# ネイティブアプリ基本設計書

目次

[ネイティブアプリ基本設計書 1](#_heading=h.z337ya)

[1. 全体基本設計 3](#_heading=h.2u6wntf)

[1-1. ネイティブアプリのシステム全体における位置付け 3](#_heading=h.19c6y18)

[1-2. ネイティブアプリの機能概要 4](#_heading=h.3tbugp1)

[1-3. ネットワーク構成 7](#_heading=h.28h4qwu)

[2. ユーザーデータの安全な保存を実現するためのシステム詳細 8](#_heading=h.nmf14n)

[2-1. 全体概要 8](#_heading=h.37m2jsg)

[2-2. 実現したい安全性の定義 9](#_heading=h.1mrcu09)

[2-3. 技術選定の背景 10](#_heading=h.46r0co2)

[2-4. 設計プロトコルの概要 12](#_heading=h.2lwamvv)

[概要 12](#_heading=h.2s8eyo1)

[主要機能 12](#_heading=h.17dp8vu)

[詳細要件 12](#_heading=h.3rdcrjn)

[安全性 13](#_heading=h.26in1rg)

[ブロックチェーンの容量制限 13](#_heading=h.lnxbz9)

[2-5. 設計プロトコルの詳細 14](#_heading=h.111kx3o)

[概要 14](#_heading=h.1ksv4uv)

[機能 14](#_heading=h.44sinio)

[シーケンス図 15](#_heading=h.2jxsxqh)

[サンプルコード 15](#_heading=h.3j2qqm3)

[3. ネイティブアプリを用いたベクトルデータ暗号化の実行検証 19](#_heading=h.3l18frh)

[3-1. 全体概要 19](#_heading=h.206ipza)

[3-2. 使用技術スタック 21](#_heading=h.4k668n3)

[3-3. ネイティブアプリ上での実行の注意点（リソース観点） 22](#_heading=h.2zbgiuw)

[注意点の概要 22](#_heading=h.3whwml4)

[具体的な対策 22](#_heading=h.2bn6wsx)

[実際の実行時間への影響 22](#_heading=h.qsh70q)

[備考 23](#_heading=h.3as4poj)

[3-4. サンプルコード 24](#_heading=h.1egqt2p)

[サンプルコード① 24](#_heading=h.49x2ik5)

[サンプルレポジトリ② 24](#_heading=h.2p2csry)

[サンプルレポジトリ③ 24](#_heading=h.147n2zr)

[4.ネイティブアプリのUI/UX 26](#_heading=h.3ygebqi)

[4-1. エントリー機能 26](#_heading=h.2dlolyb)

[4-2.   ステータス画面 & マッチング機能 28](#_heading=h.sqyw64)

[4-3.  メッセージ機能 29](#_heading=h.3cqmetx)

[4-4. アカウント/ データ設定機能 30](#_heading=h.1rvwp1q)

[5. その他ネイティブアプリ構築時の注意点 31](#_heading=h.4bvk7pj)

[5-1. ネイティブアプリに対する一般的なセキュリティ攻撃と対策 31](#_heading=h.2r0uhxc)

[5-2. ネイティブアプリの実装優先度に関する考察 32](#_heading=h.1664s55)

[普及率と対象ユーザー層 32](#_heading=h.3fwokq0)

[開発エンジニアリング観点からの差異 32](#_heading=h.1v1yuxt)

[結論 32](#_heading=h.4f1mdlm)

# 1. 全体基本設計

この章では、本ネイティブアプリの、全体システムにおける位置付けや連携関係を示し、ネイティブアプリの目的と機能概要について記載することを目的とする。

* ネイティブアプリは「実証フェーズ」のシステム設計で記載された「エントリーシステム」を、ネイティブアプリとして「本番フェーズ」用に実装するものである。
* 本基本設計では、実証フェーズでエントリーシステムが持っていた機能がDockerコンテナ内においてLinuxで動作していた一方で、これらの機能が実際にネイティブアプリを開発する環境（Swift, Kotlin)で動作するかどうかの検討、検証についても記載する
* 本基本設計では、ネイティブアプリで処理されるデータの種類、データを安全に格納し、処理するための安全上のプロトコルの詳細を技術選定の背景とともに記載する
* 本基本設計では、上記プロトコルを実現するためのサンプルコードの記載、実装の時に参照できるシーケンス図の記載を実施する
* 本基本設計では、一般的なネイティブアプリの開発で考慮すべきセキュリティ要件と対策の優先度、対策方法についても言及する

## 1-1. ネイティブアプリのシステム全体における位置付け

この項では、ネイティブアプリのシステム全体における位置付けについて記載し、全体像を把握することを目的とする。

* ネイティブアプリは候補者向けのUI/UXを提供する
* ネイティブアプリは、「実証フェーズ」のシステム設計で記載された「エントリーシステム」で実装された機能をネイティブアプリとして、候補者に提供するものである
* ネイティブアプリは個人の情報を守るためのセキュリティを考慮したプロトコルが実装される
* ネイティブアプリは後述の個人データに後述のプロトコルに則した処理を実行し、出力されたデータをプロセッシングシステム、及びサービスのバックエンドに送信する

## 1-2. ネイティブアプリの機能概要

この項では、ネイティブアプリが担う機能の概要について記載する。各機能について、データの入出力の有無（加えて入出力先）や、機能の説明については以下の通りである。

|  |  |  |
| --- | --- | --- |
| **機能概要** | **入力データ/入力元** | **出力データ/出力先** |
| 成績データの取得 | LMS（学習システム）から成績データを取得する | 「成績データ、個人情報の暗号化及び分割化」及び「成績データの準同型暗号による暗号化」機能に出力 |
| 個人情報の取得 | ユーザーの入力から、ユーザーの個人情報等を取得する | 「成績データ、個人情報の暗号化及び分割化」及び「成績データの準同型暗号による暗号化」機能に出力 |
| 暗号化のための鍵生成（※） | 暗号化に必要な設定パラメータ | * 後述の暗号化、分割化プロトコルに必要なAES暗号の鍵 * 後述の準同型暗号に必要な公開鍵ペア |
| 成績データ、個人情報の暗号化及び分割化 | 「成績データの取得  」及び「個人情報の取得」機能から取得 | 後述の暗号化、分割化プロトコルに基づき処理されたデータが出力され、バックエンドのDB及びブロックチェーン上に送信される  （暗号データの分割化、ブロックチェーン上への保存はオプションの機能） |
| 成績データの準同型暗号による暗号化 | 「成績データの取得  」及び「個人情報の取得」機能から取得- | 後述の準同型暗号プロトコルに基づき暗号化されたデータが、プロセッシングシステムに送信され、スコア計算に用いられる |

また、上記説明を補助するシーケンス図を２つ提示する。それぞれ、

* 成績データ、個人情報の暗号化及び分割化を行うパターン
  + ユーザー（学習者）自身が自身の成績データをプラットフォームではなく、個人で管理するために、ブロックチェーン及びデータベースに分割して格納することを希望する場合
* 成績データ、個人情報の暗号化は行うが、分割化は行わないパターン
  + ユーザー（学習者）自身が自身の成績データを個人で管理することを希望せず、ブロックチェーン及びデータベースに分割して格納しないが、採用エントリーのための秘密計算処理のために暗号化する場合

のシーケンス図である。

（※）暗号化のために必要な鍵（秘密鍵・公開鍵）は、モバイルのデバイス上に保存される。

**【成績データ、個人情報の暗号化及び分割化を行うパターン】**

タイムライン

自動的に生成された説明

**【成績データ、個人情報の暗号化は行うが、分割化は行わないパターン】**

タイムライン

自動的に生成された説明

## 1-3. ネットワーク構成

以下に本実証で利用したネットワーク構成を示す。

ダイアグラム

自動的に生成された説明

# 2. ユーザーデータの安全な保存を実現するためのシステム詳細

## 2-1. 全体概要

この章では、ユーザーに関わるデータの安全な保存を実現するためのシステムの詳細について記載する。具体的には、

* 実現したい安全性
* プロトコルの概要
* プロトコルの詳細
* 技術選定の背景

について言及する。

まず初めに、実現したい安全性について定義する。さらに、実現したい安全性を達成するために利用する暗号化、分割化、保存方法について概要を説明し、実装のベースとなるコードスニペットなどについても解説する。最後に、総括としてこのデータの安全な保存を実現するためのシステムアーキテクチャーと、実アプリケーションに必要な各処理フローについて図解などを加えて記載する。

加えて、暗号化、分割化、保存方法に使用する技術を選定するに至った背景について言及する。

※ 取り扱いデータに関しては、Webアプリ基本設計書と同様となるため、データ詳細に関してはWebアプリ版設計書の「1-3. 取り扱う具体的なデータと各システムの入出力の詳細」を参照。

## 2-2. 実現したい安全性の定義

この項では、前項で定義したデータに対して実現したい安全性の定義について言及する。安全性の定義についてまとめたものは以下の通りである。

|  |  |  |
| --- | --- | --- |
| **安全性のカテゴリ** | **目的** | **達成するための手段** |
| 秘匿性 | 外部ハッカーなどの攻撃に対して、万が一システムに侵入されたとしてもデータの流出を防ぐ | 暗号化 |
| データアクセスの困難性 | 外部ハッカーなどの攻撃に対して、万が一システムの一部に侵入されたとしても、データの一部にしかアクセスできず、流出を限定的にする | 分割化 |
| 乱数注入によるデータの秘匿 | 外部ハッカーなどの攻撃に対して、万が一システムに侵入されたとしても、閲覧、取得できるデータの内容を理解できないようにする | 無意味化、ランダム化 |
| 保存データの改竄防止 | 保存されているデータが第三者や攻撃者によって故意に変更されたり、改竄されることを防ぐ | ブロックチェーン上へのデータ保存 |

## 2-3. 技術選定の背景

この項では、前述のデータに対して所望の安全性を達成するために、前述の技術を選定するに至った背景について記載することを目的とする。技術の選定については、大きく３つのカテゴリに絞り、それぞれ選定に至った経緯を表でまとめるものとする。

この項の結論として、

* データの暗号化
  + AES256
* データの分割化
  + シード値を用いたランダムシャッフリングによる分割
* データの保存方法
  + データベースとブロックチェーンのハイブリッド型

を採用することと決定した。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **カテゴリ** | **調査対象** | **利点** | **欠点** | **今回の採用可否** |
| 暗号化 | AES | 一般的に使われるアルゴリズムであり、ウェブ、ネイティブ開発言語問わず標準となるライブラリが存在する。 | 特になし | 採用 |
| 分割化 | シャミアの秘密分散法 | 秘密分散法の中ではメジャーな手法であり、数学的な検証も十分に行われている。 | ネイティブアプリを考慮した時にSwiftやKotlinで標準となるライブラリが無く、自作する必要がある可能性が高い。  分割されるデータのサイズは、一般的にそれぞれのパーツが同様のデータサイズとなる必要があり、今回パーツの片方がブロックチェーン上に保存され、サイズの制限がある場合にはあまり向かない。 | 不採用 |
| 分割化 | シード値を用いたランダムシャッフリングによる分割 | シンプルで単純なアルゴリズムであるが、ランダム性に基づいた独自のアルゴリズムであるため、パーツのサイズを設定できたりカスタマイズ性が高い。 | 比較的単純なアルゴリズムであるが、ライブラリなどは存在しないため、Swift,Kotlin両方で関数を自作する必要がある。 | 採用 |
| データの保存方法 | IPFS(web3) | 分散型のストレージであり、今回のアプリケーションと親和性が高い。  IPFSに保存した際のアドレスをブロックチェーンに保存したり、ウェブ３技術を組み合わせて新しいデータの保存方法を実装できる。 | 技術的にあまりメジャーでは無く、ドキュメントがあまり豊富でなかったり、情報が少ない。  基本的にデータの削除を考慮した設計にはなっておらず、データの削除を行う際は全てのノードで対象データのホワイトリスト化、全ノードの定期アップデートなどの操作が必要であり、非常に煩雑。  また、データが確実に削除された保証をすることが難しく、個人情報の削除請求などの要件に適さない。 | 不採用 |
| データの保存方法 | DB+ブロックチェーン(web2+web3) | ウェブ３ソリューションであるブロックチェーン上にデータを保存しつつ、残りの断片はデータベースに保存することで、改竄不可能性などを担保したまま機能面でも実装に負担が比較的かかりにくい。 | ブロックチェーン上に保存できるデータサイズの上限があまり大きくない（チェーン依存）ため、それを考慮したデータの分割化が必要 | 採用 |

## 2-4. 設計プロトコルの概要

### 概要

本機能は、指定されたファイルを安全に暗号化し、ブロックチェーンとストレージに格納するために2つの部分に分割することとを目的としています。このシステムは、秘密鍵に基づくAES暗号化方式を使用し、ファイルのセキュリティとブロックチェーンの容量制限を考慮します。

### 主要機能

1. **ファイル暗号化**: ユーザーが指定したファイルをAES暗号化アルゴリズムを使用して暗号化します。
2. **ファイル分割**: 暗号化されたファイルをシード値に基づいてシャッフルし、2つの部分に分割します。片方はブロックチェーンに格納するため、特定の容量制限を満たす必要があります。
3. **ファイル結合**: 分割されたファイルを再結合します。
4. **ファイル復号化**: 結合されたファイルを復号化します。

### 詳細要件

1. **秘密鍵に基づくAESキー生成**:
   * システムは、ユーザーによって提供される秘密鍵を使用してAESキーを生成する必要があります。
   * AESキーは、256ビットの安全なキーである必要があります。
2. **シード値の生成**:
   * 秘密鍵からシード値を生成する機能が必要です。
   * このシード値は、ファイルの分割および結合時のシャッフルに使用されます。
3. **暗号化プロセス**:
   * 指定されたファイルをAESを使用して暗号化する機能が必要です。
   * 暗号化されたファイルは、後の処理で使用するために一時的に保存される必要があります。
4. **分割プロセス**:
   * 暗号化されたファイルをシード値に基づいてシャッフルし、2つの部分に分割する機能が必要です。
   * 分割された片方のファイルはブロックチェーンの容量制限内に収まるようにサイズを調整する必要があります。
   * 分割されたファイルは、後の結合プロセスで使用するために保存される必要があります。
5. **結合プロセス**:
   * 分割されたファイルをシード値に基づいて再シャッフルし、一つのファイルに結合する機能が必要です。
6. **復号化プロセス**:
   * 結合されたファイルを元のファイル形式に復号化する機能が必要です。
   * 復号化されたファイルは、最終的な出力としてユーザーに提供されます。

### 安全性

* システムはAES-256暗号化アルゴリズムを使用し、ファイルの安全性を保証する必要があります。
* シード値に基づくシャッフルは、分割プロセスの安全性を高めるために使用されます。

### ブロックチェーンの容量制限

* 分割されたファイルの一部はブロックチェーン上に格納されるため、ブロックチェーンの容量制限に適合するサイズである必要があります。

## 2-5. 設計プロトコルの詳細

この項では、設計プロトコルの詳細について、プロトコルで使用される関数レベルでの機能の詳細、およびそれらの呼び出し手順などを記したシーケンス図を記載する。さらに、これらを実現するためのベースとなるサンプルコード(Python)を記載する。

サンプルコードで使用される関数とアルゴリズム（後述）は一般的なものであり、ネイティブアプリ上でも同様のプロトコルに従うことを前提としている。

また、分割されたファイルはサンプルコード内ではメモリ上に保存されているが、システムでは前述の通り、データベースとブロックチェーン上にそれぞれ保存される。

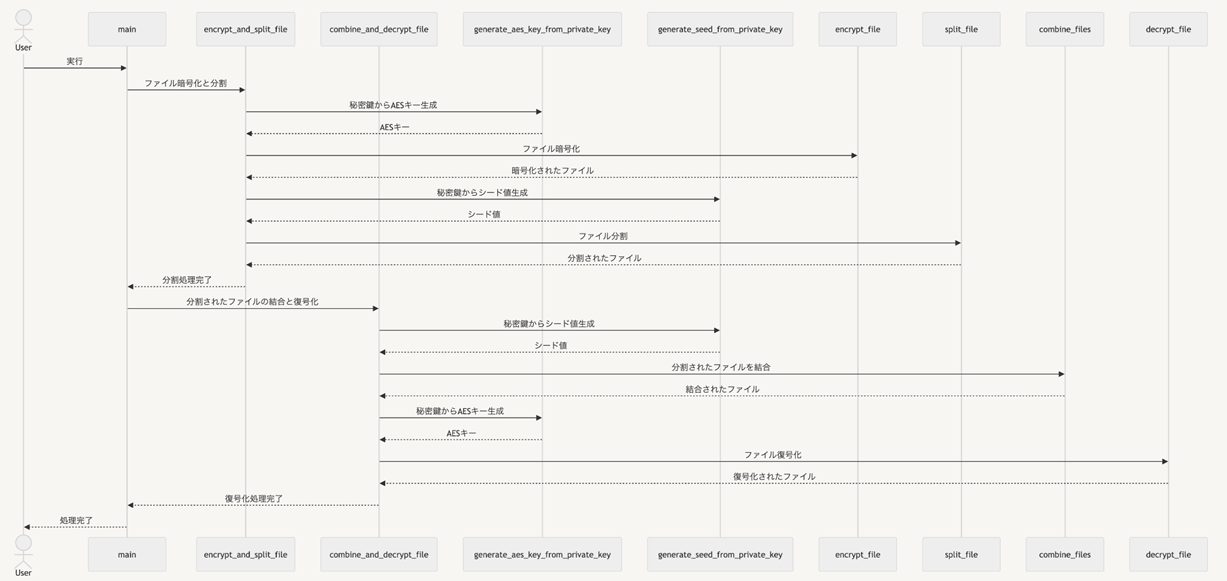
### 概要

この設計書は、ファイルの暗号化、分割、結合、および復号化のプロセスを説明します。このプロセスは、AES暗号化アルゴリズムを使用し、特定の秘密鍵に基づいてAESキーとシード値を生成します。

### 機能

1. **AESキー生成**: 秘密鍵から安全なAESキーを生成します。
2. **シード値生成**: 同じく秘密鍵からシード値を生成します。
3. **ファイル暗号化**: 指定されたファイルをAESを使用して暗号化します。
4. **ファイル分割**: 暗号化されたファイルをシード値に基づいてシャッフルし、2つの部分に分割します。
5. **ファイル結合**: 分割されたファイルを再結合します。
6. **ファイル復号化**: 結合されたファイルを復号化します。
7. **ファイル暗号化分割**：指定されたファイルを暗号化し、それを2つの部分に分割します。
8. **ファイル複合化結合**：分割されたファイルを結合し、復号化します。

### シーケンス図



サンプルコード

以下はサンプルコードとして、１つのファイルを２つの断片に分割し断片化するPythonのサンプルコード。

一般的なAESアルゴリズムやハッシュ関数をライブラリ pycryptodome を用いて実装しており、

同様のコードをSwift,Kotlinで実装することは可能であり、ここではPythonでのサンプルコードのみの記載としている。

#### Pythonでの実行（パッケージのインストール）

|  |
| --- |
| pip install pycryptodome |

#### ソースコード

|  |
| --- |
| import os  import random  import binascii  from Crypto.Cipher import AES  from Crypto.Hash import SHA256  from Crypto.Random import get\_random\_bytes  # 秘密鍵からAESキーを生成する関数  def generate\_aes\_key\_from\_private\_key(private\_key):  hash\_obj = SHA256.new(private\_key.to\_bytes(32, 'big')) # SHA256ハッシュオブジェクトを生成  aes\_key = hash\_obj.digest() # AES-256キーとして使用  return aes\_key  # 秘密鍵からシード値を生成する関数  def generate\_seed\_from\_private\_key(private\_key):  hash\_obj = SHA256.new(private\_key.to\_bytes(32, 'big')) # SHA256ハッシュオブジェクトを生成  # ハッシュの最初の数バイトを使用してシードを作成  return int.from\_bytes(hash\_obj.digest()[:4], 'big')  # ファイルを暗号化する関数  def encrypt\_file(input\_file, encrypted\_file, key):  cipher = AES.new(key, AES.MODE\_EAX) # AES暗号オブジェクトを生成  with open(input\_file, 'rb') as f\_in:  data = f\_in.read()  ciphertext, tag = cipher.encrypt\_and\_digest(data) # データを暗号化  with open(encrypted\_file, 'wb') as f\_out:  f\_out.write(cipher.nonce) # ノンスを書き込み  f\_out.write(tag) # タグを書き込み  f\_out.write(ciphertext) # 暗号文を書き込み  # 暗号化されたファイルを分割する関数  def split\_file(encrypted\_file, part1\_file, part2\_file, seed, max\_size=262144):  random.seed(seed) # 再現性のためのシード設定  with open(encrypted\_file, 'rb') as f:  data = list(bytearray(f.read()))  shuffled\_indices = list(range(len(data)))  random.shuffle(shuffled\_indices) # データのインデックスをシャッフル  shuffled\_data = [data[i] for i in shuffled\_indices]  shuffled\_data = bytearray(shuffled\_data)  # データサイズがmax\_size以下の場合、二等分に分割  if len(shuffled\_data) <= max\_size:  part1\_size = len(shuffled\_data) // 2 # データサイズの半分  else:  part1\_size = max\_size # 最初の部分をmax\_sizeに制限  with open(part1\_file, 'wb') as f1:  f1.write(shuffled\_data[:part1\_size])  with open(part2\_file, 'wb') as f2:  f2.write(shuffled\_data[part1\_size:])  # 分割されたファイルを結合する関数  def combine\_files(part1\_file, part2\_file, seed, combined\_file):  random.seed(seed) # シャッフルされたインデックスを再生成するためにシードをリセット  with open(part1\_file, 'rb') as f1, open(part2\_file, 'rb') as f2:  shuffled\_data = list(bytearray(f1.read() + f2.read()))  shuffled\_indices = list(range(len(shuffled\_data)))  random.shuffle(shuffled\_indices) # インデックスをシャッフル  combined\_data = [None] \* len(shuffled\_data)  for original\_index, shuffled\_index in enumerate(shuffled\_indices):  combined\_data[shuffled\_index] = shuffled\_data[original\_index]  combined\_data = bytearray(combined\_data)  with open(combined\_file, 'wb') as f:  f.write(combined\_data)  # ファイルを復号化する関数  def decrypt\_file(combined\_file, decrypted\_file, key):  with open(combined\_file, 'rb') as f\_in:  nonce, tag, ciphertext = f\_in.read(16), f\_in.read(16), f\_in.read()  cipher = AES.new(key, AES.MODE\_EAX, nonce) # AES暗号オブジェクトを生成  data = cipher.decrypt\_and\_verify(ciphertext, tag) # データを復号化  with open(decrypted\_file, 'wb') as f\_out:  f\_out.write(data)  # ファイルを暗号化して分割する関数  def encrypt\_and\_split\_file(filename, private\_key):  aes\_key = generate\_aes\_key\_from\_private\_key(private\_key)  print("AES Key:", binascii.hexlify(aes\_key).decode())  encrypt\_file('./input/' + filename, './temp/encrypted.dat', aes\_key)  seed = generate\_seed\_from\_private\_key(private\_key)  split\_file('./temp/encrypted.dat', './splits/part1.dat', './splits/part2.dat', seed)  # 分割されたファイルを結合して復号化する関数  def combine\_and\_decrypt\_file(filename, private\_key):  seed = generate\_seed\_from\_private\_key(private\_key)  combine\_files('./splits/part1.dat', './splits/part2.dat', seed, './temp/combined.dat')  aes\_key = generate\_aes\_key\_from\_private\_key(private\_key)  print("AES Key:", binascii.hexlify(aes\_key).decode())  decrypt\_file('./temp/combined.dat', './output/' + filename, aes\_key)  if os.path.getsize('./input/' + filename) == os.path.getsize('./output/' + filename):  print("Success: Decrypted file matches the original file.")  else:  print("Failure: Decrypted file does not match the original file.")  # メイン関数  def main():  private\_key = 0xf8f8a2f43c8376ccb0871305060d7b27b0554d2cc72bccf41b2705608452f315 # 秘密鍵の設定  filename = 'memo.pdf' # ファイル名の設定  # ディレクトリの作成とクリア  for directory in ['./temp', './splits', './output']:  os.makedirs(directory, exist\_ok=True)  for file in os.listdir(directory):  os.remove(os.path.join(directory, file))  # ファイルを暗号化して分割  encrypt\_and\_split\_file(filename, private\_key)  # 分割されたファイルを結合して復号化  combine\_and\_decrypt\_file(filename, private\_key)  if \_\_name\_\_ == "\_\_main\_\_":  main() |

# 3. ネイティブアプリを用いたベクトルデータ暗号化の実行検証

この章では、ネイティブ上でのエントリーシステムの実行において、特に

* 準同型暗号ライブラリ Lattigo

を用いてユーザーの成績データであるベクトルデータを暗号化し、プロセッシングシステムへと送信する際の実行を詳細に検証し、サンプルコードを示すことを目的とする。

記載する理由としては、プロトタイプで開発するウェブアプリと、ネイティブアプリを開発する際の開発環境や言語が異なることがボトルネックとなり、実現したい「準同型暗号ライブラリ Lattigo」の活用が出来ないことを事前に防ぐためである。

## 3-1. 全体概要

この項では、ベクトルデータをユーザー端末からネイティブアプリを経由して暗号化し、プロセッシングシステムへと送信する際の

* システムアーキテクチャー
* データの暗号化手法
* 対象とするデータ
* 開発環境
* 実行されるプロセス
* ネイティブアプリ開発における焦点

について概要を記載する。

* **システムアーキテクチャー**
  + プロトタイプ開発フェーズのWebアプリ基本設計書と同様のため参照すること
* **データの暗号化手法**
  + Lattigo (準同型暗号 / 格子暗号のOSSライブラリ)を用いて格子暗号を用いて暗号化する
* **対象とするデータ**
  + 成績データ（数値を要素としたベクトルデータ）
* **開発環境**
  + Swift (iOSアプリの開発)
  + Kotlin (Androidアプリの開発)
* **実行されるプロセス**
  + **鍵生成**
    - Lattigoライブラリの関数をGo言語ランタイムで呼び出し
    - Swift、Kotlinからも同様のプロセスで鍵生成を行う
  + **鍵のシリアライズ・デシリアライズ**
    - Lattigoライブラリの関数をGo言語ランタイムで呼び出し
    - Swift、Kotlinからも同様のプロセスでシリアライズ・デシリアライズを行う
  + **暗号化**
    - Lattigoライブラリの関数をGo言語ランタイムで呼び出し
    - Swift、Kotlinからも同様のプロセスでデータの暗号化を行う
  + 暗号化プロセス
    - Go言語ランタイムを使用してLattigoの関数が呼ばれ、鍵生成、鍵のシリアライズ・デシリアライズ、暗号化が実行される
* **ネイティブアプリ開発における焦点**
  + **技術スタックの整合性**:
    - Swift、Kotlinでのエントリーシステム機能の実現において、プロトタイプとのミスマッチを防ぐ
  + **Lattigoライブラリの適用**:
    - プロトタイプ構築時と同様に、Lattigoライブラリを使用して安全な暗号化を実現
  + **ネイティブアプリ上でのプロセス実行**:
    - Swift、KotlinからLattigo関数を呼び出す実装が期待通りに動作することを確認

## 3-2. 使用技術スタック

この項では、ネイティブアプリ上でエントリーシステムを構築し、成績データを暗号化しプロセッシングシステムに送信するために必要な技術スタックについて記載し、実装時のフローについて言及することを目的とする。

* **ネイティブアプリ開発言語**
  + Swift (iOSアプリの開発)
  + Kotlin (Androidアプリの開発)
* **ベクトルデータの暗号化ライブラリ**
  + Lattigo ライブラリ
    - Go言語で実装されており、格子暗号の形式として CKKS 形式を採用
    - ベクトルデータの扱いに長け、準同型暗号を用いたアプリケーション開発に広く利用されている
* **バインディングツール**
  + gomobile ([リンク](https://github.com/golang/go/wiki/Mobile))
    - Go言語で書かれたライブラリをネイティブアプリの開発に利用
    - Go言語のライブラリをKotlin（Java）やSwift（Objective-C)のインターフェースとして活用

以上の技術スタックを用い、以下のようなフローに沿って開発を進める。

1. **ネイティブアプリ開発**
   * iOSアプリ: Swiftを使用
   * Androidアプリ: Kotlinを使用
2. **ベクトルデータの暗号化**
   * Lattigoライブラリを採用
   * CKKS形式を使用し、ベクトルデータを安全に暗号化
3. **gomobileを利用したバインディング**
   * Go言語で実装されたライブラリをgomobileを用いてバインディング
   * KotlinやSwiftのインターフェースとして、ネイティブアプリ開発に組み込み

## 3-3. ネイティブアプリ上での実行の注意点（リソース観点）

この項では、ネイティブアプリ上でLattigoを実行する際に、

* ネイティブアプリ上ではDockerコンテナよりもリソースが限られる

という観点から、いくつかの注意点を記載し、それらに対する対策などを提示することを目的とする。

### 注意点の概要

1. **計算リソースの制約**:
   * ネイティブアプリ上での構成では、計算リソースが制約される可能性がある。
   * 暗号化や鍵生成にかかる時間が増加し、ユーザーの体験に影響を与える可能性がある。
2. **シリアライズとデシリアライズのオーバーヘッド**:
   * gomobileを使用する場合、GoパッケージからのオブジェクトをSwiftやKotlinから直接認識できない場合があり、代替としてシリアライズやデシリアライズが必要。
   * データの変換に伴うオーバーヘッドが発生し、実行時間が影響を受ける可能性がある。

### 具体的な対策

1. **ユーザー体験の向上**:
   * ローディングアイコンの設置など、ユーザーへのUXを損なわないように注意。
   * バックグラウンドスレッドを使用して、計算タスクをUIスレッドとは別に実行する。
2. **シリアライズ・デシリアライズの最適化**:
   * シリアライズやデシリアライズの回数を最小限に抑える。
   * パフォーマンスの検証を通じて、適切な最適化手法を採用する。

### 実際の実行時間への影響

* **サンプルコードの実行結果**:
  + ネイティブアプリ上での鍵生成や暗号化において、オーバーヘッドがそれほど大きくないことが確認された。
    - ネイティブアプリをそれぞれ実機（iPhone14Pro, Pixel 4 XL）で起動し、メモリの使用量をモニタリング
    - ネイティブアプリ単体のメモリ使用量が２GBを超えないことを確認
  + サンプルコードのパフォーマンス検証を通じて、実運用に耐えうるレベルの実行時間が確認された。

### 備考

* **リアルタイム性の要件**:
  + ユーザーの期待値やアプリの要件によっては、リアルタイム性が求められる場合があり、それに応じた最適化が必要。
* **ベンチマークとネイティブアプリ上での実験**:
  + ネイティブアプリ上での実験やベンチマークを通じて、具体的なオーバーヘッドを確認し、必要に応じて最適化を行う。

## 3-4. サンプルコード

この項では、Lattigoライブラリをネイティブアプリ上で動かす際の実装のベースとなるサンプルコードを記載することを目的とする。

サンプルとなるコードはgitレポジトリにアップロードされており、後述の手順を持って環境構築を行うものとする。

### サンプルコード①

* [lattigo\_wasm\_ios\_android](https://github.com/kenmaro3/lattigo_wasm_ios_android) レポジトリをクローン

このレポジトリは、Lattigo を gomobile を使ってiOS開発環境(swift), Android開発環境(kotlin)から使用可能にするためのレポジトリである。

レポジトリのsimpleフォルダに有るsimple.go の中に、今回gomobile を用いてバインディングされる一連の関数

* 暗号パラメータの設定
* 鍵ペアの生成
* 暗号化
* 復号
* 通信に載せるためのシリアライズ、デシリアライズ

がサンプルとして記載されている。

クローンしたレポジトリ直下で、

|  |
| --- |
| gomobile init  go get golang.org/x/mobile/cmd/gobind |

としてgomobile をイニシャライズし、

|  |
| --- |
| gomobile bind -target ios |

のコマンドでios 用のバインディングを実行する。

また、

|  |
| --- |
| gomobile bind -target android |

でandroid 用のバインディングを実行する。

### サンプルレポジトリ②

1. [ios\_lattigo\_sample](https://github.com/kenmaro3/ios_lattigo_sample) レポジトリをクローン

このレポジトリには、Swiftコードからgomobileで生成されたライブラリを呼び出し、鍵生成や暗号化を実行するサンプルが含まれている。UIはサンプル用に簡易的なものをSwiftUIをインターフェースとして構築している。

クローンしたレポジトリ直下で

|  |
| --- |
| open SwiftWithGo.xcodeproj |

を実行することで、XCode上で開発を行う必要がある。

### サンプルレポジトリ③

1. [android\_lattigo\_sample](https://github.com/kenmaro3/android_lattigo_sample) レポジトリをクローン

このレポジトリには、Kotlinコードからgomobileで生成されたライブラリ(aarファイル）を呼び出し、鍵生成や暗号化を実行するサンプルが含まれている。UIはサンプル用に簡易的なものをJetpackComposeをインターフェースとして構築している。

クローンしたレポジトリをAndroid Studio で開き開発を行う必要がある。

# 4.ネイティブアプリのUI/UX

候補者向けに提供するUI/UXは主に以下の4つの要素で構成される。

* エントリー画面と付随する機能
* オファーおよびマッチング状況のステータス表示画面
* 候補者と企業担当者間のメッセージ機能
* 候補者アカウント設定機能

以下それぞれの要素におけるネイティブアプリのワイヤーフレームと機能概要について説明をしていく。

## 4-1. エントリー機能

グラフィカル ユーザー インターフェイス, アプリケーション

自動的に生成された説明

全ての候補者にとってエントリー画面が採用マッチングシステム利用の起点となる。候補者のエントリー機能は主に３つの画面で構成される。

* 1つ目は、企業に対する個人情報開示許可画面である。候補者が個人情報開示許可に同意をすることで、エントリーを開始することができる。
  + この画面から、個人情報取り扱い方針を明記したプライバシーポリシーに遷移することができ、プライバシーポリシー上では本プロダクトが秘密計算技術等を利用することによりユーザーの個人情報が秘匿化された状態で国際間のマッチングが可能になることが示されている。その上で、秘密計算技術などを利用した場合でも、ユーザーにとって起こり得る不利益などに関しても、黒字かつ下線で強調されており、ユーザーがきちんとプロダクト利用のリスクを十分に理解した上で、プロダクト利用ができるように配慮している。実際の本実証でベトナムのユーザー向けに作成したプライバシーポリシーは以下である。
  + <https://4447336.fs1.hubspotusercontent-na1.net/hubfs/4447336/PrivacyPolicy_TrustedWeb_IGS.pdf>
* 2つ目は、同意後のローディング画面である。このローディング中に成績データを暗号化し、PSに送信するといった処理がアプリ上で実行される。また、サーバーサイドではPSで秘密計算の処理等が実行される。詳細なロジックに関しては、後述のエントリーシステムの詳細または、Webアプリ版の要件定義書および基本設計書を参照。
* 3つ目は、ローディング終了後のエントリー完了画面である。こちらの画面から採用マッチングシステムのステータス画面に推移することができる。

**補足:**

* エントリー機能を利用できるのは、エントリーするための条件を満たした候補者のみである。条件を満たさない候補者に対しては「エントリーできません」といった文言のみが表示され、エントリーすることはできない。
* エントリー機能は初回のみのプロセスである。1度エントリーが完了したら、上記画面は表示されない。

## 4-2.   ステータス画面 & マッチング機能

グラフィカル ユーザー インターフェイス, テキスト, アプリケーション, チャットまたはテキスト メッセージ

自動的に生成された説明

ステータス画面では、候補者は希望年収、マッチング済み企業、候補者にオファー申請をした企業情報を閲覧することができる。

**ステータス表示画面**

* 最上部には、候補者の希望年収が表示される。希望年収は候補者がアップデート可能
* 中段にマッチング済み企業が表示される。マッチング済み企業とは自由にメッセージを送受信することができる。
* 下段にオファー申請した企業のリストが表示される。この時点ではマッチングはしていないため、自由にメッセージを送受信することはできない。

**マッチング機能**

* 候補者はオファー申請企業リストから企業名をクリックするとオファー画面に推移し、企業名とオファー金額に加えて、メッセージを閲覧することができる。この時点ではまだ候補者はメッセージに返信することができない。
* 候補者がマッチング許可をすると、マッチング成立画面が表示される。
* マッチング成立後該当企業はマッチング済み企業のリストに移動する。
* 候補者および企業担当者はマッチング後に、アプリ上でメッセージのやり取りができるようになる。

## 4-3.  メッセージ機能

グラフィカル ユーザー インターフェイス, アプリケーション

自動的に生成された説明

メッセージ画面では、マッチング済みの候補者と企業担当者が自由にアプリ上で面談日時や条件のすり合わせなどのやり取りをすることができる。

* マッチング済みのリストにある企業を選択すると、メッセージ画面に推移する。
* メッセージ画面上から、企業のアイコン等を選択すると企業情報および担当者の情報などをモーダル形式で閲覧することができる。
* 候補者はテキスト入力エリアから、テキストメッセージを送信できるだけではなく、画像等のメディアファイルも送信することができる。

## 4-4. アカウント/ データ設定機能

グラフィカル ユーザー インターフェイス, アプリケーション

自動的に生成された説明

データ設定画面では「成績データ」や「受講したコース」のデータを編集することができる。

* 候補者は成績データの企業に対する開示許可の設定をすることができる。
* 成績データの表示・非表示はスキル別（KnowledgeやResponsibility）に設定することができる。
* 候補者が成績データを非表示に設定した場合、企業担当者はシステムを通じて候補者の成績データを閲覧することができなくなる。
* 候補者は成績情報が含まれているコースデータを削除することができる。
* コースデータを削除した場合、削除した内容に応じて能力スコア等が再計算される。
* 上記再計算の具体的なロジックに関しては詳細設計となるため、開発のフェーズで検討。

# 5. その他ネイティブアプリ構築時の注意点

この章では、ネイティブアプリ構築時の注意点として、以下を記載することを目的とする。

* ネイティブアプリに対する一般的なセキュリィ攻撃と対策
* ネイティブアプリ実装時の、対象OS（iOS、Android）間の実装優先度に関する考察

## 5-1. ネイティブアプリに対する一般的なセキュリティ攻撃と対策

この項では一般的なネイティブアプリに対する攻撃の種類と、それらに対するリスク対策を記載

|  |  |  |
| --- | --- | --- |
| **リスク** | **解決策** | **対策の優先度** |
| データ漏洩 | * HTTPSの使用 * データの暗号化 * セキュアな認証メカニズムの実装 * 不要なデータの保存を避ける * セキュリティアップデートの迅速な適用 | 高 |
| マルウェア感染 | * アプリの信頼性確認 * スキャンツールの組み込み * 不正なコードの検知と削除 * ユーザーへのセキュリティ教育 | 高 |
| 不正なアクセス | * 2要素認証の実装 * 不審なアクティビティの監視 * アカウントロックアウトの実装 * ログの詳細な監査 | 中 |
| リバースエンジニアリング | * コードの難読化 * 定期的なアプリの更新と再署名 * デバッガの無効化 | 中 |
| インターセプト攻撃 | * トランスポート層セキュリティ * データの暗号化 * 中間者攻撃への対策 | 低 |
| 不正な権限の利用 | * 最小権限の原則を適用 * アプリの権限要求を検討 * ユーザーへの権限の説明 | 低 |

## 5-2. ネイティブアプリの実装優先度に関する考察

ここでは、ネイティブアプリ開発時のiOS, Android の２つのOSについて、以下の２つの観点から、実装する上でのOS間の優先度について考察し、結論づけることを目的とする。

* ネイティブアプリの対象となるユーザーの観点
* エンジニアリング工数の観点

### 普及率と対象ユーザー層

* **日本における普及率**:
  + iOS: 70%
  + Android: 30%
* **ベトナムにおける普及率**:
  + iOS: 30%
  + Android: 70%
* **優先度の考慮**:
  + 日本ではiOSが優勢だが、ベトナムではAndroidが主流。
  + ターゲットユーザー層や地域によって優先度を検討。

### 開発エンジニアリング観点からの差異

* **ライブラリ依存性**:
  + Gomobile使用で、SwiftとKotlinの開発で依存性の大きな差は予想されない
* **実装難易度**:
  + Gomobile経由のモジュール呼び出しは、Swift、Kotlinで類似し、難易度は同じ
* **審査工数**:
  + iOS審査はAndroidよりも厳しく、暗号化アルゴリズムの提出時に注意が必要

### 結論

* **ネイティブアプリの優先度**:
  + ターゲットユーザー層や地域によってiOSとAndroidの優先度を検討。
  + 開発コストや審査工数の違いを考慮し、実装の順序を設定。
* **クロスプラットフォーム開発**:
  + クロスプラットフォームを検討する場合も、Go言語のバインディングで共通の実装を行うことで、効率的な開発が可能。
* **地域差やターゲットユーザーの特性**:
  + ユーザー層や普及率の地域差を考慮し、戦略的な展開を検討。
* **審査工数への対応**:
  + iOSの審査がAndroidよりも厳しいため、スケジュールに余裕を持たせる。