L-III
準同型暗号アプリケーション開発ゼミ

2021/08/29

松本 直樹

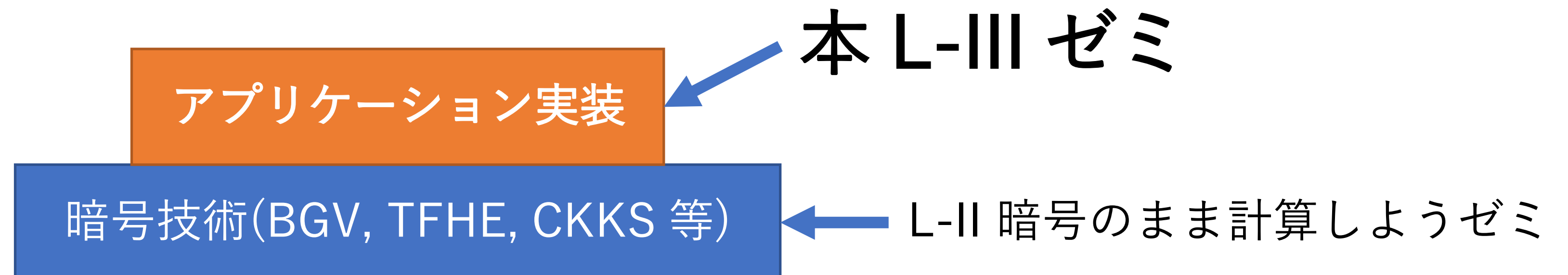
KYOTO UNIVERSITY

今日の予定

1. L-III ゼミについて
2. ここまでの振り返り
3. 実装上のアドバイス

L-III ゼミについて

- **FHE(完全準同型暗号) を用いたアプリケーション開発**
  - **センシティブなデータ**(DNA, 個人情報等)を暗号化した状態で処理
  - いわゆる秘匿計算(ゼミでは触れないが他には 秘密分散, Garbled Circuit 等もある)
  - 速度的な問題等、発展途上の分野(LHE の方が活発に研究されている)
- 本ゼミでは暗号部分には深く触れず実装メインで取り扱う
  - **とりあえず作ってみよう！**の精神
  - 暗号部分が気になる場合は L-IIの資料を参考に(https://nindanaoto.github.io/)

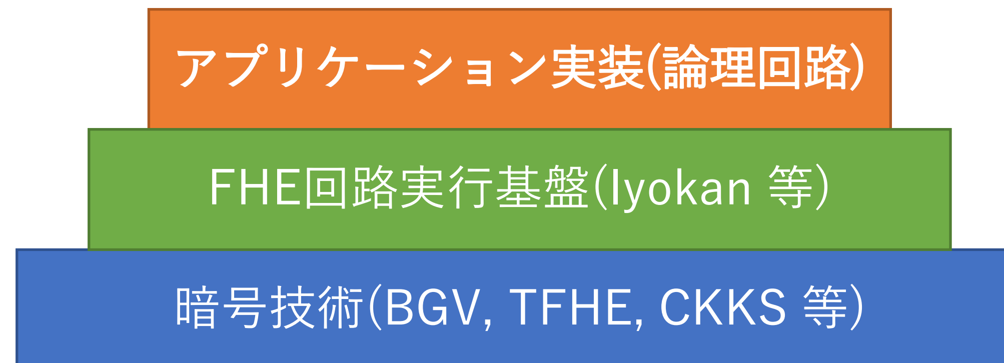


FHEアプリケーション

- FHEを利用した秘匿計算アプリケーションの開発
  - BGV, TFHE, CKKS 等の FHE 手法を利用
- <u>**非常に高度な暗号学的な知見**</u>が必要
  - 暗号学的手法を用いた処理アルゴリズムの実装
  - パラメータの選定,安全性の検証 等

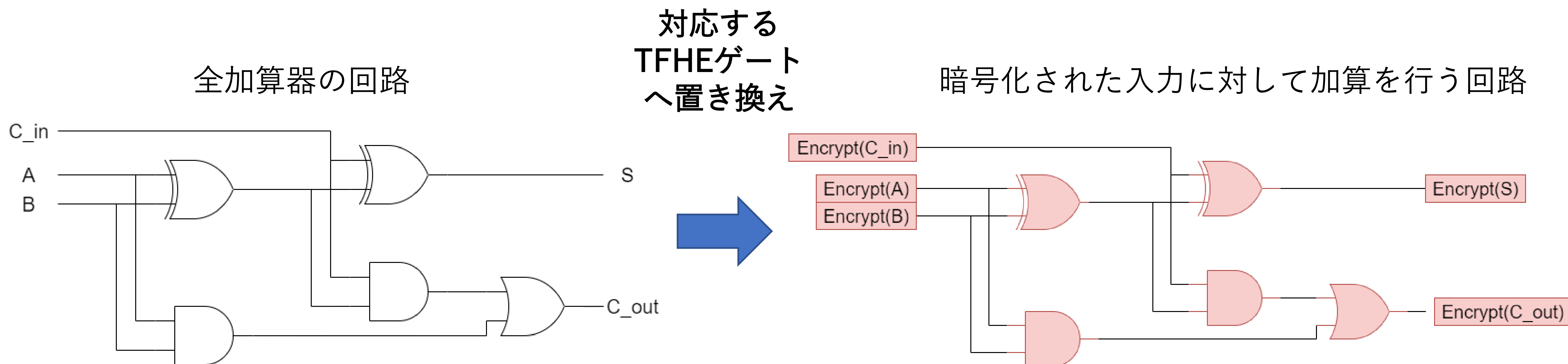
FHE アプリケーション

- TFHE : 高速にFHE上の**論理演算**が可能
- 論理回路のゲートをTFHEのゲートで置き換える
  - 論理回路の設計手法には既存のHW実装手法が利用可能
  - 従来の FHE では不可能だった処理も可能(ReLU 関数などの if 分岐)
- **下のレイヤー(暗号技術等)を意識することなく実装可能**



論理回路とTFHE

- 論理ゲートをTFHEゲートに置き換えることでFHEによる処理に対応できる



論理回路とTFHE

- 利点
  - アプリケーションの開発に**暗号学的な知識を必要としない**
  - **任意の演算**が実現可能
- 欠点
  - 特定の処理に最適化された手法には**速度の面で劣る**
  - **論理回路に関する理解**を要する

ここまでの振り返り

- FHE アプリケーションの開発手法について簡単に学んだ
  - 演習 1 (論理回路の開発環境の準備)
  - 演習 2 (簡単な処理を論理回路で記述し FHE 上で処理)
  - 演習 3 (フロントエンド実装, XLS を用いた処理の記述)

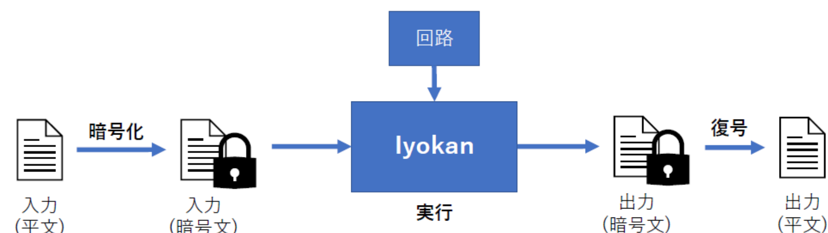
総和計算回路(Sum8bit.scala)

- **演習内容**
  - カウンター回路を参考に、前述の要件を満たす回路を Sum8bit.scala に追記してください
  - 実装後、回路の生成とテストをパスすることを確認してください
- 回路について
  - sum が総和を保存するレジスタ
  - cnt が ROM のアドレス (= 配列の添字) を保存するレジスタ
  - ROM から読みだした値は io.romData から入力される

```
import chisel3._

class Sum8bit(val num:Int, val romWidth:Int) extends Module {
    val io = IO(new Bundle{
        val romAddr = Output(UInt(romWidth.W))
        val romData = Input(UInt(8.W))

        val out    = Output(UInt(8.W))
    })

    val sum = RegInit(0.U(8.W))
    val cnt = RegInit(0.U(romWidth.W))

    io.romAddr := cnt
    io.out := sum

    when(cnt === num.U){
      // ここに処理を記述
    }.otherwise{
      // ここに処理を記述
    }
}
```

Iyokan 入門

- 以下のような流れで利用する



実装上のアドバイス(一般)

- ユニット単位でのテスト
  - Chisel の機能(PeekPokeTester 等) を利用した論理回路自体のテスト
    演習 2 のテストコードが参考になる(↓こんなやつ)

```
class Sum8bitSpec extends AnyFreeSpec with Matchers {
  "tester should returned values with interpreter" in {
    val data = Seq(30, 75, 89, 42, 65, 16, 85, 43, 60)
    Driver.execute(Array("--backend-name", "firrtl"), () => new TopSum8bit(data)) { c =>
      new Sum8bitTester(c, data)
    } should be (true)
  }
}
```

  - Iyokan の平文モードを利用したアプリケーション全体のテスト

Okay. Run Iyokan *in plaintext mode* with this packet as input to check it's correct:

```
$ iyokan plain --blueprint test/test-div-8bit.toml \
               -i request.plain.packet -o result.plain.packet -c 1
```

実装上のアドバイス(一般)

- 組み合わせ回路と順序回路
  - 組み合わせ回路のみで計算可能な場合がある
  - ベクトルの要素ごとの積

$$z_i = x_i y_i$$

  - Chisel で書くとこんな感じ

```
val x := Vec(100, UInt(16.W))
val y := Vec(100, UInt(16.W))
val z := Vec(100, UInt(16.W))

for (i <- 0 until 100) {
    z(i) := y(i) * x(i)
}
```

実装上のアドバイス(一般)

- 組み合わせ回路と順序回路
  - 要素数が大きくなる、複数の組み合わせ回路を組み合わせると、
    回路の合成(Chisel, yosys) に時間がかかる、または、合成が出来なくなることがある
  - **組み合わせ回路の最適化問題は(多分)NP-hard**
    - NP-Hardness of Circuit Minimization for Multi-Output Functions(Ilango et al., 2020)
  - **適当にレジスタ(D-FF)を挟んで分割する必要がある**
    → ある程度のかたまりで分割してテスト

実装上のアドバイス(Chisel)

- printf デバッグが出来る

```
when(write) {
    printf("[WB] MainReg(%d) <= 0x%x\n", io.rd.sel, io.rd.data)
}
```

```
[WB] MainReg(1) <= 0x2
```

- 信号線をまとめて扱う(まとめて接続,レジスタの宣言も可)

```
class MainRegPort(implicit val conf:CAHPConfig) extends Bundle {
  val inRead = Input(new MainRegInRead)
  val inWrite = Input(new MainRegInWrite)

  val out = Output(new MainRegOut)
}
```

```
class MainRegInRead(implicit val conf:CAHPConfig) extends Bundle {
  val rs1 = UInt(4.W)
  val rs2 = UInt(4.W)
}
```

```
mainReg.io.port.inRead := decoder.io.regRead
mainReg.io.port.inWrite := io.wbIn.regWrite
```

```
val portReg = RegInit(0.U.asTypeOf(new MainRegInRead(conf)))
portReg := io.inRead
```

実装上のアドバイス(BNN)

- 回路の構成
  - レイヤーごとに実装 & テスト
- パラメータの供給
  - 学習したモデルからパラメータを抽出し回路へ反映する必要がある
  - パラメータ自体を回路に埋め込む(ハードコーディング)
    → 前述の組み合わせ回路の規模問題が絡む
  - ROM からパラメータを供給する
    → 複数サイクルかけて計算する = 順序回路