1　何を作るか

1.1　理想

我々の理想はビットコインが「通貨」から「銀行」への「信用」を取り除いたように、クラウドコンピューティングおける「情報処理」から「クラウドベンダ」への「信用」を取り除くことである。これによって真の分散コンピューティングを達成できる。すなわち、我々の理想は、計算資源を特定の個人・組織に帰属するものではないようにすることである。

すべてのコンピュータが単一のネットワークに接続し、準同型暗号などによるセキュアな計算環境をそのネットワークの上で仮想的に構築することで、まるで唯一つのメインフレームにすべての人がアクセスしているような状況を作り出したいと考えている。

1.2　背景

本テーマの背景として最もわかりやすいものはクラウドコンピューティングの普及である。クラウドコンピューティングにおいては、クラウドベンダが保有するハードウェアを仮想化技術によって分割し、その上に OS などのソフトウェアをセッティングしたものを提供する形が一般的である。これは、クラウドコンピューティングとはクラウドベンダが提供する計算資源に対する信用をその基板としているということを意味する。そこに信用があるならば、当然セキュリティリスクもそこには存在する。我々が特に考えるセキュリティは、一般に耐タンパ性と呼ばれるものである。

耐タンパ性とは、端的に言えばソフトウェアのリバースエンジニアリングに対する耐性のことを言う。タンパは更にソフトウェアタンパとハードウェアタンパに分類される。ソフトウェアタンパとは OS やファームウェアの改造、専用のプログラムを用いた攻撃である。対して、ハードウェアタンパとはロジックアナライザをコンピュータに取り付けるなど、物理的な電気信号を観測して行う攻撃のことである。後者のほうが難易度は高い。

クラウドコンピューティングにおいては、OS 以下のすべての層のタンパ性についてクラウドベンダが責任を負う。それは同時に、クラウドベンダが望めば脆弱性を OS 以下のすべての層に仕込めることを意味する。

例えば OS においては、各ベンダーが自社のサービスに合わせて OS を改造することは一般的であり、この際に脆弱性を仕込むことも可能である。ベンダ提供の OS 使用を避ける方法の一つとしてベアメタルサーバの利用があるが、これは前の利用者がファームウェアを書き換えるといった脆弱性を産む。[1]。クラウドベンダはロジックアナライザなどの専用の機器をサーバに取り付けることも容易である。そうした器具がないことを保証するのもクラウドベンダの責任である。そのような器具がサプライチェーンで取り付けられていた事例としてはスーパーマイクロのものがある [2]。民生品を含んだ事例として Huawei の「余計なもの」報道もあった [3]。

図 1　クラウドコンピューティングでのセキュリティ管理責任

ハードウェアタンパに対する対策として実用化された手法には、TPM[4]、TEE[5] や Intel SGX[6] といったハードウェア的仕組みによるものがある。しかし、こうした対策は意味をなさなくなる可能性がある。現代においてプロセッサの製造は一部メーカーの専売特許ではなく、Google の TPU や AWS の Graviton のように、クラウドベンダがプロセッサを製造し運用するという状況はすでに現実のものとなっているからだ。[8] このような状況では、クラウドベンダはプロセッサのダイの中にさえ脆弱性を仕込むことがで

きることになる。

我々の問題意識は、クラウドコンピューティングの普及によって世の中の「情報処理」が「クラウドベンダー」という、多くの脆弱性を作り出すことのできる「単一障害点」に集約されつつある現状に端を発するのである。

1.3 目的

このような現状において、「クラウドベンダ」にセキュリティ管理を丸投げすることによるリスクを避けつつ、クラウドコンピューティングの恩恵を受け続けるには、ユーザが行うことができる自衛手段が必要である。つまり、アプリケーションに組み込めるセキュリティが必要である。本テーマはそのような手段が構築可能であることを実証することを意図したものである。

1.4 準同型暗号

準同型暗号（Homomorphic Encryption）は本テーマのキーとなる概念である。準同型暗号とは暗号文に対し暗号化したまま何らかの演算を行うことができるような暗号のことである。準同型暗号は以下のように 3 つに分類される。

PHE (Partially Homomorphic Encryption)　PHE は加法、または乗法だけが定義された準同型暗号のことである。加法だけが定義されたものを加法準同型暗号、乗法だけが定義されたものを乗法準同型暗号という。一般に対数・指数の関係を利用することで片方からもう片方を構成できる。

SHE (Somewhat Homomorphic Encryption, **制限付き準同型暗号**)　SHE は加法と乗法の両方が可能であるが、演算回数について制限のあるものである。

FHE (Fully Homomorphic Encryption, **完全準同型暗号**)　FHE は加法と乗法の両方が定義され、演算回数にも制限のないものを言う。Fully とついているのは、2 を法とした演算では加法は XOR・乗法は AND に相当し、結果として任意の論理演算が構成可能になるためである。現在は SHE をベースに Bootstrapping と呼ばれる手法を適用して構成するものが知られている。

PHE の中には公開鍵暗号として著名な RSA 暗号も含まれる。ここでは、説明の容易な ElGamal 暗号を用いて乗法準同型暗号が定義可能であることを示す。

平文を m_1, m_2、公開パラメータを g、秘密鍵を x、公開鍵を g^x、m_1, m_2 との組として選ばれた任意の乱数を r_1, r_2 とする。どちらにも適用される式の場合は下付きの添字を省略するものとする。

暗号化

$$Enc(m) := (g^r, mg^{rx})$$

乗算

$$Enc(m_1)Enc(m_2) := (g^{r_1+r_2}, m_1m_2g^{(r_1+r_2)x}) = Enc(m_1m_2)$$

復号化

$$Dec(Enc(m_1m_2)) := (g^{r_1+r_2})^{-x} \cdot m_1m_2g^{(r_1+r_2)x} = m_1m_2$$

加法準同型暗号を構成するには、何らかの定数 h を決めて、平文を h^m とすれば良い。

この例では ElGamal 暗号を用いていて説明したが、現在研究されている主な準同型暗号は LWE（Learning With Error）と呼ばれるものである。この手法は量子コンピュータでも解読が難しいとされている。ここでは詳細なアルゴリズムを述べないが、この手法の特徴は、名前の通り暗号文に誤差（Error）が導入されており、復号化するにはこの誤差が一定以下でなければならないことである。誤差が含まれている、ということは演算を行うとその誤差が増大するのは自然である。LWE は加算と乗法の両方を同時に定義できるが、演算を繰り返すと誤差が増大して復号できなくなるため、そのままでは SHE に分類される。

LWE から FHE を構成するためには、Bootstrapping と呼ばれる手法が必要である。これは、準同型暗号上で復号処理と同じような処理を行うことで LWE に演算によって蓄積された誤差をリセットする手法である。

本テーマでは LWE ベースの FHE をもちいる。

1.5 定式化

本テーマで考える問題は、以下のように定式化されるような 2 パーティーの問題である。

- アリス（ユーザ）は処理したいデータ（プログラムとその入力）を持っている。
- ボブ（サーバ）は与えられたプロトコル通りに計算を行ってくれるが、あらゆる手段を使って受け取ったデータを盗み取ろうとしてくる。
- ある UC（Universal Circuit; チューリング完全な汎用回路）が存在して、その回路構成は両者が共有している。
- このとき、アリスはデータ、及び UC によってそのデータを処理して得られた出力に関しての情報をボブに一切与えずに計算処理を行うことができるか。

1.6 開発するバーチャルセキュアプラットフォーム（VSP）について

1.6.1 概要

バーチャルセキュアプラットフォーム（Virtual Secure Platform; VSP）とは、ハードウェアによる機能に頼ることなく、C 言語など既存のコンピュータ向けプログラミング言語で書かれたプログラムをセキュアに実行できる環境を構成するプログラム群を指す言葉である。

今回我々がバーチャルセキュアプラットフォームの実装として実際に作るものは主に以下の 4 つの要素から構成される。

1. 準同型暗号ライブラリ
2. 準同型暗号ライブラリを用いた UC の実行可能なプログラムを Verilog から生成するツールチェイン（VSP-toolchain）
3. UC （Universal Circuit）の実装としての CPU を記述した Verilog ファイル（KRISP）
4. UC で動くバイナリへ C プログラムを変換するコンパイラ（RV32K-LLVM）

実装上の優先順位は、1. セキュリティ、2. 使いやすさ、3. 高速性である。

1.6.2 開発目標

テストケースは以下のような形に定式化される。こうした動きができるプログラムを開発するのが今回の目標である。

1. アリス（ユーザ）は実行したいデータ（RV32K-LLVM でコンパイルされた C プログラムのバイナリとその入力)、準同型暗号のための秘密鍵と公開鍵を用意する。
2. ボブは VSP-toolchain で生成された KRISP の実行ファイルを持つ。
3. 暗号化：アリスは公開鍵でデータを暗号化する。
4. 送信：アリスは公開鍵とともに暗号化されたデータを、処理するクロック数とともにボブに送信する。
5. 展開：ボブは暗号化されたデータを KRISP の実行ファイルにメモリデータとして入力する。
6. 処理：ボブは与えられたクロック数、アリスから与えられたデータを KRISP の実行ファイルによって処理する。
7. ステータス送信：ボブは、KRISP のステータスを示す特定番地のメモリをアリスに送信する。
8. 応答：アリスは受け取ったメモリデータを復号してステータスを確認し、処理が終了していない場合はその旨と次に実行するクロック数をボブへ伝え、5. に戻る。終了していた場合はその旨をボブへ伝える。
9. 結果送信：ボブは KRISP のメモリデータをアリスへ送信する。
10. 復号：アリスは受け取ったメモリデータを復号し、データの処理結果を得る。

より具体的には、アリスとボブは同一 PC 上のプログラムとして実装され、その通信はファイルを介して行うことで動作を確認する。

本テーマの中核をなすのは UC の実装である KRISP であり、その性能目標は準同型暗号ライブラリを用いて実行した際の実時間での CPU 周波数が 60Hz、RAM が 512Byte であることである。

1.6.3 準同型暗号ライブラリ

本テーマでは準同型暗号ライブラリまで作っていると時間が足りなくなるため、サードパーティ製のものを使用しようと考えている。我々が知る限り最も高速なライブラリは、CUDA を使わない場合は TFHE[12] であり、CUDA を使う場合は TFHE の CUDA

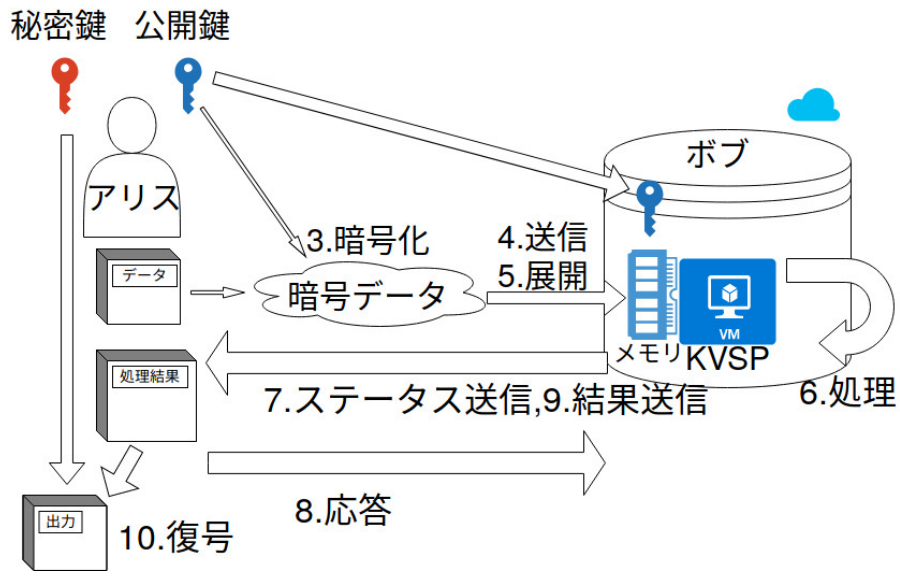

図 2　テストケースの模式図

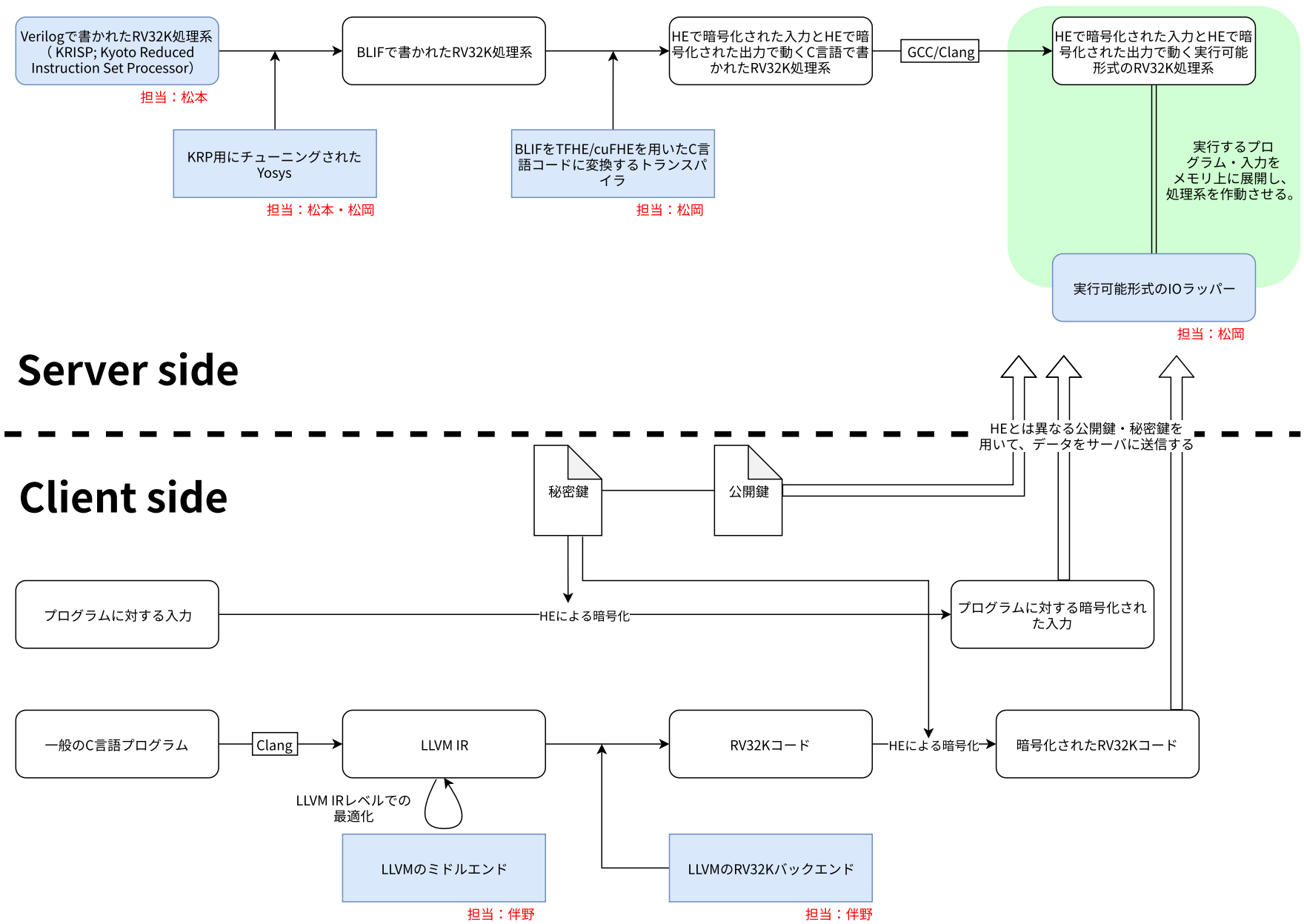

図 3　KRISP 全体の動作イメージ図

版である cuFHE[11] である。本テーマでは CPU で動作できることが好ましい初期の開発では前者を、性能のチューニングを目指す後期の開発では後者を用いる。

本テーマでは基本的に準同型暗号ライブラリに手を加えることは想定していない。一方でこのライブラリは KRISP の動作速度を決める本質的な要因の 1 つであるため、これを編集することは、時間に余裕が生まれた場合やどうしてもここで速度を稼ぐ必要が発生した場合の発展的課題である。

1.7 KRISP-toolchain

UC の実装は回路の記述であるため、本テーマではハードウェア記述言語として一般的な Verilog を想定している。

発想のベースは、Cingulata[10] である。このソフトウェアは特殊な記法の C++ をいわゆる高位合成言語として用い、フロントエンドでそれを BLIF（Berkeley Logic Interchange Format）[13] に変換、ミドルエンドで ABC[14]（独自のオープンソースライセンス）という BLIF を受け取り BLIF を返すプログラムを用いて最適化をほどこし、バックエンドで BLIF を Cingulata で独自に実装された準同型暗号ライブラリで動作するように変換する。

本テーマで作るツールチェインでは、入力を C++ ではなく Verilog とし、フロントエンドとミドルエンドを Yosys[15] とする。BLIF から TFHE または cuFHE を用いた C++ コードへ変換するバックエンド部分は自作する。なお Yosys も最適化には ABC を用いている。発想は Cingulata に端を発するが、コードはほとんど流用できないと考えている。

つまり、ツールチェインにおける主な開発対象は BLIF から準同型暗号ライブラリを用いた C++ コードへの変換部であり、次に Yosys の最適化部のチューニングである。本テーマのでは基本素子が XOR と AND である点や、フリップフロップに相当する記憶素子が基本的な要素である点などにおいて、必ずしも物理的なハードウェアに合わせた最適化が良いわけではないと考えている。

1.7.1 KRISP

1.7.1.1 ISA 設計方針

KRISP（Kyoto Reduced Instruction Set Processor）の ISA には、32bit 整数演算をサポートしつつも 16bit に命令が圧縮された RISC-V 命令セットである RV32C をベースとした独自規格 RV32K を用いる。RV32C をベースとする利点としては以下の点がある。

- 命令長を短くすることでメモリを有効活用できる。
- 既存の ISA をベースとすることで ISA 設計にかかるコストを軽減できる。
- LLVM に関して RV32I をサポートしているため LLVM の RV32K 向け改造が独自アーキテクチャに比べて容易である。
- RISC-V は BSD ライセンスによって提供されているため、今回のテーマに関して制約がかからない。

論理回路の規模の制約上、CISC 志向な x86 の実装は困難であり、RISC 系 CPU として有名な MIPS や ARM は、それらの命令をすべて実装しようとするとやはり論理回路の規模の制約に引っかかることになる。RISC-V は用途ごとに自分で最適な命令セットを組み立てることができる。このことと以上の利点から RV32C をベースとすることにした。

RV32C のシミュレータを開発し命令数や速度に関する検証をしつつ RV32C を改造し、RV32K として ISA を開発する予定である。現時点で RV32K と RV32C が違う点は、RV32K は 11 本のレジスタしか持たないことである。これは命令の構成を単純化するとともに回路サイズを小さくするためである。

1.7.1.2 マイクロアーキテクチャ設計方針

KRISP は命令長 16bit、データ長 32bit である。マイクロアーキテクチャの古典的類型として、プログラムとデータを同一メモリ上に保持するノイマン型アーキテクチャと、プログラムとデータを別々のメモリ上に保持するハーバードアーキテクチャがある。ノイマン型アーキテクチャは

- メモリを共有するため、メモリ空間の利用効率が高い。
- プログラムとデータがバスを共有するためロジックを削減できる。
- バスを共有するためデータとプログラムの読み出しが同時に行えず、パイプライン化等の最適化の障害となりうる。

という特徴を持つ。一方ハーバード型アーキテクチャは

- プログラムとデータのメモリが別々のバスにつながるため、構造が簡単である（バスの調停が必要ない）。
- それぞれのバスが干渉しないためパイプライン化が容易である。
- プログラムとデータが別々になるためメモリの利用効率は悪い。

という特徴を持つ。

KRISP では、メモリはマルチプレクサなどのロジックの塊であり出来る限り小さくすることが求められること、原理的に書き込みと読み込みが別のバスにならざるを得ない。そのため、ロジック規模を抑えながら限られたメモリ空間を最大限活用するために、ノイマン型アーキテクチャを採用する。図 4 がマイクロアーキテクチャの概念図である。マルチサイクルで命令を実行することでメモリから命令とデータの読み出しの衝突を回避し、パイプライン化も可能な設計方針とした。マイクロアーキテクチャの開発第一段階

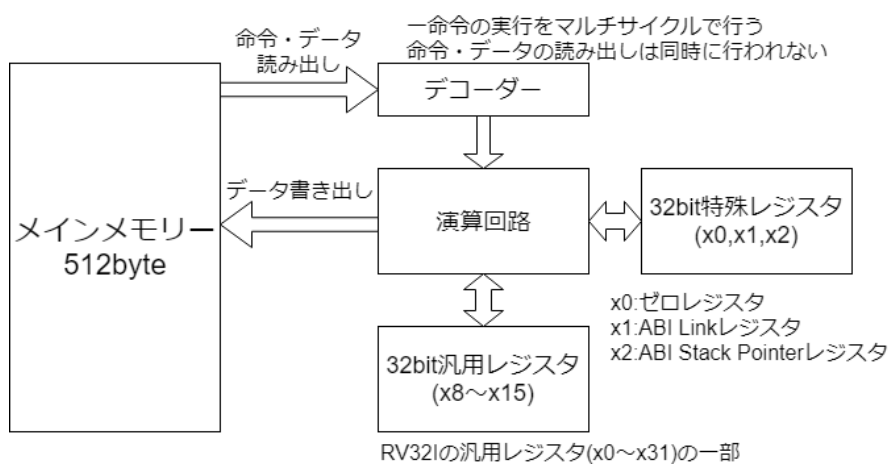

図 4　KRISP マイクロアーキテクチャ概念図

としては、図 4 をそのまま実装し性能を評価する。第二段階としては、律速している命令の複数クロック実行への変更や、パイプライン化、準同型暗号の演算自体を並列化つまりロジックの評価を並列化することにより目標の性能を達成する。

1.7.2　RV32K コンパイラにおける LLVM IR の採用

1.7.2.1　LLVM IR の特徴と採用理由

本テーマでは、ターゲットである ISA の RV32K を生成するために LLVM IR 及びそれに関するツール群を使用する。

準同型暗号を使用しセキュアにプログラムの実行を行うような既存実装は、得てしてプログラムに使用する言語として独自実装の DSL を採用している [10]。このことは、その実装を使用するハードルを一段上げることに直結してしまう。一方、本テーマの目標の一つは「簡単に使用できる」セキュアプラットフォームを開発することである。したがってこの目標を達成するためには、実際に広く使われている言語——例えば C 言語——を RV32K に変換できることが望ましい。しかしコンパイラ全てを作成することは難易度が高く、時間もかかる。我々の興味の向くところはそのコード生成部のみであり、それ以外の部分は既存の実装を利用することが理想的である。また AST から RV32K コードへの変換はどの言語においても本質的に共通していると考えられるため、可能な限り多言語においてその実装を共通したものにしたい。

そこで本テーマでは、LLVM IR を RV32K に変換するような LLVM バックエンドを開発する。

LLVM IR とは、高水準言語を表現可能な一種の中間言語表現であり、型安全性・低水準操作性・柔軟性を兼ね備えている [?]。一般的に LLVM IR を使用する場合、まず対象の高水準言語を LLVM IR に変換する。この変換器をフロントエンドと呼ぶ。次に得られた LLVM IR を最適化する。この部分をミドルエンドと呼ぶ。その後、LLVM IR を出力対象の低水準言語に変換する。この変換器をバックエンドと呼ぶ [22]。従って、本テーマで我々が主に作成するのは RV32K のためのバックエンドということになる。一度バックエンドを構成してしまえば、理想的には、どのようなフロントエンドが受け取るプログラムについても、それと同等の RV32K のコードを得ることができる。LLVM IR のフロントエンドとして、C・C++ などのコンパイラである Clang が挙げられる。我々の当面の目標は、C 言語コードを KRISP 上で実行することである。RV32K のバックエンドが完成すれば、C 言語コードを Clang に通すことで同等の LLVM IR コードを得、さらにこれを開発したバックエンドに通すことで同等の RV32K コードを得ることができる。これを KRISP 上で実行することで目標を達成する。

LLVM における「フロントエンド——LLVM IR——バックエンド」という統一された構造は、第三者がその内容を把握・修正することを容易にする。したがって本テーマ終了後異なる ISA・実装などにより VSP を新たに構築する際に、我々の未踏事業において得た知見を流用しやすいという点でも LLVM を採用する価値がある。

1.7.2.2 準同型暗号上での計算を見越した最適化手法

RV32K コンパイラを作成する際の最も重要な事柄の 1 つとして、コンパイラが出力する RV32K コードは、物理的なプロセッサ上ではなく、準同型暗号を利用する KRISP 上で実行されるという事実がある。準同型暗号における演算の計算量的観点から、KRISP が使用できるメモリ量は 512Byte に限られており、また KRISP の動作周波数も大きくはない。従って本テーマにおいて開発する RV32K コンパイラが出力する実行形式が現実的な速度で動作するためには、最終出力コード量及びその使用メモリ量が可能な限り抑えられている必要がある。これを実現するためにはコンパイラにおける最適化が欠かせない。コードの実行速度と使用メモリ量は一般的にトレードオフの関係にあるため、最もバランスが取れる程度を実験などにより見極め、LLVM バックエンド及びミドルエンドにおいて最適化を施す必要がある。

本テーマにおけるコンパイラ開発が典型的なコンパイラ開発と異なる点として、本テーマでは ISA に手を加えることが可能であることが挙げられる。KRISP の実装とコンパイラ実装の双方を勘案し、最良の ISA を定義することが必要である。

2 どんな出し方を考えているか

本テーマの成果は利用するサードパーティのライブラリ等のライセンスに反しない限り Apache2.0 ライセンスのもとで公表するものとする。これは我々の目指す分散コンピューティングのためには、バーチャルセキュアプラットフォーム自体が特定の組織の信用に基づいたものであってはならないためである。

我々が所属する京都大学は 11 月に学祭を行うため、そこで何かしらの中間発表を行いたいと考えている。

我々は本テーマが十分に新規性のあるものと考えているため、未踏での開発期間終了後に論文にまとめたいと考えている。開発時間の確保の都合上、期間中に執筆活動等を行う予定はない。

3 斬新さの主張、期待される効果など

3.1 分散コンピューティング

ブロックチェーンの分散処理能力を利用して分散アプリケーションを達成するための仕組みに EVM[7] があるが。そのセキュリティを改善するために立ち上げられたプロジェクトが MIT のεnigma であり、現在存在するものの中では最も我々の理想に近いものである [9]。このプロジェクトでは EVM での暗号化された情報処理を達成するために、準同型暗号による計算処理を実装している。しかし、εnigma で使われている準同型暗号は後述する SHE に分類され、また組み合わせ回路的に扱うことを想定している点で本テーマで目指すものとは異なるものになっている。

我々の想定している具体的目標の 1 つに、このεnigma の計算処理部を置き換えることがある。

しかし我々としては、アプリケーションだけを分散するのではなく、コンピュータそれ自体を分散することこそが理想であると考えている。すなわち、リモートデスクトップ環境を分散するような状況である。この世のすべてのコンピュータが単一のネットワークで繋がり、まるでひとつのメインフレームのように振る舞うわけである。そうしたものができれば、人々は高性能なコンピュータを所有する必要がなくなり、シンクライアント端末を蛇口とした、水道のように計算資源を使えるような世界が実現できると考えている。

そのための第一歩として、本テーマがあると考えている。

3.2 先行研究

世界初と考えるのであれば、その根拠を述べなければならないので、ここでは先行研究の内、特に重要なものをまとめる。

まず上げるべきは、”Perennial secure multi-party computation of universal Turing machine”[16] である。この論文は昨年 10 月に公開されたものだが、我々が考えているような、複数のクラウドサーバーが分散して暗号化されたコンピューティングにおけるセキュリティを考える際の理論的枠組みを提案している。逆に言えば、今我々が理想としているものは、その理論的枠組みすら確立されていないものとも言える。

より今回のテーマで達成するものに近しいものとしては、GarbledCPU[17] と ARM2GC[18] がある。これらは、Garbled Circuit という準同型暗号とは異なる仕組みを用いたセキュアな仮想プロセッサである。前者は MIPS を、後者は ARM を実装している。GC は一回使うとセキュアでなくなってしまうために、順序回路を作るためにはクロック数分の GC を用意しなければならない欠点

がある。

準同型暗号上での仮想プロセッサ実装を見つけることはできなかったが、Oblivious Turing Machine の亜種を作ったという論文は存在した。[20] この論文の場合と本テーマの違いの一つとしては、この論文はチューリングマシンであるためにメモリに相当するテープをサーバー側に実装せざるを得ず、メモリアクセスをいかに秘匿するかが問題になったが、本テーマの場合はレジスタファイルとして準同型暗号上に回路の一部として実装するためにその問題が発生しないことがある。

準同型暗号上で、通常のプログラミングに近い環境を提供するという発想のものでは、先に紹介した Cingulata[10] 以外にも ALCHEMY[21] がある。

以上に見たように、GC でのプロセッサ実装は見られるものの、準同型暗号でのプロセッサ実装は新規なものであると考えられる。これはおそらく、現在の準同型暗号の利用法の研究が順序回路的使い方ではなく、組み合わせ回路的使い方の方に主眼をおいているためであると思われる。

4　具体的な進め方と予算

4.1　使用する言語・ツール

VSP-toolchain は Cingulata が C++ で書かれているため、C++ で記述する予定である。KRISP は Verilog で記述される。RV32K-LLVM は C++ と LLVM 用の DSL である TableGen によって記述される [22]。

4.2　分担について

図 3 も参考のこと。

担当者名	内容
松岡	BLIF から C++ コードへの変換部の開発 暗号化されたデータを KRISP のメモリへ展開するための IO プログラム 全体の統括
伴野	RV32K-LLVM（KRISP 用の LLVM ミドルエンド・バックエンド）の開発
松本	KRISP の Verilog 記述の開発
松岡・松本	Yosys の最適化チューニング
伴野・松本	KRISP シミュレータ（Verilog での開発時のリファレンスとしての暗号化されていないシミュレータの開発） RV32K の策定

4.3　開発線表

参考文献

[1] https://twitter.com/utshina2/status/1100498441938055169

[2] https://www.bloomberg.co.jp/news/articles/2018-10-04/PG2CZY6TTDS801

[3] https://www.fnn.jp/posts/00397920HDK

[4] https://docs.microsoft.com/ja-jp/windows/security/information-protection/tpm/how-windows-uses-the-tpm
[5] https://news.mynavi.jp/article/20131209-arm_tee/
[6] https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-1-foundation
[7] https://github.com/ethereum/wiki/wiki/%5BJapanese%5D--Ethereum-Development-Tutorial
[8] https://www.itmedia.co.jp/enterprise/articles/1812/05/news059.html
[9] https://enigma.co/
[10] https://github.com/CEA-LIST/Cingulata
[11] https://github.com/vernamlab/cuFHE
[12] https://github.com/tfhe/tfhe
[13] http://www1.cs.columbia.edu/~cs6861/sis/blif/
[14] http://people.eecs.berkeley.edu/~alanmi/abc/
[15] https://en.wikipedia.org/wiki/ISC_license
[16] https://www.sciencedirect.com/science/article/pii/S0304397518306297
[17] http://www.aceslab.org/sites/default/files/GarbledCPU_0.pdf
[18] https://arxiv.org/abs/1902.02908
[19] https://eprint.iacr.org/2016/654.pdf
[20] https://arxiv.org/pdf/1312.3146.pdf
[21] https://web.eecs.umich.edu/~cpeikert/pubs/alchemy.pdf
[22]『きつねさんでもわかる LLVM〜コンパイラを自作するためのガイドブック〜』（柏木 餅子・風薬・矢上 栄一、株式会社インプレス、2013 年）