ゲノム秘匿検索のbootstrap処理における適切な暗号ライブラリの評価

辻　有紗†　　小口　正人†

† お茶の水女子大学　〒 112–8610 東京都文京区大塚 2–1–1
E-mail: †{arisa-t,oguchi}@ogl.is.ocha.ac.jp

あらまし　クラウドでのデータ解析に向けて完全準同型暗号 (以下：FHE) が注目を集めている．FHE は暗号化された状態で任意の演算が可能な暗号方式である．実用に向けた課題点として，時間空間計算量が膨大であることが挙げられる．特に bootstrap と呼ばれる，演算により蓄積する暗号文のノイズを取り除く処理の負荷が大きい．様々な実装 (方式) が提案されており，暗号文の形式や，鍵のサイズ，bootstrap 処理の方法が異なる．本研究では，BFV， BGV，CKKS 方式が実装された openFHE, Lattigo と，TFHE 方式が実装された TFHEpp， FHEW の分析を行う．また，暗号方式により高速化やメモリの効率化の方法が異なる．本研究では，CPU 使用率や実行時の関数の負荷の分析により，SSD の活用に適したライブラリの検討を行う．FHE を用いたシステムとして，ゲノム秘匿検索を対象とする．PBWT と呼ばれるゲノムビックデータを用いて演算を行うシステムである．以前の研究により，HElib ライブラリを用いた場合，並列性が高く，CPU と DRAM 間の load/store 処理がボトルネックであることが示されている．本研究では，その他のライブラリの実行時の状態を分析し，ゲノム秘匿検索の SSD を用いた効率化に適したライブラリを検討する．

キーワード　完全準同型暗号，bootstrap，SSD

1　は じ め に

クラウド上でのビックデータ解析の際にデータの安全性を保証する手段として完全準同型暗号 (以下：FHE) が注目を集めている．FHE とは，データが暗号化された状態で任意の演算を行うことが可能な暗号方式である．従来の暗号方式では暗号化された状態での演算が不可能なため，データをクラウドに保存する際に暗号化を行うが，クラウド上の分析時に暗号化されたデータを一度復号する必要がある．FHE を用いることで，データが復号される必要がなくなり，第三者からの攻撃を防ぐことができる．

FHE が実用に至らない課題点の 1 つに，時間空間計算量が膨大であることが挙げられる．特に bootstrap と呼ばれる処理の負荷が大きい．FHE では格子暗号を用いており，平文に小さいノイズを加えることで暗号文を作成する．また，ある閾値より小さなノイズを取り除くことで平文に復号する．ただし，一定回数以上演算を繰り返す場合，蓄積されたノイズが閾値を超えると正しい値に復号することができない．bootstrap とは暗号化された状態で，複雑な演算を行うために，適切なタイミングで暗号文に蓄積したノイズを取り除く処理を指す．

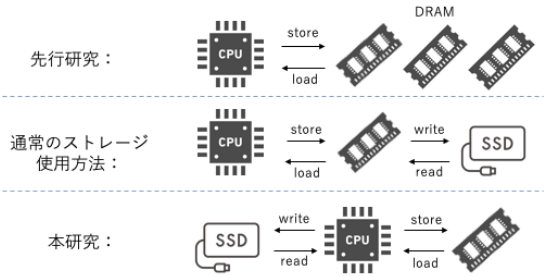

図 1　DRAM・ストレージの利用方法の比較

暗号方式により bootstrap の方法や暗号化の手順が異なる．それぞれの特徴により世代に分けられている．第二世代に当たる暗号方式では，RLWE(Ring Lerning Error) の形に暗号化し，の BFV 方式，BGV 方式で算術演算を行う，bootstrap を行わずに LHE として演算を行うことや SIMD 演算が可能である．HElib [1]， Microsoft SEAL [2]， Lattigo [3] などのライブラリで実装されている．第三世代では新たに TRGSW(Torus Ring-GSW) を用いて bootstrapping key を作成し，Blind Rotation と呼ばれる暗号化された状態で，平文の情報を元にした値を取り出す操作などを用いて bootstrap を高速化した．演算を論理回路として評価を行い，それぞれのゲートで bootstrap を行う．TFHE [4] [5]，FHEW [6]，TFHEpp [7] [8] で実装されている．第四世代では第二世代の方式を拡張し，固定小数点数を扱うことができる．算術演算として SIMD 演算も可能であり，深層学習などに用いられる．HElib, Microsoft SEAL, Lattigo などで実装されている．

暗号方式により CPU 使用率やメモリ使用量が異なるため，高速化やメモリの効率化の方法が異なる．本研究では実行時の状態を分析し，SSD を有効的に活用することが可能な暗号方式を検討する．また，FHE を用いたシステムとして，ゲノム秘匿検索 [9] を対象とする．ゲノムデータベース PBWT と患者のゲノムデータを照合し，病気の原因となる塩基配列の特定

を行うシステムである．HElib ライブラリを用いた場合，並列性が高く，CPU と DRAM 間の load/store 処理がボトルネックであることが示されている．本研究では，その他のライブラリ，TFHEpp， FHEW， openFHE， Lattigo の分析を行い，SSD を用いたゲノム秘匿検索の効率化に適したライブラリを検討する．

2 完全準同型暗号

2.1 特　　徴

準同型暗号とは，データを暗号化した状態で計算し，完了後に復号することで，暗号化せずに計算した場合と同じ結果が得られる暗号方式のことである．(1) を加法準同型暗号，(2) を乗法準同型暗号という．

$Encrypt(m) \oplus Encrypt(n) = Encrypt(m+n)$ ... (1)

$Encrypt(m) \otimes Encrypt(n) = Encrypt(m \times n)$ ... (2)

加法準同型暗号の代表的なものに，合成数剰余判定仮定を用いた Paillier 暗号 [10]，乗法準同型暗号の代表的なものに，大きい値の素因数分解の困難性を用いた RSA 暗号 [11] などがあり， これらは古くから知られている．一方，(1)，(2) 共に満たす暗号方式は，実行可能な演算回数に応じて，SHE(Somewhat Homomorphic Encryption)，LHE(Leveled Homomorphic Encryption)，FHE(完全準同型暗号， Fully Homomorphic Encryption) に分けられる [12] [13] [14]. SHE， FHE は 2009 年， Graig Gentry によって提案された．

SHE は乗算回数を比較的小さい定数回に制限された暗号方式である．Gentry はイデアル格子に基づき，暗号化する際に小さいノイズを加えることで実現した．その後，RLWE 問題に基づくものや GCD 問題に基づくものなど，様々な方法が提案されている．

FHE は任意回，加算・乗算可能な暗号方式であり，SHE と bootstrap を元に構成される．SHE を用いて一定回数以上演算を行う場合，暗号文のノイズ量が閾値を超え，正しく復号できなくなる．ここで，bootstrap と呼ばれる，暗号文に蓄積するノイズを取り除くため複数の鍵を用いて暗号化・復号化を行う処理を行うことで，任意回の暗号文同士の演算が可能となる．ただし，bootstrap には膨大な時間空間計算量が必要であるため，実用に至っていない．

SHE と FHE の中間にあたる，事前に決めた任意回の乗算を可能とする LHE も， Brakskil，Gentry，Vercauteren により提案されている．モジュラススイッチ・鍵スイッチという仕組みにより構成され，乗算によるノイズや暗号文サイズの爆発的な増加を防いでいる．現在は暗号可能関数が比較的大きいことや，時間空間計算量の面から，LHE が利用される場面が多い．

2.2 CRT パッキングと Rotation

BFV，BGV 形式では，CRT パッキングと呼ばれる手法により，平文の集合 $(m_1, m_2, m_3...)$ は m 次円分多項式 $(\phi_m(X))$ 上の 1 つの平文多項式 $(f(x))$ で定義される．図 2 に CRT パッキングと Rotation の仕組みを示す．CRT パッキングは中国剰余定理に基づく方法であり，$f(x) \equiv$m_1 mod$F_1(x)$，$f(x) \equiv m_2 \bmod F_2(x)$，$f(x) \equiv m_3 \bmod F_3(x)$，... $(F_1(x)F_2(x)F_3(x)... = \phi_m(X))$ を満たす $f(x)$ がただ 1 つ存在することを用いて，複数の平文を 1 つの多項式として表現する．多項式内に格納されたそれぞれの平文をスロットと呼ぶ．

また，CRT パッキングされた要素は，ベクトルのような SIMD 演算が可能である．つまり，2 つの平文多項式同士の演算 (足したり掛けること) は，各平文多項式の同じ位置のスロット同士の演算することを意味する．

さらに，平文多項式に automorphism という処理を行うことで，平文多項式内の要素 (ベクトルの要素であるスロット) を回転させることが可能である．この処理を Rotation とよぶ．Rotation と SIMD 演算を組み合わせることで，2 つの暗号文内の任意の位置のスロット同士の演算が可能である．

2.3 LWE，RLWE 問題

FHE の暗号化には LWE, RLWE 問題を用いる．行列 $\mathbf{a}$，$\mathbf{b}$，$\mathbf{s}$，$\mathbf{e}$ が行列であると仮定する．

$$\mathbf{b} = \mathbf{a} \cdot \mathbf{s} + (\mathbf{m} + \mathbf{e})$$

が与えられたとき，$(\mathbf{a}, \mathbf{b})$ のみから $\mathbf{s}$, $\mathbf{m}+\mathbf{e}$ を求める問題を LWE 問題という．この問題を以下の用に秘密鍵を用いる FHE の暗号化の手順に適用することができる．平文空間 (平文がとることができる値の最大値) を p とする．平文を $m \in \mathbb{Z}_p$，秘密鍵を $\mathbf{s} \in \mathbb{B}^n$, ノイズを $e \in \mathcal{D}_{\mathbb{Z}_{p,\alpha}}$, 公開鍵の一部を $\mathbf{a} \in \mathbb{Z}_p^n$ とし，LWE の式から公開鍵の一部を $b \in \mathbb{Z}_p$を求める．ベクトルの長さ n とノイズレベルαを適切に選ぶと暗号文から，平文と秘密鍵を求めることが難しい．

LWE を多項式に拡張した問題を RLWE という．平文を $m[X] \in \mathbb{Z}_{p\mathrm{N}}[X]$, 秘密鍵を $s[X] \in \mathbb{B}^n[X]$, ノイズを $e[X] \in \mathbb{Z}_{p\mathrm{N}}[X]$, 公開鍵の一部を $a[X] \in \mathbb{Z}_{p\mathrm{N}}[X]$ とし，$b[X] = a[X] \cdot s[x] + m[X] + e[X]$ から公開鍵の一部 $b[X]$ を求める．CRT パッキングを用いて， 1 つの暗号文 $\mathbb{Z}_{\mathrm{pN[X]}}$で N 個の平文を表すことで暗号文サイズを削減することができる．

2.4 FHE の暗号方式

エンコーディング，暗号化，乗算など暗号文の主要な処理を以下に示す．

2.5 BFV 方式

2.5.1 エンコーディング

メッセージ $m \in \mathbb{Z}$ を平文空間である多項式環 $\mathcal{M} \in R_t$

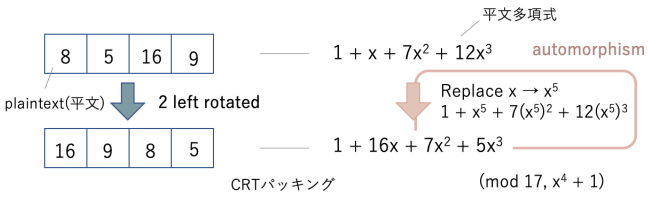

図 2　CRT パッキングと Rotation

に変換する．CRT パッキングを用いてメッセージの集合 $(m_1, m_2, m_3 ... m_N)$ は m 次円分多項式 $\phi_m(X)$ 上の 1 つの平文多項式 $(f(x) \in R_t)$ に変換される．

$$\mathcal{M} = a_0 + a_1 x + a_2 X^2 + a_3 x^3 + ... + a_{n-1} x^{n-1}$$

$(a_0, a_0, ..., a_{n-1} \in [0, t-1])$

演算結果が正しいことを保証するため，平文多項式 $f(x)$ の係数が t を超えて回り込まないように $a_0, ...a_{n-1}$ を設定する必要がある．

2.5.2 鍵の作成

秘密鍵 $\mathrm{SK} \in \mathbb{B}^{\ltimes}$ と公開鍵の一部 $\mathrm{a} \in R_q$ をランダムに作成する．また，ノイズ $e \in R_q$ をガウス分布を元にサンプルする．2 つの多項式からなる公開鍵 PK = (PK1，PK2) は以下のように作成される．

$$\mathrm{PK_1} = [\mathrm{a} \cdot \mathrm{SK} + \mathrm{e}]_\mathrm{q}$$

$$\mathrm{PK_2} = \mathrm{a}$$

PK1, PK2 から SK，e を求めることの困難性は RLWE 問題で保証されており，ノイズ e を調整することで SK の求めやすさが変化する．

2.5.3 暗号化

平文 $\mathcal{M} \in R_t$ を暗号文 $\mathcal{C} \in R_q$ にマッピングする．初めに，u, ノイズ $e1$, ノイズ $e2$ ($\in \mathbb{B}^{\ltimes}$) をランダムに作成する．$\Delta = [q/t]$ とすると，暗号文 $\mathcal{C}$ = (C1,C2) は以下のように表される．

$$\mathrm{C1} = [\mathrm{PK1} \cdot \mathrm{u} + \mathrm{e1} + \Delta\mathcal{M}]_\mathrm{q}$$

$$\mathrm{C2} = [\mathrm{PK2} \cdot \mathrm{u} + \mathrm{e2}]_\mathrm{q}$$

Δ によりメッセージ m は MSB(Most Significant Bit) に配置される．また，$\mathcal{M} = [[\frac{t[\mathrm{C1+C2 \cdot SK}]_\mathrm{q}}{q}]]_t$ により暗号文 C から平文 M に復号可能である．

2.5.4 乗算

暗号文 $\mathcal{C} \in R_q$ は RLWE 問題に基づいており，

$$\mathrm{C1} - \mathrm{SK} \cdot \mathrm{C2} = [\Delta\mathcal{M} + \mathrm{e1} + \mathrm{e2}]_\mathrm{q}$$

が成り立つ．ここで，2 つの暗号文 $\mathrm{C}^{(0)}$，$\mathrm{C}^{(1)}$ の乗算を考える．

$$\mathrm{C1}^{(0)} - \mathrm{SK} \cdot \mathrm{C2}^{(0)} = [\Delta\mathcal{M}^{(0)} + \mathrm{e}_1^{(0)} + \mathrm{e}_2^{(0)}]_\mathrm{q}$$

$$\mathrm{C1}^{(1)} - \mathrm{SK} \cdot \mathrm{C2}^{(1)} = \Delta[\mathrm{M}^{(1)} + \mathrm{e}_1^{(1)} + \mathrm{e}_2^{(1)}]_\mathrm{q}$$

の各辺をかけると，

$$\mathrm{C1}^{(0)}\mathrm{C1}^{(1)} - \mathrm{SK}(\mathrm{C2}^{(0)}\mathrm{C1}^{(1)} + \mathrm{C1}^{(0)}\mathrm{C2}^{(1)}) + \mathrm{SK}^2\mathrm{C2}^{(0)}\mathrm{C2}^{(1)} = [\Delta\mathcal{M}^{(0)}\Delta\mathcal{M}^{(1)} + e^{'}]_q$$

となる．

$\mathrm{D0} = \mathrm{C1}^{(0)}\mathrm{C1}^{(1)}$, $\mathrm{D1} = \mathrm{C2}^{(0)}\mathrm{C1}^{(1)} + \mathrm{C1}^{(0)}\mathrm{C2}^{(1)}$, $\mathrm{D2} = -\mathrm{C2}^{(0)}\mathrm{C2}^{(1)}$ とおくと，

$$\mathrm{D0} - \mathrm{SK} \cdot \mathrm{D1} - \mathrm{SK}^2 \cdot \mathrm{D2} = [\Delta\mathcal{M}^{(1)}\Delta\mathcal{M}^{(2)} + \mathrm{e}^{'}]_\mathrm{q}$$

と表され，(D0,D1,D2) は秘密鍵 $(\mathrm{SK}, \mathrm{SK}^2)$ に関する $\Delta\mathcal{M}^{(1)}\Delta\mathcal{M}^{(2)}$ の暗号文となっている．

2.5.5 Relinearization

乗算後の暗号文 $\mathcal{C}$ は 3 つの多項式 (D0,D1,D2) となり，乗算に伴い暗号文のサイズが増加する．そこで，$[\mathrm{D0} - \mathrm{SK} \cdot \mathrm{D1} - \mathrm{SK}^2 \cdot \mathrm{D2}]_\mathrm{q} \approx [\mathrm{C1}^{(2)} - \mathrm{SKC2}^{(2)} + \mathrm{r}]_\mathrm{q}$ となる $\mathrm{C3}=(\mathrm{C1}^{(2)}, \mathrm{C2}^{(2)})$ を求める．この操作は Relinerization と呼ばれる．ここで，Relinearization では SK^2 を用いるため，公開鍵に SK^2 の暗号文も含める必要がある．SK^2 の暗号文を $\mathrm{EK} = (\mathrm{EK1}, \mathrm{EK2})$ とする．RLWE 問題に基づき，$\mathrm{EK1} - \mathrm{EK2} \cdot \mathrm{SK} = \mathrm{SK}^2 - \mathrm{e}$ を満たす EK は

$$\mathrm{EK1} = \mathbf{a} \cdot \mathrm{SK} - \mathrm{e} + \mathrm{SK}^2$$

$$\mathrm{EK2} = \mathbf{a}$$

と表せる．EK を用いて $[\mathrm{D0} - \mathrm{SKD1} - \mathrm{SK}^2\mathrm{D2}]_\mathrm{q}$ を変形すると

$$\mathrm{C1}^{(2)} = [\mathrm{D0} + \mathrm{EK1} \cdot \mathrm{D2}]_\mathrm{q}$$

$$\mathrm{C2}^{(2)} = [\mathrm{D1} + \mathrm{EK2} \cdot \mathrm{D0}]_\mathrm{q}$$

が求まり，$\mathrm{C3}=(\mathrm{C1}^{(2)}, \mathrm{C2}^{(2)})$ に一致する．

2.6 BGV 方式

BGV 方式では，modulus chain と呼ばれる小さな mod の組 $\mathcal{Q} = p_0, p_1, p_2, ..., p_L$ が定義され，暗号文の modulus q_l は Level に応じて変化する．新たに暗号化された暗号文は Level$L$ で始まり，計算に伴い l から $l-1$ に下がる．

2.6.1 鍵の作成

秘密鍵 $\mathrm{SK} \in \mathbb{B}^{\ltimes}$ と公開鍵の一部 $\mathrm{a} \in R_{q_L}$ をランダムに作成する．また，$e \in R_{q_L}$ をガウス分布を元にサンプルする．2 つの多項式からなる公開鍵 PK=(PK1,PK2) は以下のように作成される．

$$PK1 = [a \cdot \mathrm{SK} + \mathrm{te}]_{\mathrm{q_L}}$$

$$PK2 = a$$

2.6.2 暗号化

平文 $\mathcal{M} \in R_t$ を暗号文 $\mathcal{C} \in R_{q_L}$ にマッピングする．初めに，u, ノイズ $e1$, ノイズ $e2$ ($\in \mathbb{B}^{\ltimes}$) をランダムに作成する．暗号文 C=(C1,C2) は以下のように表される．

$$C1 = [\mathrm{PK1} \cdot \mathrm{u} + \mathrm{e1} + \Delta\mathcal{M}]_{\mathrm{q_L}}$$

$$C2 = [\mathrm{PK2} \cdot \mathrm{u} + \mathrm{e2}]_{\mathrm{q_L}}$$

t によりノイズが MSB(Most Significant Bit)，メッセージが LSB(Least Significant Bit) に配置される．また，$\mathcal{M} = [[\mathrm{C1} + \mathrm{C2} \cdot \mathrm{SK}]_{\mathrm{q_l}}]_\mathrm{t}$ により平文 M に復号される．

2.6.3 乗算

暗号文 C は RLWE 問題に基づいており，

$$\mathrm{C1} - \mathrm{SK} \cdot \mathrm{C2} = [\Delta\mathcal{M} + \mathrm{te1} + \mathrm{te2}]_\mathrm{q}$$

が成り立つ．ここで，2 つの暗号文 $\mathrm{C}^{(0)}$，$\mathrm{C}^{(1)}$ の乗算を考える．

$$\mathrm{C1}^{(0)} - \mathrm{SK} \cdot \mathrm{C2}^{(0)} = [\Delta\mathcal{M}^{(0)} + \mathrm{t} \cdot \mathrm{e}_1^{(0)} + \mathrm{t} \cdot \mathrm{e}_2^{(0)}]_\mathrm{q}$$

$$\mathrm{C1}^{(1)} - \mathrm{SK} \cdot \mathrm{C2}^{(1)} = [\Delta\mathcal{M}^{(1)} + \mathrm{t} \cdot \mathrm{e}_1^{(1)} + \mathrm{t} \cdot \mathrm{e}_2^{(1)}]_\mathrm{q}$$

の各辺をかけると，

$$\mathrm{C1}^{(0)}\mathrm{C1}^{(1)} - \mathrm{SK}(\mathrm{C2}^{(0)}\mathrm{C1}^{(1)} + \mathrm{C1}^{(0)}\mathrm{C2}^{(1)}) + \mathrm{SK}^2 \cdot \mathrm{C2}^{(0)}\mathrm{C2}^{(1)} = [\Delta\mathcal{M}^{(0)}\Delta\mathcal{M}^{(1)} + e'']_q$$

となる．計算に伴いノイズ (e'') が指数的に増加するため，ModSwich と呼ばれる操作でノイズの調整を行う．

2.6.4 Mod Switch

秘密鍵 SK に関して暗号文空間 modulus q で作成された暗号文 C を modulus q' で作成された暗号文 $\mathcal{C}'$ に変換する．

$$[\mathcal{C}(\mathrm{SK})]_\mathrm{q} = [\mathrm{C}' \cdot (\mathrm{SK})]'_\mathrm{q}$$

暗号文 C の多項式の係数を q'/p でスケーリングすることでノイズを削減する．

$$\mathcal{C}' = [q'/q \cdot \mathcal{C}]$$

ただし，q_l は $[q_l = q_{l-1}]_t$ を満たすことで，平文に Mod Switch の影響を与えない必要がある．

＊ エンコーディングは BFV 方式と同様である．

2.7 CKKS 方式

実数の準同型演算が可能な暗号方式である．CKKS 方式ではメッセージ m にノイズ e を含めたものに対して暗号処理が行われる．ノイズがメッセージに対してはるかに小さい場合，一定の精度に関してはメッセージに影響を与えない．スケーリングパラメータ，ノイズに対してメッセージが大きくなるよう設定し，暗号文の Level に対応する modulus q_l を変換しながら演算を行う．CKKS では Rescale と呼ばれる操作で，m とノイズを含む暗号文に対して下位 bit の切り捨てを行う．

2.7.1 エンコーディング

小数を含むベクトルを 1 つの平文の値に変換する．メッセージを含むベクトル $z \in \mathbb{C}^{\frac{n}{2}}$, 平文 $a \in R$, スケーリングパラメータ Δ，フーリエ変換の畳み込み処理を行う関数 π を用いて以下のように表せる．

$$\mathrm{ENCODE(z, \Delta) = [\Delta \cdot ß^{-1}(z)]}$$

$$\mathrm{DECODE(a, \Delta) = ß(\frac{1}{\Delta} \cdot a)}$$

2.7.2 暗　号　化

平文 $\mathcal{M} \in R_t$ から暗号文 $\mathcal{C} \in R_{q_L}$ を求める．初めに，u, ノイズ $e1$, ノイズ $e2$ ($\in \mathbb{B}^\kappa$) をランダムに作成する．暗号文 C=(C1,C2) は以下のように表される．

$$C1 = [\mathrm{PK1 \cdot u + e1} + \mathcal{M}]_{\mathrm{q_L}}$$

$$C2 = [\mathrm{PK2 \cdot u + e2}]_{\mathrm{q_L}}$$

BFV との違いは平文 $\mathcal{M}$ が Δ でスケーリングされていない点である．ノイズとメッセージの間に境界はなく，LSB に配置される．

2.7.3 Rescale

BFV や BGV 方式で用いられる ModSwich と同様に，暗号文の modulus を変換する．CKKS 方式ではスケーリングされた暗号文が乗算により Δ^2 にスケーリングされるため，Rescale を用いて乗算前のスケーリングの割合に変換する．暗号文 $\mathcal{C}^{(0)} \in R^2_{q_l}, \mathcal{C}^{(1)} \in R^2_{q_{l-1}}$, を用いて以下のように表される．

$$\mathrm{RESCALE}(\mathcal{C}^{(0)}, \Delta) = \frac{1}{\Delta} \cdot [\mathrm{C1}^{(0)}, \mathrm{C2}^{(1)}]_{\mathrm{q_l}} = [\mathcal{C}^{(1)}]_{\mathrm{q_{l-1}}}$$

スケーリングによりノイズのΔ分の LSB が削除される．

＊ 鍵の作成，乗算，Relinearinzation は BFV 方式と同様である．

2.8 bootstrap

暗号文に蓄積したノイズを取り除くには, 複合可能な状態で一度暗号文を復号し, 復号結果を再び暗号化して, 暗号文演算処理を続ければ良い．しかし, この場合平文を生成することになり, データの安全性が保障されない．そこで, 完全準同型暗号の, データを暗号化した状態で処理できる性質 (2.1 節 (1), (2)) を用いる．

平文を m, 鍵ペア (pk, sk) に対応する暗号文を c, 鍵ペア (pk', sk') に対応する暗号文を c' とすると, (3) が成り立つ．

$$Encrypt(m, pk') = Decrypt(Encrypt(c, pk'), Encrypt(sk, pk')) \ldots (3)$$

暗号文 c, 秘密鍵 sk それぞれについて, pk^ı で暗号化した状態で, 復号処理をすることにより, 暗号文に蓄積したノイズの除去が行われる.ただし, 暗号文c, 秘密鍵skは pk'で暗号化されているため, 平文も pk'で暗号化されている.また, KDM 安全性を満たす場合,sk = sk^ı, $pk = pk'$ としてよく, 複数の鍵を用意する必要はない．上の処理を, 複数回の準同型演算により暗号文のノイズが上限に近づくたびに行い, FHE を実現する．

3 TFHE (Torus Fully Homomorphic Encryption)

bootstrap で実行される処理を以下に示す．

3.0.1 BlindRotation

TFHE では平文 $b - \sum_{i=0}^{n-1} a_i s_i = \Delta m + e$ を単項式 X の指数として計算 ($X^{-\Delta m+e}$) する．平文を暗号化した値が複数格納された多項式である LUT に対して $X^{-\Delta m+e}$ をかけることで，Δm に対応する暗号文が 0 次の項に移動される．以下のように計算する．

$$\begin{aligned} V \cdot X(-\Delta m + e) &= V \cdot X^{-b+\sum_{i=0}^{n-1} a_i s_i} \\ &= V \cdot X^{-b} \cdot X^{a_0 s_0} \cdot \ldots \cdot X^{a_{n-1} s_{n-1}} \end{aligned}$$

ここで $X^{a_i s_i}$ を計算するため，s_i を TRGSW で暗号化した bootstrapping key と，TRGSW と RLWE の乗算を活用した CMUX 回路を使用する．

3.0.2 bootstrapping key の作成

秘密鍵 SK=$(s_0, s_i, ..., s_{n-1})$ を TRGSW で暗号化したものを bootstrapping key と呼ぶ．TBGSW 暗号文は GLev 暗号文を用いて作成される．

GLev

GLev 暗号文はメッセージ M に対して異なるスケーリングパラメータ Δ を用いて暗号化した RLWE による暗号文を含むベクトルである．

$$(GLWE_{S,\sigma}(\frac{q}{\beta^1}M) \times ... \times GLWE_{S,\sigma}(\frac{q}{\beta^l}M)) = GLev_{S,\sigma}^{\beta,l}(M) \subseteq R_q^{l\cdot(k+1)}$$

βを大きくすることで，丸め誤差によるノイズが削減される．このように多項式を複数に分解し，ノイズを減少させる操作は *Decomposition* と呼ばれる．

TRGSW

TRGSW は GLev 暗号文を含むベクトルである．メッセージ m と秘密鍵 $\mathrm{SK_i}(\mathrm{i} \in [0,\ \mathrm{n}-1])$ の乗算に対する GLev 暗号文と，メッセージ m に対する GLev 暗号文をかけたものがメッセージ m に関する TRGSW 暗号文と定義される．

3.0.3 CMUX

CMUX は 1 つの GGSW 暗号文 b の値により，GLWE の d0 または d1 を出力するマルチプレクサである．内部では以下の計算が実行される．

$$b \cdot (d1 - d0) + d0 = db$$

3.1 ライブラリ

ライブラリによりサポートしている暗号方式や実装方法が異なる．

OpenFHE

オープンソースの FHE ライブラリである．BFV，BGV，CKKS，FHEW，TFHE 方式をサポートしており，bootstrap も実装されている．PALISADE，HElib，HEAAN，FHEW ライブラリの作成者が開発を行なっている．

Lattigo

Go 言語のライブラリで，RLWE を基にした BFV，BGV，CKKS 方式をサポートしている．それぞれの方式について秘密分散処理も実装されている．

TFHEpp

TFHE 方式をサポートしている．TFHE のより高機能な programmable bootstrap と呼ばれる機能が実装されている．従来の TFHE では論理回路で評価を行うため，非線形演算が難しいとされているが，programmable bootstrap では LUT に非線形演算による計算結果を保存することで，任意の演算を可能にしている．

FHEW

FHEW 方式をサポートしている．FHEW は TFHE の元になった方式であり，bootstrap を FHE の倍の速度で実行することが可能である．論理回路で評価を行う．

表 1　FHE，TFHE のライブラリの比較

	openFHE	Lattigo	TFHEpp	FHEW
BFV	○	○	×	×
BGV	○	○	×	×
CKKS	○	○	×	×
TFHE	○	×	○	×
programmable	×	×	○	×
FHEW	○	×	×	○
秘密分散	×	○	×	×
言語	C++	Go	C++	C++

4 先 行 研 究

4.1 ゲノム秘匿情報検索

アプリケーションについて述べる．クライアントサーバ 型通信で, クライアントがサーバのゲノムデータベース (PBWT) [15] に対して特定の塩基配列が最長マッチを持つか問い合わせを行う．具体的な手順を以下に示す．

（1） クライアントはクエリを全文暗号化し，その他のパラメータと一緒にサーバに送信する．

（2） サーバはクエリを元に FHE 演算を行い，結果をクライアントに送信する．

（3） クライアントはサーバから送られてきたデータを復号し, 結果を得る．

この時, お互いに持っているゲノム塩基配列や調べたい内容を秘匿しながら行うことが FHE により実現され, 個人情報の漏洩が防がれている. ここで, クエリ長や検索ポジション数に関わらず実装を可能にするには, FHE の演算で bootstrap の導入もしくは LHE でより多くの演算を可能にするための大きなパラメータの指定が必要となる．また, これらの暗号化処理には膨大なメモリ容量が必要な一方で, クラウドでは適当に DRAM が割り当てられないことが多く, スワップ処理が発生する場合がある. [16]

4.2 HElib を用いた場合の評価

4.2.1 並　列　度

論理コア数 12 まで実行時間が減少した．特に暗号文同士の照合処理と bootstrap 処理の並列度が高い．HElib を用いた実装では CPU のマルチコア性能の活用が可能である．また，Hyperthead を用いて実行時間の比較を行った．Hyperthead が有効になるのに伴い実行時間が増加することから，CPU と DRAM 間の load/store 命令がボトルネックであることがわ

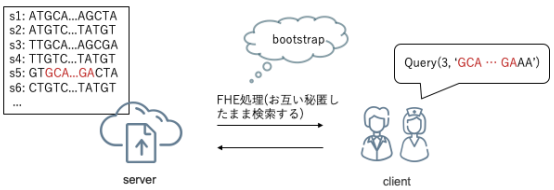

図 3　ゲノム秘匿情報検索の流れ

表 2　評 価 環 境

CPU	Xeon E5-2643 v3
	(3.40 GHz × 6 Cores) × 2
L1 (i,d)cache	32KB, 64B/line
L2 (i,d)cache	256KB, 64B/line
L3 cache	20480KB, 64B/line
DRAM	DDR4, 512GB
Docker1	memory 512GB

かった．SSD では CPU から Read/write 命令を並列に受け取ることが可能なため，SSD を活用できる可能性が高い．

5　評　　価

5.1　評 価 環 境

表 2 に評価環境を示す．

5.2　ゲノム秘匿検索における HElib ライブラリの負荷

図 4 にゲノム秘匿検索実行時の関数ごとの負荷の内訳を示す．perf コマンドを用いて計測を行った．HElib ライブラリを用いた場合，暗号文を円分多項式として処理を行うため，係数の小さい高次の多項式同士の演算を表す NTL::FFT の負荷が処理全体の約 50 %を占める．そこで，NTL::FFT の負荷が HElib のどの関数により引き起こされているのか分析を行った．

5.2.1　関数名と処理内容

図 5，図 6 で参照される，HElib 関数を用いた代表的な処理内容を示す．

- addconstant_equal_lookup

クライアントにより送信された，暗号化された LUT を参照するための指標を，サーバが LUT のインデックスに重ね合わせる．暗号化した状態で LUT 上の検索位置を特定する際に発生する．暗号文と LUT のインデックスの値を持つ多項式，1 を持つ多項式を足し合わせ，該当するブロック内のみ 1 を立てる．

- rotate_equal_lookup

サーバが，クライアントから送信された，暗号化された LUT の検索位置を，暗号化した状態で特定する照合処理の際に発生する．インデックスを表すブロック内の全ての値を掛けたものを求めるため，暗号化された多項式に対し rotation を行う．

- multiplyBy_equal_lookup

rotate_equal_lookup により作成した多項式同士を掛け合わせる処理．求めたいインデックスを表すブロック内のみ 1 が残り，その他のブロック内は 0 で埋められる．

- rotate_lookup

上の multiplyBy 関数により発生した求めたいインデックス内の 1 を，インデックスを表すブロック内の全ての値にコピーするため，多項式に対し rotation を行う．

- others_lookup

上記以外の lookup 処理に含まれる，HElib 関数による暗号処理．

- initialization_ini

initialization で行われる処理全体．client による FHE 公開鍵・秘密鍵の作成や，server による client から送信された情報を基に FHE コンテキストを設定する処理，FHE 公開鍵の作成などを行う．

5.2.2　負荷の推移 -検索クエリ長

図 5 にクエリ長を変化させた時の関数毎の負荷の推移を示す．検索開始ポジション数は 1 に設定した．検索クエリ長が 1 伸びるにつれて，PBWT と検索クエリの照合回数は 2 回ずつ増加するが，1 回の照合にかかる負荷は変わらない．bootstrap 処理も 2 回増加するが，1 回毎の処理の重さはクエリ長に依存せず一定である．クエリ長が 1 の時，NTL::FFT の処理全体に対して，initialization が 2.95 %，lookup が 76.69 %，bootstrap が 20.36 %を占める．その後，クエリ長が伸びるのに伴い，lookup の割合は緩やかに減少し，bootstrap の割合は増加する．クエリ長が 10 の時点で，initialization は 0.70 %，lookup は 64.37 %，bootstrap は 34.93 %となる．実際は検索クエリ長が長いことも想定されるが，その場合も lookup の負荷は一定以上で推移することがわかる．特に，暗号化された状態で適切な検索位置を特定するための rotation の処理 (rotate_equal_lookup，rotate_lookup) が重い．実行回数が多いことが原因である．HElib ライブラリの暗号方式では，暗号化された状態で 2 つの値を比較することができない点を，rotation と暗号文を表す多項式，0 を表す多項式，1 を表す多項式を使い補っている．暗号化された状態で値の比較が可能な TFHE を用いることで

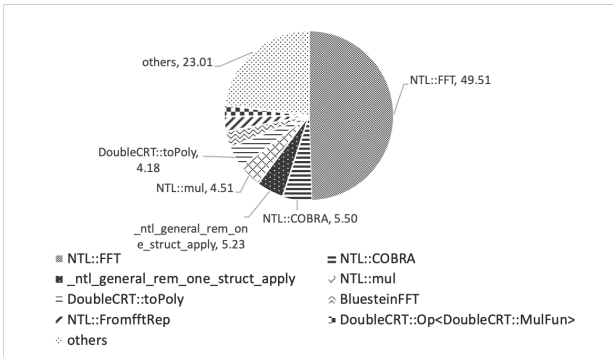

図 4　実行時の関数ごとの負荷の内訳

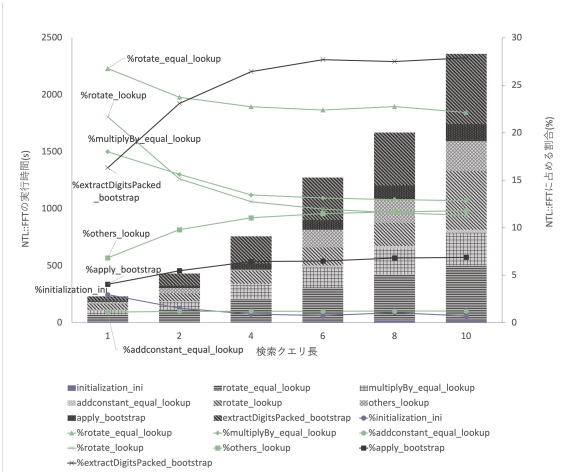

図 5　検索クエリ長の変化に伴う関数ごとの負荷の推移

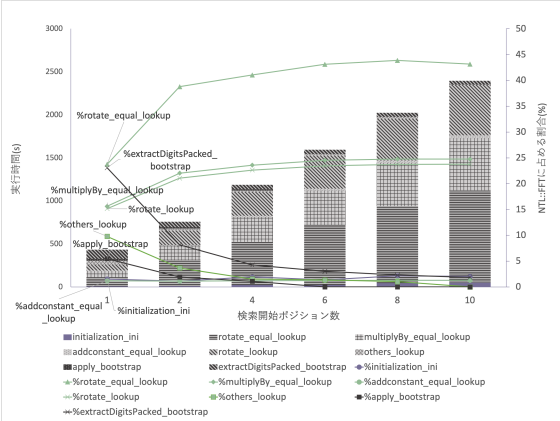

図 6　検索開始ポジション数の変化に伴う関数ごとの負荷の推移

rotation の負荷は削減できる．

lookup で発生する暗号文同士の掛け算 (multiplyBy_equal_lookup) は，暗号化された状態で適切な検索位置を特定する処理に伴い，rotate_equal_lookup と同じ回数だけ発生する．multiplyBy 関数は，rotation に比べて 1 回の実行にかかる時間が少ない．また，bootstrap 処理の負荷がクエリ長に依存して増加する．特に，extractDigitPacked 関数の負荷が大きい．実行回数は (検索クエリ長)×2＋1 と少ないが，1 回にかかる負荷が大きいことが原因である．EvalMap_apply 関数も 1 度の負荷が大きい．

5.2.3　負荷の推移 -検索開始ポジション数

図 6 に検索開始ポジション数を変化させた時の関数毎の負荷の推移を示す．検索クエリ長は 2 に設定した．検索開始ポジション数の変化により，照合処理や bootstrap の回数は変化しない．一方，照合処理を行う暗号文サイズは (PBWT 内のクエリのサンプル数)+1 ずつ増加するため，1 回の処理にかかる負荷は増加する．bootstrap は複数の暗号文の照合結果を，決まったサイズにまとめたもの 1 つに対して行うため，負荷は変化しない．従ってポジション数が増えるにつれて，bootstrap に比べて，lookup の照合処理の負荷が大きくなる．検索ポジション数 10 では rotation の負荷が大きいことが顕著である．使用可能なメモリに余裕がある場合は，未使用の暗号文に rotation 済みのデータを保存しておくことで，rotation の回数を減らし高速化が可能である．

5.3　その他の暗号ライブラリの評価

5.3.1　FHE

（1）　加算

（a）　速度・メモリ使用量

図 7，8，9 に BFV・BGV・CKKS 方式に関して OpenFHE または Lattigo 使用時の暗号文の次数の変化に伴う加算の実行時間またはメモリ使用量の推移を示す．OpenFHE では次数の増加に伴い実行時間が指数関数的に増加するが，Lattigo では変化が緩やかである．BFV 方式では暗号文の係数の処理に関する並列化の割合が小さいことがわかる．OpenFHE では BFV，CKKS，BGV 方式の順に高速である．図 7，8 から BFV と BGV の処理の内訳が等しいため，OpenFHE の BGV 方式では暗号化により作成される暗号文サイズが大きいため処理時間が長いことがわかる．Lattigo では暗号方式による速度の差は見られない．また，N=2^10〜N=2^14 では BFV，CKKS 方式では OpenFHE のメモリ使用量がそれぞれ 0.4GB 0.7GB，0.2GB 0.6GB ほど少ない．OpenFHE では

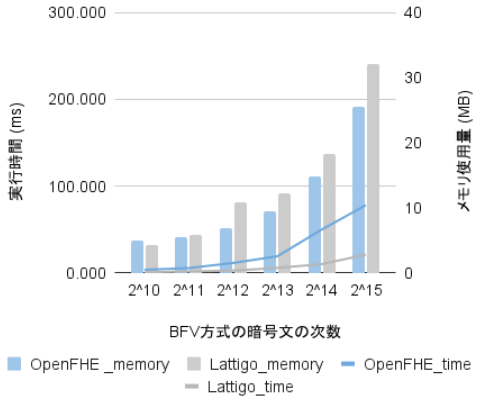

図 7　OpenFHE・Lattigo の BFV 方式を用いた際の加算の実行時間とメモリ使用量

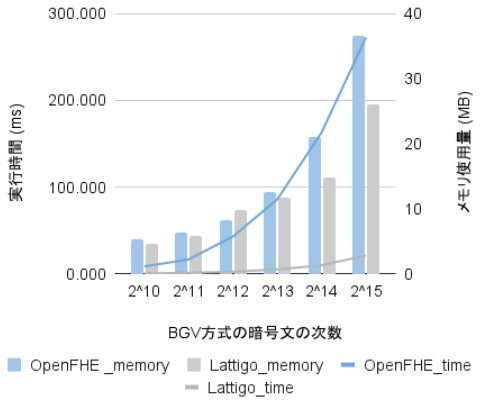

図 8　OpenFHE・Lattigo の BGV 方式を用いた際の加算の実行時間とメモリ使用量

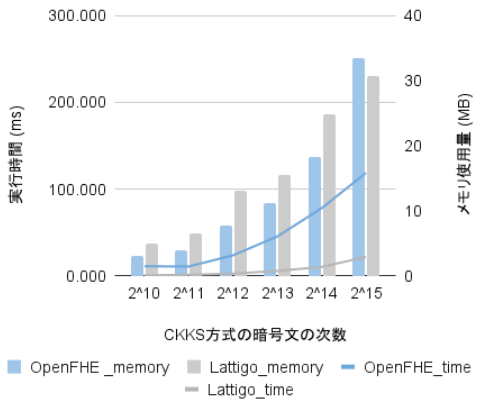

図 9　OpenFHE・Lattigo の CKKS 方式を用いた際の加算の実行時間とメモリ使用量

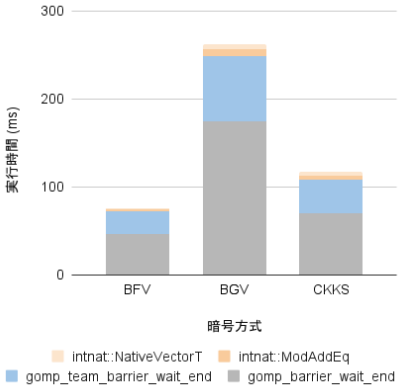

図 10　OpenFHE を用いた加算による負荷の内訳

BFV，CKKS，BGV の順にメモリ使用量が少ない．Lattigo では方式によるメモリ使用量の差は見られない．

（b）　ライブラリ内の負荷

図 10 に暗号方式に関する加算に伴う処理の負荷の比較を示す．負荷が大きい関数の処理内容は以下である．

- gomp_barrier_wait_end

GCC で用いられる OpenMP のライブラリである GOMP 内の関数である．並列実行できないタスクの実行に伴い，スレッドを停止する．

- gomp_team_barrier_wait_end

GOMP 内の関数で，並列実行できないタスクの実行に伴い，スレッドからなるグループを停止する．

- intnat::NativeVectorT::ModAddEq

ベクトルに対しスカラーの剰余加算を行う．

- intnat::NativeVectorT::NativeVectorT

正の整数から成るベクトルを作成する．

- clear_page_c_e

カーネルのページング処理で用いられるページを初期化する．

暗号処理による負荷の違いは見られない．全ての暗号方式で GOMP によるスレッドの停止時間が 90 %以上を占め，並列度が低いことがわかる．全ての暗号方式で同様の方法で暗号文の加算を行っているため，共通の方法で GOMP による処理時間を減少することが可能である．ModAddEq は暗号文同士の加算により呼び出される．NativeVectorT は暗号文同士の加算結果を入力された暗号文に保存する lbcrypto::LeveledSHERNS::EvalAddInPlace で暗号文 vector を複製する際の代入処理に実行される．

（2）　乗算

（a）　暗号文の次数の変化に伴う速度・メモリ使用量

図 11，12，13 に BFV・BGV・CKKS 方式に関して OpenFHE または Lattigo 使用時の暗号文の次数の変化に伴う加算の実行時間またはメモリ使用量の推移を示す．BFV 方式では OpenFHE と Lattigo による速度の差は見られない．BGV，CKKS 方式では暗号文の次数の増加に伴い OpenFHE に比べ Lattigo が高速である．Lattigo は，暗号文の係数に対する処理が並列化されている割合が高いことが原因である．OpenFHE では暗号方式による速度の差は見られない．Lattigo では CKKS，BGV，BFV の順に高速である．また，Lattigo では次数の増加に伴いメモリ使用量が指数関数的に増加するが，OpenFHE では変化が緩やかである．Lattigo では係数の処理の並列性が高いことが原因である．OpenFHE では BFV，BGV，CKKS の順にメモリ使用量が少ない．Lattigo では BGV，CKKS に比べ BFV

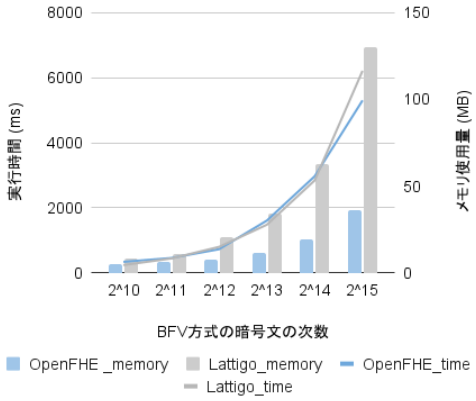

図 11　BFV 方式による乗算実行時の暗号文の次数の変化に伴うメモリ使用量の変化

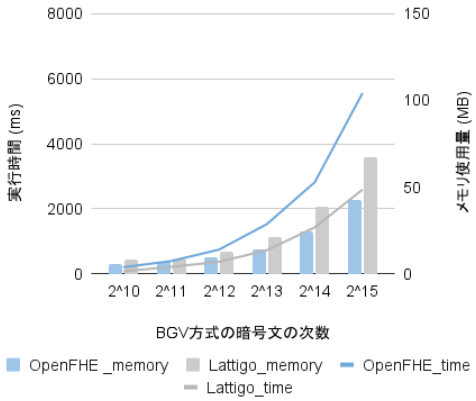

図 12　BGV 方式による乗算実行時の暗号文の次数の変化に伴うメモリ使用量の変化

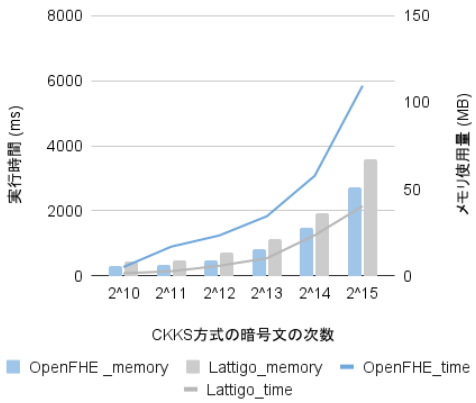

図 13　CKKS 方式による乗算実行時の暗号文の次数の変化に伴うメモリ使用量の変化

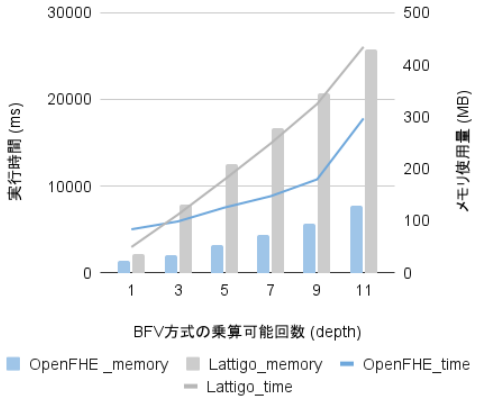

図 14　BFV 方式による乗算実行時の乗算可能回数の変化に伴うメモリ使用量の変化

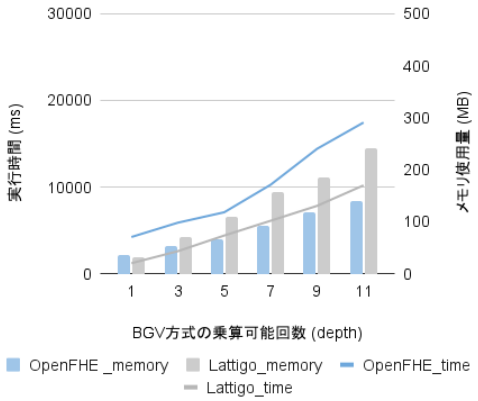

図 15　BGV 方式による乗算実行時の乗算可能回数の変化に伴うメモリ使用量の変化

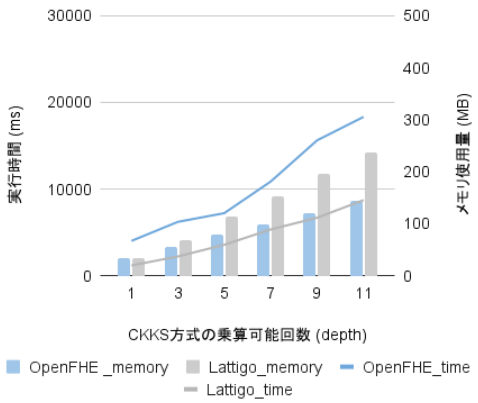

図 16　CKKS 方式による乗算実行時の乗算可能回数の変化に伴うメモリ使用量の変化

のメモリ使用量が高い．

（b）　乗算可能回数の変化に伴う速度・メモリ使用量

図 14，15，16 に BFV・BGV・CKKS 方式に関して OpenFHE または Lattigo 使用時の乗算可能回数の変化に伴う加算の実行時間またはメモリ使用量の推移を示す．BFV 方式では，乗算可能回数の増加に伴い Lattigo に比べ OpenFHE が高速である．Lattigo の BFV 方式ではそれぞれの Level で用いられる暗号文に対する処理が並列化されている割合が小さいことがわかる．OpenFHE では暗号方式による実行時間の差は見られない．Lattigo では乗算可能回数の増加に伴い BFV に比べ BGV，CKKS が高速である．また，全ての暗号方式で乗算可能回数の増加に伴い Lattigo に比べ OpenFHE のメモリ使用量が増加する．OpenFHE では暗号方式によるメモリ使用量の差は見られない．Lattigo では BFV に比べ，BGV，CKKS のメモリ使用量が少ない．

（c）　乗算可能回数の変化に伴うライブラリ内の負荷の推移

図 17 に暗号方式に関する乗算による負荷の内訳の比較を示す．負荷が大きい関数の処理内容は以下である．

- gomp_team_barrier_wait_end

GCC で用いられる OpenMP のライブラリである GOMP 内の関数である．並列実行できないタスクの実行に伴い，スレッドからなるグループを停止する．

- gomp_team_barrier_wait_end

GOMP 内の関数で，並列実行できないタスクの実行に伴い，スレッドを停止する．

- ForwardTransformToBitReverseInPlace

CRT パッキングで作成された Zq[x]/(X^n+1) に対する処理の 1 つで，NTL の剰余の乗算のため bit 反転を行う．

- InverseTransformFromBitReverseInPlace

ForwardTransformToBitReverseInPlace の対となる bit 反転を行う．

- intnat::NativeVectorT::ModMulEq

ベクトルに対しスカラーの剰余乗算を行う．

- lbcrypto::DCRTPolyImpl::ScaleAndRound

値を平文空間 (modp) から暗号文空間 (modq) に拡張する．

- lbcrypto::DCRTPolyImpl::SwitchCRTBasis

CRT パッキングにおける基数 (mod) を変更する．暗号文空間 $Q = q_1, ..., q_l$ から $P = p_1, ..., p_l$ に変換する際に用いる．

- ApproxSwitchCRTBasis

CRT パッキングにおける基数 (mod) を変更する際に呼び出される．

暗号方式による関数の負荷の割合の違いは見られない．乗算について，暗号方式による乗算の内部処理の違いが小さいため，全ての暗号方式で同じ方法で効率化が可能である．図 17 から乗算可能回数が小さいほど GOMP によるスレッドの停止時間が長い．Modulus switch を使用した処理では，それぞれの乗算可能回数 (Level) 用に対応する暗号文が作成され，全てに対して処理が実行される．複数の暗号文に対する処理で並列処理できる割合が高いことがわかる．また，乗算可能回数の増加に伴い，ForwardTransformToBitReverseInPlace が増加する．BFV，BGV，CKKS 共に係数から CRT パッキングを行う lbcrypto::DCRTPolyImpl::SwitchFormat 関数に呼び出される．BFV では CRT パッキングされた関数の Decomposition を行う lbcrypto::DCRTPolyImpl::CRTDecompose による呼び出しが 1/2 を占める．

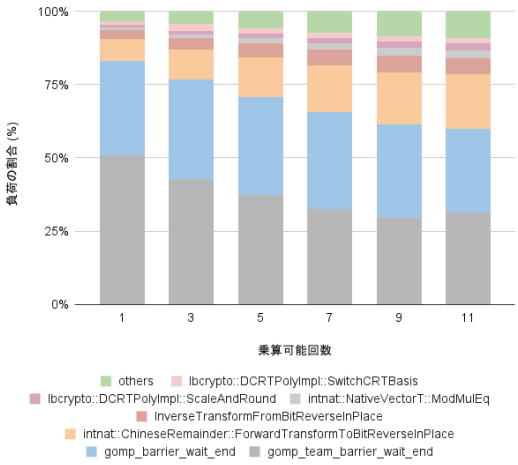

図 17　OpenFHE を用いた乗算による負荷の内訳

5.3.2　bootstrap

OpenFHE の CKKS 方式における暗号文では字数が N=2^10 の時 1340.43ms と高速であるが，N の増加に伴い実行時間が指数関数的に増加し，N=2^15 では 353068ms となる．

5.4　TFHE

5.4.1　演　　算

bootstrap を行わない場合 0.0014ms，circuit bootstrapping を用いる場合 4843.54ms，gate bootstrapping を用いる場合 28788ms の順に高速である．加算はノイズの増加の割合が少ないため，circuit bootstrapping を使用し，適宜 bootstrapping を行うことで高速に計算される．乗算について，必要なゲート数は桁数の 2 乗となり，gate bootstrapping に比べ circuit bootstrapping が 10149ms と高速である．メモリ使用量は加算・乗算共に 3.0GB 程度で推移する．

5.4.2　bootstrap

（1）　実行時間

図 18 にライブラリ・暗号方式に関する (演算処理を除く) 暗号処理に伴う実行時間の比較を示す．bootstrap では TFHEpp(TFHE 方式)69.21ms，FHEW(DM 方式) 1243.0125ms，OpeFhe(TFHE 方式) 21723ms，OpenFHE(DM 方式) 33403ms の順に高速である．Gate Key の作成は TFHEpp(TFHE 方式) 10047ms，OpenFHE(TFHE 方式) 1156ms，FHEW(DM 方式) 106221ms，OpenFHE(DM 方式) 202303ms の順に高速である．また，パラメータを変更することによる速度の差は見られなかった．

（2）　メモリ使用量

TFHEpp(TFHE 方式) 3GB，OpenFHE(TFHE 方式) 7.4GB，FHEW(DM 方式) 21.7GB，OpenFHE(DM 方式) 107.7GB の順にメモリ使用量が少ない．図 19 にライブラリ・暗号方式に関する (演算処理を除く) 暗号処理に伴う鍵サイズの比較を示す．TFHE 方式では secret key，bootstrap key 共に，TFHEpp のサイズが大きい．OpenFHE(TFHE 方式) と OpenFHE(DM 方式) の比較から，DM 方式の bootstrap key が大きいことがわかる．

（3）　TFHE 方式の負荷の比較

図 20，21 にそれぞれ OpenFHE，Lattigo における TFHE 方式実行時の処理の負荷の比較を示す．負荷が大きい関数の処理内容は以下である．

OpenFHE

- ForwardTransformToBitReverseInPlace

CRT パッキングで作成された多項式に対する処理の 1 つで，NTL の剰余の乗算のため bit 反転を行う．

- intnat::NativeVectorT::ModMulEq

ベクトルに対しスカラーの剰余乗算を行う．

- RingGSWAccumulator::SignedDigitDecompose

RGSW 暗号文の作成時にノイズを削減するため多項式を複数に分解する．

- intnat::NativeVectorT::ModAddEq

ベクトルに対しスカラーの剰余加算を行う．

TFHEpp

- TFHEpp::trgswfftExternalProduct

Decomposition により作成される TRGSW と TRLWE の積を計算する．CMUX など，暗号化した状態で比較演算を行う際に用いられる．

- TFHEpp::IdentityKeySwitchWITHEncoder

秘密鍵の変換を行う．bootstrap により新たな秘密鍵 S' を元に作成された暗号文を，bootstrap 前の秘密鍵 S による暗号文に変換する際などに用いられる．

- execute_reverse_int

離散フーリエ変換を行う．暗号文同士の乗算を畳み込み処理で高速化する際に用いられる．

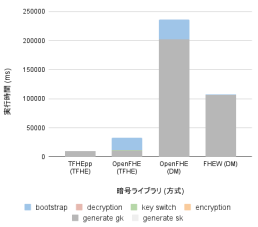

図 18　ライブラリ・暗号方式による実行時間の比較

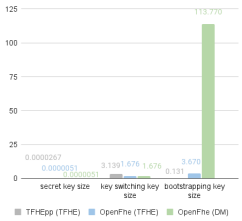

図 19　ライブラリ・暗号方式による鍵サイズの比較

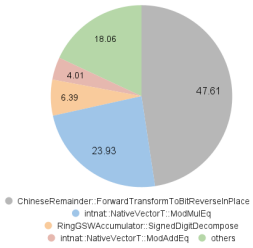

図 20　OpenFHE の bootstrap に伴う負荷の内訳

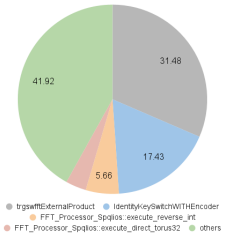

図 21　TFHEpp の bootstrap に伴う負荷の内訳

OpenFHE の 47.61 %を占める ForwardTransformToBitReverseInPlace のうち，大部分は bootstrap 内の SampleExtractIndex を行う AddToAccCGGI で呼び出される．CMUX により出力された RLWE の暗号文を LWE に変換する際に用いられる．23.29 %を占める intnat::NativeVectorT::ModMulEq も AddToAccCGGI 内の CMUX と sampleExtractIndex の計算で呼び出される．TFHEpp では CMUX の計算を行う trgswfftExternalProduct が 31 %を占める．OpenFHE に比べ THFEpp では CMUX の計算負荷が小さい．また，TFHEpp では TFHEppKey switching の負荷は 6.06 %と計測され，OpenFHE より負荷が大きい．CPU 使用率は OpenFHE，TFHEpp 共に 10 %以下にとどまり，処理の並列性が低いことがわかる．また，1 つのコアで 80 %程度の iowait が発生しており，並列処理できないプロセスが待ち状態であることがわかる．

5.5 ライブラリの比較に関する考察

加算に関して，FHE では GOMP ライブラリの負荷が 90 %以上を占め，ボトルネックになっている．また全ての暗号方式で OpenFHE に比べ，Lattigo が高速である．OpenFHE では暗号文の次数の増加に伴い実行時間が大幅に上昇することから，暗号文の次数に対する並列性が低いことがわかる．TFHEpp ライブラリを用いた TFHE 方式の加算は，bootstrap を行わない場合，0.0014ms と高速である．一方，FHE の BFV，BGV，CKKS ではパッキングにより複数の暗号文を並列で加算することが可能なため，暗号文サイズが 2^{11} より大きい場合は，FHE が高速である．加算に伴うメモリ使用量は FHE では最大 0.13MB，TFHE では 3GB 使用となり，FHE がメモリ使用率が良いことがわかった． 乗算では Lattigo の CKKS 方式が OpenFHE，Lattigo，TFHEpp，FHEW の中で最速である．乗算のボトルネックである，ForwardTransformToBitReverseInPlace や intnat::NativeVectorT::ModMulEq は TFHE のボトルネックと共通している． bootstrap は OpenFHE の CKKS 方式を用いた場合の次数が 2^10 の暗号文に対する処理が 1340.43ms と高速である．多項式の次数が大きい場合は TFHE が高速である．N が小さいほど暗号文が脆弱になるため，適切な暗号文長を検討する必要がある．

6 まとめと今後の課題

本稿では, FHE と TFHE を実装したライブラリについて，SSD の活用に適したライブラリの検討を行った．ゲノム秘匿検索では暗号文同士の比較ができないことによる BlindRotation と bootstrap の負荷が大きい．暗号ライブラリの評価により，算術演算については TFHE に比べ FHE が高速であることがわかった．bootstrap は暗号文の次数が小さい場合は CKKS，大きい場合は TFHE が高速である．ゲノム秘匿検索では暗号文同士の値が比較できることにより，処理量が大幅に削減されるため，TFHE の比較できる性質を用いることで高速化されることが期待される．今後は TFHE 又は FHE の暗号方式で高速であった CKKS を用いた実装を行い，SSD に適切な暗号処理の検討を行う．

謝　　辞

本研究は一部，キオクシア株式会社，JST CREST JPMJCR22M2 の支援を受けたものである．

文　　献

[1] HElib. https://github.com/homenc/HElib.
[2] SEAL. https://github.com/microsoft/SEAL.
[3] Lattigo. https://github.com/tuneinsight/lattigo.
[4] TFHE. https://tfhe.github.io/tfhe/.
[5] zama. https://www.zama.ai/post/tfhe-deep-dive-part-1.
[6] fhew. https://github.com/lducas/FHEW.
[7] tfhepp. https://github.com/virtualsecureplatform/TFHEpp.
[8] TFHEpp. https://qiita.com/nindanaoto/items/98335-\\ad4d32b927effa9.
[9] Yu Ishimaki, Hiroaki Imabayashi, Kana Shimizu, and Hayato Yanama. Privacy-preserving string search for genome sequences with fhe bootstrapping optimization. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 3989–3991, 2016.
[10] P.Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. of EURO CRYPT 1999, Lecture Notes in Computer Science , Vol.1592*, pages 223–238, 1999.
[11] Cynthia Dwork. Differential privacy. In *Languages and Programming, LNCS, vol. 4052*, pages 1–12, 2006.
[12] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. In *ACM Computing Surveys, Vol. 51, Issue 4*, 2018.
[13] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjosteen, Angela Jaschke, Christian A. Reuter, and Martin Strand. A guide to fully homomorphic encryption. In *Cryptology ePrint Archive 2015/1192*, 2015.
[14] Craig Gentry. *A FULLY HOMOMORPHIC ENCRYPTION SCHEME.* PhD thesis, Stanford University, 2009.
[15] R. Durbin. Efficient haplotype matching and storage using the positional burrows-wheeler transform (pbwt). In *Bioinformatics, vol. 30, no. 9*, pages 1266–1272, 2014.
[16] Muhammad Tirmazi, Adam Barker, Nan Deng, Md.E Haque, Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. Borg: the next generation. In *Fifteenth European Conference on Computer Systems (EuroSys '20)*, pages 1–4, 2020.