DEIM Forum 2022 J41-4

準同型暗号処理で多用される Trace-Type Function の AVX512 による高速化

井上紘太朗† 鈴木 拓也† 山名 早人††

† 早稲田大学 大学院基幹理工学研究科 〒 169–8555 東京都新宿大久保 3-4-1
†† 早稲田大学 理工学術院 〒 169–8555 東京都新宿大久保 3-4-1
E-mail: †{kinoue,t-suzuki,yamana}@yama.info.waseda.ac.jp

あらまし クラウドの普及により，データ分析を外部に委託する事例が増え，データを情報漏洩やプライバシー侵害のリスクから保護する必要性が高まっている．特にデータを暗号化した状態で計算を行うことができる準同型暗号は，データ保護の一手法として活発に研究が進められている．その中でも Ring Learning With Errors 方式は，複数の値を一つの平文とみなすパッキングと呼ぶ手法により，各要素（スロット）ごとの並列計算が可能であり，多くの準同型暗号アプリケーションで採用されている．パッキングされた各要素に対する操作としては，スロットのシフト（rotation）と準同型加算を組み合わせる trace-type function と呼ばれる演算が多用され，同演算の高速化がホットな話題となっている．本稿では，trace-type function の高速化として従来提案されている loop unrolling に加え，AVX512 を活用した SIMD 演算によりさらなる高速化を狙う．評価実験の結果，AVX512 未適用の unrolled trace-type function に対して，1.05-2.30 倍の速度向上を確認した．

キーワード 準同型暗号，セキュリティ，並列処理，SIMD

1 はじめに

インターネットの普及に伴い，市場に流通するデータの量は爆発的に増大している．IT 調査会社 IDC の報告によると，2020 年に全世界で 64.2ZB のデータが作成または複製された [1]．また，クラウドサービスの普及に伴い，大量のデータを収集・分析し，活用するまでを一貫して行うことが容易になった．近年では，保有するデータを活用したい企業や個人が，分析のノウハウを持つ第三者へ分析業務を委託するという事例が増えてきている．扱われるデータの中には，遺伝子情報等の個人情報や，企業の機密情報等も存在する．こうしたデータを，プライバシー侵害や情報漏洩のリスクから保護しつつ，安全に利活用する手法について，近年盛んに研究が進められている．その要素技術の一つとして，準同型暗号が挙げられる．

準同型暗号（Homomorphic Encryption:HE）は，データを暗号化した状態で計算を行うことができる暗号方式である．CKKS（Cheon-Kim-Kim-Song）方式 [2], [3] は，復号結果に誤差が含まれることを許容することによって実数や複素数を扱うことのできる準同型暗号の一方式であり，機械学習等の実アプリケーションで幅広く利用されている．また、準同型暗号は，安全性を保証する基盤として数学的な問題を採用している．その問題の一つである Ring Learning With Errors（RLWE）において，暗号文は多項式で表現されている．RLWE ベースの暗号方式では，ベクトル全体を一つの平文とみなすパッキング [2], [4] という手法が存在する．パッキングを用いることで，SIMD（Single Instruction Multiple Data）の要領でベクトルの各要素（スロット）ごとに並列で演算を行うことが可能である．また，準同型暗号上でスロットをシフトする演算を rotation と呼ぶ．実用的なアプリケーションの中には，rotation と準同型加算を連続して適用する演算が用いられることが多く，trace-type function と呼ばれている [5]．例えば，ベクトルの総和をとる演算がこれにあたる．しかし，準同型暗号上の演算は，通常の演算と比較して時間・空間計算量が大きいという問題を抱えている．準同型加算（準同型乗算）は，通常の加算（乗算）と比較し 168 倍 (27,721 倍) の実行時間を要する [6]．また，rotation は準同型乗算と比較して 7.15〜8.88 倍の実行時間を要する [7]．特に，trace-type function は $\mathcal{O}(\log N)$ の rotation を必要とするため，準同型乗算と比較して 85.83〜124.26 倍の実行時間を要する．

準同型暗号の実用化に向けて，計算量の問題は大きな障害となる．そのため，性能向上に向けて理論と実装の双方から盛んに研究が行われている．準同型暗号の性能向上においては，並列実行可能な部分をハードウェアの支援によって並列化していくという方法が研究されている．その一つとして，CPU の SIMD 拡張命令セットの一つである Intel AVX512（Advanced Vector eXtensions）が挙げられる．AVX512 は，AVX2 の後継として Intel が x86 命令セットに実装した SIMD 拡張命令である．AVX512 のレジスタ長は 512bit であり，AVX2 より効率良く処理を行えるようになった．この AVX512 を活用し，準同型暗号で多用される剰余演算を高速化したものが Intel HEXL である [8]．しかし，AVX512 を用いた trace-type function の演算最適化と性能評価に関する報告はこれまでに無く，AVX512 を用いた trace-type function の演算最適化への有効性を示すことは本分野の実用化において喫緊の課題である．

そこで，本研究では trace-type function の演算に焦点を絞り，AVX512 を活用した SIMD 化による最適化を行い，その効果を確認することを目的とする．特に，trace-type function の最適化手法の一つである loop-unrolling と AVX512 による SIMD 演算の親和性について，シングルスレッドとマルチスレッドの双方の側面から考察を行う．

2 節では，本研究の内容理解に必要な要素技術について述べる．3 節では，関連研究を紹介する．4 節では，本研究の提案手法について説明する．5 節では，本研究で行った実験と結果，得られた結果についての考察を行い，6 節でまとめる．

2 関連技術

2.1 記号の定義

$\mathbb{N}, \mathbb{Z}, \mathbb{R}, \mathbb{C}, \mathbb{Q}$ を, それぞれ自然数, 整数, 実数, 複素数, 有理数を表す集合とする．N を 2 べきの整数とし，$K = \mathbb{Q}[X]/(X^N+1)$ を $2N$-円分体，およびその整数環を $R = \mathbb{Z}/(X^N+1)$ とする．すなわち，R 上の任意の元は，たかだか $(N-1)$ 次の多項式の係数列である．R の元を $a(X)$ のように表す．また，整数 q を法（modulus）とする R 上の剰余環 R/qR を R_q と表記する．ベクトルは特に断りのない限り行ベクトルとし，ベクトル $\mathbf{a}$ の i 番目の要素を $\mathbf{a}[i]$ と表記する．整数 q について，$\mathbb{Z} \cap [-q/2, q/2)$ を $\mathbb{Z}_q$ と表記する．ただし，実装上は $\mathbb{Z}_q$ を $\mathbb{Z} \cap [0, q)$ としている．q を法として，整数 a の剰余をとる演算を $[a]_q$ と表記し，$[a]_q = 0$ が成立することを $q|a$ と表記する．また，正の整数 a について，$[0, 1, ..., a-1] \cap \mathbb{Z}$ の範囲を $[a]$ と表記し，$\mathbb{Z}^*_{2N} = 2k + 1_{k \in [N]}$ を $2N$ より小さい正の整数の集合とする．ベクトル $\mathbf{a}, \mathbf{b}$ の要素ごとの積を $\mathbf{a} \circ \mathbf{b}$，内積を $\langle \mathbf{a}, \mathbf{b} \rangle$ と表記する．集合 S から一様にランダムな要素をサンプリングすることを $\xleftarrow{u} S$，ある分布 X からサンプリングすることを $\leftarrow X$ と表記する．また，$|\cdot|$ は配列や集合の要素数を表す．$\oplus$ と $\otimes$ は，それぞれ準同型加算と準同型乗算を表す．

2.1.1 剰余数系（Residue Number System）

準同型暗号では，暗号文の法がワードサイズ（x86-64 環境では 64bit）に収まらないほど大きい場合が多く，通常では実装に多倍長整数が必要となる．しかし，多倍長整数を用いた演算は計算コストが大きいため，中国剰余定理（Chinese Remainder Theorem: CRT）を用いて，巨大な法をワードサイズに収まる複数の小さな法に分割する剰余数系（Residue Number System: RNS）と呼ばれる手法がよく用いられる [9]．Q_l を暗号文の法とし，$Q_l = \prod_{i=0}^{l} q_i$ を満たす互いに素な整数 q_i をとる．この時，q_i はワードサイズに収まるように選択すると，$\mathbb{Z}_{Q_l}$ 上の計算は $\mathbb{Z}_{q_i}$ ごとに並列で行うことができる．このように，RNS により多倍長演算にかかる計算コストを削減することができる．本稿では，$x \in R_{Q_l}$ について，$\{x^{(i)}\}_{i \in [l+1]}$ を x の RNS 表現（representation）と呼び，$x^{(i)} = [x]_{q_i}$ を RNS component，$\{q_0, ..., q_l\}$ を RNS 表現の基底（base）と呼ぶ．

RLWE ベースの方式では，多項式環と呼ばれる数学的構造が用いられており，平文や暗号文は多項式の係数列として表現されている．用いる多項式の次数 N について，準同型加算の計算量は係数同士の和で表現されるため $\mathcal{O}(N)$ である．対して，準同型乗算は多項式同士の畳み込みを行う必要があるため，計算量は $\mathcal{O}(N^2)$ である．準同型乗算の計算コストを削減するために用いられる手法が，数論変換（Number Theoretic Transform: NTT）である．本稿では, NTT の逆変換を inverse-NTT（invNTT）と表記する．N を多項式の次数とすると，NTT 及び invNTT の計算量は $\mathcal{O}(N \log_2 N)$ である．NTT を適用した暗号文同士の乗算は係数ごとに計算でき，全体の計算量は $\mathcal{O}(N \log N)$ となる．ゆえに，多くの準同型暗号の実装では準同型乗算の最適化に NTT が用いられている．

RNS と NTT を組み合わせた手法は，Full-RNS と呼ばれている．本稿で用いる CKKS 方式についても，Full-RNS を適用した実装 [3] を使用した．PALISADE [10] の実装において，通常の状態の暗号文を COEFFICIENT format，NTT を適用した状態の暗号文を EVALUATION format と呼称する．Full-RNS を用いた暗号方式の実装では，しばしば基底変換（base conversion）[11] と呼ばれる演算が用いられる．Full-RNS 版の CKKS においては，近似基底変換（approximate base conversion）[3] と呼ばれる演算が定義されている．これは，RNS で表現された値を別の基底の RNS 表現へ変換する演算である．$\mathcal{C} = \{q_0, ..., q_{l-1}\}, \mathcal{B} = \{p_0, ..., p_{k-1}\}$ を，それぞれ $Q = \prod_{i=0}^{l-1} q_i, P = \prod_{j=0}^{k-1} p_j$ を満たす RNS の基底とする．$x \in \mathbb{Z}_Q$ とした時, $\mathcal{C}$ を基底とする RNS 表現 $x_i = [x]_{q_i}$ から，$\mathcal{B}$ を基底とする RNS 表現 $x_j = [x]_{p_j}$ への近似基底変換 $\mathsf{Conv}_{\mathcal{C} \to \mathcal{B}}$ は以下のように定義される．ただし，$e \in \mathbb{Z}, e \leqq l/2$ とした時，$x_j = [x + Q \cdot e]_{p_j}$ を満たす．

$$\mathsf{Conv}_{\mathcal{C} \to \mathcal{B}} = \left(\sum_{i=0}^{l-1} [x_i \cdot \frac{q_i}{Q}]_{q_i} \cdot \frac{Q}{q_i} \mod p \right)_{p \in \mathcal{B}} \tag{1}$$

2.1.2 CKKS 方式

CKKS 方式は，任意回数の準同型加算と指定した回数の準同型乗算が実行可能な Leveled 準同型暗号の一種である．CKKS 方式では，暗号文に対して実行可能な準同型乗算の回数は multiplicative depth と呼ばれるパラメータ L によって決定される．後述の Rescale を暗号文に対して適用すると，暗号文の「レベル」と呼ばれるパラメータが 1 減少する．法が Q_l である暗号文のレベルは l である．レベルの初期値は L であり，レベルが 0 となると Rescale はそれ以上実行できない．

CKKS 方式では，固定小数点を整数係数多項式として表現するため，スケーリングを行っている．計算の精度は scaling factor と呼ばれるパラメータ Δ により決定される．準同型乗算の出力は，入力された暗号文のスケール値の積となり，準同型乗算を繰り返し適用するとスケール値は指数的に増加する．表現可能な桁数には限りがあるため，スケール値の増加によって整数部の表現に必要な桁数が不足してしまう．そこで，Rescale によりスケール値を削減することにより，準同型乗算によるスケール値の指数的な増加を緩和することができる．

2.2 Key-Switching と Rotation

key-switching は，$s, s' \in R$ とした時，2 つの異なる秘密

鍵 $sk = (1, s), sk' = (1, s')$ について，sk で暗号化された暗号文 ct を sk' で復号可能な暗号文 ct' へ変換する処理である．key-switching には，key-switching key（evaluation key）と呼ばれる鍵 **swk** と，暗号文の法 Q_l とは別に用意された特殊な法（special modulus）$P = \prod_{i=0}^{k-1} p_i$ が必要となる [12]．ここで，k は整数とし，p_i は互いに素な整数とする．χ を離散ガウス分布とし，$a_i \xleftarrow{u} R_{PQ_l}, e_i \leftarrow \chi$ とした時，s から s' への key-switching を行う evaluation key は $\mathbf{swk}_{s\to s'} = \{(e - as' + PB_i s, a)\}_{i\in[d]} \in R_{PQ_l}^{d\times 2}$ で定義される．

rotation は automorphism と呼ばれる演算によって構成されている．$t \in \mathbb{Z}_{2N}^*$ とした時，automorphism は写像 $\psi_\kappa : R \to R, a(X) \mapsto a(X^\kappa) \mod \Phi_{2N}(X)$ として定義される．ただし，$\Phi_{2N}(X)$ は 1 の原始 2N 乗根に対する円分多項式とする．この時，κ を automorphism index と呼ぶ．暗号文に対して automorphism を適用すると，秘密鍵が s から $\psi_\kappa(s)$ に変わり，副作用としてスロットが特定の数だけシフトされる [13]．秘密鍵が異なる暗号文は計算を行うことができないため，automorphism の適用後には key-switching を行う必要がある．本稿では automorphism と key-switching は rotation を構成する一連の演算とみなす．

2.3 Trace-Type Function

total-sums は, 全スロットの総和を各スロットに格納する演算であり，Halevi ら [14] により初めて構成され，rotate-and-sum を再帰的に適用することにより実現されている．また，total-sums の亜種として，部分和をとるものも存在する．total-sums は，しばしば秘匿 DB 検索 [15] や秘匿文書分類 [16] 等の準同型暗号を用いたアプリケーションで使用されている．

trace-type function のアルゴリズムをアルゴリズム 2.1，2.2 に示す．ここでは，rotate-and-sum を M 回（$M \leqq \log_2 N$）適用することを想定する．CKKS 方式において，$M = \log_2 N$ の時は total sums に，$M \leqq \log_2 N$ の時は部分和に対応する．なお，多項式の次数を N，k は key-switching 用の特殊な法の数とし，$d = (L+1)/k$，$\alpha = (L+1)/d$，$\beta = \lceil (l+1)/\alpha \rceil$，$\hat{Q}_j = \{\prod_{i=0}^{\alpha-1} q_{j\alpha+i}\}_{j\in[d]}$，$Q'_j = Q_L/\hat{Q}_j$ とする．ただし，$d|(L+1)$ とする．また，$\mathcal{C}_l = \{q_0, q_1, ..., q_l\}$，$\mathcal{B} = \{p_0, p_1, ..., p_{k-1}\}$，$\mathcal{C}'_j = \{q_{j\alpha}, ..., q_{(j+1)\alpha-1}\}$ を，それぞれ Q_l ，P，$\hat{Q}_j$ に対応する基底とし，$\mathcal{D}_i = \{\cup_{j\in[i]}\mathcal{C}'_j\} \cup \mathcal{B}$，$\mathcal{C}''_j = \mathcal{D}_\beta \setminus \mathcal{C}'_j$ とする．アルゴリズム 2.2 は，$\mathcal{O}(\log N)$ の automorphism および準同型加算を含む．

2.4 Intel Advanced Vector Extensions 512 (Intel AVX512)

Intel AVX512(Intel Advanced Vector Extensions 512) は，Intel 製の CPU に搭載されている SIMD 拡張命令セットの一つである．AVX512 を活用することで, 64bit 整数の 8-way SIMD 演算が可能となる．Intel AVX512 の命令セットを利用するには，コンパイラの自動ベクトル化を活用するか，intrinsic 関数を用いて利用する命令とタイミングを直接指定する．本研究では，intrinsic 関数を用いて AVX512 を利用する．intrinsic 関数が存在する．[17]

アルゴリズム 2.1 Automorphism and Key-Switching (AKS), Algorithm 1 in [5]

Input: κ is an automorphism index, $c = (c_0, c_1) \in R_{Q_l}^2$ is a ciphertext in EVALUATION format, $\mathbf{swk}_{\psi_\kappa(s)\to s} \in (R_{PQ_l}^{\beta\times 2})$ is an evaluation key in EVALUATION format.

Output: $\mathbf{c}' \in R_{Q_l}^2$ is a ciphertext in EVALUATION format

1: $(c'_0, c'_1) \leftarrow \mathbf{c}$
2: $(d_0, d_1) \leftarrow (\psi_\kappa(c'_0), \psi_\kappa(c'_1))$
3: $\hat{\mathbf{d}} \leftarrow \mathsf{RNSDecompModUp}(d_1)$
4: $\tilde{c_0} \leftarrow \psi_{k_i}(\langle \hat{\mathbf{d}}, \mathbf{swk}_{\psi_\kappa(s)\to s}[0] \rangle)$
5: $\tilde{c_1} \leftarrow \psi_{k_i}(\langle \hat{\mathbf{d}}, \mathbf{swk}_{\psi_\kappa(s)\to s}[1] \rangle)$
6: $(c''_0, c''_1) \leftarrow (\mathsf{RNSModDown}(\tilde{c_0}), \mathsf{RNSModDown}(\tilde{c_1}))$
7: $(c'_0, c'_1) \leftarrow (d_0, 0) \oplus (c''_0, c''_1)$
8: **return** c'

アルゴリズム 2.2 Homomorphic Trace-Type Function Algorithm 2 in [5]

Input: $c = (c_0, c_1)$ is a ciphertext in EVALUATION format, $\mathbf{k} = (k_0, k_1, ..., k_{|\mathbf{k}|-1})$ is an array of automorphism indices such that $|\mathbf{k}| = M(\leqq \log_2 N)$

Output: c' is a ciphertext in EVALUATION format

1: $c' \leftarrow c$
2: **for** $i = 0$ to $M - 1$ **do**
3: $\quad c' \leftarrow c' \oplus \mathsf{AKS}(\mathbf{k}[i], c')$
4: **end for**
5: **return** c'

- `__m512i _mm512_mullo_epi64(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，128bit の積 xy の下位 64bit を返却する．
- `__m512i _mm512_add_epi64(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，$x + y$ を返却する．
- `__m512i _mm512_sub_epi64(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，$x - y$ を返却する．

AVX512 を準同型暗号の高速化に応用する試みとして，準同型暗号で多用される剰余演算の実装を提供する Intel HEXL [8] と呼ばれるライブラリが開発されている．Intel HEXL は，以下に示す AVX512 の補助関数を実装している．ただし，`__m512i` は 8 つの 64bit 整数がパックされた 512bit 整数型である．

- `__m512i _mm512_hexl_mullo_epi<64>(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，128bit の積 xy の下位 64bit を返却する．内部では `_mm512_mullo_epi64` を直接呼んでいる．
- `__m512i _mm512_hexl_mulhi_epi<64>(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x, y について, 128bit の積 xy の上位 64bit を返却する．
- `__m512i _mm512_hexl_shrdi_epi64<bit_shift>(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，x と y を結合して生成した 128 ビットの中間結果を `bit_shift` ビットだけ右シフトし，下位 64 ビットを返却する．

- `__m512i _mm512_hexl_small_mod_epi64<k>(__m512i x, __m512i q, __m512i* q_times_2, __mm512i* q_times_4)`: $k \in \{1,2,4,8\}$ とした時，パックされた符号なし 64bit 整数 x，q の各要素が $0 \leqq x[i] \leqq k \cdot q[i]$ を満たすとする．この時，$x \mod q$ を計算し返却する．$k=2$ の時，$x < 2q$ であることを利用して，$x \mod q = \min(x-q, x)$ となる．`q_times_2` $= 2q$，`q_times_4` $= 4q$ は，それぞれ $k=4$，$k=8$ の時に必要となる入力であり，それぞれ前述の計算が再帰的に $\log_2 k$ 回実行される．例えば，$k=4$ の時は，$x \mod q = \min(\min(x-2q, x)-q, \min(x-2q, x))$ となる．
- `__m512i _mm512_hexl_mulhi_approx_epi<64>(__m512i x, __m512i y)`: パックされた符号なし 64bit 整数 x，y について，128bit の積 xy を計算し 128bit の中間結果を得る．中間結果の上位 64bit を返却する．ただし結果はたかだか 1 の誤差を含む．

3 関 連 研 究

3.1 SIMD を用いた準同型暗号の高速化

準同型暗号に限らず，SSE（Streaming SIMD Extensions）や AVX2（Advanced Vector eXtensions）といった Intel 系 CPU の SIMD 拡張命令セットを用いた高速化は広く用いられている．NTL[1]や NFLlib [18] といった数論ライブラリについても，SSE や AVX2 を使用した実装が組み込まれている．PALISADE [10][2]や HElib[3]，HEAAN[4]等の準同型暗号ライブラリは NTL に，FV-NFLlib[5]は NFLlib にそれぞれ依存しているため，間接的に SIMD 拡張命令セットによる高速化の恩恵を受けることができる．

近年は，AVX512 を活用した実装についても盛んに研究されている．Jung ら [6] は，AVX512 を用いて準同型乗算を実装し，従来実装の HEAAN と比較して 2.06 倍の計算速度向上を実現した．Boemer ら [8] は，AVX512-IFMA52（Integer Fused Multiply Add）を用いてベクトル同士の剰余演算と NTT/invNTT を実装，これらを用いて PALISADE v1.11.3 を対象に準同型乗算を実装し，従来実装の 2.59 倍の計算速度向上を実現した．

3.2 準同型暗号における Trace-Type Function の高速化

Halevi ら [14] は, total-sums を初めて構成すると共に, rotate-and-sum を再帰的に適用する repeated doubling（rotate-and-sums とも呼ばれる）を提案した．Halevi ら [19] は，baby-step/giant-step（BS/GS）を用いて automorphism の適用回数を削減する手法，および，hoisting を用いて automorphism 自体の実行コストを削減する手法を提案した．なお，hoisting とは，同じ暗号文に対して複数行う処理を処理順序の入れ替えと事前計算によって 1 回に削減する手法を指す．Bossuat ら [20] は，Halevi らの hoisting の手法を改善し，鍵生成の際にあらかじめ事前計算を行うことで automorphism のコストをさらに削減する手法を提案した．Ishimaki ら [5] は，これらの手法に加え，loop-unrolling と lazy modulus-down を用いて，ベースラインである repeated doubling と比較して 1.32-2.12 倍の性能向上を実現した．

アルゴリズム 3.1 HoistKS，Algorithm 4 in [5]

Input: $c = (c_0, c_1)$ is a ciphertext in EVALUATION format, $\mathbf{k} = (k_0, k_1, ..., k_{|\mathbf{k}|-1})$ is an array of automorphism indices, and $\mathbf{ek} = \{\mathbf{swk}_{s \to s(X^{(k[i])^{-1}})}\}_{i \in [|\mathbf{k}|]} \in (R_{PQ_l}^{\beta \times 2})^{|\mathbf{k}|}$ is an pre-computed value from evaluation key in EVALUATION format.
Output: c' is a ciphertext in EVALUATION format

1: $(c'_0, c'_1) \leftarrow c$
2: $\hat{\mathbf{d}} \leftarrow \mathsf{RNSDecompModUp}(c'_1)$ ▷ $\hat{\mathbf{d}} \in R_{PQ_l}^{\beta}$
3: $\mathbf{v} \leftarrow []$ ▷ Empty array of R_{Q_l} of length $|\mathbf{k}|$
4: $\tilde{\mathbf{v_0}}, \tilde{\mathbf{v_1}} \leftarrow []$ ▷ Empty array of R_{PQ_l} of length $|\mathbf{k}|$
5: **for** $i = 0$ to $|\mathbf{k}| - 1$ **do**
6: $\quad \mathbf{v}[i] \leftarrow \psi_{k_i}(c'0)$
7: $\quad \tilde{\mathbf{v_0}} \leftarrow \psi_{k_i}(\langle \hat{\mathbf{d}}, \mathbf{ek}[i][0] \rangle)$
8: $\quad \tilde{\mathbf{v_1}} \leftarrow \psi_{k_i}(\langle \hat{\mathbf{d}}, \mathbf{ek}[i][1] \rangle)$
9: **end for**
10: $d_0 \leftarrow \mathbf{v}[0]$
11: $\tilde{c_0} \leftarrow \tilde{\mathbf{v_0}}[0]$
12: $\tilde{c_1} \leftarrow \tilde{\mathbf{v_1}}[0]$
13: **for** $i = 0$ to $|\mathbf{k}| - 1$ **do**
14: $\quad d_0 \leftarrow d_0 + \mathbf{v}[i]$
15: $\quad \tilde{c_0} \leftarrow \tilde{c_0} + \tilde{\mathbf{v_0}}[i]$
16: $\quad \tilde{c_1} \leftarrow \tilde{c_1} + \tilde{\mathbf{v_1}}[i]$
17: **end for**
18: $(c''_0, c''_1) \leftarrow (\mathsf{RNSModDown}(\tilde{c_0}), \mathsf{RNSModDown}(\tilde{c_1}))$
19: $(c'_0, c'_1) \leftarrow (d_0, 0) \oplus (c''_0, c''_1)$ ▷ EVALUATION format
20: **return** (c'_0, c'_1)

本稿では Ishimaki らの手法 [5] をベースとするため，loop-unrolling の手法について詳しく述べる．アルゴリズム 3.1，3.2 に Ishimaki らの手法を示す．以降，loop-unrolling を適用した trace-type Function を unrolled trace-type function と呼称する．アルゴリズム 2.1 で必要なイテレーション数を $M(\leqq \log_2 N)$ とし，loop-unrolling を行なった後のイテレーション数を $h(\leqq M)$ とする．[5] によると，アルゴリズム 3.2 は $\mathcal{O}(h\sqrt[h]{N})$ の rotate-and-sum を含む．

4 提 案 手 法

本節では，unrolled trace-type function をベースとし，AVX512 を組み合わせた実装を提案する．AVX512 を用いた実装を行うにあたり，ライブラリとして Intel HEXL を使用した．さらに, 新たにベクトル同士の融合積和演算（FMA:fused multiply-add）を Intel HEXL へ追加し，Trace-Type Function へ適用した．

4.1 ベクトル同士の融合積和演算

Intel HEXL には，ベクトル対スカラーの融合積和演算が既に存在する．しかし，ベクトル同士の融合積和演算については

1：https://libntl.org/
2：厳密にはデフォルトで NTL に依存していないが，コンパイル時にオプションを与えることで NTL を使用した実装が有効になる．
3：https://github.com/homenc/HElib
4：https://github.com/snucrypto/HEAAN
5：https://github.com/CryptoExperts/FV-NFLlib

アルゴリズム 3.2 Unrolled version of Trace-Type Function, Algorithm 5 in [5]

Input: $c = (c_0, c_1)$ is a ciphertext in EVALUATION format, h is a number of iterations after unrolling, $\mathbf{K} = (\mathbf{k}_0, ..., \mathbf{k}_{h-1})$ is an array of vectors of automorphism indices per iteration, and $\mathbf{E} = (\mathbf{ek}'_0, ..., \mathbf{ek}'_{h-1})$ is an array of vectors of evaluation keys in EVALUATION format for each of the automorphisms, where $\mathbf{ek}'_i = \{\mathbf{swk}_{s \to s(X^{(k[i])^{-1}})}\}_{j \in [|\mathbf{k}_i|]} \in (R_{PQ_l}^{\beta \times 2})^{|\mathbf{k}_i|}$ for each $i \in [h]$.

Output: c' is a ciphertext in EVALUATION format

1: $c' \leftarrow c$
2: **for** $i = 0$ to $h - 1$ **do**
3: $\quad c' \leftarrow c' \oplus \mathsf{HoistKS}(c', \mathbf{k}_i, \mathbf{ek}'_i)$
4: **end for**
5: **return** c'

アルゴリズム 4.1 EltwiseVectorVectorFMAMod, based on Algorithm 3 in [8]

Input: `q` is a 512-bit packed value contains $q < 2^{62}$ modulus in all 8-lane, `X,Y,Z` is a 512-bit packed value contains $0 < X, Y, Z < q < 2^{63}$, and `barr_lo` is a 512-bit packed value contains $\lfloor 2^L/q \rfloor$ across all 8-lane.

Output: `X'` is a 512-bit packed value contains $0 < X' < q$.

1: `prod_hi` ← `_mm512_hexl_mulhi_epi<64>(X,Y)`
2: `prod_lo` ← `_mm512_hexl_mullo_epi<64>(X, Y)`
3: `c1` ← `_mm512_hexl_shrdi_epi64<L-1>(prod_lo, prod_hi)`
4: `c3` ← `_mm512_hexl_mulhi_epi<64>(c1, barr_lo)`
5: `c4` ← `_mm512_hexl_mullo_epi<64>(c3, q)`
6: `c4` ← `_mm512_sub_epi64(prod_lo, c4)`
7: `c5` ← `_mm512_add_epi64(c4, Z)`
8: `q_times_2` ← $2 \cdot$ `q`
9: `q_times_4` ← $4 \cdot$ `q`
10: `X'` ← `_mm512_hexl_small_mod_epu64<8>(c5, q, q * 2,q * 4)`
11: **return** `X'`

未実装であり，別々の関数として定義されている剰余加算と剰余乗算を順次適用する必要がある．

今回新たに実装したベクトル同士の融合積和演算は，剰余加算と剰余乗算を内部的に順次適用するものである．その際，剰余乗算と剰余加算の結果出力時で合計 2 回必要な剰余演算を，関数出力時の 1 回のみに集約するよう実装を行なった．

4.2 Unrolled Trace-Type Function に対する AVX512 の適用

準同型暗号ライブラリに定義されているベクトル同士の剰余演算に対して Intel HEXL を適用することにより，Unrolled Trace-Type Function の実行速度向上を試みる．本研究では，準同型暗号のライブラリ実装として PALISADE v1.9.2 [10] をベースとし，剰余演算の実装に対して Intel HEXL を適用した．ただし，PALISADE v1.11.5 [21] にすでに実装されているコード（剰余乗算，NTT/invNTT，modulus switching）については新たに実装を行わず，v1.9.2 で動作するように改修の上，移植を行なった．加えて，新たに剰余加算，剰余減算，剰余融合積和の実装を行なった．また，アルゴリズム 3.1 において，for ループについてはそれぞれ並列化を行った．

表 1　実 験 環 境

項目		値
CPU	型番	3rd Gen Intel(R) Xeon(R) Gold 6334
	動作周波数（定格） [GHz]	3.60
	L1 キャッシュ [KiB/Core]	48(データ)/ 32(命令)
	L2 キャッシュ [KiB/Core]	1280
	L3 キャッシュ [MiB/CPU]	18
	コア数	8
	ソケット数	2
RAM	サイズ [GB]	128
OS		Ubuntu 20.04 LTS
g++(GCC) version		9.3.0
OpenMP version		4.5
PALISADE version		1.9.2
Intel HEXL version		1.2.3
googlebench[10] commit hash		f730846

5　評価実験と考察

本節では，提案手法の評価を行う．まず，本研究で新たに追加したベクトル同士の融合積和演算を対象に実行時間の評価を行う．次に，AVX512 を適用した trace-type function を対象に実行時間の評価を行う．測定に用いた実験環境を表 1 に示す．実験時は，numactl コマンドを用いて NUMA1 の CPU と RAM が使用されるように設定した．なお，ベンチマークの安定性を考慮し，Turbo Boost と Hyper-Threading を無効化した．さらに，動作周波数を定格周波数の 3.60GHz へ固定した．実験に用いるプログラムは，全て C++17 で記述した [6]．コンパイルオプションとして，`-march=native -O3 -fopenmp -Wall -NDEBUG` を指定した．なお，並列化ライブラリとして OpenMP[7] 4.5 を使用した．

本研究では，準同型暗号のライブラリ実装として，PALISADE v1.9.2 [10] を使用する．実験では，PALISADE のネイティブ実装をベースとし，剰余演算の実装を AVX512 を利用するよう修正を行なったものを使用した [8]．このパッチには，PALISADE v1.11.5 にすでに実装されているコード（剰余乗算，NTT/invNTT，modulus switching）を移植したものが含まれている．加えて，新たに剰余加算，剰余減算，剰余融合積和の実装を行なった．なお，今回使用した PALISADE は，NTL 等の数論ライブラリには依存していない．比較のため，実験条件は Ishimaki ら [5] と同じものとする．また，AVX512 による実装を補助するライブラリとして，Intel HEXL v1.2.3 [8] を使用した [9]．

6：https://github.com/yamanalab/FastHETrace
7：https://www.openmp.org
8：https://github.com/yamanalab/PALISADE
9：https://github.com/yamanalab/HEXL

5.1 ベクトル同士の融合積和演算

5.1.1 実験方法

本実験では，新たに実装したベクトル同士の融合積和演算の実行速度の評価をするために，乗算と加算を順番に適用したもの（以下，MultAdd）と，今回新たに実装した融合積和演算（以下 FMA）との比較を行った．MultAdd は，Intel HEXL が提供している剰余加算・剰余乗算の関数を呼び出す形で実装を行った．ゆえに，MultAdd と FMA はともに AVX512 を使用した実装である．MultAdd と FMA は，多項式の次数 N，l を自然数とした時，要素数 $N*l$ の 64bit 整数配列 a, b, c，および要素数 l の 64bit 整数配列 m を入力にとり，$i \in [N*l], a[i] = [a[i]*b[i]+c[i]]_{m[\lfloor i/N \rfloor]}$ を計算する．$\log_2 N$ と l のパラメータの組を $(\log_2 N, l) \in \{10, 11, 12, 13, 14, 15\} \times \{1, 2, 3, ..., 14, 15\}$ と定義する．なお，実験には 60-bit modulus を使用する．それぞれのパラメータ対に対して 10,000 回実行し，その実行時間の平均を比較する．

5.1.2 実験結果

図 1 は、乗算と加算を順番に適用したもの（MultAdd）と，新たに実装した融合積和演算（FMA）との平均実行時間を比較したものである．入力データサイズ（input_data_size）は，$64*(3*N*l+l)/8$[bytes] で算出し，横軸を入力データサイズ，縦軸を実行時間としている．

図 1 によると，入力データサイズが 2^{19}bytes 付近で実行時間が逆転している．2^{19}bytes 付近より小さいデータサイズの際は MultAdd の方が性能が高く，対して 2^{19}bytes 付近を超えると FMA の方が性能が高くなる．これは，実装とキャッシュサイズに依るものと考えられる．図 1 より，逆転が起きている箇所は，L2 キャッシュサイズ付近であることがわかる．FMA は，処理を一つの関数にまとめた上で，剰余演算を返却時の 1 回に集約を行なっている．しかし，FMA の実装はナイーブであるのに対し，MultAdd で使用している剰余乗算の実装は，内部処理を手動で unrolling している．ゆえに，FMA は MultAdd に比べて L1〜L2 までのキャッシュを有効に使うことができず，入力データサイズが L2 キャッシュに収まる範囲では速度が劣っていると考えられる．一方，入力データサイズが L2 キャッシュに乗り切らなくなった際には，別々の関数呼び出しとなっている MultAdd が速度的に不利となる．これは，それぞれの関数で SIMD レジスタに入力を読み込んでいるため，L3 キャッシュアクセスのレイテンシによる影響を受けていると考えられる．対して，FMA は一度の関数呼び出しのため，SIMD レジスタへの読み込みは 1 回で済む．実際は L2 キャッシュサイズよりも小さいサイズで実行時間の逆転が起こっている．これは，入力データサイズに考慮されていない関数内部の一時変数が影響していると考えられる．

5.2 Trace-Type Function

5.2.1 実験方法

シングルスレッドとマルチスレッドの双方で実行時間の測定を行う．マルチスレッドでは，使用可能なコアを全て活用するため，使用するスレッド数は 8 に設定した．多項式の次数 N，scaling factorΔ は，それぞれ $\log_2 N = 15$，$\Delta = 2^{40}$ に設定した．測定は各パラメータ（rotate-and-sum の適用回数 M，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k，loop-unrolling 後のイテレーション数 h）の組 $(M, l, k, h) \in \{7\} \times \{1, 4, 9\} \times \{1, 5, 10\} \times \{1, 2, 3, ..., 14, 15\}$ ごとに 100 回ずつ行い，その平均実行時間により比較を行う．なお，初回実行時の影響を排除するため，測定の前段階として 1 回余分に実行を行なっている．

5.2.2 実験結果（シングルスレッド）

シングルスレッドでの提案手法による実行時間の改善率を表 2 に示す．normal と hexl は，それぞれ unrolled trace-type function（AVX512 未適用）と提案手法（AVX512 適用済）を指す．なお，改善率は normal の実行時間/hexl の実行時間 で算出した．実行時間の改善率を表したグラフが図 2 である．

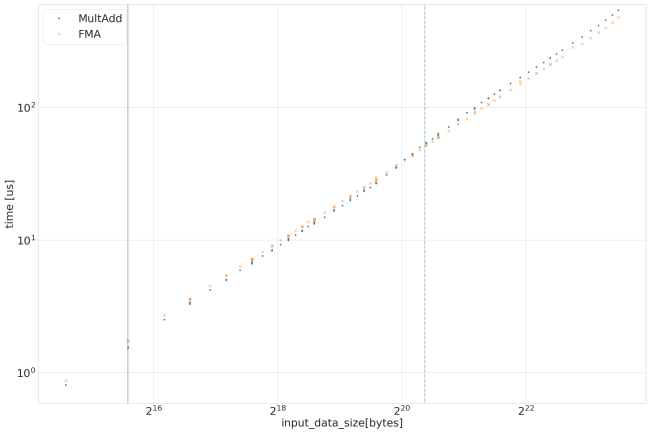

図 1 乗算と加算を順次適用したもの（MultAdd）と融合積和演算（FMA）の入力データサイズ（input_data_size）ごとの平均実行時間（シングルスレッドで 10,000 回実行．60-bit modulus を使用．入力データは N を多項式の次数，l を整数とした時，$N*l$ 要素の 64bit 整数配列 3 つ，および l 要素の 64bit 整数配列（moduli）1 つとする．入力データサイズは，$64*(3*N*l+l)/8$[bytes] で算出した．縦実線と縦破線は，それぞれ L1 キャッシュと L2 キャッシュのサイズを示す．）

表 2 rotate-and-sum の適用回数 M=7 の時の Unrolled Trace-Type Function 実行時間の改善倍率（シングルスレッドで 100 回実行．(改善倍率) = (normal の実行時間)/(hexl の実行時間) で算出．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	1.75	2.04	2.30	1.51	1.53	1.69	1.52	1.46	1.44
2	1.44	1.60	1.79	1.29	1.29	1.42	1.29	1.29	1.27
3	1.28	1.39	1.50	1.19	1.18	1.25	1.20	1.18	1.17
4	1.17	1.28	1.33	1.14	1.11	1.17	1.14	1.12	1.11
5	1.17	1.19	1.26	1.11	1.09	1.13	1.10	1.10	1.09
6	1.10	1.16	1.18	1.09	1.06	1.10	1.09	1.07	1.07
7	1.09	1.13	1.16	1.06	1.05	1.08	1.06	1.06	1.05

10：https://github.com/google/benchmark

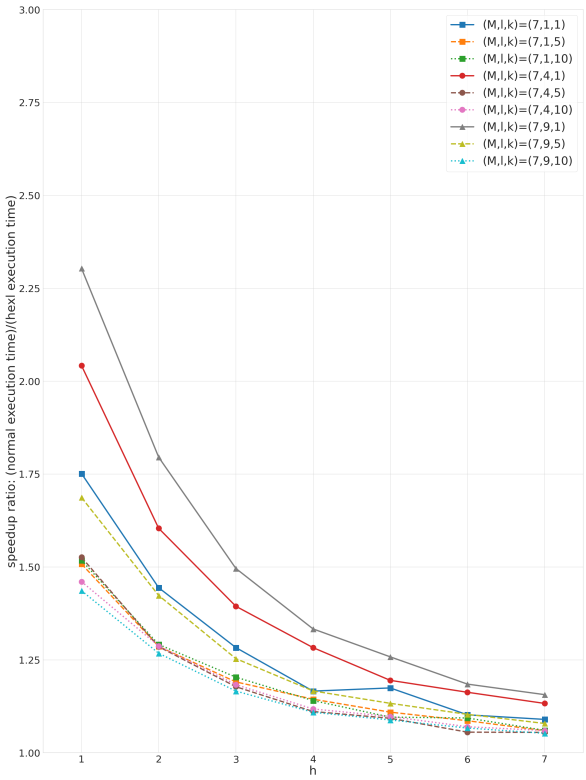

図 2　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間の高速化率（スレッド数 1，rotate-and-sum の実行回数 M=7）

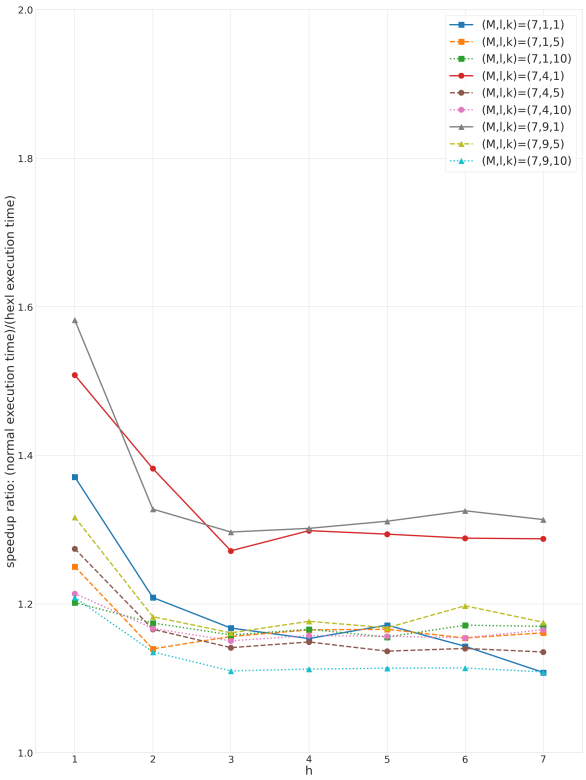

図 3　loop-unrolling 後のイテレーション数 h を変化させた際の実行時間の高速化率（スレッド数 8，rotate-and-sum の実行回数 M=7）

表 3　rotate-and-sum の適用回数 M=7 の時の Unrolled Trace-Type Function 実行時間の改善倍率（スレッド数 8 で 100 回実行．（改善倍率）＝ (normal の実行時間)/(hexl の実行時間) で算出．ただし N/A はメモリ不足により測定不能であることを示す．loop-unrolling 後のイテレーション数 h，暗号文のレベル l，KeySwitch に使用する特殊な法の数 k）

$h\backslash k/l$	1/1	1/4	1/9	5/1	5/4	5/9	10/1	10/4	10/9
1	1.37	1.51	1.58	1.25	1.27	1.32	1.20	1.21	1.21
2	1.21	1.38	1.33	1.14	1.17	1.18	1.17	1.17	1.14
3	1.17	1.27	1.30	1.16	1.14	1.16	1.16	1.15	1.11
4	1.15	1.30	1.30	1.17	1.15	1.18	1.17	1.16	1.11
5	1.17	1.29	1.31	1.17	1.14	1.17	1.16	1.16	1.11
6	1.14	1.29	1.33	1.15	1.14	1.2	1.17	1.15	1.11
7	1.11	1.29	1.31	1.16	1.14	1.18	1.17	1.17	1.11

5.2.3　実験結果（マルチスレッド）

マルチスレッドでの提案手法による実行時間の改善率を表 3 に示す．ただし，normal と hexl は，それぞれ Ishimaki ら [5] の従来手法（AVX512 未適用）と提案手法（AVX512 適用済）を指す．なお，改善率は normal の実行時間/hexl の実行時間 で算出している．図 3 は，実行時間の改善率をグラフとして図示したものである．

5.2.4　考　　察

図 2，3 を見ると，k を固定した時，l が大きくなるにつれて高速化率は高くなっている．これは，l が大きくなる，すなわち暗号文のデータサイズが増加することにより，SIMD によって一度に処理されるデータが増加することに起因するものであると考えられる．

また，図 2 について，l を固定した時の k，および，各パラメータ対における h に注目すると，k や h が大きくなるにつれて高速化率は下がっている．これは，SIMD によって一定の高速化が見込めるものの，全体の計算量は増加するため，SIMD で一度に処理されるデータの量に変化がないことが原因であると考えられる．

マルチスレッドの結果（図 3）を確認すると，前述の特徴は軽微に現れた．しかし，全体として高速化率はほぼ一定，かつシングルスレッドよりも低いことがわかる．これは，マルチスレッドにおける同期コストによるものと考えられる．

6　ま　と　め

本稿では，trace-type function の演算に焦点を絞り，AVX512 を活用した SIMD 化による最適化を行った．実装にあたって，Intel HEXL [8] には実装されていないベクトル同士の剰余積和演算を新たに実装し，Ishimaki らの unrolled trace-type function [5] へ適用した．実験の結果，AVX512 未適用の unrolled trace-type function と比較して 1.05-2.30 倍の計算速度向上を実現した．

現状の課題としては，新たに実装したベクトル同士の剰余積和演算の性能が，パラメータが小さい場合に悪化することが挙げられる．また，実際に trace-type function を活用したアプリケーションに適用した上での性能評価及び，本実験で無効化したハイパースレッディングやターボブーストを有効化した上での検証等，実運用を想定した検証を行っていく必要がある．

謝　　辞

本研究は，JST CREST（JPMJCR1503）の支援を受けたものである．

文　　献

[1] International Data Corporation (IDC): Data Creation and Replication Will Grow at a Faster Rate than Installed Storage Capacity, According to the IDC Global DataSphere and StorageSphere Forecasts, https://www.idc.com/getdoc.jsp?containerId=prUS47560321 (2021). (Accessed on 01/06/2022).

[2] Cheon, J. H., Kim, A., Kim, M. and Song, Y.: Homomorphic encryption for arithmetic of approximate numbers, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10624 LNCS, pp. 409–437 (2017).

[3] Cheon, J. H., Han, K., Kim, A., Kim, M. and Song, Y.: A Full RNS Variant of Approximate Homomorphic Encryption, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 11349 LNCS, pp. 347–368 (2019).

[4] Smart, N. P. and Vercauteren, F.: Fully homomorphic SIMD operations, *Designs, Codes and Cryptography*, Vol. 71, No. 1, pp. 57–81 (2014).

[5] Ishimaki, Y. and Yamana, H.: Faster Homomorphic Trace-Type Function Evaluation, *IEEE Access*, Vol. 9, pp. 53061–53077 (2021).

[6] Jung, W., Lee, E., Kim, S., Kim, J., Kim, N., Lee, K., Min, C., Cheon, J. H. and Ahn, J. H.: Accelerating fully homomorphic encryption through architecture-centric analysis and optimization, *IEEE Access*, Vol. 9, pp. 98772–98789 (2021).

[7] Boemer, F., Cammarota, R., Demmler, D., Schneider, T. and Yalame, H.: MP2ML: A Mixed-Protocol Machine Learning Framework for Private Inference, Cryptology ePrint Archive, Report 2020/721 (2020). https://ia.cr/2020/721.

[8] Boemer, F., Kim, S., Seifu, G., de Souza, F. and Gopal, V.: Intel HEXL: Accelerating Homomorphic Encryption with Intel AVX512-IFMA52, *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '21, New York, NY, USA, pp. 57–62 (2021).

[9] Mohan, P. V. A.: *Residue Number Systems*, Springer International Publishing (2016).

[10] PALISADE Lattice Cryptography Library (release 1.9.2), https://palisade-crypto.org/ (2020).

[11] Bajard, J. C., Eynard, J., Hasan, M. A. and Zucca, V.: A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 10532 LNCS, pp. 423–442 (2017).

[12] Han, K. and Ki, D.: Better bootstrapping for approximate homomorphic encryption, *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Vol. 12006 LNCS, pp. 364–390 (2020).

[13] Gentry Craig, Halevi, S. and Smart Nigel P: Fully Homomorphic Encryption with Polylog Overhead, *Advances in Cryptology – EUROCRYPT 2012* (Pointcheval David and Johansson, T., eds.), Berlin, Heidelberg, pp. 465–482 (2012).

[14] Halevi, S. and Shoup, V.: Algorithms in HElib, *Advances in Cryptology – CRYPTO 2014* (Garay, Juan, A. and Gennaro, R., eds.), Berlin, Heidelberg, pp. 554–571 (2014).

[15] Gentry, C., Halevi, S., Jutla, C. and Raykova, M.: Private Database Access with HE-over-ORAM Architecture, *Applied Cryptography and Network Security* (Malkin Tal, Kolesnikov, V. and Lewko Allison BishopPolychronakis Michalis, eds.), Cham, pp. 172–191 (2015).

[16] Badawi, A. A., Hoang, L., Mun, C. F., Laine, K. and Aung, K. M. M.: PrivFT: Private and Fast Text Classification With Homomorphic Encryption, *IEEE Access*, Vol. 8, pp. 226544–226556 (2020).

[17] Intel Corporation: Intel ® Intrinsics Guide, https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html. (Accessed on 01/22/2022).

[18] Aguilar-Melchor Carlos, Barrier, J. G. S., Guinet Adrien, Killijian Marc-Olivier and Lepoint Tancrède: NFLlib: NTT-Based Fast Lattice Library, *Topics in Cryptology - CT-RSA 2016* (Sako, K., ed.), Cham, pp. 341–356 (2016).

[19] Halevi, S. and Shoup, V.: Faster Homomorphic Linear Transformations in HElib, *Advances in Cryptology – CRYPTO 2018* (Shacham, H. and Boldyreva, A., eds.), Cham, pp. 93–120 (2018).

[20] Bossuat Jean-Philippe, Mouchet, C., Troncoso-Pastoriza Juan and Hubaux Jean-Pierre: Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys, *Advances in Cryptology – EUROCRYPT 2021* (Canteaut Anne and Standaert, F.-X., eds.), Cham, pp. 587–617 (2021).

[21] PALISADE Lattice Cryptography Library (release 1.11.5), https://palisade-crypto.org/ (2021).