SCIS 2002 The 2002 Symposium on Cryptography and Information Security
Shirahama,Japan, Jan.29-Feb.1,2002
The Institute of Electronics,
Information and Communication Engineers

Batch 検証と大規模冪乗アルゴリズムについて
Batch Verification and Multiple Exponentiation Algorithm

星野 文学* 阿部 正幸 *
Fumitaka Hoshino Masayuki Abe

あらまし Multiple Exponentiation は署名, ゼロ知識証明 あるいは 暗号文のような暗号アイテムの検証コストを大幅に削減する Batch 検証等を実現する為のアルゴリズムである. 本論文では現在 楕円離散対数ベースの暗号として利用可能な最も高速なソフトウェア実装の一つである 61 bit OEF [1] 上で Multiple Exponentiation を実装結果を報告し, Batch 検証の有意性について検討する. また同じ群上で Domain Test を実装し Strict Batch Verification [2] の有意性についても考察する. また 61 bit OEF のような高速実装技術をを出来るだけ使って, Batch 検証, あるいは [3] のようなプロトコルが, 現時点でどのレベルの規模まで実現可能か検討する.

キーワード Batch Verification, Multiple Exponentiation Algorithm

1 はじめに

近年, 電子現金, 電子投票, ネットワークゲームといった様々なアプリケーションを実現する為の大規模な暗号プロトコルがしばしば提案されている. これらの暗号プロトコルでは, 署名, ゼロ知識証明 あるいは 暗号文のような幾つかの暗号アイテムの組み合わせが沢山のクライアントから少数のサーバに対して送信され, サーバは送信内容に関係する何らかのサービスを実行するような, クライアント-サーバ型のモデルがしばしば利用される. こうしたクライアント-サーバ型のプロトコルに大規模な数のクライアントが参加するとサーバに負荷が一局集中してしまうことがある.

例えば日本の国勢選挙の場合, 東京都の世田谷区には有権者数 50 万人規模の開票区が存在している. この規模の選挙をネットワーク型電子投票で実現する場合, 何らかの形で 50 万規模の署名やゼロ知識証明の検証をサーバで行う事となり, 離散対数ベースでプロトコルを設計するならこの検証を行うのに 100 万回程度の冪乗が必要となる. Batch 検証はこのような大規模な検証処理を行う時にサーバの負荷を削減する有効な道具である.

さらに最近, 全体検証可能匿名通信路の研究に於いて大量のアイテムに対して一括してゼロ知識証明を行うようなプロトコルが幾つか提案されている. [3, 4] このようなプロトコルは離散対数ベースの Batch 検証と良く似た構造を持っており, どちらも Multiple Exponentiation と呼ばれる手続きを実行する.

本論文では, 現在 楕円離散対数ベースの暗号として利用可能な最も高速なソフトウェア実装の一つである 61 bit OEF [1] を用いて Multiple Exponentiation を実装し, Batch 検証がどのくらい有意に働くかを検証する. またこの群に関して 文献 [2] で提案された Strict Batch Verification に必要な Domain Test も実装し Strict Batch Verification の有意性検証も行い, これらの Batch 検証あるいは [3] のようなプロトコルが, 現時点でどのレベルの規模まで実現可能か検討する.

2 Batch 検証に関して

離散対数ベースの暗号アイテムに関する Batch 検証は次の 2 つに分けて説明する.

Lenient Batch Verification

- 従来の Batch 検証法 [5, 6, 7, 8, 9]
- 現実的な群に適用した場合必ずしも基の暗号アイテムを保証しない [9]

Strict Batch Verification

- 基の暗号アイテムを高い確率で保証する [2]

2.1 Separate Verification

g を位数 q の巡回群の生成元とする. $i \in \{1, \ldots, n\}$ に関する暗号アイテム列

$$I_i^k = (e_{i1}, \ldots, e_{ik}, h_{i0}, h_{i1}, \ldots, h_{ik}) \in Z_q^k \times \langle g \rangle^{k+1}$$

* NTT 情報流通プラットフォーム研究所 情報セキュリティプロジェクト, 〒 239-0847 神奈川県 横須賀市 光の丘 1-1 NTT Information Sharing Platform Laboratories, NTT Corporation, 1-1 Hikarino-oka, Yokosuka-shi, Kanagawa-ken, 239-0847 Japan

が与えられた場合, Separate Verification とは $i \in \{1,\ldots,n\}$ に関して,

$$\text{Domain test:} \quad h_{ij} \stackrel{?}{\in} \langle g \rangle \quad \text{for } 1 \le j \le k$$

$$\text{Separate relation test:} \quad h_{i0} \stackrel{?}{=} \prod_{j=1}^{k} h_{ij}^{e_{ij}}$$

を検証することを云う. 最近 1 アイテムの基底の数が十分小さい時, Separate Verification を従来考えられていたよりも効率良く行える事が分かった. [2]

2.2 Lenient Batch Verification

$h_{ik_x},\ldots,h_{ik}$ が全ての i に関して共通であるとし, 残りの基底は共通ではないとする. Separate Verification に対して Lenient Batch Verification とは 乱数列 r_i に関して,

$$\text{Domain test:} \quad h_{ij} \stackrel{?}{\in} \langle g \rangle \quad \text{for} \quad \begin{matrix} j \in \{1,\ldots,k\}, \\ i \in \{1,\ldots,n\} \end{matrix}$$

$$\text{Batch relation test:} \quad 1 \stackrel{?}{=} \left(\prod_{i=1}^{n} \prod_{j=0}^{k_x-1} h_{ij}^{r_i e_{ij}} \right) \left(\prod_{j=k_x}^{k} h_j^{\hat{e}_j} \right),$$

を検証することを云う. 共通基底の数 $k_o = k - k_x + 1$ は Batch 検証の効率に大きく影響を与える. [2]

2.3 Strict Batch Verification

上記 Lenient Batch Verification は現実的な群に適用した場合, 必ずしも基の暗号アイテムを保証しない事, 及び暗号アイテムが署名のみで構成されるような場合なら, 群を選べば Lenient Batch Verification でも十分実用的である事が知られている. [9] しかしながら, 暗号アイテムはいつでも必ず署名であるとは限らない. そのような場合は以下の Strict Batch Verification を用いると良い. [2] Strict Batch Verification とは 乱数列 r_i に関して,

$$\text{Domain test:} \quad h_{ij} \stackrel{?}{\in} \langle g \rangle \quad \text{for} \quad \begin{matrix} j \in \{0,\ldots,k\}, \\ i \in \{1,\ldots,n\} \end{matrix}$$

$$\text{Batch relation test:} \quad 1 \stackrel{?}{=} \left(\prod_{i=1}^{n} \prod_{j=0}^{k_x-1} h_{ij}^{r_i e_{ij}} \right) \left(\prod_{j=k_x}^{k} h_j^{\hat{e}_j} \right),$$

を検証することを云う. Lenient Batch Verification との違いは, Domain test の j の添字が 0 から始まっている事である. 例えば $\langle g \rangle$ が何らかの一般的な群の部分群である場合は Domain test が 1 回増えることは, 冪乗が 1 回増えることに相当し, Strict Batch Verification は必ずしも Separate Verification より高速であるとは云えない. しかしながら, 多くの実用的な群で Strict Batch Verification が実効的にに働く事が示されている. [2]

3 Multiple Exponentiation

Batch 検証あるいは [3] のようなプロトコルでは次のような冪乗演算が必要とされる.

$$\prod_{i=1}^{n} g_i^{e_i}$$

本論文ではこのような演算を Multiple Exponentiation と呼ぶことにする. n が大きい場合, この演算を実行するには, 冪を一つづつ計算してかけるよりもずっと高速な方法が存在する. 以下に良く知られているものを挙げる.

3.1 BME

本論文では冪乗アルゴリズムの Binary 法の拡張にあたる以下の Multiple Exponentiation の計算方法を Binary Multiple Exponentiation あるいは BME と呼ぶことにする. この方法は Multiple Exponentiation の基本となる.

名称: BME
入力: 群要素 $g_1,\ldots,g_n$ 及び非負整数 $e_1,\ldots,e_n$
出力: $g_1^{e_1}\cdots g_n^{e_n}$
設定: e_i の 2 進表現 e_{ij} を, $e_i = \sum 2^j e_{ij}$, $e_{ij} \in \{0,1\}$ として定義し, $t = \lfloor \log_2 \max e_i \rfloor$ とする. `mul`,`sqr` はそれぞれ群演算, 自乗 1 回分の演算コストとする.
処理: Step 1: $A \leftarrow 1$
Step 2: $j = t$ から 0 まで以下を繰り返す
Step 2.1: $A \leftarrow A^2$
Step 2.2: $i = 1$ から n まで以下を繰り返す
Step 2.2.1: もし $e_{ij} = 1$ なら $A \leftarrow A \cdot g_i$
Step 3: A を出力する.
平均演算コスト:
$\frac{1}{2}nt \times \texttt{mul} + t \times \texttt{sqr}$
記憶領域コスト:
$1 \times$ 群要素サイズ + 入力

BME が冪乗を n 回繰り返すよりずっと高速に実行できるのは, 平均演算コストの `sqr` の係数が n に依らずに一定だからである. 冪乗アルゴリズムを square & multiply をベースにして設計する場合 `mul` の係数を $\frac{1}{2}$ より改善する generic な方法は良く研究されており, 比較的簡単に行なうことが出来る. 一方 `sqr` の係数を改善するのは一般には難しい. 従って単なる冪乗の場合は generic な方法で `mul` の係数をいくら改善しても `sqr` の方が律速段階となり全体の速度の改善に継らない.
ところが Multiple Exponentiation のような特殊な状況設定を用いると, 冪一個当たりの `sqr` の係数を小さくできる. このような特殊な状況設定は

- 固定基底の場合 [10]
- Multiple Exponentiation の場合

– 特殊な群を用いる場合 [11, 12]
– 高速離散対数検証法を用いる場合 [13]

等が挙げられる. 一般に sqr の係数を無視し得るほど小さくできるなら, mul の係数の改善で直接全体の速度を改善できる. Multiple Exponentiation の場合, 冪 1 個あたりの sqr の係数を $1/n$ と見做すことが出来るので, n を大きく取れば, sqr の係数は無視できる. 従って mul の係数の改善によって全体の速度の向上が期待できる.

3.2 HAC 14.88

文献 [10] pp.617–618 記載のアルゴリズム 14.88 (以降 HAC 14.88) は Multiple Exponentiation を効率的に行う方法として, 多くの論文で参照されている.

名称: HAC 14.88
入力: 群要素 $g_1, \ldots, g_n$ 及び非負整数 $e_1, \ldots, e_n$
出力: $g_1^{e_1} \cdots g_n^{e_n}$
処理: 文献 [10] を参照.
平均演算コスト:
$t \times (\mathtt{sqr} + \frac{2^n-1}{2^n}\mathtt{mul}) + (2^n - n - 1) \times \mathtt{mul}$
記憶領域コスト:
$(2^n - n) \times$ 群要素サイズ $+$ 入力

HAC 14.88 は基底の数 n に対して, 事前演算テーブルの作成及び記憶領域のコストが およそ 2^n となる. 2^n 個のテーブルエントリーが t 回の square & multiply の間に何度も使われる為には最低でも $2^n < t$ が必要であり, 従って n は $n < \log_2 t$ でなければならない. ゆえに HAC 14.88 は n が大きい場合には適用できない.

名称: HAC 14.88 改 (自明な改良)
入力: 群要素 $g_1, \ldots, g_n$ 及び非負整数 $e_1, \ldots, e_n$
出力: $g_1^{e_1} \cdots g_n^{e_n}$
処理: Step 1:
$j \in \{0, 1, \ldots, \lceil \frac{n}{m} \rceil - 1\}, i_1, \ldots, i_n \in \{0, 1\}$ の全パターンに関して事前演算テーブル

$$G[j][i_1] \cdots [i_n] = g_{mj+1}^{i_1} g_{mj+2}^{i_2} \cdots g_{mj+m}^{i_m}$$

を用意する.
Step 2: $A \leftarrow 1$
Step 3: $i = t$ から 0 まで以下を繰り返す
Step 3.1: $A \leftarrow A^2$
Step 3.2: $j = 0$ から $\lceil \frac{n}{m} \rceil - 1$ まで $A \leftarrow A \cdot G[j][e_{mj+1,i}][e_{mj+2,i}] \cdots [e_{mj+m,i}]$ を繰り返す
Step 4: A を出力する.
平均演算コスト:
$\lceil n/m \rceil (\frac{2^m-1}{2^m} t + 2^m - m - 1) \times \mathtt{mul} + t \times \mathtt{sqr}$
記憶領域コスト:
$(\lceil n/m \rceil \times (2^m - m - 1) + 1) \times$ 群要素サイズ $+$ 入力

HAC 14.88 の自明な改良として, 以下の方法がある. テーブルサイズを決めるパラメタ m を n と独立に $1 < m < \log_2 t$ の範囲で設定し, square & multiply の multiply を n/m 回に分割する方法である. 本論文ではこのアルゴリズムを HAC 14.88 改 と呼ぶことにする.
平均演算コストに関して, mul の係数は Binary 法の 1/2 に比べて HAC 14.88 改では $(2^m - 1)/m2^m$ に改善されている. しかしながら記憶領域コストがおよそ $\lceil n/m \rceil \times 2^m \times$ 群要素サイズ であり, n が大きいときは記憶領域コストが大きくなりすぎる為 m の値を大きく出来ない場合がある. 例えば典型的な 1024 bit = 128 Byte の乗法群を想定し $n = 10^6$ でこの記憶領域コストを見積もるなら, 入力群要素だけで 128 MByte 必要で, $m = 5$ とするとテーブルのサイズは 819 MByte 必要となる. 現状利用可能な安価な 32 bit 計算機の主記憶で実現するのは不可能ではないが無視できない大きさの記憶領域コストとなる.

4 実装の詳細

4.1 演算スケジュールの変更

名称: 提案法
入力: 群要素 $g_1, \ldots, g_n$ 及び非負整数 $e_1, \ldots, e_n$
出力: $g_1^{e_1} \cdots g_n^{e_n}$
設定: m は $m < \log_2 t$ を満たす十分小さい正整数とし, $j > n$ の時は $g_j = 1, e_j = 0$ とする.
処理: Step 1: $i \leftarrow 0$ とし $j \in \{0, 1, \ldots, t\}$ に関して $A_j \leftarrow 1$ とする.
Step 2: $i < n$ なら以下を実行.
Step 2.1: $i_1, \ldots, i_m \in \{0, 1\}$ の全パターンに関して事前演算テーブル

$$G[i_1][i_2] \cdots [i_m] = g_{i+1}^{i_1} g_{i+2}^{i_2} \cdots g_{i+m}^{i_m}$$

を用意する.
Step 2.2: $j \in \{0, \ldots, t\}$ に関して $A_j \leftarrow A_j \times G[e_{i+1,j}] \cdots [e_{i+m,j}]$ とする.
Step 2.3: 事前演算テーブル G を破棄する.
Step 2.4: $i \leftarrow i + m$ とし Step 2 へ.
Step 3: $A \leftarrow 1$ とする.
Step 4: $j = t$ から 0 まで以下を繰り返す.
Step 4.1: $A \leftarrow A^2$ とする.
Step 4.2: $A \leftarrow A \cdot A_j$ とする.
Step 5: A を出力する.
平均演算コスト:
$\lceil n/m \rceil \times (t \times \frac{2^m-1}{2^m}\mathtt{mul} + (2^m - m - 1) \times \mathtt{mul}) + t \times \mathtt{sqr}$
記憶領域コスト:
$(2^m - m + t) \times$ 群要素サイズ $+$ 入力

HAC 14.88 改を使う代わりに 単に HAC 14.88 を n/m 回呼ぶとすれば, 記憶領域コストは HAC 14.88 を m 入力で使うのと同じで済むはずである. 従って, HAC 14.88 改のような大きな記憶領域コストはテーブル参照法で必ず必要なコストではない. 但し HAC 14.88 を n/m 回呼ぶのでは, 平均演算コストの `sqr` の係数が n/m に比例して大きくなってしまう. 仮に `sqr` $\sim$ `mul` とするなら, これは HAC 14.88 改 よりも 2 倍の平均演算コストを必要とする. これを避けるにはテーブル参照中の自乗演算を止めれば良い. 即ち提案法の様に演算スケジュールを変更する.

5 実装結果

以下の環境等にて冪乗, BME, HAC 14.88 改, 提案法 をそれぞれ実装し, $n = 10^3 \sim 10^6$ の Multiple Exponentiation を計算するのに必要な CPU 時間を計測した. それから使用した群の Domain Test に必要な CPU 時間も測定した.

ハード: COMPAQ 21264 500MHz Memory 512MByte
OS: OSF1 V4.0 1091 alpha
言語: DEC C V5.8-011
群: OEF($2^{61}-1$ の 3 次拡大,(位数 183 bit)) [1]

表 1: 実装結果 (単位 sec)

n	冪乗	BME	提 1	改 5	提 5	DOM
10^3	0.505	0.190	0.185	0.092	0.087	—
10^4	5.050	1.896	1.862	0.940	0.876	0.030
10^5	50.25	21.36	18.42	10.63	8.621	0.133
10^6	502.7	215.9	183.7	—	86.34	1.048

冪乗 : 冪乗 (Binary 法) n 回
BME : Binary Multiple Exponentiation
提 1 : 提案法 $m = 1$ の場合
改 5 : HAC 14.88 改 $m = 5$ の場合
提 5 : 提案法 $m = 5$ の場合
DOM : Domain Test n 回

表 1 には次の着目すべきポイントがある.

- 今回実装した $n \gg t \sim 160$ の領域では n 個の冪乗と n 個の Multiple Exponentiation の演算コストは桁が変る程度の差はある. (「提 5」と「冪乗」の比較)
- 今回実装した群では Multiple Exponentiation に比べて Domain Test の演算コストは殆んど無視できる.
- BME や HAC 14.88 改 の $n = 10^5 \sim 10^6$ における速度が $n = 10^3 \sim 10^4$ から予想される値よりもやや遅い.
- 提案法では $n = 10^5 \sim 10^6$ における速度が $n = 10^3 \sim 10^4$ から予想される値とほとんど矛盾が無い.

$n = 10^3 \sim 10^4$ では BME が冪乗より, 2.66 倍程度高速であり, `sqr` と `mul` の演算コスト等の違いを考えると妥当な値と考えられる. $n = 10^5 \sim 10^6$ では BME は $n = 10^5 \sim 10^6$ から予想される値よりもやや遅い. 原因は BME における群の元に対するメモリアクセスが局所化されていない事に依ると思われる.
現状では 500MHz で動作しているような CPU は, 主記憶よりもずっと早いタイミングで反応するメモリキャッシュを装備している事が多く, 今回実験に使用した 21264 も例外ではない. そのような場合は 1 度キャッシュに読み込まれた群の元を読み込んだタイミングで何度も使い回せばそれだけ高速に動作させることが出来るはずである. 試しに提案法で $m = 1$ の場合 (提案法の Binary Method 版) を計測してみると, $n = 10^3 \sim 10^6$ の間でほぼ矛盾の無い値になっている. 此の事は BME が必ずしも大規模冪乗アルゴリズムに向いている訳ではない事を意味している. 今回は特には実装していないが, 入力の記憶階層が主記憶よりずっと遅い外部記憶装置に代われば BME と提案法の違いは歴然とするはずである.
今回使用した群で HAC 14.88 改 $m = 5$ の場合は注意深く実装を行えば主記憶 512 MByte で $n = 10^6$ を実現可能と考えられるが, 今回は実装の都合により, テーブルを含めてデータセグメントを 512 MByte に納める事が出来ず, 実行不可能となってしまった.
Domain Test で $n = 10^3$ の場合は実行時間が計測誤差の範囲に埋もれてしまい有意な値が得られなかった.

6 まとめ

本論文では以下の事を行った

- 61 bit OEF 上で Multiple Exponentiation を実装し同じ基底数の冪乗に比べて Multiple Exponentiation が桁違いに高速であることを示した. これにより, Lenient Batch Verification は Separate Verification よりずっと高速に実行可能であることが分かる.
- 61 bit OEF 上で Domain Test を実装し同じ数 (基底数) の Multiple Exponentiation 比べて Domain Test が桁違いに高速であることを示した. これにより, Strict Batch Verification は Lenient Batch Verification と比べて実行効率の面でほとんど遜色が無い事が分かる.
- 大規模冪乗アルゴリズムとして従来よく紹介されていた BME や HAC 14.88 改 は大規模冪乗アルゴリズムに必ずしも向いている訳ではない事を示し, 演算スケジュールの変更でこれらのアルゴリズムの欠点を解決できることを示した.

— 今回の実装結果より, $n = 10^6$ の規模のアプリケーション, 例えば有権者数 100 万人 規模のネットワーク型電子投票の即日開票等が計算量的には実現可能であることを示した. [3]

参考文献

[1] 青木和麻呂, 小林鉄太郎, 星野文学: 楕円曲線暗号の最高速実装. In: SCIS2000. (2000) SCIS2000–B05

[2] Hoshino, F., Abe, M., Kobayashi, T.: Lenient/strict batch verification in several groups. In Davida, G., Frankel, Y., eds.: Information Security Conference2001. Volume 2200 of Lecture Notes in Computer Science., Berlin; Heidelberg; New York, Springer-Verlag (2001) 81–94

[3] 千田浩司, 星野文学: 事前処理を仮定した mix-net の高速検証方式. In: SCIS2002. (2002) –

[4] Furukawa, J., Sako, K.: An efficient scheme for proving a shuffle. In: Advances in Cryptology — CRYPTO2001. Volume 2139 of Lecture Notes in Computer Science., Berlin; Heidelberg; New York, Springer-Verlag (2001) 368–387

[5] Sako, K., Kilian, J.: Receipt-Free Mix-Type Voting Scheme — A practical solution to the implementation of a voting booth —. In Guillou, L.C., Quisquater, J.J., eds.: Advances in Cryptology — EUROCRYPT'95. Volume 921 of Lecture Notes in Computer Science., Berlin; Heidelberg; New York, Springer-Verlag (1995) 393–403

[6] Naccache, D., M'Raïhi, D., Vaudenay, S., Raphaeli, D.: Can D.S.A be Improved ? - Complexity Trade-Offs with the Digital Signature Standard -. In Santis, A.D., ed.: Advances in Cryptology — EUROCRYPT'94. Volume 950 of Lecture Notes in Computer Science., Berlin; Heidelberg; New York, Springer-Verlag (1995) 77–85

[7] Sung-Ming Yen, Chi-Sung Laih: Improved Digital Signature Suitable for Batch Verification. IEEE Transactions on Computers **44** (July 1995) 957–959

[8] Bellare, M., Garay, J.A., Rabin, T.: Fast Batch Verification for Modular Exponentiation and Digital Signatures. In Nyberg, K., ed.: Advances in Cryptology — EUROCRYPT'98. Volume 1403 of Lecture Notes in Computer Science., Berlin; Heidelberg; New York, Springer-Verlag (1998) 236–250

[9] Boyd, C., Pavlovski, C.: Attacking and Repairing Batch Verification Schemes. In Okamoto, T., ed.: Advances in Cryptology — ASIACRYPT2000. Volume 1976 of Lecture Notes in Computer Science., Berlin; Heidelberg; New York, Springer-Verlag (2000) 58–71

[10] Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of applied cryptography. CRC Press (1997)

[11] Koblitz, N.: CM-Curves with Good Cryptographic Properties. In Feigenbaum, J., ed.: Advances in Cryptology — CRYPTO'91. Volume 576 of Lecture Notes in Computer Science., Berlin; Heidelberg; New York, Springer-Verlag (1992) 279–287

[12] Kobayashi, T., Morita, H., Kobayashi, K., Hoshino, F.: Fast Elliptic Curve Algorithm Combining Frobenius Map and Table Reference to Adapt to Higher Characteristic. In Stern, J., ed.: Advances in Cryptology — EUROCRYPT'99. Volume 1592 of Lecture Notes in Computer Science., Berlin; Heidelberg; New York, Springer-Verlag (1999) 176–189 (A preliminary version was written in Japanese and presented at SCIS'99-W4-1.4).

[13] 小林鉄太郎, 星野文学: 離散対数検証法. In: SCIS2002. (2002) –