ．

加算 vadd：
$$[[a_{32}+b_{32}]_d, [a_{10}+b_{10}]_d]_q \leftarrow [[a_{32}]_d, [a_{10}]_d]_q + [[b_{32}]_d, [b_{10}]_d]_q$$

Long 加算 vaddl：
$[[a_1+b_1]_d, [a_0+b_0]_d]_q \leftarrow [a_1, a_0]_d + [b_1, b_0]_d$

Word 加算 vaddw：
$[[a_{32}+b_1]_d, [a_{10}+b_0]_d]_q \leftarrow [a_{32}, a_{10}]_q + [b_1, b_0]_d$

乗算 vmull：
$[[a_1 b_1]_d, [a_0 b_0]_d]_q \leftarrow [a_1, a_0]_d \times [b_1, b_0]_d$

Long 移動 vmovl： $[[a_1]_d, [a_0]_d]_q \leftarrow [a_1, a_0]_d$

d-レジスタから r-レジスタへの移動 vmov：
$[a_0]_r, [a_1]_r \leftarrow [a_1, a_0]_d$

ベクタ置換 vext：
$[(b_{10} \ll (64-8i)) | (a_{10} \gg 8i)]_d \leftarrow ([a_{10}]_d, [b_{10}]_d, i)$

メモリロード vld1.64： $[a]_q = q_* \leftarrow a$

メモリストア vst1.64： $a \leftarrow q_* = [a]_q$

ここで，i を $0 \le i < 8$ の整数とする．NEON 命令の 64-bit 加算命令 vaddw, vadd では，キャリーが発生した場合は無視される．

4 NEON 上での 256-bit 整数の乗算手法

本節では，256-bit 整数 a, b を入力とする多倍長整数乗算 $c = a \times b$ に対する NEON を利用した高速化手法を示すことで，最大 256-bit の素数 p における素体 $\mathbb{F}_p$ 上の乗算を高速化する．

256-bit の整数集合を $\mathbb{Z}_{2^{256}}$ とする．ARM のワード長 $W = 32$，サイズ $N = 8$ に対して，$\mathbb{Z}_{2^{256}}$ 上の元 a は，

$$a = \sum_{i=0}^{N-1} a_i 2^{iW} \in \mathbb{Z}_{2^{256}} \ (0 \le a_i < 2^W)$$

と表現される．このとき入力 $a, b \in \mathbb{Z}_{2^{256}}$ とした多倍長整数乗算は，$c = a \times b = (\sum_{i=0}^{N-1} a_i 2^{iW}) \times (\sum_{i=0}^{N-1} b_i 2^{iW})$ で計算される．この計算は，通常 W-bit 毎の乗算命令と加算命令を用いて構成する．

4.1 従来手法

素朴な多倍長整数乗算手法として，b_j を固定し，$a_0, ..., a_{N-1}$ に対して，$(c_{i+j+1}, c_{i+j}) + b_j \times a_i$ を順次計算し結果 c を更新するアルゴリズムがある．また乗算結果を下位の c_0 から順次計算するアルゴリズムとして Comba 法がある．Comba 法では，出力のメモリへの一時的な保存が不要で，メモリロード，ストアを削減できる．

また SIMD 命令を用いた多倍長整数乗算の手法として，入力を $(a_{i+T-1}, ..., a_i)$，$(b_{j+T-1}, ..., b_j)$ と適切なサイズ T のブロックに分割し，ブロック毎の乗算を順次計算する手法が，Intel CPU の SIMD 拡張命令 SSE2 を用いて実装されている [9].

4.2 提案手法

本節では，256-bit 整数 a, b を入力とする多倍長整数乗算 $c = a \times b$ について，NEON を利用した高速なアルゴリズムを示す．提案手法は，Comba 法と同様に結果

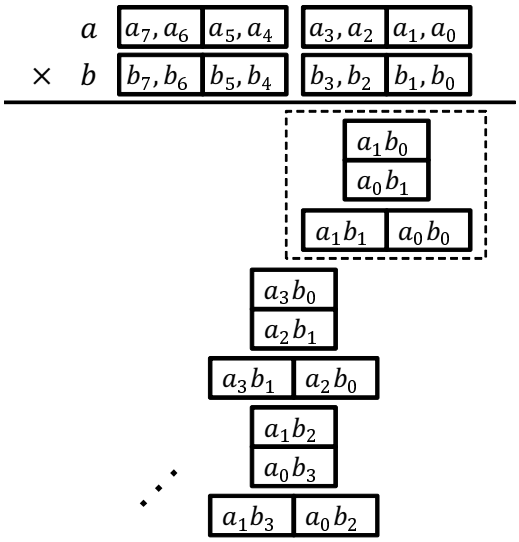

図 2: NEON 乗算命令の計算順序

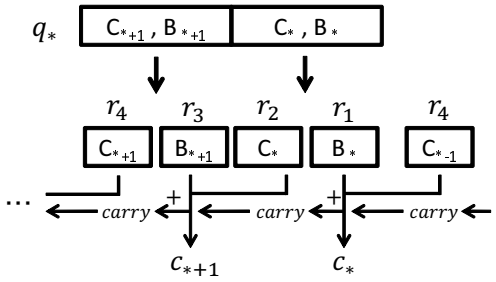

図 3: ARM アセンブリでの $\mathtt{C}_*$, $\mathtt{B}_*$ の加算処理

を下位から順次計算する．ARM アセンブリと NEON の両方の環境を用いて，次の手順で計算する．

1. 入力 a, b を 4 個の q-レジスタに保存する．
2. $[b_{*+1}, b_*]_d$ に対して，$[b_*, b_{*+1}]_d$ を計算し，2 個の q-レジスタに保存する．
3. 図 2 の順序で，2 回の NEON 乗算命令を単位として計算し，2 個の q-レジスタに格納する．
4. 乗算命令結果に対して，64-bit 単位で同様の 2^{*W} を持つ結果を，3 個の q-レジスタを用いて一時加算処理を行う．一時加算処理では，オーバーフローを抑制するために，64-bit を 32-bit 毎に分割し，それぞれを 64-bit 加算命令で計算する．加算命令後は，64-bit の上位にキャリー $\mathtt{C}_*$，下位にボディ $\mathtt{B}_*$ が格納される．
5. 同様の 2^{*W} を持つ乗算命令の分 Step 3–4 を繰り返し，一時加算処理の結果を更新する．
6. 一時加算処理での 3 個の q-レジスタに対して，32-bit 単位で同様の 2^{*W} を持つ d-レジスタを統合する．d-レジスタはキャリー $\mathtt{C}_*$ とボディ $\mathtt{B}_*$ で構成される．結果として，2 個の q-レジスタが構成される．

Algorithm 1 多倍長整数乗算 $c = a \times b$

INPUT: $a, b \in \mathbb{Z}_{2^{256}}$, r_0 is the pointer of c for **str**
OUTPUT: $c = a \times b$

1: $q_1, q_0 \leftarrow (a_7, ..., a_4), (a_3, ..., a_0)$
2: $q_3, q_2 \leftarrow (b_7, ..., b_4), (b_3, ..., b_0)$
3: $d_{11}, d_{10}, d_9, d_8 \leftarrow (b_6, b_7), (b_4, b_5), (b_2, b_3), (b_0, b_1)$
4: **for** $n \leftarrow 0$ **to** $N-2$ **do**
5: **if** $n < N/2$ **then** $i = 0$, $j = n$
else $i = n - N/2 + 1$, $j = N/2 - 1$
6: **while** $i < N/2$ and $j \geq 0$ **do**
7: $q_6, q_7 \leftarrow$ mul_mul(d_i, d_{j+4}, d_{j+8})
8: **if** $i = 0$ or $j = N/2 - 1$ **then**
9: $q_8, q_9, q_{10} \leftarrow$ shift_blk(q_6, q_7, q_9)
10: **else**
11: $q_8, q_9, q_{10} \leftarrow$ add_long$(q_6, q_7, q_8, q_9, q_{10})$
12: **end if**
13: $i \leftarrow i + 1$, $j \leftarrow j - 1$
14: **end while**
15: $q_8, q_9 \leftarrow$ marge(q_8, q_9, q_{10})
16: $c_{2n}, c_{2n+1}, carry, r_4 \leftarrow$ add_str$(q_8, r_4, carry)$
17: **end for**
18: $c_{2N-2}, c_{2N-1} \leftarrow$ add_str$(q_9, r_4, carry)$ $(n = N-1)$
19: **return** $c = (c_{2N-1}, ..., c_0)$

Algorithm 2 mul_mul(d_i, d_{j+4}, d_{j+8})

INPUT: $d_i = [a_{2i+1}, a_{2i}]_d$,
$d_{j+4} = [b_{2j+1}, b_{2j}]_d$, $d_{j+8} = [b_{2j}, b_{2j+1}]_d$
OUTPUT: $q_6 = [[a_{2i+1}b_{2j+1}]_d, [a_{2i}b_{2j}]_d]_q$,
$q_7 = [[a_{2i+1}b_{2j}]_d, [a_{2i}b_{2j+1}]_d]_q$

1: $q_6 \leftarrow$ vmull d_i, d_{j+4}
2: $q_7 \leftarrow$ vmull d_i, d_{j+8}

7. 下位の値を保持する q-レジスタを図 3 のように ARM アセンブリへ移動し，キャリー有加算命令とメモリストア命令を実行する．
8. Step 3–7 の操作を $N-1$ 回繰り返す．
9. 最上位の値を保持する q-レジスタを図 3 のように ARM アセンブリへ移動し，キャリー有加算命令とメモリストア命令を実行する．

以上の手順による 256-bit 入力の多倍長整数乗算を，Algorithm 1, ..., 6 で詳細に記述する．

NEON 乗算命令 vmull の実行クロックは 2clk であるが，出力結果が利用可能となるまでに 7clk 必要とする．従って，乗算命令直後に結果を参照する場合，5clk の遅延が発生してしまう．これを回避するため，Algorithm 1 の mul_mul で利用する q-レジスタを 2 個追加し，shft_blk,

Algorithm 3 shift_blk(q_6, q_7, q_9)

INPUT: $q_6 = [[a_{2i+1}b_{2j+1}]_d, [a_{2i}b_{2j}]_d]_q$,
$q_7 = [[a_{2i+1}b_{2j}]_d, [a_{2i}b_{2j+1}]_d]_q$,
$q_9 = [[\mathtt{C}_{2n+1}, \mathtt{B}_{2n+1}]_d, [\mathtt{C}_{2n}, \mathtt{B}_{2n}]_d]_q$

OUTPUT: $q_8 = [[\mathtt{C}'_{2n+1}, \mathtt{B}'_{2n+1}]_d, [\mathtt{C}'_{2n}, \mathtt{B}'_{2n}]_d]_q$,
$q_9 = [[\mathtt{C}'_{2n+3}, \mathtt{B}'_{2n+3}]_d, [\mathtt{C}'_{2n+2}, \mathtt{B}'_{2n+2}]_d]_q$,
$q_{10} = [[\tilde{\mathtt{C}}'_{2n+2}, \tilde{\mathtt{B}}'_{2n+2}]_d, [\tilde{\mathtt{C}}'_{2n+1}, \tilde{\mathtt{B}}'_{2n+1}]_d]_q$

1: $q_8 \leftarrow$ vaddw q_9, d_{12}
2: $q_9 \leftarrow$ vmovl d_{13}
3: $q_{10} \leftarrow$ vaddl d_{14}, d_{15}

Algorithm 4 add_long($q_6, q_7, q_8, q_9, q_{10}$)

INPUT: $q_6 = [[a_{2i+1}b_{2j+1}]_d, [a_{2i}b_{2j}]_d]_q$,
$q_7 = [[a_{2i+1}b_{2j}]_d, [a_{2i}b_{2j+1}]_d]_q$,
$q_8 = [[\mathtt{C}_{2n+1}, \mathtt{B}_{2n+1}]_d, [\mathtt{C}_{2n}, \mathtt{B}_{2n}]_d]_q$,
$q_9 = [[\mathtt{C}_{2n+3}, \mathtt{B}_{2n+3}]_d, [\mathtt{C}_{2n+2}, \mathtt{B}_{2n+2}]_d]_q$,
$q_{10} = [[\tilde{\mathtt{C}}_{2n+2}, \tilde{\mathtt{B}}_{2n+2}]_d, [\tilde{\mathtt{C}}_{2n+1}, \tilde{\mathtt{B}}_{2n+1}]_d]_q$

OUTPUT: $q_8 = [[\mathtt{C}'_{2n+1}, \mathtt{B}'_{2n+1}]_d, [\mathtt{C}'_{2n}, \mathtt{B}'_{2n}]_d]_q$,
$q_9 = [[\mathtt{C}'_{2n+3}, \mathtt{B}'_{2n+3}]_d, [\mathtt{C}'_{2n+2}, \mathtt{B}'_{2n+2}]_d]_q$,
$q_{10} = [[\tilde{\mathtt{C}}'_{2n+2}, \tilde{\mathtt{B}}'_{2n+2}]_d, [\tilde{\mathtt{C}}'_{2n+1}, \tilde{\mathtt{B}}'_{2n+1}]_d]_q$

1: $q_8 \leftarrow$ vaddw q_8, d_{12}
2: $q_9 \leftarrow$ vaddw q_9, d_{13}
3: $q_{10} \leftarrow$ vaddw q_{10}, d_{14}
4: $q_{10} \leftarrow$ vaddw q_{10}, d_{15}

Algorithm 5 marge(q_8, q_9, q_{10})

INPUT: $q_8 = [[\mathtt{C}_{2n+1}, \mathtt{B}_{2n+1}]_d, [\mathtt{C}_{2n}, \mathtt{B}_{2n}]_d]_q$,
$q_9 = [[\mathtt{C}_{2n+3}, \mathtt{B}_{2n+3}]_d, [\mathtt{C}_{2n+2}, \mathtt{B}_{2n+2}]_d]_q$,
$q_{10} = [[\tilde{\mathtt{C}}_{2n+2}, \tilde{t}_{2n+2}]_d, [\tilde{\mathtt{C}}_{2n+1}, \tilde{\mathtt{B}}_{2n+1}]_d]_q$

OUTPUT: $q_8 = [[\mathtt{C}'_{2n+1}, \mathtt{B}'_{2n+1}]_d, [\mathtt{C}'_{2n}, \mathtt{B}'_{2n}]_d]_q$,
$q_9 = [[\mathtt{C}'_{2n+3}, \mathtt{B}'_{2n+3}]_d, [\mathtt{C}'_{2n+2}, \mathtt{B}'_{2n+2}]_d]_q$

1: $d_{17} \leftarrow$ vadd d_{17}, d_{20}
2: $d_{18} \leftarrow$ vadd d_{18}, d_{21}

add_long では，前々回の乗算命令の結果を参照することで，遅延が軽減され，より高速化される．

4.3 高速化の評価

提案したアルゴリズムにより，以下のような高速化の効果が得られる．

ワード乗算の計算効率化 ARM アセンブリにおける乗算命令 mull では，入力レジスタ 2 個と出力レジスタ 2 個により，32-bit × 32-bit → 64-bit の計算がされる．一方，NEON 乗算命令 vmull では，前述の計算を 2 並列で計算可能であるため，乗算命令の呼出回数が 1/2 となる．また mull は 3clk，vmull は 2clk であり，乗算命令自体の実行サイクルも削減される．

Algorithm 6 add_str($q_*, r_4, carry$)

INPUT: $q_* = [[\mathtt{C}_{2n+1}, \mathtt{B}_{2n+1}]_d, [\mathtt{C}_{2n}, \mathtt{B}_{2n}]_d]_q$, r_4, $carry$

OUTPUT: $c_{2n}, c_{2n+1}, r_4, carry$

1: $r_1, r_2 \leftarrow$ vmov d_{2*}
2: $c_{2n}, carry \leftarrow$ adcs $r_1, r_4, carry$
3: $r_3, r_4 \leftarrow$ vmov d_{2*+1}
4: $c_{2n+1}, carry \leftarrow$ adcs $r_3, r_2, carry$

表 1: Android 端末のスペック

Device	docomo GALAXY S III SC-06D
SoC	Qualcomm Snapdragon S4 MSM8960
CPU	Dual Krait (1.5 GHz, 2 core)
Architecture	ARMv7-A
RAM	2 GB
OS ver.	Android 4.0.4
gcc ver.	4.4.3

表 2: PC のスペック

CPU	Intel Core i7-2600 (3.4 GHz, 4 core)
Architecture	Intel 64
RAM	16 GB
OS ver.	Linux Mint 14 (64-bit)
gcc ver.	4.7.2

入力 a, b のメモリロードの削減 最大 256-bit の入力 a, b をメモリから q-レジスタに格納する場合，高々4 個の q-レジスタで入力を保持できる．計算中は，これらのレジスタへの書込を禁止することで，入力 a, b に対するメモリロードを最小とする．

加算処理の効率化 NEON における加算命令では，32-bit 毎のキャリー $\mathtt{C}_*$ とボディ $\mathtt{B}_*$ に分割し，64-bit 加算命令を使用することでオーバーフローを防止した．また最終的なキャリー $\mathtt{C}_*$ とボディ $\mathtt{B}_*$ の加算処理は，キャリービットが利用可能な ARM アセンブラで計算し，効率的な加算処理を構成した．

出力 c のメモリロード，ストアの削減 Comba 法と同様に c_* を下位から順次計算することで，計算中の c_* へのメモリストア，ロードを削減した．結果として，c_* へのアクセスはメモリストア $2N$ 回となった．

5 実験結果

本節では，Android 端末および PC 上におけるペアリング暗号および関数型暗号の評価結果を示す．表 1, 2 に実験で使用する Android 端末と PC のスペックを示す．

本稿での計算時間の測定は Android NDK 上で行い，C やアセンブリと Java の連携によるオーバヘッドは考

表 3: 素体 $\mathbb{F}_{p254}$ 上での多倍長乗算の計算時間 (μsec)

Function	PC	Android		
	Assembly	C	Assembly	NEON
Multiplication	—	0.930	0.335	0.113

表 4: ペアリング暗号の主要な関数の計算時間 (msec)

Function	PC	Android		
	Assembly	C	Assembly	NEON
Pairing	0.85	30.70	11.38	6.77
Point Mult. in $E(\mathbb{F}_p)$	0.57	12.74	5.03	3.04
Point Mult. in $E'(\mathbb{F}_{p^2})$	0.92	25.89	9.93	5.84
Exp. on $\mathbb{F}_{p^{12}}$	1.25	47.29	17.62	9.99

表 5: 関数型暗号での Setup, Keygen, Encrypt, Decrypt の計算時間 (msec)

Function	PC	Android		
	Assembly	C	Assembly	NEON
Setup	473.1	12,941.5	5,133.3	3,231.7
Keygen	149.6	4,315.9	1,636.9	989.8
Encrypt	200.8	4,647.7	1,831.7	1,140.1
Decrypt	72.6	2,638.8	980.1	584.5

* 属性数：$d = 5$, 属性ベクトルの次元：$n_{d_i} = 2$, 述語：AND 10 個，平文サイズ：10-byte，共通鍵暗号 Alg.：AES-128-CBC，ハッシュ関数 Alg.：SHA-256

慮しなかった．これは，実際に利用する場合は，暗号化，復号という単位でのリクエストが主で，連携部分は十分に無視できると考えられるためである．

5.1 ペアリング暗号の評価結果

ペアリングの構成には，Beuchat ら [7] と同様で，126-bit 安全な素体 $\mathbb{F}_{p254}$ における BN 曲線上の Optimal Ate ペアリングを利用した．C, アセンブリ, NEON での実装範囲は，素体 $\mathbb{F}_{p254}$ 上の加減算，2 倍算，乗算，剰余算であり，他の演算は NTT ペアリング演算ライブラリを使用した．剰余算では，モンゴメリ乗算を利用し，内部で提案アルゴリズムを用いている．

素体 $\mathbb{F}_{p254}$ 上の乗算結果を表 3 に示す．本稿の素体 $\mathbb{F}_{p254}$ 上の乗算では，C，アセンブリでの実装に素朴な乗算手法を，また NEON での実装に提案アルゴリズムを使用した．結果として，NEON による素体 $\mathbb{F}_{p254}$ 上の乗算は，C との比較で約 8.2 倍，アセンブリとの比較で約 3.0 倍高速化された．

ペアリング暗号で利用される主要な関数である，ペアリングの計算，$E(\mathbb{F}_p)$，$E'(\mathbb{F}_{p^2})$ 上のスカラ倍算，および $\mathbb{F}_{p^{12}}$ 上のべき乗算の計算結果を表 4 に示す．ペアリングの計算では，素体 $\mathbb{F}_{p254}$ 上の乗算の計算量が大きいため，NEON での計算時間は約 6.77 ミリ秒と，C と比較して 4.5 倍程度高速となった．また他の演算でも 4 倍以上高速となり，主要な関数の全てで高速化された．PC と Android 端末の比較では，アセンブリで比べると各演算で 10 倍以上の差があるが，NEON での高速化により各演算の差が 6〜8 倍まで抑えられた．

5.2 関数型暗号の評価結果

関数型暗号は，鍵識別子 id_k，受信者識別子 id_r，関係 R に対して，$R(id_k, id_r) = \text{True}$ であるときに復号可能な方式であり，柔軟な復号制御が可能である．述語に利用するロジックは AND，OR, NOT, 大小比較などが記述できる．関数型暗号は Setup, Keygen, Encrypt, Decrypt の 4 個のアルゴリズムから構成され，これらの計算には，属性数や，属性ベクトルの次元，述語などのパラメータと，ペアリング，$E(\mathbb{F}_p)$，$E'(\mathbb{F}_{p^2})$ 上のスカラ倍算，および $\mathbb{F}_{p^{12}}$ 上のべき乗算を必要とする．

今回，ペアリング暗号の応用例として，Android 端末上で関数型暗号の計算時間を評価した．関数型暗号の計算には NTT インテリジェント暗号ライブラリを用いた．評価実験の設定として，属性数 $d = 5$, ベクトルの次元 $n_{d_i} = 2$，平文サイズ 10-byte とし，述語には AND を 10 個連結したものを用いた．また共通鍵暗号アルゴリズムとハッシュ関数はそれぞれ AES-128-CBC，SHA-256 を利用した．関数型暗号は吉田ら [19] が提案した暗号文ポリシ方式を用いる．この設定による Setup, Keygen,

Encrypt, Decrypt の計算時間を表 5 に示す.

Andoroid 端末上 NEON による実装で，Setup は約 3 秒であり，これは逆行列の計算が主である．実際の利用時には，Setup および Keygen は初期化にのみ利用されるため，これらはこの程度の計算時間でも許容可能と考えられる．また Encrypt, Decyrpt はそれぞれ約 1.1 秒，約 0.6 秒であり，暗号化，復号の両方で，1 秒程度と十分に実用的な計算時間を示すことができた.

6 おわりに

本稿では，Android 端末上でのペアリング暗号の高速化を目的として，NEON と呼ばれる ARMv7 アーキテクチャでサポートされる SIMD 命令を利用した高速実装を示した．BN 曲線上の Optimal Ate ペアリングでは，素体 $\mathbb{F}_p$ 上の乗算が支配的な計算量であるため，我々は，NEON を利用することで素体 $\mathbb{F}_p$ 上の高速な乗算アルゴリズムを構成した.

提案手法では，NEON 乗算命令により乗算命令の呼出回数を 1/2 に削減し，Comba 法と同様に乗算結果を下位から計算することによりメモリロード，ストアを削減した．また加算処理は，NEON での 64-bit 加算命令と ARM アセンブリでのキャリー有 32-bit 加算命令を併用することで効率的に計算した．結果として，NEON を用いた素体 $\mathbb{F}_p$ 上の乗算は C での実装より約 8.2 倍高速化された.

また提案手法を 126-bit セキュリティの安全性における BN 曲線上の Optimal Ate ペアリングへ適応し，暗号システム構築での主要な関数および関数型暗号の計算時間を評価した．結果，ペアリングの計算，$E(\mathbb{F}_p)$ 上のスカラ倍算，$E'(\mathbb{F}_{p^2})$ 上のスカラ倍算，および $\mathbb{F}_{p^{12}}$ 上のべき乗算で，それぞれ約 6.77 ミリ秒，3.04 ミリ秒，5.84 ミリ秒，9.99 ミリ秒であり，また関数型暗号における暗号化，復号は約 1.1 秒，0.6 秒であった．本稿の実験結果より，Android 端末上でのペアリング暗号や関数型暗号は十分に実用的な計算時間で利用可能であることを示した.

参考文献

[1] Android Developers. http://developer.android.com/index.html.

[2] D.F. Aranha, K. Karabina, P. Longa, C.H. Gebotys, J.López, “Faster Explicit Formulas for Computing Pairings over Ordinary Curves,” EUROCRYPT 2011, LNCS 6632, pp. 48–68, 2011.

[3] ARM, “Cortex-A9 NEON Media Processing Engine Technical Reference Manual,” 2012. Available at http://infocenter.arm.com/help/topic/com.arm.doc.ddi0409i/DDI0409I_cortex_a9_neon_mpe_r4p1_trm.pdf

[4] ARM, “Cortex-A9 Technical Reference Manual,” 2012. Available at http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388i/DDI0388I_cortex_a9_r4p1_trm.pdf

[5] ARM, “RealView Compilation Tools Assembler Guide,” Version 4.0, 2010. Available at http://infocenter.arm.com/help/topic/com.arm.doc.dui0204j/DUI0204J_rvct_assembler_guide.pdf

[6] P.S.L.M. Barreto, M. Naehrig, “Pairing-Friendly Elliptic Curves of Prime Order,” SAC 2005, LNCS 3897, pp. 319–331, 2006.

[7] J.-L. Beuchat, J.E. González-Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, T. Teruya, “High-speed Software Implementation of the Optimal Ate Pairing over Barreto-Naehrig Curves,” Pairing 2010, LNCS 6487, pp. 21–39, 2010.

[8] D. Boneh, M. Franklin, “Identity-based encryption from the Weil pairing,” SIAM Journal on Computing, vol. 32, no. 3, pp. 586–615, 2003.

[9] Intel Corporation, “ストリーミング SIMD 拡張命令 2 (SSE2) を使用した大数の乗算の実行,” 2000. Avairable at http://www.intel.co.jp/content/dam/www/public/ijkk/jp/ja/documents/developer/w_big_mul_j.pdf

[10] 井山政志, 福島和英, 清本晋作, 三宅優, 高木剛, “Android 携帯電話におけるペアリングの高速実装,” SCIS 2012, 1B1-5, 2012.

[11] 井山政志, 清本晋作, 福島和英, 田中俊昭, 高木剛, “携帯電話におけるペアリング暗号の実装,” 電子情報通信学会論文誌, vol. J95-A, no. 7, pp. 579–587, 2012.

[12] V.S. Miller, “Short Programs for Functions on Curves,” Unpublished manuscript, 1986.

[13] 光成滋生, 照屋唯紀, 岡本栄司, “BN 曲線上の Optimal Ate ペアリングのソフトウェア実装,” SCIS 2012, 1B1-4, 2012.

[14] 森佑樹, 根角健太, 野上保之, “BN 曲線を用いたペアリングの NTL による iPhone 実装,” SCIS 2012, 1B1-2, 2012.

[15] T. Okamoto, K. Takashima, “Fully Secure Functional Encryption with General Relations from the Decisional Linear Assumption,” CRYPTO 2010, LNCS 6223, pp.191–208, 2010.

[16] 境隆一, 大岸聖史, 笠原正雄, “楕円曲線上のペアリングを用いた暗号方式,” SCIS 2001, 7B-2, 2001.

[17] 高橋克巳, 星野文学, 小林鉄太郎, 山本剛, 山本具英, 宮澤俊之, 吉田麗生, 冨士仁, 横森正利, 永井彰, “関数型暗号: 実装の現在と実用化への展望,” SCIS 2012, 1B1-3, 2012.

[18] F. Vercauteren, “Optimal Pairings,” IEEE Transactions on Information Theory, vol. 56, no. 1, pp. 455–461, 2010.

[19] 吉田麗生, 星野文学, 小林鉄太郎, “Fully Secure Ciphertext Policy Functional Encryption with Fast Decryption,” SCIS 2013, 2013.