FHE in Action

Self Introduction

- 陸　文杰 (Wen-jie Lu)。中国広東出身。
- 2009 - 2013 中国 JiNan University（広州）情報科学学部卒.
- 2013 - 2014 言語学校
- 2014 - 2019 筑波大学博士課程。指導教員：佐久間 淳。DC2
- 研究： secure multi-party computation, homomorphic encryption.

FHE In Action

- 現在の 完全準同型暗号 (FHE) について自分の理解
- 既存の FHE 実装をどう使う my-know-how

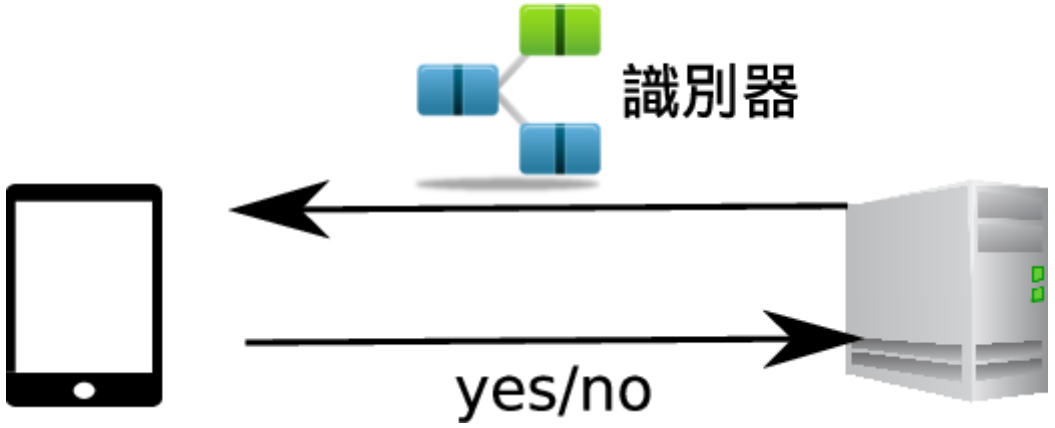
なぜ準同型暗号を使おうとするのか?

- データや情報を秘密したまま何らかの計算を行いたいから

なぜ準同型暗号を使おうとするのか?

- データや情報を秘密したまま何らかの計算を行いたいから

例：app のユーザは転売屋かどうかを検知する

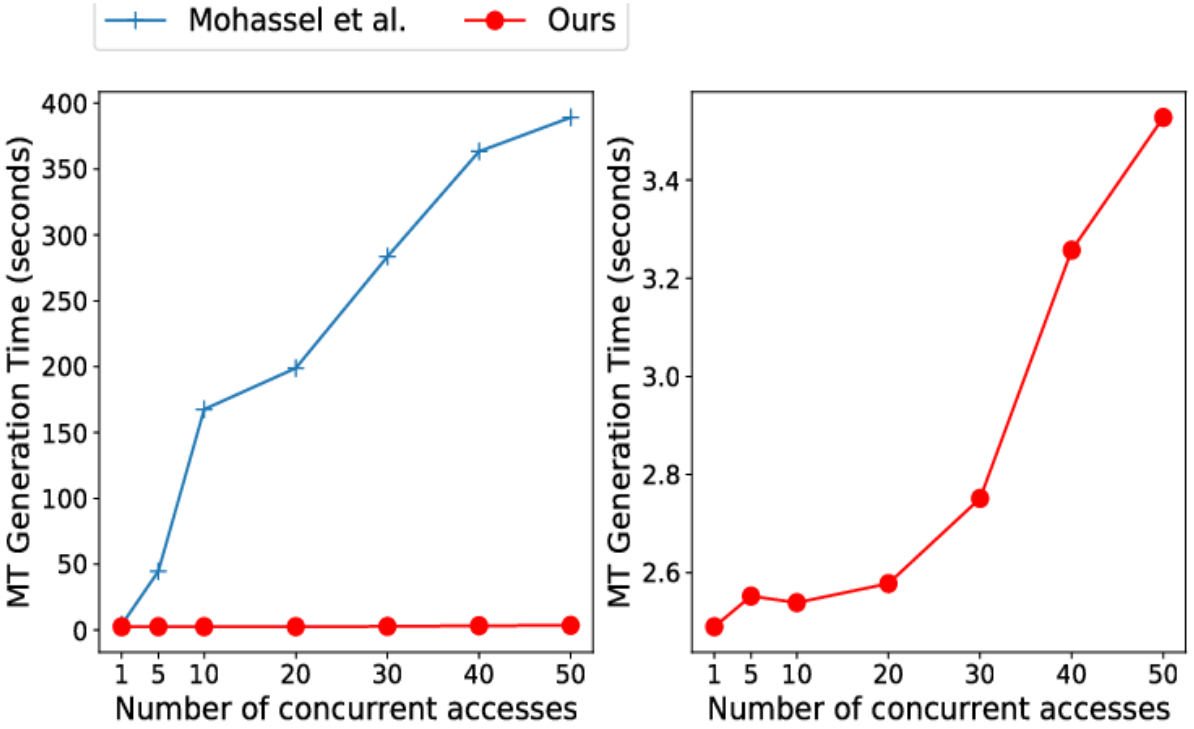


サーバーから識別器をユーザへに push

- ユーザの情報 (i.e., 携帯の物理情報など) はサーバへに収集できない
- 識別器は reverse engineering される恐れがある
- 現在は コードの obfuscation ベースでやっているらしい

なぜ秘密分散や GC ベースだけならいけないのか

- 通信 latency や通信量に厳しい
- 帯域より計算力の方が scalable
- e.g. 10 Gbps 帯域においてサーバー一台 v.s. 50 クライアント. 同時に行列積の秘密計算。OT vs. FHE



FHE を使う他の場面

- Secure Multi-party Computation
- Secure Outsourcing
- SMC -> Threshold-FHE + Secure Outsourcing
- etc.

FHE: A Black Box View

Fully Homomorphic Encryption

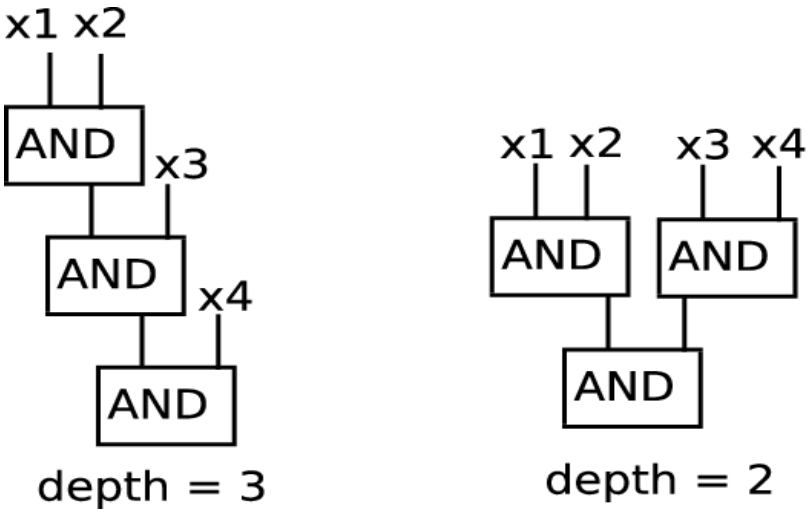
- 暗号化したまま「足し算」と「掛け算」が可能な暗号方式

『クラウドを支えるこれからの暗号技術』@herumi さん

- $GF(2)$上 +, x あれば完備なので $\Rightarrow$ 任意の Boolean 関数を暗号文上で計算できる
- The glory of cryptography で呼ばれる理由 (?)

FHE and Boolean Circuit

- 一ビット $b_1 + b_2 \mod 2 = b_1 \mathrm{X}OR\, b_2$
- $b_1 \times b_2 \mod 2 = b_1 \mathrm{A}ND\, b_2$
- 関数 F を boolean 回路に変換して評価する。
- key: 回路の深さをどれだけ小さくできるのか.

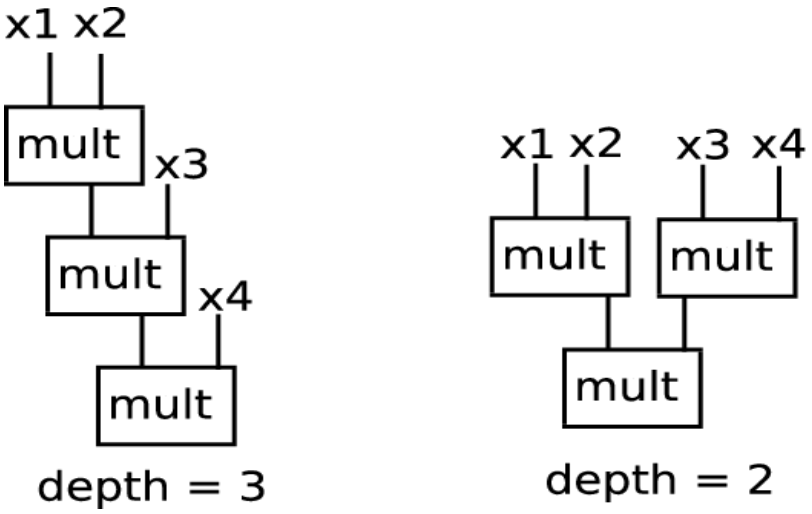


同じ関数でも異なる boolean circuit がある

- 電子回路と類似、深いほど計算時間がかかる (理由は違うが)

FHE and Arithmetic Circuit

- 平文はビットではなく 有限体の元である. $\mathbb{Z}_t$ for a prime t.
- $x_1, x_2, x_3, x_4 \in \mathbb{Z}_t$
- $x_1 \times x_2 \times x_3 \times x_4 \mod t$ を計算する AC



- 同様、AC の深さは重要なパラメータ

BC vs. AC: Pros and Cons

- Boolean circuit (BC) 的な考え方

pros: 任意の関数を評価可能
cons: メモリコスト と 計算コスト
- ciphertext expansion が悪い. 1 bit を暗号すると 数 KB の暗号文.

- Arithmetic circuit (AC) 的な考え方

pros:
- 線形計算な関数をより浅いな AC を作れる.
- ciphertext expansion がいい (SIMDを利用する)

cons:
- 非線形関数はそのまま評価できない. e.g., max, mux

- どちらも準同型の演算を早くする必要がある.
- 計算コストを削減するテクニックはいくつがある. (これから紹介する)

A Buggy but Informative Construction

A Buggy but Informative Construction

- Gaussian 分布から キーをランプリングする. $\mathsf{sk} \leftarrow \chi^n$.
- Encryption: $m \in \mathbb{Z}_t, (e.g., t = 2)$. $a \leftarrow \mathbb{Z}_q^n, e \leftarrow \chi'$.

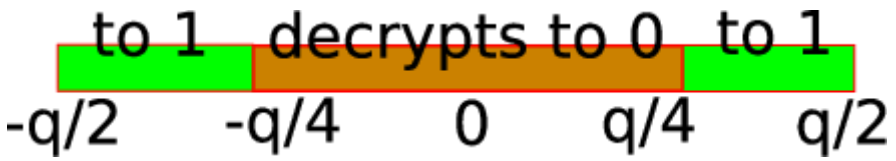
$$\mathsf{ctx} = (a, b = m + e - a \cdot \mathsf{sk} \mod q).$$

- Learning with Errors (LWE) に帰着
- ここの · はまだ定義しない

A Buggy but Informative Construction

- Decryption, ctx $= (a, b = m + e - a \cdot \mathsf{sk})$.

$$\tilde{m} = b + a \cdot \mathsf{sk} \mod q. \text{ outputs } \begin{cases} 0 & |\tilde{m} - 0| < |\tilde{m} - q/2| \\ 1 & \text{o.w.} \end{cases}$$



- $|e| < q/4$ なら正しく復号できる

A Buggy but Informative Construction (Cont'd)

- $\mathsf{ctx}_1 = (a_1, b_1), \mathsf{ctx}_2 = (a_2, b_2)$ とする
- Addition : $\mathsf{ctx}_1 \oplus \mathsf{ctx}_2 = (a_1 + a_2, b_1 + b_2)$
- Multiplication: $\mathsf{ctx}_1 \otimes \mathsf{ctx}_2 = (a_1 \cdot a_2, a_1 \cdot b_2 + a_2 \cdot b_1, b_1 \cdot b_2)$

四つの特徴
1. noisy な暗号文
2. 計算によって暗号文ノイズが増加。しきい値を上回ったら復号できなくなる。
3. 復号の関数の回路は "浅い"
4. 掛け算によって暗号文のレングスも変化する (e.g., 2から3になる)

ノイズの増やす方

1. 初期のノイズ(の分散)は $O(n^2 + nt^2W)$ (W は hamming weight)
2. 準同型足し算 $\mathrm{var}_1 + \mathrm{var}_2$
3. 準同型掛け算によってノイズの分散 $\mathrm{var}_1 \times \mathrm{var}_2 \times \mathrm{length}$
4. addPlain: $\mathrm{var} + O(nt^2)$
5. multPlain: $\mathrm{var} \times O(nt^2)$

- 平文の t はあまり大きくしない
- big number な平文が欲しい時、小さい t で表現する (e.g., CRT decomposition)

計算効率がいい FHE へ

計算効率がいい FHE への道

- Ring-LWE ベースなスキームの方が 準同型演算がより効率だ.

FFTにより $O(n^2) \to O(n \log n)$
例えここの a, b, e はすべて $\in \mathbb{Z}_q[X]/(X^n+1)$ である

$$\mathsf{ctx} = (a, b = m + e - a \cdot \mathsf{sk} \mod (q, X^n + 1))$$

- 次数は n 以下かつ係数は $\mod q$ の多項式の集合
- "·" は多項式環上の掛け算で定義される

計算効率がいい FHE への道

1. 多項式ベースな RLWE スキームは LWE スキームより効率. $O(n^2) \to O(n \log n)$

2. 暗号文のノイズを抑えるテクニック

- Mod-switch [BGV11]
- reLinearization [BV11b]
- Bit-decomposition [BV11a]
- Bootstrapping. 暗号文をさらに暗号化して、復号関数を暗黙的に評価する

計算効率がいい FHE への道

1. 多項式ベースな RLWE スキームは LWE スキームより効率. $O(n^2) \to O(n \log n)$

2. 暗号文のノイズを抑えるテクニック

- Mod-switch [BGV11]
- reLinearization [BV11b]
- Bit-decomposition [BV11a]
- Bootstrapping. 暗号文をさらに暗号化して、復号関数を暗黙的に評価する

3. 実装面の高速化

- Packing (SIMD) [SV11]
- Residue Number System (RNS)

Modulus Switching (Mod-Switch)

技その１：Mod-Switch

文字通り：$\bmod\ q \to \bmod\ q'\ (q > q')$の暗号文に変換することによってノイズを削減する

- 暗号文の累積ノイズeは準同型演算によって増加してまう。
- 初期のノイズ$|e| < B$とする。

1. 一回足し算$|e_+| < 2B$。
2. 一回掛け算$|e_\times| < B^2$. 更に掛け算すると$|e_\times| < B^4$.

- ノイズ$|e| > q/4$なると正しく復号できなくなる
- $q \approx 4B^L$とする。1-st Gen な FHE は$\log L$の深さの回路しか評価できない

技その１：Mod-Switch

- 初期のノイズ $|e| < B$ をとする。

Mod-Switch: 準同型掛け算した (する) 時、暗号文ctx $= (a, b)$ をBで割る

1. $a' = a/B, b' = b/B$ (実際は少し複雑)
2. ノイズのスケールも削減される $|e_\times| < B^2 \to |e_\times| < B$
3. 法 q も小さくになる $q \approx 4B^{L-1}$.
4. 初期の法 は $q \approx 4B^L$なので深さ L の回路を評価できる

技その１：Mod-Switch (Cont'd)

pros

1. 評価できる回路の深さ $logL \to L$ まで

cons

1. 異なる法に従う号文が同時に存在しうるので暗号文の法をトラックする必要がある.

暗号文に割り算をする idea と近い. *invariant* なスキーム (e.g., FV スキーム, YASHE スキームなど)

- 法は変わらないのでトラックする必要もない
- 実装面的はシンプル

初期ノイズ $O(n^2 + nt^2W)$ の W について

技その2：Hamming Weighted Secret Key

- 秘密鍵 sk は $\{0,1\}^{n+1}$ から生成する。かつ$|\mathsf{sk}| = W$ where $W \ll n$. e.g., $W = 64$とか.
- RLWE なスキームで 法の円分多項式 $\Phi_m(X)$ の $m = 2^k$　を使わないと安全性が問題ありそうな論文もあったみたい.
- e.g., $X^{2^{k-1}} + 1 = \phi_{2^k}(X)$

reLinearization (or Key Switching)

技その3: reLinearization

- 掛け算によって暗号文のレングスが増える。$\mathsf{ctx}_1(m_1) \otimes \mathsf{ctx}_2(m_2) = (a', b', c')$
- レングスの増加によって、次の準同型演算のコストも増える
- そして、掛け算に増加するノイズはこのレングスにも依存する

平文を保持しつつ、レングスを縮む。

$$\mathrm{reLinearization}(a', b', c') \to (a'', b''),\ \text{while } \mathrm{Dec}((a'', b'')) \to m_1 m_2$$

- レングス 2 の暗号文は canonical form で呼ばれる

技その3: reLinearization

$$\mathrm{reLinearization}(a', b', c') \to (a'', b''),\ \text{while}\ \mathrm{Dec}((a'', b'')) \to m_1 m_2$$

- レングス 3 の暗号文はどう復号するのか？ $\Rightarrow c' + b' \cdot \mathsf{sk} + a' \cdot \mathsf{sk}^2$

main idea: sk^2 を事前に暗号化して evaluation key に加える

1. Enc(sk^2) = (α, β), i.e., $\beta + \alpha \cdot \mathsf{sk} \to \mathsf{sk}^2$
2. $b'' = c' + \beta \cdot a', a'' = b' + \alpha \cdot a'$.
3. $b'' + a'' \cdot \mathsf{sk} = c' + \beta \cdot a' + (b' + \alpha \cdot a') \cdot \mathsf{sk} = c' + b' \cdot \mathsf{sk} + a' \cdot \mathsf{sk}^2$

ノイズは増加される。

技その3: reLinearization

- 計算のステップ $b'' = c' + \beta \cdot a', a'' = b' + \alpha \cdot a'$ を見直す

しかし α, β は大きい値 ($\mathbb{Z}_q^n$) をとりうるため累積ノイズがかなり増加しまう

- bit-decomposition を使う

$$\alpha := \sum_k \alpha_k \omega^k, \beta := \sum_k \beta_k \omega^k, \omega \ll q$$

- 極端に $\omega = 2$ (i.e., bit-decomposition).
- だが $\omega = 2$ 時 α_k, β_k を保存するコストは高すぎ
- 実装上は $\omega > 2, k = 4$ など使う (HElib). あるいは $|\omega| \approx 2^{16}$ (SEAL)

技その3: Key-Switching

- 暗号文のレングスを 2 に戻るため sk^2 を暗号化する
- もっと一般的な使い方がある. $\mathsf{ctx}_{\mathsf{sk}'}(m)$ をある鍵sk'で mの暗号文.

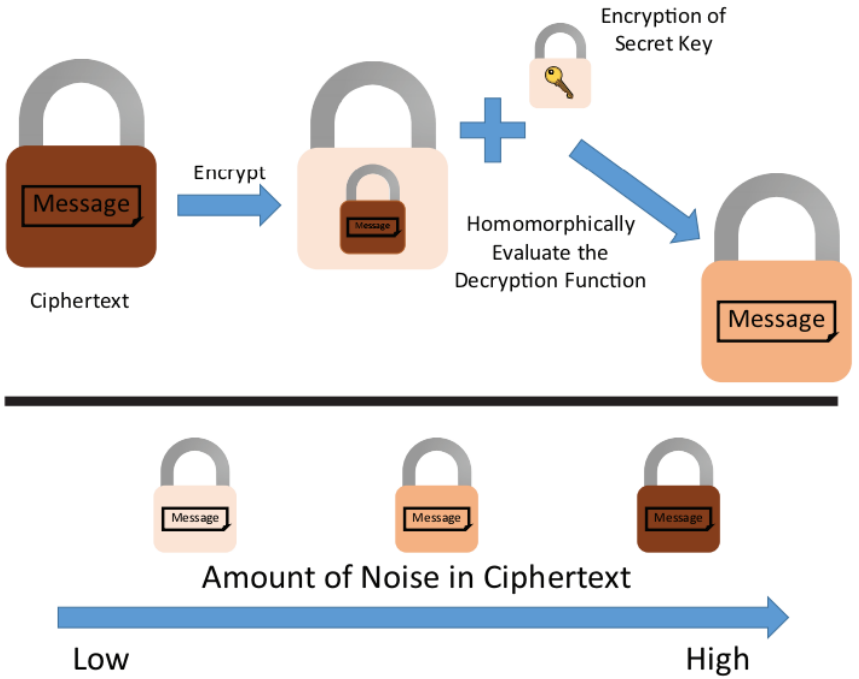
$$\mathsf{KeySwitch} : \mathsf{ctx}_{\mathsf{sk}'}(m) \to \mathsf{ctx}_{\mathsf{sk}}(m)$$

別の鍵sk の暗号文に変換することができる。

例え
1. $\mathsf{sk}' = \mathsf{sk}^2$ (i.e., reLinearization)
2. $\mathsf{sk}'(X^j) = \mathsf{sk}(X)$ (i.e., automorphism. homomorphic rotation で使う. まだ議論する)

Bootstrapping

- 復号回路もBCで表示できる。準同型性を使って復号回路を暗号文上で評価する。



from David Wu cs.stanford.edu/~dwu4/papers/XRDSFHE.pdf (https://cs.stanford.edu/~dwu4/papers/XRDSFHE.pdf)

- 復号鍵を公開鍵を使って暗号化し、boostrapping キーを作る ⇒ circular security

Bootstrapping 流派

1. RLWE-based なスキーム, e.g., BGV, FV
pros:

- > 2 bits の平文を暗号化するができる
- bootstrapping するまでに L-level の回路を評価することができる

cons:

- bootstrapping はかなり重い。e.g., 数分単位/一暗号文

2. LWE-based + GSW 方式なスキーム, e.g., TFHE
pros:

- 1-bit の平文に特化するため bootstrapping は高速. e.g., 0.1sec / 一暗号文

cons:

- gate ごとに bootstrapping する必要がある

Bootstrapping の困難点

1. 丸めの操作は非線形. 復号関数の深さは厳しい

$$m \leftarrow \underbrace{c - \sum s_i \cdot u'_i}_{\text{“simple part”}} - \underbrace{\left\lceil 2^{-\kappa} \cdot \sum s_i \cdot u''_i \right\rfloor}_{\text{“complicated part”}} \bmod p.$$

Cite from "Fully Homomorphic Encryption without Squashing Using Depth-3 Arithmetic Circuits"

2. 法 p は大きい、e.g., $p \approx 2^{1000}$. この暗号文を暗号化するため、もっと大きい FHE パラメータが必要

Bootstrapping の困難点 (Cont'd)

- 復号回路を BC 回路で表現する時深さは結構ある

cyclotomic ring m	21845 =257·5·17
lattice dim. $\phi(m)$	16384
plaintext space	$\mathrm{GF}(2^{16})$
number of slots	1024
security level	76
before/after levels	22/10
initialization (sec)	177
linear transforms (sec)	127
digit extraction (sec)	193
total recrypt (sec)	**320**
space usage (GB)	3.4

Cite from "Bootstrapping in HElib"

- That is, each 10 level multiplication needs one bootstrapping

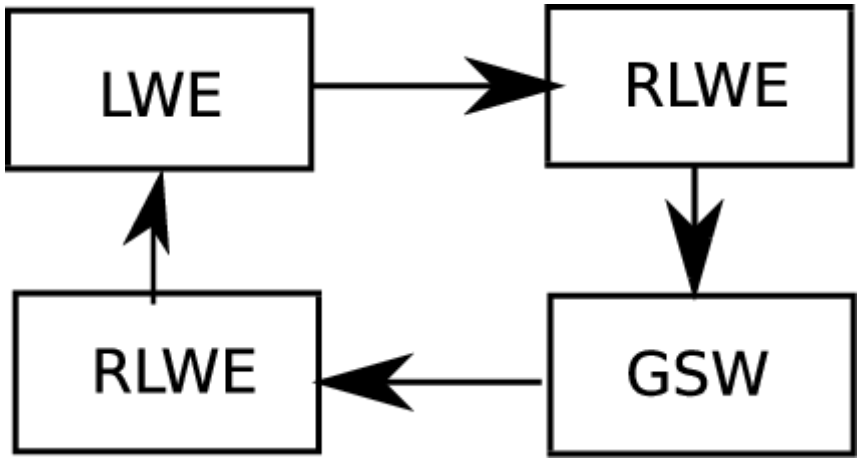
最速な 1 ビットの bootstrapping (TFHE)

Bootstrapping within 0.1sec [TFHE]

- メッセジーが二値の LWE 暗号文 $(\vec{a}, b)$ の bootstrapping
- 復号回路：$|b + \vec{a}^\top \vec{s}| >^? q/4$

1. 内積はどう計算するのか
2. $>^?$ をどう計算するのか

- main idea: GSW方式に経由して高速化
- $>^?$ の計算には $X^n + 1$ の構造をうまく利用する



前置き：ベクトル内積と多項式積

- length-n のベクトル積と法 $X^n + 1$ において多項式積は結んでいる
- $\vec{a}^\top \vec{b} = (\sum_{i=0}^{n-1} a_i X^i \times \sum_{j=0}^{n-1} -b_j X^{n-j})[0] \mod X^n + 1$
- i.e., $X^n = -1 \mod X^n + 1$
- bootstrapping 以外の色々使い道がある

Bootstrapping within 0.1sec [TFHE]

- もっとも重要なトリック

$$X^{\beta} \times (1 + X + X^2 \cdots + X^{n-1}) \times X^{-n/2} \times 2^{-1} + 2^{-1} \mod X^n + 1$$

の定数項 (i.e., $??X^0$)

- $\beta > n/2$ なら定数項は 0
- o.w. 定数項は 1

次はどう $\beta = \vec{b} + \vec{a}^{\top}\vec{s}$ を多項式の次数に計算するについて

Bootstrapping within 0.1sec [TFHE]

- 三種類の暗号文

LWE(m) : (bootstrapping の対象)

$$(\vec{a}, b = m + e - \vec{a}^{\top}\vec{s}), m \in \{0,1\}\ \vec{a}, \vec{b} \in \mathbb{Z}_q^{n-1}$$

RLWE(m'): (中間表現役)

$$(a', b' = m' + e' - a's')\ a', s', m' \in \mathbb{Z}_q[X]/(X^n + 1)$$

GSW(m'): (boostrappingキーを暗号化)

$$\begin{bmatrix} RLWE_{s'}(0) \\ \vdots \\ RLWE_{s'}(0) \end{bmatrix} + m'H\ (H \text{ is a fixed poly matrix })$$

Three Hacks in TFHE

Bootstrapping within 0.1sec [TFHE]

TFHE Hack 1: approximate gadget decomposition

- 多項式ベクトル $\vec{u} := (a', b')$ を分解する. $a', b' \in \mathbb{Z}_q[X]/(X^n + 1)$
- Decomp: $\vec{u} \mapsto \vec{v}$ s.t. $\vec{v}H \approx \vec{u}$ かつ 誤差はバンドされる

Bootstrapping within 0.1sec [TFHE]

TFHE Hack 1: approximate gadget decomposition

TFHE Hack 2: External Product

- $LWE(m) = (\vec{a}, b)$ を $m \in \{0, 1\}$ の LWE 暗号文とする
- $GSW(m')$ を $m' \in \mathbb{Z}_t[X]/(X^n + 1)$ の GSW 暗号文とする
- Decomp($[0, \underbrace{\vec{a} \| b}_{\text{多項式と見なす}}]$) $\cdot GSW(m')$ は $RLWE(mm')$になる

Bootstrapping within 0.1sec [TFHE]

TFHE Hack 1: approximate gadget decomposition

TFHE Hack 2: External Product

TFHE Hack 3: Extract LWE from RLWE

- $RLWE(mm')$ から定数項を $LWE(mm'[0])$ の形で抽出できる。

1. $RLWE(mm') := (a', b' = mm' + e' - a's')$, s' は秘密鍵
2. 抽出した $LWE(mm'[0]) := (\vec{c}, d)$ とする.
3. $\vec{c} = [-a'[n-1], -a'[n-2], \ldots, -a'[0]]$ (次数の逆順と符号 flip).
4. $d = b'[0]$ (多項式 b' の定数項)

- $(\vec{c}, d)$ を復号する時. $d + \underbrace{\vec{c}^\top \vec{s}}_{(a's')[0]}$ を計算する

- よって $d + \vec{c}^\top \vec{s} = mm'[0] + e'[0]$

Bootstrapping within 0.1sec

まとめ

1. LWE の秘密鍵 $\vec{s} \in \{0,1\}^{n-1}$ をGSWで暗号化. boostrappingキーとする
bk $:= \{GSW(\vec{s}_i)\}_i$.
- GSW の鍵は? LWE と異なる鍵を使ったりする (この場合 KeySwitchが必要)
2. boostrapping の対象 LWE の暗号文 $(\vec{a}, b)$ とする. $(|b + \vec{a}^\top \vec{s}| >^? q/4)$
2.1 積を計算する。$(H + X^{\vec{a}_i} - 1) \cdot GSW(\vec{s}_i) \to \mathrm{GSW}(X^{\vec{a}_i \vec{s}_i})$
- 整数 $\vec{a}_i$ を次数に置くこともポイント
- ここでは $\vec{s}_i \in \{0,1\}$ ので, $\vec{s}_i = 1$の場合 $X^{\vec{a}_i \vec{s}_i} = X^{\vec{a}_i \vec{s}_i}$

Bootstrapping within 0.1sec

まとめ

1. LWE の秘密鍵 $\vec{s} \in \{0,1\}^{n-1}$ をGSWで暗号化. boostrappingキー bk $:= \{GSW(\vec{s}_i)\}_i$.

2. boostrapping をするLWE の暗号文 $(\vec{a}, b)$ とする. $(|b + \vec{a}^\top \vec{s}| >^? q/4)$

2.1 積を計算する。$(H + X^{\vec{a}_i} - 1) \cdot GSW(\vec{s}_i) \to \mathrm{GSW}(X^{\vec{a}_i \vec{s}_i})$

- ここでは $\vec{s}_i \in \{0,1\}$ ので, $\vec{s}_i = 1$の場合 $X^{\vec{a}_i \vec{s}_i} = X^{\vec{a}_i \vec{s}_i}$

2.2 summation を計算する。

Decomp(RLWE(X^b)) GSW($X^{\vec{a}_i \vec{s}_i}$) $\to$ RLWE($X^{b+\vec{a}_i \vec{s}_i}$)

- 再帰的に RLWE($X^{b+\vec{a}^\top \vec{s}}$) を計算できる

Bootstrapping within 0.1sec

1. LWE の秘密鍵 $\vec{s} \in \{0,1\}^{n-1}$ をGSWで暗号化. boostrappingキー bk $:= \{GSW(\vec{s}_i)\}_i$.
- GSW の鍵は?
- LWE と異なる鍵を使ったりする (この場合 KeySwitchが必要)
2. boostrapping をするLWE の暗号文 $(\vec{a}, b)$ とする. $(|b + \vec{a}^\top \vec{s}| >^? q/4)$
2.1 積を計算する。$(H + X^{\vec{a}_i} - 1) \cdot GSW(\vec{s}_i) \to \mathrm{GSW}(X^{\vec{a}_i \vec{s}_i})$
- ここでは $\vec{s}_i \in \{0,1\}$ ので, $\vec{s}_i = 1$の場合 $X^{\vec{a}_i}\, \vec{s}_i = X^{\vec{a}_i \vec{s}_i}$
2.2 summation を計算する。Decomp(RLWE(X^b)) · GSW($X^{\vec{a}_i \vec{s}_i}$) $\to$ RLWE($X^{b+\vec{a}_i \vec{s}_i}$)
- 再帰的に RLWE($X^{b+\vec{a}^\top \vec{s}}$) を計算できる
2.3 トリック：RLWE($X^{b+\vec{a}^\top \vec{s}}$) $\cdot (1 + X + \cdots + X^{N-1}) \cdot X^{-N/2} \times 2^{-1} + 2^{-1}$
2.4 定数項を抽出して LWE に戻る

- $>^? q/4$ ではなく $>^? N/2$ の rescaling ステップは省略した
- ノイズの解析は論文に参考

これから実装っぽいの話です

既存のもの

1. HElib by IBM. 最近 Apache 2.0 ライセンスに変わった
- 基本 Shai Halevi さんだけメンテナンスしている. わりと inactive
- 機能はかなりある. mod switch, reLinearization, rotation, boostrapping など
- NTL, GMP などの外部ライブラリーに依存. ビルドはやや大変

2. Simple Encrypted Arithmetic Library (SEAL) by Microsoft. Microsoft のライセンス
- 2-3 人がメンテナンス + インターンを雇っているらしい.
- 機能は上々に増加. rotation -> bootstrapping (まだreleaseされてない)
- 外部の依存なし、out-of-the-box で使える
- 完全openではなくメールを送ってgitの招待をお願いする

3. TFHE
- bootstrapping が一番高速.
- AND, XOR など色々な boolean gate が実装された.
- フランスの大学の学生さん3人がメンテナンス。commentもちょこちょこフランス語

その他：PALISADE by NJIT, $\Lambda \circ \lambda$

「FHE の計算を早くする」

1. 暗号化した後の optimization

- big number の表示
- 多項式の表示

2. 暗号化する前の optimization

- SIMD による平文の前処理

回路的な考え方すると

- 1 はゲートを早く動かす
- 2 は良い回路を設計する

暗号化した後にやっていること

「FHE の計算を早くする」その 1: 大きい法 q を表現する.

- $\mathbb{Z}_q[X] \to \mathbb{Z}_{q_1}[X] \times \cdots \times \mathbb{Z}_{q_j}[X]$, each $q_j < 2^{64}$
- for $i \neq j$ q_i, q_j は co-prime
- Residue number system (RNS) でも呼ばれる
- Almost parallelizable

[CBHHJLL18] は logistic regression を暗号文上で学習させるため $q \approx 2^{1020}$ を使った

- BigNumber 「できるだけ」使用しない. rounding 操作が必要になる時一回 BigNumber に戻る.

例え $0 < c < q$, $\lfloor c/t \rceil$ など

- [BEHZ16] は BigNumber に戻らせず rounding を行う方法は SEAL (v2.~) に実装された (big number free)

「FHE の計算を早くする」その 2: 多項式掛け算

- RLWE-based のスキームでは FFT と NTT を利用して計算を高速化にする
- 多項式を二つの表現がある

1. シンプルな係数方式 $A(X) = \sum a_i X^i \in \mathbb{Z}_q[X]/(X^n + 1)$.

- SEAL, TFHEなどのライブラリーに採用される
- 一回掛け算のコスト $O(n \log n)$ (Numeric Theoretic Transform)

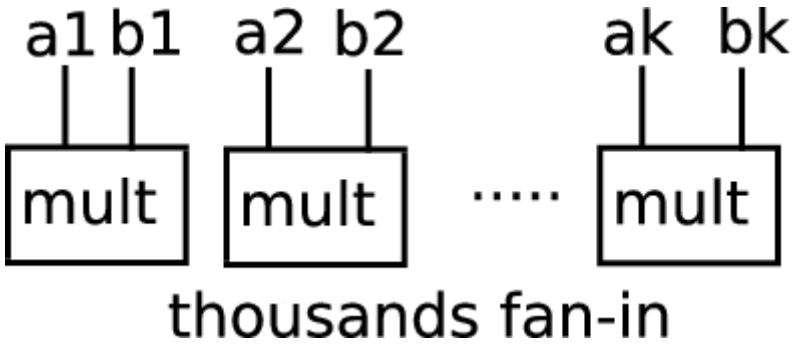
2. DoubleCRT 方式 (HElibの実装)

- $A(X) \to L \times n$ の整数行列 (L 回 FFT経由)
- 掛け算のコストは $O(Ln)$. (i.e., 行列の要素ごと積)
- メモリ使用量は 係数方式の L 倍.

暗号する前にやっていること

「FHE の計算を早くする」その 3: Single Instruction Multiple Data (SIMD) によって前処理

- SIMD は fan-in が大きいな回路に対する optimization
- e.g., お互い独立な batch 処理



- 暗号化する前に $\{a_k\}, \{b_k\}$ をそれぞれ encode をかける

1. $\{a_k\} \to A \to \mathsf{Enc}(A)$, $\{b_k\} \to B \to \mathsf{Enc}(B)$
2. $\mathsf{Enc}(A) \otimes \mathsf{Enc}(B) \to \mathsf{Dec} \to AB \to \{a_k \times b_k\}_k$

- encodeすることによって k-fan-in の回路は準同型掛け算は一回で済む
- 具体的には多項式版の Chinese Remainder Theorem を利用

多項式版 CRT

$A = 8 \mod X + 15, \quad A = 5 \mod X + 9$
$A = 16 \mod X + 2, \quad A = 9 \mod X + 8$
$\rightarrow A = 1 + X + 7X^2 + 12X^3$

- $[8, 5, 16, 9] \Leftrightarrow A \in \mathbb{Z}_{17}[X]/(X^4 + 1)$

p.s. $X^4 + 1 = (X + 15)(X + 9)(X + 2)(X + 8) \mod 17$

多項式版 CRT

- $[8, 5, 16, 9] \Leftrightarrow A \in \mathbb{Z}_{17}[X]/(X^4+1)$
- 一つの大きい集合を幾つの小さい集合の積集合で合成する

$$\mathbb{Z}_{17}[X]/(X^4+1) \cong \mathbb{F}_1 \otimes \mathbb{F}_2 \cdots \otimes \mathbb{F}_\ell (i.e., \ell = 4)$$

- 大きい集合において一回の演算(+, x)は同時に ℓ 個小さい集合に反映する

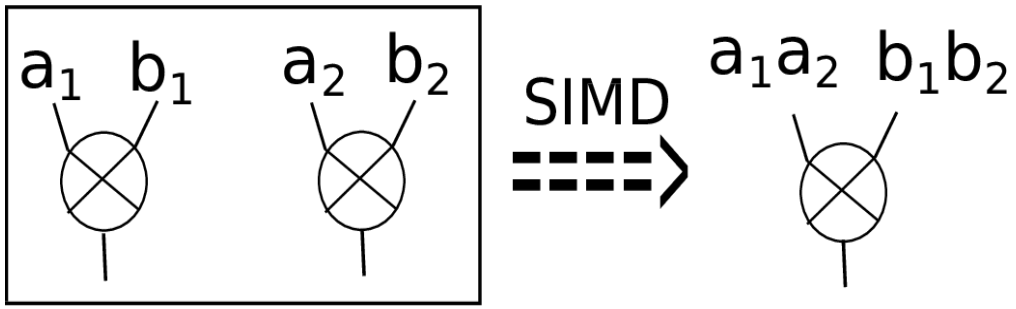
1. $3 \times A \Leftrightarrow 3 \times [8, 5, 16, 9]$
2. $A^2 \Leftrightarrow [8^2, 5^2, 16^2, 9^2]$

- 平文のデータはここのベクトルに想定、暗号化するのは多項式 A
- 合成する集合の個数は $\ell = n/\text{multiplicative order}(t, 2n)$.

Deeping in SIMD

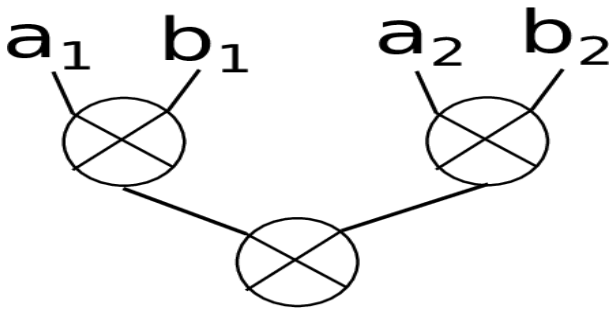
Deeping in SIMD: rotation

- SIMD によって評価するゲート数を削減する



SIMD によって評価するゲート数を削減する

- Intersection がある回路は SIMD できない



intersection がある回路は SIMD できない

Deeping in SIMD: rotation

- 一つの暗号文の中にパッキングされたメッセージを操作したい

1. 例え：extraction. $\mathsf{Enc}([a, b, c]) \to \mathsf{Enc}([a]), \mathsf{Enc}([b]), \mathsf{Enc}([c])$
2. rotation. $\mathsf{Enc}([a, b, c]) \ll 1 \to \mathsf{Enc}([b, c, a])$ (2. ができれば、1. はできる)

- rotation ができることは SIMD でベクトル内積を計算することができる

Deeping in SIMD: rotation

- ここは automorphism を使う: $A(X) \to A(X^j), j \in \mathbb{Z}_n^*$
- 例：$[8,5,16,9] \Leftrightarrow = 1 + X + 7X^2 + 12X^3 \in \mathbb{Z}_{17}[X]/(X^4+1)$

1. $A(X^5) = 1 + X^5 + 7(X^5)^2 + 12(X^5)^3 = 1 + 16X + 7X^2 + 5X^3$.
2. ベクトルに戻ると $1 + 16X + 7X^2 + 5X^3 \Leftrightarrow [16,9,8,5]$.
- 右方向に2 単位 rotation されたと等価 $[8,5,16,9] \ll 2$

Rotation and KeySwitch

RLWE 暗号文に rotation を適用する

- RLWE 暗号文 ctx $= (A(X), B(X))$ は多項式の tuple
- 直接多項式を操作することで rotation を行う. ctx$' = (A(X^j), B(X^j))$.
- ほどんとノイズに影響なし
- しかし. ctx$'$ を復号する時,鍵 sk(X) ではなく sk(X^j)を使うべき

$$B(X) + A(X) \cdot \mathsf{sk}(X) = B(X^j) + A(X^j) \cdot \mathsf{sk}(X^j)$$

- よって、異なる offset でrotationされた暗号文間は正しく計算できなさそう

NG: $\mathsf{Enc}(M_1) \ll 1 \oplus \mathsf{Enc}(M_2) \ll 3$

Rotation and KeySwitch

RLWE 暗号文に rotation を適用する

- RLWE 暗号文 ctx $= (A(X), B(X))$ は多項式の tuple
- 直接多項式を操作することで rotation を行う. ctx$' = (A(X^j), B(X^j))$.
- ほどんとノイズに影響なし
- しかし. ctx$'$ を復号する時,鍵 sk(X) ではなく sk(X^j)を使うべき

$$B(X) + A(X) \cdot \mathsf{sk}(X) = B(X^j) + A(X^j) \cdot \mathsf{sk}(X^j)$$

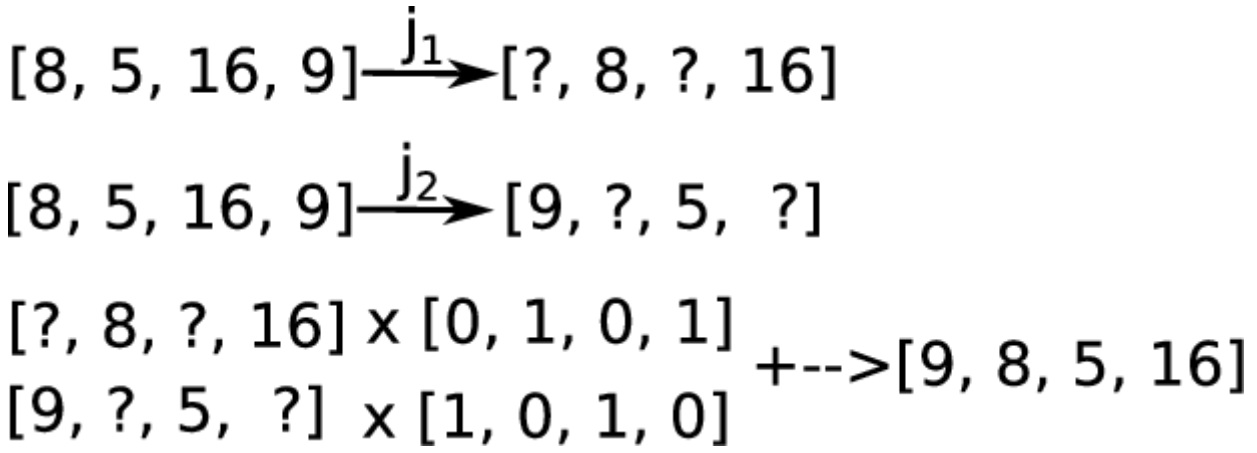
- よって、異なる offset でrotationされた暗号文間は正しく計算できなさそう

NG: $\mathsf{Enc}(M_1) \ll 1 \oplus \mathsf{Enc}(M_2) \ll 3$

- ctx$'$ を鍵 sk$'(X) :=$ sk(X^j) の暗号文と見なし、KeySwitch を使えば, 元鍵skの暗号文に戻れる.
- automorphism の後に KeySwitch すれば、 異なる offset でrotationされた暗号文間でも計算可能になる

Deeping in SIMD: rotation

- 常に一回の automorphism で欲しいな rotation ができることはない
- 一回の automorphism でどのスロットを動かせるのは n, t から分かる
- `masking` と合わせて任意の rotation はできる



viewing rotation in hybercude

- know-how: rotation しやすい n, t のコンビネーションを使う. (コード例あり)

少しコード

HElib & SEAL

HElib
- 出身 IBM
- 実装スキーム BGV's leveled homomorphic encryption

SEAL
- 出身 Microsoft
- 実装スキーム FV's somewhat homomorphic encryption

共有点

- BigNumber p を表現するため RNS を使っている
- reLinearization で暗号文のレングスを抑える

相違点

- HElib は leveled なスキーム ⇔ SEAL は invariant なスキーム
- HElib は DoubleCRT 方式で多項式を表現する ⇔ SEAL は 係数方式
- HElib は 一般な$\Phi_m(X)$をサポートする ⇔ SEAL は 2のべき乗な m のみ

KeyGen/Enc/Dec/ (HElib 篇)

```
1  FHEcontext context(m, t, /*r =*/1); // Z_t[X]/(Phi_m(X))
2  buildModChain(context, L, /*3*/); // bit-decomposition は 3 桁に分割する
3  FHESecKey sk(context);
4  sk.add
5  sk.GenSecKey(64, /*3*/); // hamming weight = 64, 3乗までのKS matrix が作られる. s^3, s^2 -> s
6  //sk.GenKeySWmatrix(1, j, 0, 0); sk(X^j) -> sk(X) の KS matrix を追加する
7  FHEPubKey pk(sk);
```

- HElib は一般の円分多項式 $\Phi_m(X)$ をサポートする. 平文の空間は *FHEcontext* で定義される $\mathbb{Z}_{t^r}[X]/(\Phi_m(X))$
- *buildModChain* は Mod-Switch で使う L 個の modulo を見つける.

```
1  ZZX plain;
2  Ctxt ctx(pk);
3  pk.Encrypt(ctx, plain); // asymmetric
4
5  Ctxt ctx2(sk);
6  sk.Encrypt(ctx2, plain); // symmetric
7  ZZX plain2;
8  sk.Decrypt(plain2, ctx);
```

KeyGen/Enc/Dec/ (SEAL 篇)

```
1  EncryptionParameters parms;
2  parms.set_poly_modulus("1x^2048 + 1"); // n = 2048
3  parms.set_coeff_modulus(coeff_modulus_128(2048)); // n に応じて kappa = 128-bit のパラメータを探してくれる
4  parms.set_plain_modulus(t); // Z_t[X]/(X^n+1)
5  SEALContext context(parms);
6
7  KeyGenerator keygen(context);
8  PublicKey public_key = keygen.public_key();
9  SecretKey secret_key = keygen.secret_key();
10  EvaluationKeys ev_keys;
11  keygen.generate_evaluation_keys(16, 1, ev_keys); // KS は 16-bit まで分割する. s^2 -> s を作る
12
13  Encryptor encryptor(context, public_key);
14  Evaluator evaluator(context);
15  Decryptor decryptor(context, secret_key);
```

- 同じで context を定義する。しかし法は $X^n + 1$ のみ (i.e., m は2乗の円分多項式)
- 操作は *Encryptor*, *Evaluator*, *Decryptor* に経由

```
1  Plaintext plain;
2  Ciphertext ctx;
3  encryptor.encrypt(plain, ctx);
4  decryptor.decrypt(plain, ctx);
```

Addition / Multiplication (HElib 篇)

```
1  void hom_add(Ctxt &c1, Ctxt const& c2) {
2      c1 += c2; // addition
3      // or c1.addCtxt(c2);
4  }
5  void hom_mult(Ctxt &c1, Ctxt const& c2) {
6      c1 *= c2; // multiplication w/o reLinearization,
7      // or c1.multipyBy(c2); // multiplication with reLinearization
8  }
```

(time costly) reLinearization は何時行うのか調整できる.

```
1  Ctxt inner_product(std::vector<Ctxt> const& ctx_a,
2                     std::vector<Ctxt> const& ctx_b,
3                     FHEPubKey const& pk) {
4      Ctxt prod(pk);
5      for (i = 0; i < ctx_a.size(); ++i) {
6          Ctxt tmp(ctx_a[i]);
7          tmp *= ctx_b[i]; // reLinearization なし
8          prod += tmp; // レングス-3の暗号文間の足し算、コストは少し増加
9      }
10
11      prod.reLinearize(); // 最後一回 reLinearize、レングス2に戻れる
12      return prod;
13  }
```

Addition / Multiplication (SEAL 篇)

```
1  // Ciphertext c1, c2, c3
2  evaluator.add(c1, c2); // c1 = c1 + c2
3  evaluator.multiply(c1, c2); // c1 = c1 * c2
4  evaluator.relinearize(c1, evaluation_key, c3); // c3 = relinearize(c1);
5  evaluator.multiply_plain(c2, plain); // c2 = c2 * plain
```

- 掛け算は reLinearization が付いていない. 明確で呼び出し

SIMD & Rotation (HElib 篇)

```
1  const EncryptedArray *ea = context.ea;
2  std::vector<long> vec_a(ea->size()); // ea->size() = \ell
3  vec_a[0] = 8; vec_a[1] = 5;
4  vec_a[2] = 16; vec_a[3] = 9; // vec_a = [8, 5, 16, 9]
5  ZZX A;
6  ea->encode(A, vec_a); // A = 1 + X + 7X^2 + 12X^3
7  Ctxt ctx(pk);
8  pk.Encrypt(ctx, A);
9
10  ctx.multiplyBy(ctx);
11  sk.Decrypt(A, ctx);
12  ea->decode(A, vec_a); // vec_a = [64, 25, 196, 81]
```

Rotation は Automorphism と KeySwitch 両方行う

```
1  Ctxt tmp1(ctx), tmp2(ctx);
2  ea->rotate(tmp1, 1); // left-rotate
3  ea->rotate(tmp2, -1); // right-rotate
4  tmp1 += tmp2; // 既に Keyswitch 済み
5
6  ctx.automorph(j); // A(X) -> A(X^j)
7  ctx.reLinearize(j); // Keyswitch
8  // tmp1.automorph(j1) += tmp2.automorph(j2) <- NG
```

Good & Bad Parameters for Rotation

- m, t のコンビネーションによって rotation の効率が違う

```
1  // m = 16384, t = 8191
2  FHEcontext context(1683, 8191, 1);
3  ea->size(); // 4096
4  ea->dimension(); // 1
5  ea->rotate(ctx, ?); // 100 rotations took 0.98 sec
6  /*-----------------------------------------*/
7  // m = 16384, t = 40961
8  FHEcontext context(1683, 40961, 1);
9  ea->size(); // 4096
10  ea->numOfGens(); // 2 <-- needs more key-switch and masking
11  ea->rotate(ctx, ?); // 100 rotations took 4.21 sec
```

- 試行錯誤で dimension が少ない t を探す

IO

- HElib の暗号文は DoubleCRT フォーマットで $O(Ln)$ のスペースを使う
- ネット通信する時、まず係数フォーマットして通信量を削減する.
- しかし HElib の IO ルーチンにはこの機能がない。じまいした

```
1  @https://github.com/fionser/HElib/blob/master/src/DoubleCRT.cpp#L918
2  std::ostream& operator<< (std::ostream &str, const DoubleCRT &d)
3  { ...
4    NTL::ZZX poly;
5    d.toPoly(poly, true); // use iFFT
6    const FHEcontext &context = d.context;
7    double bits = context.logOfProduct(set);
8    bits /= std::log(2.);
9    long bytes = long(std::ceil(bits) + 7) >> 3;
10    long phim = context.zMStar.getPhiM();
11    std::vector<uint8_t> buff(bytes);
12    for (long i = 0; i < phim; i++) {
13        const NTL::ZZ &e = NTL::coeff(poly, i);
14        BytesFromZZ(buff.data(), e, bytes);
15        str.write(reinterpret_cast<char *>(buff.data()), bytes);
16    }
17    ...
18 }
```

Hack into SIMD (CSS'17 の内容)

目的：暗号化されたベクトルの内積をバッチ的に計算したい. $\rightarrow$ 行列積を計算するため

- $X^n + 1 \mod t$ は他のいい感じの分解ができる

n, t の combination によって $X^n + 1 = \prod_j X^d + \beta_j \mod t$, e.g.,
$X^{1024} + 1 = (X^{256} + 10)(X^{256} + 41)(X^{256} + 96)(X^{256} + 127) \mod 137$

- すなわち、サイズ 256 のベクトル 4 個を一つの多項式にencodeすることができる

```
1  FHEcontext context(1024 * 2, 137, 1); // Phi_{2048}(X) = X^{1024} + 1
2  const EncryptedArray *ea = context.ea;
3  long ell = ea->size(); // 4
4  long d = ea->getDegree(); // 256
5  std::vector<NTL::ZZX> vectors(ell);
6  // ... fill in vectors
7  // e.g. vectors[0] = 1 + 2X + 3X^2 ... + 256X^{255}
8  NTL::ZZX poly;
9  ea->encode(poly, vectors);
```

Hack into SIMD (CSS'17 の内容)

目的：暗号化されたベクトルの内積をバッチ的に計算したい. $\longrightarrow$ 行列積を計算するため

- $X^d + \beta$ の形にする理由: サイズ d のベクトルの内積、準同型掛け算一回で済む

$X^3 + 4$を例としましょう.
1. $(1 + 2X + 3X^2) \times (5 + 6X + 7X^2) = \ldots + 34X^2 \mod X^3 + 4$
2. X^{d-1} の係数 34 はベクトル内積 $[1, 2, 3]^\top [7, 6, 5]$ になる
暗号化する前にベクトルを二通りで encoding すると内積の計算は早くできる

Hack into SIMD (CSS'17 の内容)

目的：暗号化されたベクトルの内積をバッチ的に計算したい. $\rightarrow$ 行列積を計算するため

- ベクトル $\{\vec{a}_i, \vec{b}_i\}, 1 \leq i \leq 4, |\vec{a}_i| = |\vec{b}_i| = 256$
- 4 個の内積 $\vec{a}_i^\top \vec{b}_i$, 準同型掛算一回で計算する (結果の暗号文も一つしかない)

1. encoding I: $[\vec{a}_1, \ldots \vec{a}_4] \rightarrow A(X)$
2. encoding II: $[\vec{b}_1, \ldots \vec{b}_4] \rightarrow B(X)$
3. $\mathsf{Enc}(A(X)) \otimes \mathsf{Enc}(B(X)) \rightarrow \mathsf{Dec} \rightarrow \mathsf{decode} \rightarrow \{\vec{a}_i^\top \vec{b}_i\}$

- 実際 $n \geq 4096, t$ によって d を調整することができる

1. e.g., $n = 4096, t = 82561$ なら サイズ 64 のベクトルを 64 個分畳み込める

n と t 組み合わせはどう見つけるのか?

- n は security level に関連なので given とする。 t は平文のサイズ. e.g, 16-bit 以上であれば ok. (https://github.com/OpenSMP/SMP)

```
1  void DoublePacking(long m, long slots) { // slots 数を指定
2      long t = NTL::NextPrime(1 << 16); // 16-bit 以上
3      long n = phi_N(m); // assert(n * 2 == m)
4      while (true) {
5          long d = multOrd(t, m);
6          long s = n / d;
7          if (s == slots) {
8              FHEcontext context(m, t, 1);
9              const std::vector<NTL::ZZX> &ftrs = context.alMod.getFactorsOverZZ();
10              bool ok = true;
11              for (const auto& f : ftrs)
12                  ok &= check(f); // check whether X^d + beta
13              if (ok) // all factors follow the form X^d + beta
14                  break;
15          }
16          do { // try next prime
17              t = NTL::NextPrime(t + 2);
18          } while (m % t == 0);
19      }
20  }
```

SCIS, AsiaCCS'18 の内容

- 大小比較
- $\mathsf{Enc}(X^a), \mathsf{Enc}(X^b) \to \mathsf{Enc}(a >^? b)$
- TFHE のトリックと類似

$$\mathsf{Enc}(X^a) \otimes \mathsf{Enc}(X^{-b}) \times (1 + X + X^2 + \cdots + X^{n-1}) \times 2^{-1} + 2^{-1}$$

- $\mathsf{Enc}(X^b)$ から $\mathsf{Enc}(X^{-b})$ を作るのに $j = 2n - 1$ のautomorphism を使う
- $(X^b)^{2n-1} = X^{2nb-b} = (-1)^{2b} X^{-b} \mod X^n + 1$

```
1  sk.GenKeySWmatrix(1, 2n - 1, 0, 0);
2  ctx.smartAutomorph(2n - 1);// X^b --> X^{-b}
```

- よって one-round な 決定木とU 検定の秘密計算プロトコルを作った

少し展望

- TFHE の利点は：パラメータ m, q はかなり固定。GPU などの最適化に便利
- GSW 方式 (so call 3rd gen FHE) もいい選択肢でしょう。

1. reLinearization が要らない. (so called evaluation key free)
2. mod-switch もなし (flatten, unflattenでノイズをコントロール)
3. 暗号文は行列だが、0-1の行列なので、最適化の余地はまだあるでしょう.

References

- [BV11a] Brakerski et al. Fully Homomorphic Encryption from ring-LWE and security for key dependent messages.
- [BV11b] Brakerski et al. Efficient Fully Homomorphic Encryption from (Standard) LWE.
- [BGV11] Brakerski et al. Fully Homomorphic Encryption without Bootstrapping.
- [GHS12] Gentry et al. Homomorphic Evaluation of the AES Circuit.
- [FV12] Fan et al. Somewhat Practical Fully Homomorphic Encryption
- [SV11] Smart et al. Fully Homomorphic SIMD Operations
- [BEHZ16] Bajard et al. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes
- [CBHHJLL18] Chen et al. Logistic regression over encrypted data from fully homomorphic encryption

Thank you

@rbklwj (http://twitter.com/rbklwj)
riku@mdl.cs.tsukuba.ac.jp (mailto:riku@mdl.cs.tsukuba.ac.jp)