2019年度未踏IT人材発掘・育成事業
準同型暗号によるバーチャルセキュアプラットフォームの開発
成果報告書

クリエータ：松岡 航太郎
伴野 良太郎
松本 直樹

担当PM：首藤 一幸

2020年3月6日

目 次

1　要約

本プロジェクトでは準同型暗号を用いてプログラムの秘匿を実現する、バーチャルセキュアプラットフォームを提案・実装した。C プログラムを暗号化により秘匿しつつ実行できることが本プロジェクトの特徴である。本プロジェクトでは暗号・CPU 設計・コンパイラなど全面からの最適化により 1 クロック約 1.5 秒の実行速度を実現した。

2　背景及び目的

2.1　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では、計算資源を提供するクラウドベンダが保有するクラウドサーバに、利用者であるクライアントが計算処理を移譲する（図 1）。

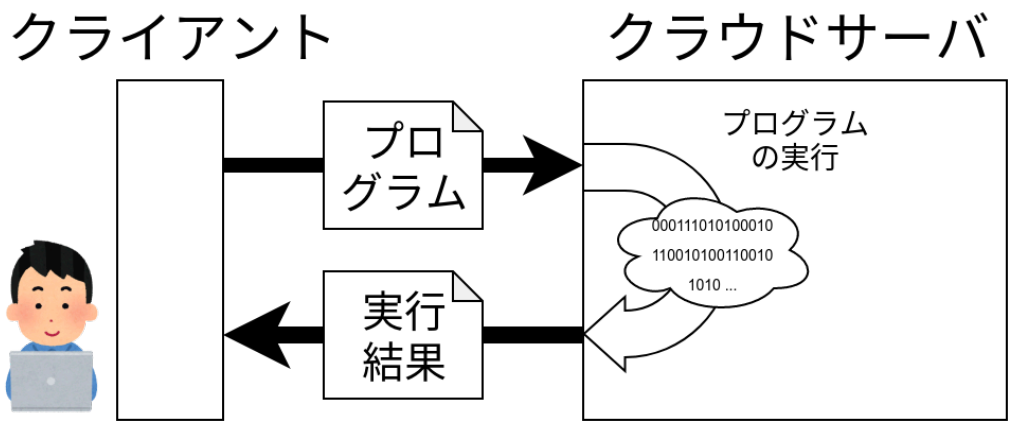

図 1: クラウドコンピューティングの模式図

ここで盗聴者がクラウドサーバに直接アクセスできるような場合を考える。ここで言う盗聴者は悪意ある第三者も含むが、特にクラウドベンダ自身のことを想定している。現在普及しているコンピュータのハードウェアにおいては、少なくとも CPU のチップ内において、プログラムの実行時にはプログラムとその入力であるデータは平文になっていなければならない。そのため、ロジックアナライザやサイドチャネル攻撃などの手法によってハードウェアを流れる電気信号を観測できた場合、平文になっているプログラムやデータを抜き取られる恐れがある。

クラウドコンピューティングにおいては、クラウドサーバの OS 以下全ての層の安全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、専用の器具を取り付けるなど様々な脆弱性をクラウドサーバに仕込めることを意味する（図 2）。

このような現状では、クライアントはクラウドサーバに脆弱性が存在しないことを信じてクラウドコンピューティングを利用する他ない。これは取りも直さず、ク

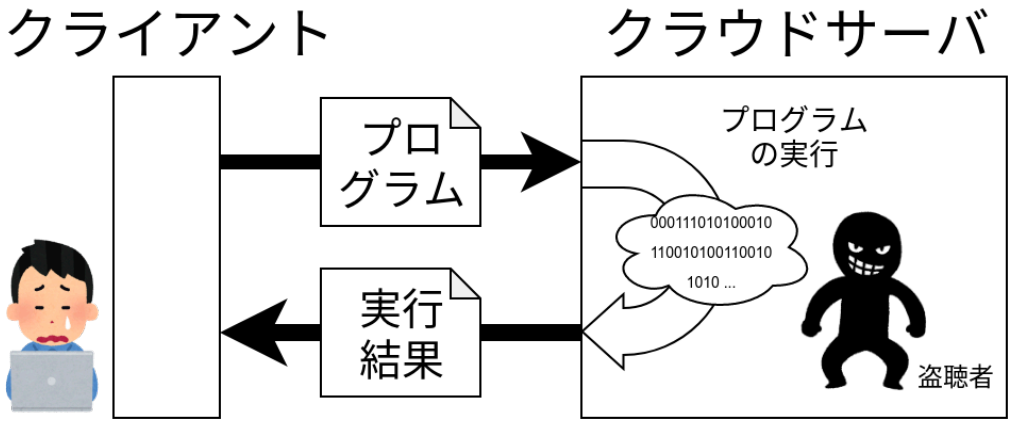

図 2: クラウドサーバは脆弱性を持ちうる

ラウドベンダへの信用がクラウドコンピューティングの基盤となっていることを意味している。

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」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。

2.2 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、つまりアプリケーションに組み込める、ソフトウェアだけで実現できるセキュリティが必要である。

本プロジェクトではそのようなセキュリティの一つとして、プログラムの秘匿がソフトウェアだけで達成可能であることを示すことを目的としている。より具体的には、プログラムとデータの両方を暗号化したまま計算処理を行えるような仕組みをソフトウェアだけで実現する。そうすることによって、たとえ任意の位置の電気信号を読み取ることができたとしても暗号文しか抜き取られないために、計算量的安全性を保証することができる（図 3)。

この目的を達成する仕組みが、我々が提案するバーチャルセキュアプラットフォーム（Virtual Secure Platform; VSP）である。VSP の特徴は以下の 3 つである。

1. データだけでなくプログラムも秘匿したまま計算処理ができること
2. 暗号学のみに基づいて安全性を保証できること
3. 高級言語で書いたプログラムが動作すること

1 は本プロジェクトが対応するセキュリティの範囲の定義である。プログラムの秘匿という点に重きを置くのは、データの秘匿だけであれば既存の手法でも十分に可能であるためである。2 はクラウドベンダを信用せずに済むように、暗号学だけ

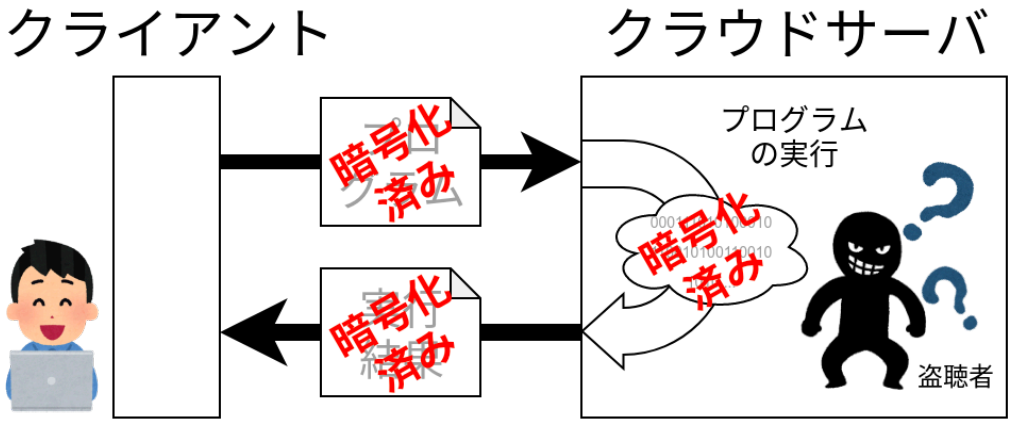

図 3: 本プロジェクトで実現する暗号化によるプログラムの秘匿

に依拠することを要請する。3 はセキュリティと利便性というのは得てしてトレードオフの関係にあると考えるために要請される。

2.3 先行研究

2.3.1 brainfreeze・ArcaneVM

brainfreeze[1] と ArcaneVM[2] は準同型暗号上で一種の CPU を実行しているという点で、我々のプロジェクトに近いものである。しかしながら、両者とも難解プログラミング言語 Brainfuck を命令セットとして採用しており、VSP が言うところの「高級言語の実行」からは外れたものとなっている。

2.3.2 GarbledCPU・ARM2GC

GarbledCPU[SSZ+16]・ARM2GC[SRH+19] は、Garbled Circuit によって CPU エミュレーションを行うことでプログラムの秘匿を実現するような手法である。前者は MIPS、後者は ARM を ISA として採用している。これらの手法では、実行するクロックが増えれば増えるほどクライアントが生成しなければならない暗号文の量が増え、クライアントの負荷が増大してしまう。他方 VSP は暗号文のサイズが一定、つまりクライアントの負荷が一定であるため、計算処理の移譲に、より適している。

2.3.3 Intel SGX

Intel SGX は Intel 社が提供しているセキュリティ用の機能であり、メモリ上に「Enclave」と呼ばれる暗号的に厳重に保護された領域を生成することで、センシティブデータを保護しつつプログラムを実行する為の CPU の拡張機能である [3]。この機能は Intel 社の提供するハードウェアに依存しており、Intel 社を信用する必要があるという点で VSP とは異なるものとなっている。

2.3.4 Homomorphic Processor

Homomorphic Processor とは、準同型暗号上での演算を行える物理的な CPU を構成する手法である。データの秘匿だけに注力して暗号文に対する加算や乗算といった拡張命令を通常の CPU に実装する [IMP18]、暗号化された命令はデコードできないために命令を 1 つしかもたないプロセッサにする [TM14][CS15] などのアプローチが存在する。これらのアプローチは、物理的な論理ゲートをどのように組み合わせれば準同型暗号を処理できるプロセッサを作れるかという立場に立ったものであり、物理的な論理ゲートを準同型暗号によって置き換えるという VSP の立場とは異なるものと考えられる。

2.3.5 Cingulata

Cingulata[4] は、C++コードを高位合成によって論理回路に変換し、その論理回路を準同型暗号上で実行するというアプローチをとっている。論理回路の構成自体は漏れてしまうため、プログラムは秘匿することができない。

2.3.6 ALCHEMY

ALCHEMY[CPS18] は Cingulata（2.3.5 節）に似ているが、独自の DSL を用いてプログラムを記述する点で異なる。

3 プロジェクト概要

バーチャルセキュアプラットフォーム（Virtual Secure Platform; VSP）の基本的なアイデアは、CPU を準同型暗号（12.1 項参照）上でエミュレーションする、というところにある。ここでは、このアイデアがどのようなものであるかについて説明する。

通常のプログラムの実行は、CPU が解釈できる 0 と 1 のバイナリになったプログラムを CPU に流し込むと、結果が出てくるという流れとして解釈できる。ここで CPU の物理的内部構造に着目すると、CPU は論理回路と呼ばれる電子回路で表現されることが知られている。更に論理回路は、論理ゲートと呼ばれる回路をつなぎ合わせたものとして表現できる。つまり、CPU は論理ゲートをつなぎ合わせたものとして表現できる。CPU の回路の一部を切り取ると図 4 のようになる。

目指すところは、CPU が解釈できる 0 と 1 のバイナリになったプログラムを暗号化して渡すと、暗号化されたままそれが処理されて、暗号化された結果が出てくることである。つまり、CPU の処理を暗号化したまま行うことである。これは、CPU を表現した論理回路と等価な処理を暗号化したまま行うということと同値である。このことを図 4 と対応させると図 5 となる。この四角で囲われた「？」、つまり 0 と 1 のどちらかを暗号化したものを受け取って、その位置に本来あるべき論理ゲート

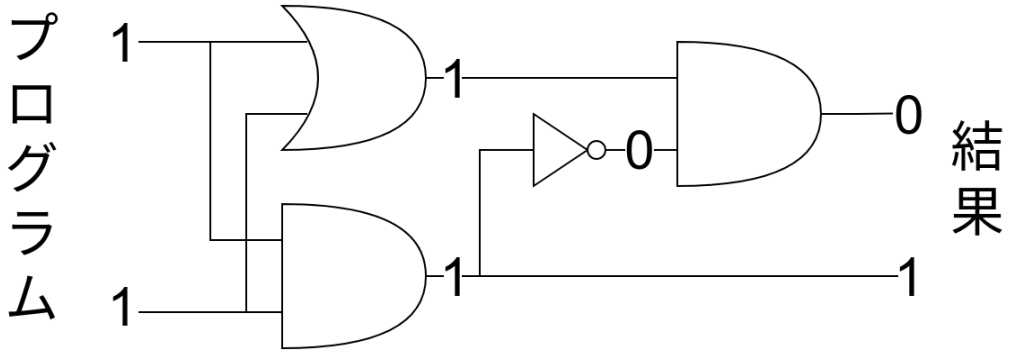

図 4: CPU の論理回路の一部

と等価な処理を暗号化したまま行えるものがあれば、CPU 全体を準同型暗号上でエミュレーションすることができる。

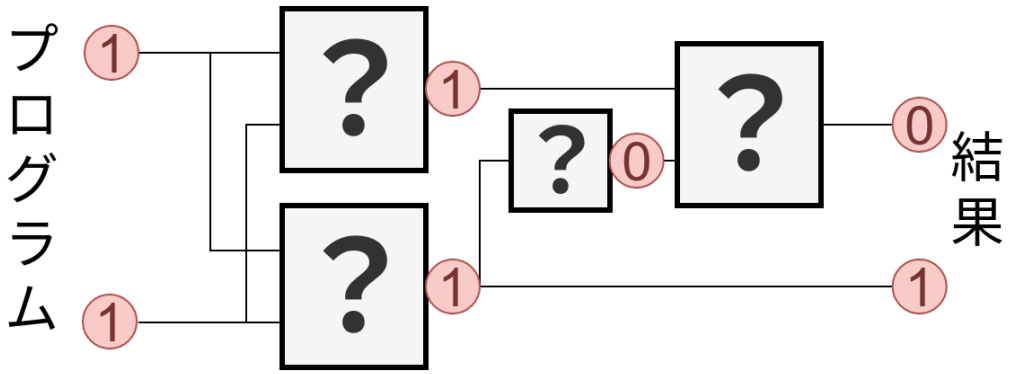

図 5: 図 4 の入出力と途中結果を暗号化した場合の概念図
暗号は赤丸で囲った。

「？」の位置に本来あるべき論理ゲートと等価な処理を暗号化したまま行えるものとして、我々は準同型暗号を用いた。準同型暗号は図 6 で表されるような暗号である。図 5 の「？」を図 6 の準同型暗号上の演算で置き換えることで、暗号化したまま等価な演算ができるようになる（図 7）。

以上をまとめると、我々の基本的なアイデアは次のように表現できる。まず CPU を論理ゲートの集合として表現し、全ての論理ゲートを準同型暗号上の演算に置き換える。ここに CPU が解釈できる 0 と 1 のバイナリになったプログラムを暗号化して渡すと、暗号化されたままそれが処理されて、暗号化された結果を得ることができる。

ここで重要なのは、準同型暗号上の演算は一般的な CPU で実行可能なプログラムとして表現でき、特殊なハードウェアを要しないことである。このアイデアは図 8 のように表現される。一般的なサーバのハードウェア上で準同型暗号上の演算を行うプログラムを走らせ、それによって CPU の論理ゲートを準同型暗号上の演算で置き換えたものを実行し、その準同型暗号上の演算で表現された CPU の論理回路[1]へ暗号化されたプログラムを流し込むことで暗号化された結果を得る。

[1]本プロジェクトの発想の流れを追うための補足を述べておく。CPU を論理回路、つまり論理ゲー

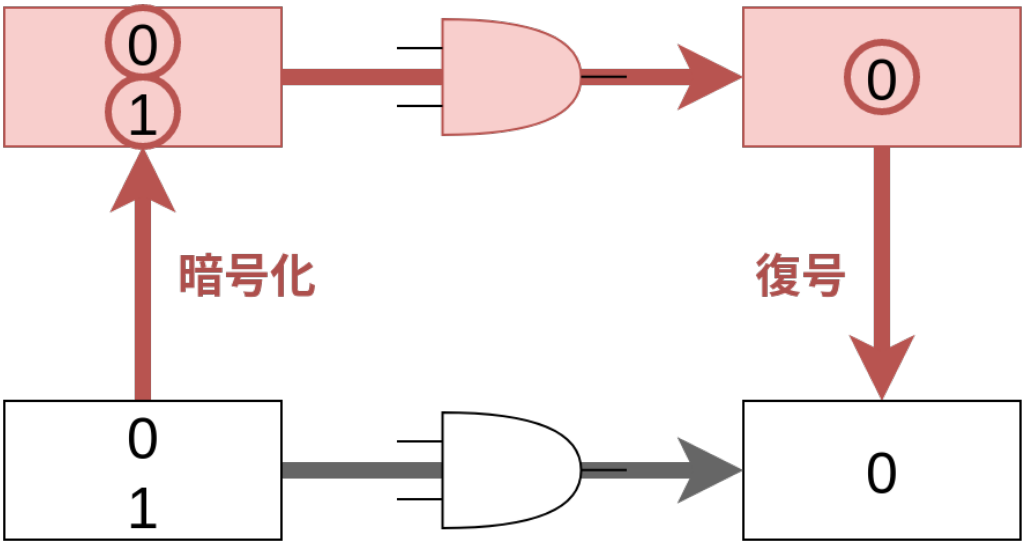

図 6: 準同型暗号による論理ゲートと等価な処理
平文を処理するゲートを白色で、暗号を処理するゲートを赤色で示した。

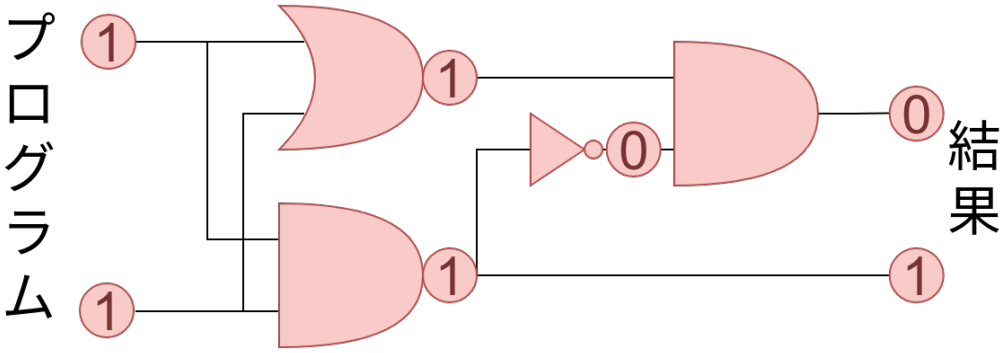

図 7: CPU の論理回路の一部を準同型暗号上の演算で置き換えたもの

本プロジェクトでは、VSP の具体的な実装として、C 言語で書かれたプログラムをセキュアに実行できる KVSP（Kyoto Virtual Secure Platform）を制作した（図 9）。

4 開発内容

4.1 概要

4 節では本プロジェクト期間内に行ったバーチャルセキュアプラットフォームの開発について、その詳細を述べる。開発したソフトウェアの概要図を図 10 に掲げる。

トの集まりとして表現する最大のメリットは、NAND と呼ばれる 1 種類の論理ゲートさえあれば任意の論理回路を表現できることが知られていることである。つまり、CPU は NAND というただ 1 つの論理ゲートの集まりとして表現できるのである。ここのことから、「CPU と同じ処理を準同型暗号上で実現できるか」という問題の「CPU」を「NAND」という至極単純なものに置き換えることができると気づいたのである。実際には NAND だけでは回路表現が複雑になるので、他のゲートも実装では用いている。

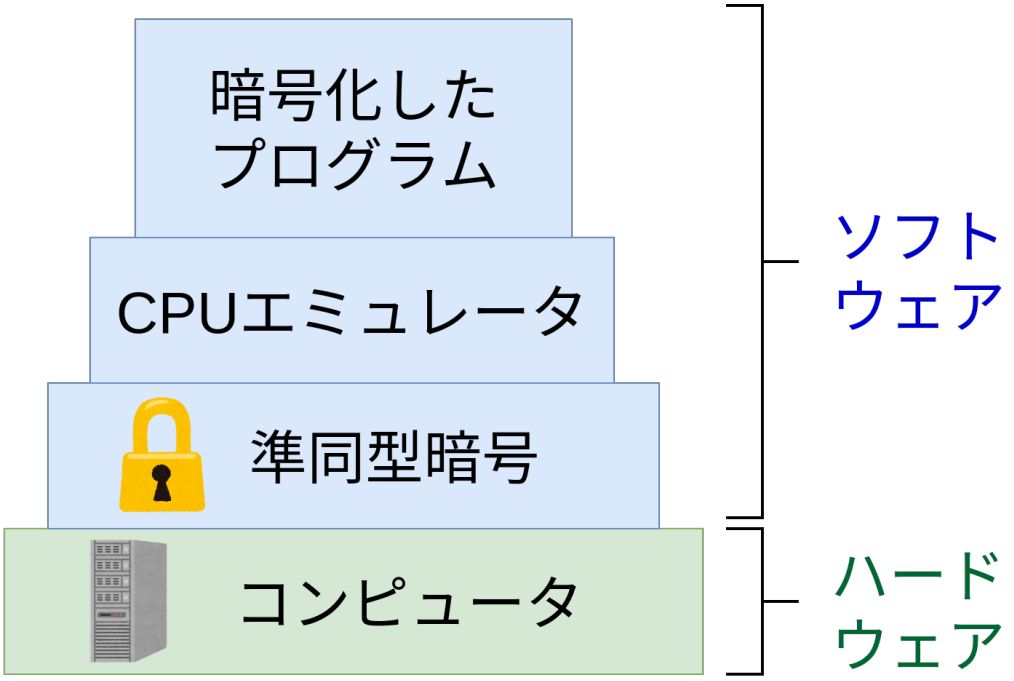

図 8: アイデアのスタック図

これら全体が KVSP であり、またバーチャルセキュアプラットフォームのリファレンス実装でもある。なお図 10 には本成果報告書の執筆時点で最新の KVSP（v8[21]）に使用されているコンポーネントのみを表示している。4 節では開発途中で放棄された、または完成したが現在使用されていない成果物についても述べる。

4.2 準同型暗号ライブラリの開発

4.2.1 pyFHE

pyFHE[5] は、Python による TFHE のフルスクラッチ実装である。TFHE の原理を確認する目的で制作された。原論文者実装 [6] に比べると、10 倍ほど遅くなっている。しかし、原論文者実装では別リポジトリに PoC があるのみ [7] で統合されていなかった Circuit Bootstrapping と呼ばれる機能を実装している。Circuit Bootstrapping の用途については 4.8 項も参考のこと。

4.2.2 TFHEpp

TFHEpp[8] は pyFHE をベースに、原論文者実装から C との互換性を廃し、同時に C++のモダンな機能を取り入れることでコード量が少なく拡張しやすい実装としてフルスクラッチで開発されたものである。この実装も pyFHE と同様に Circuit Bootstrapping を実装している。

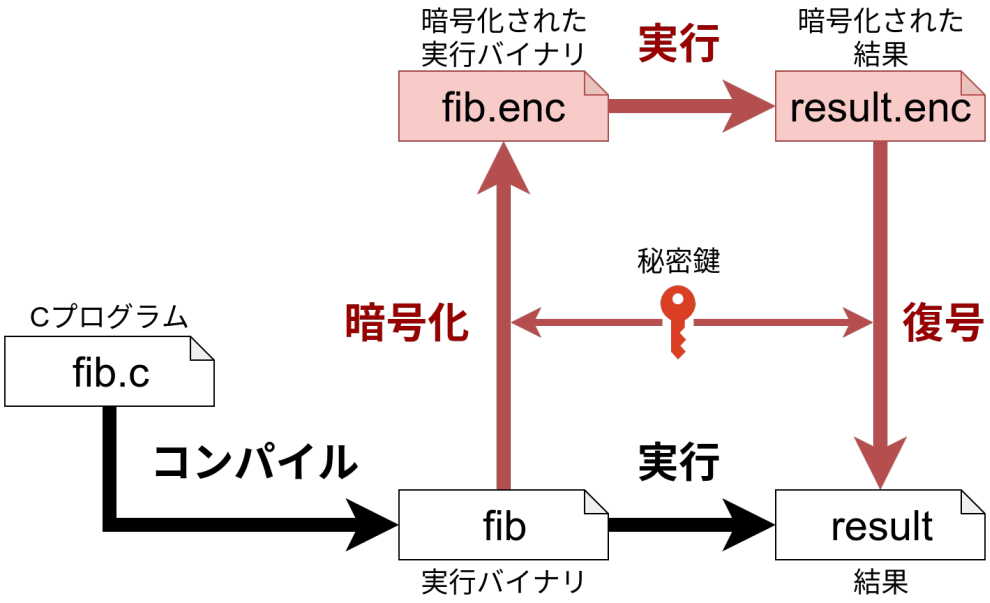

図 9: バーチャルセキュアプラットフォームの模式図

TFHEpp は原論文者実装に比べると 10%ほど高速である。これは暗号の強度や暗号文に対して演算を行う時の計算量を決めるパラメータをヘッダファイルに定数として記述したために、コンパイラの最適化が効きやすかったことが要因と考えられる。

4.2.3 cuFHE

cuFHE[9] は TFHE の CUDA 実装であって、同名の既存実装 [10] を fork してバグを修正すると共にマルチ GPU への拡張などを施したものである。

4.2.4 安全性と誤り確率の検証

TFHE においては暗号の安全性は選択するパラメータによって変動し、それは TFHE の論理ゲートの誤り確率にも影響する。そのため、それらを検証するためのコードを用意した [11]。安全性に関しては Homomorphic Encryption Security Standard[ACC+18] にならって、lwe-estimator を用いた [12]。KVSPv8 時点でのパラメータでは、古典的コンピュータで 154.5bit、量子コンピュータで 95.8bit の安全性を有している。

4.3 ISA の開発

本プロジェクトで想定した VM に搭載された ROM・RAM は、容量が各々512 バイトのみであった。プログラマの努力次第でその使用量を削減できる RAM と異なり、ROM の使用量はプログラムが使用する命令自体のサイズに依存する。したがっ

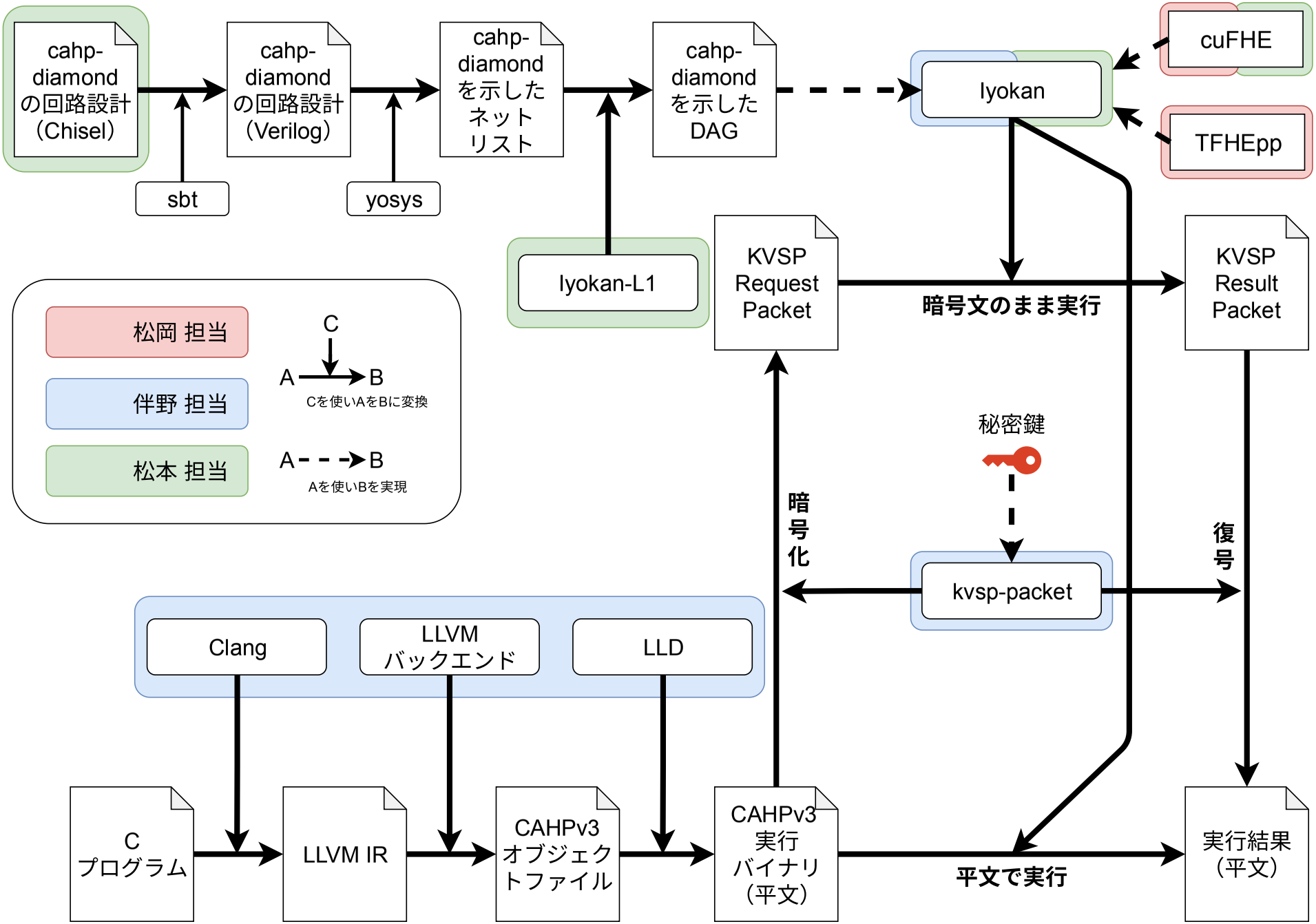

図 10: KVSP の概要図

て、コンパイル後のコードサイズが大きくなるような ISA では、この ROM を有効に使用することはできない。

そこで我々は、コンパイル後のコードサイズが小さくなるように独自に ISA を作成した。その過程として RV32Kv1 と RV16Kv2 という ISA を作り、最終的に CAHPv3 という ISA を作ってこれを採用した。

以下ではこれらの ISA について詳述する。

4.3.1 RV32Kv1

RISC-V[28] の RV32IC という ISA は他の ISA と比べコードサイズが削減されることが報告されている [16]。そこで我々は RV32IC の命令のうち命令長が 16bit のもののみを取り出し、これに不足した命令を独自に付け足して RV32Kv1 という ISA を 6 月に新しく作成した（図 11）。したがって RV32Kv1 は 32bit アーキテクチャで、2 オペランドの命令が主体である。

16bit のもののみを取り出すことによって、RV32IC よりもコードサイズが削減されることを期待したが、RV32Kv1 の枠組みでは関数呼び出しを行うことができないと後日分かったため、放棄された。

4.3.2 RV16Kv2

RV32Kv1 の反省を踏まえ、RV32Kv1 に 2 オペランドの命令を大幅に補填した RV16Kv2 を 8 月に策定した（図 12）。RV16Kv2 が RV32Kv1 と大きく異なる点は（RV32Kv1 と比較して）充実した命令と、アーキテクチャが 16bit になったことである。

ROM・RAM が 512 バイトしかない都合上、32bit アーキテクチャにおける長大なアドレス空間（32bit）は全く不要であり、命令サイズを増やす要因となっていた。そこで RV16Kv2 では 16bit アーキテクチャを採用し、アドレス空間を 16bit とした。これによって命令中の即値 bit 幅を削減することができ、主要な命令のサイズを 16bit に抑えることに成功した。一方で 16bit 即値の読み込みなどでは大きい即値 bit 幅が必要なため「即値ぶら下げ」という機構を開発した。これは 16bit の命令の後ろに続けて 16bit の即値を入れることで擬似的に 32bit 命令を実現するものであった。

しかし実際にコンパイラを作成して出力結果を確認すると、RISC-V と比較してコードサイズが小さくなるどころか、むしろ大きくなってしまっていることが分かった（4.4.7 節）。この原因の一つは、RV16Kv2 が 2 オペランドを採用したことである。2 オペランドの命令の場合、オペランドとして指定した 2 つのレジスタ（やメモリ）のうち片方は値が上書きされてしまう。そのためこれを保持しておくためには一度メモリに格納する必要があり、この操作に必要なコードのためにコードサイズが大きくなってしまっていた。

また他の原因として、即値ぶら下げが起こる頻度が想定よりも多く、結果として疑似 32bit 命令が比較的大量に使われてしまっていた。

RV32Kv1

Ops	How it works	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
lw rs1, uimm7(rd)	rd <- [rs1 + uimm7]	0	1	0	uimm[5:3]			rs1			uimm[2|6]			rd		0	0
sw rs2, uimm7(rs1)	[rs1 + uimm7] <- rs2	1	1	0	uimm[5:3]			rs1			uimm[2|6]			rs2		0	0
nop		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
jal simm12	ra <- ra + 2, pc <- pc + simm12	0	0	1	imm[11|4|9:8|10|6|7|3:1|5]											0	1
li rd, simm6	rd <- simm6	0	1	0	imm[5]			rd					imm[4:0]			0	1
sub rs1, rs2	rs1 <- rs1 - rs2	1	0	0	0	1	1	rs1			0	0	rs2			0	1
slt rs1, rs2	rs1 <- (rs1 < rs2 ? 1 : 0)	1	0	0	1	1	1	rs1			1	0	rs2			0	1
j simm12	pc <- pc + simm12	1	0	1	imm[11|4|9:8|10|6|7|3:1|5]											0	1
beqz rs1, simm9	if rs1==0 then pc <- pc + simm9	1	1	0	imm[8|4:3]			rs1			imm[7:6|2:1|5]					0	1
bnez rs1, simm9	if rs1!=0 then pc <- pc + simm9	1	1	1	imm[8|4:3]			rs1			imm[7:6|2:1|5]					0	1
lwsp rd, uimm8(sp)	rd <- [sp + uimm8]	0	1	0	uimm[5]			rd				uimm[4:3|8:6]				1	0
jr rs1	pc <- rs1	1	0	0	0			rs1			0	0	0	0	0	1	0
mv rd, rs2	rd <- rs2	1	0	0	0			rd					rs2			1	0
jalr rs1	ra <- ra + 2, pc <- rs1	1	0	0	1			rs1			0	0	0	0	0	1	0
add rs1, rs2	rs1 <- rs1 + rs2	1	0	0	1			rs1					rs2			1	0
swsp rs2, uimm8(sp)	[sp + uimm8] <- rs2	1	1	0			uimm[5:2|7:6]						rs2			1	0

図 11: RV32Kv1 の命令リスト

RV16Kv2

name	operation	note	#bytes	Instruction C		Regist	Imm F	MISC	ALU Opcode			Source Register				Destination Register				imm type	instruction format		imm
				15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0				
M命令																							
lw rd, imm(rs)	rd <- [rs + imm]		4	1	0	1	1	0	0	1	0	rs				rd				simm16_lsb0	RR32	B2	
lwsp rd, imm(sp)	rd <- [sp + U(imm)]		2	1	0	1	0	imm[8:1]								rd				uimm9_lsb0	SL	A	
lbu rd, imm(rs)	rd <- zext([rs + imm])		4	1	0	1	1	1	0	1	0	rs				rd				simm16	RR32	BA	
lb rd, imm(rs)	rd <- sext([rs + imm])		4	1	0	1	1	1	1	1	0	rs				rd				simm16	RR32	BE	
sw rs, imm(rd)	[rd + imm] <- rs	store word	4	1	0	0	1	0	0	1	0	rs				rd				simm16_lsb0	RR32	92	
swsp rs, imm(sp)	[sp + U(imm)] <- rs		2	1	0	0	0	imm[8:5]				rs				imm[4:1]				uimm9_lsb0	SS	8	
sb rs, imm(rd)	[rd + imm] <- rs	store byte	4	1	0	0	1	1	0	1	0	rs				rd				simm16	RR32	9A	
R命令																							
mov rd, rs	rd <- rs		2	1	1	1	0	0	0	0	0	rs				rd					RR16	C0	
add rd, rs	rd <- rd + rs		2	1	1	1	0	0	0	1	0	rs				rd					RR16	E2	
sub rd, rs	rd <- rd - rs		2	1	1	1	0	0	0	1	1	rs				rd					RR16	E3	
and rd, rs	rd <- rd & rs		2	1	1	1	0	0	1	0	0	rs				rd					RR16	E4	
or rd, rs	rd <- rd | rs		2	1	1	1	0	0	1	0	1	rs				rd					RR16	E5	
xor rd, rs	rd <- rd ^ rs		2	1	1	1	0	0	1	1	0	rs				rd					RR16	E6	
lsl rd, rs	rd <- rd << rs	logical left shift	2	1	1	1	0	1	0	0	1	rs				rd					RR16	E9	
lsr rd, rs	rd <- rd >> rs	logical right shift	2	1	1	1	0	1	0	1	0	rs				rd					RR16	EA	
asr rd, rs	rd <- rd >>> rs	arithmetic right shift	2	1	1	1	0	1	1	0	1	rs				rd					RR16	ED	
cmp rd, rs	rd - rs		2	1	1	0	0	0	0	1	1	rs				rd					RR16	C3	
I命令																							
li rd, imm	rd <- imm		4	0	1	1	1	1	0	0	0	0				rd				simm16	RR32	78	
addi rd, imm	rd <- rd + S(imm)		2	1	1	1	1	0	0	1	0	imm				rd				simm4	RI	F2	
cmpi rd, imm	rd - S(imm)		2	1	1	0	1	0	0	1	1	imm				rd				simm4	RI	D3	
J命令																							
j imm	pc <- pc + imm + 2		4	0	1	0	1	0	0	1	0	0	0	0	0	0	0	0	0	simm16_lsb0	RR32	52	
jal imm	RA <- pc + 4, pc <- pc + imm + 2	jump and link	4	0	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0	simm16_lsb0	RR32	73	
jalr rs	RA <- pc + 2, pc <- rs	jump and link register	2	0	1	1	0	0	0	0	1	rs				0	0	0	0		RR16	61	
jr rs	pc <- rs		2	0	1	0	0	0	0	0	0	rs				0	0	0	0		RR16	40	
jl imm	if SF != OF then pc <- pc + imm	signed <	2	0	1	0	0	0	1	0	0	0	imm							simm8_lsb0	B	44	
jle imm	if SF != OF || ZF == 1 then pc <- pc + imm	signed <=	2	0	1	0	0	0	1	0	0	1	imm							simm8_lsb0	B	44	
je imm	if ZF == 1 then pc <- pc + imm	equal to	2	0	1	0	0	0	1	0	1	0	imm							simm8_lsb0	B	45	
jne imm	if ZF != 1 then pc <- pc + imm	not equal to	2	0	1	0	0	0	1	0	1	1	imm							simm8_lsb0	B	45	
jb imm	if CF == 1 then pc <- pc + imm	unsigned <	2	0	1	0	0	0	1	1	0	0	imm							simm8_lsb0	B	46	
jbe imm	if CF == 1 || ZF == 1 then pc <- pc + imm	unsigned <=	2	0	1	0	0	0	1	1	0	1	imm							simm8_lsb0	B	46	
nop			2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0		RR16	0	

flags	SF:Sign Flag	ZF:Zero Flag
	OF:OverFlow Flag	CF:Carry Flag

FLAGS = SZCV
FLAGS[3] = Sign
FLAGS[2] = Zero
FLAGS[1] = Carry
FLAGS[0] = OverFlow

initial pc = 0x0
initial sp is set in _start

図 12: RV16Kv2 の命令リスト

4.3.3 CAHPv3

RV16Kv2 の反省を踏まえ、3 オペランドを採用した 16bit アーキテクチャとして CAHPv3 を策定した（図 13）。CAHPv3 では、RV32IC に含まれる命令のうち 32bit のものを 24bit に縮め、これによってコードサイズの削減を図った。RV32IC が 32bit アーキテクチャであるのに対して CAHPv3 は 16bit アーキテクチャであるため、冗長な情報（余分なレジスタや即値の指定など）を命令から排除することでこれは達成された。

実際コンパイラ開発後のテストによって、RV32IC や RV16Kv2 と比べても CAHPv3 ではコードサイズが削減されていることがわかった（4.4.7 節）。

4.4 コンパイラの開発

4.4.1 概要

本プロジェクトでは 3 種類の ISA（RV32Kv1・RV16Kv2・CAHPv3）を新規に作成した。これらの ISA は独自のものであるため、この上で C プログラムを動作させるためには、C プログラムからこれら ISA 用の実行バイナリを生成する C コンパイラが必要である。

本プロジェクトでは LLVM[27] を用いて独自 ISA 用の C コンパイラを作成した。このうち RV16Kv2 と CAHPv3 については実用的なレベルに達し、後述する 10 個のテストプログラムに対する正常な動作を確認できた。なお RV32Kv1 用のコンパイラについては、ISA に含まれる命令が不十分であったことが開発開始後にわかり、放棄された。

また LLVM による C コンパイラ実装の他に、証明支援系 Coq を用いた形式的に安全な C コンパイラ CompCert の改変による C コンパイラの実装も試みたが、16bit アーキテクチャに CompCert 本体が対応していないことが実装開始後にわかり、放棄された。

以下では、まず LLVM における C コンパイラ開発について述べた上で、実際に各 ISA にて作成したものとその相違点について考察する。また CompCert の改変の試行についても述べる。

4.4.2 LLVM とは

LLVM はオープンソースのコンパイラ・ツールチェイン実装基盤である。その構造は図 14 のようになっている。C などのプログラムはまずフロントエンド（FE）によって LLVM IR と呼ばれる中間言語表現に変換される。この LLVM IR は、バックエンド（BE）によってアセンブリ・オブジェクトファイルに変換される。我々のプロジェクトではこの LLVM バックエンドにあたる部分を新規に開発し、LLVM IR が我々独自の ISA に変換されるようにした。

CAHPv3

Notes	Ops	How it works	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	inA	inB	op	inst	Test
																					EX Opcode			Instruction Category		24bit Inst FLAG					
	M-Instruction																														
	lw rd, simm10(rs)	rd <- [rs + simm10]				simm10[7:0]						rs				rd			simm10[9:8]		0	1	0	1	0	1	rs	simm10	ADD	0	o
	lb rd, simm10(rs)	rd <- [rs + simm10]				simm10[7:0]						rs				rd			simm10[9:8]		1	0	0	1	0	1	rs	simm10	ADD	1	o
	lbu rd, simm10(rs)	rd <- [rs + simm10]				simm10[7:0]						rs				rd			simm10[9:8]		0	0	0	1	0	1	rs	simm10	ADD	1	o
	sw rs, simm10(rd)	[rd + simm10] <- rs				simm10[7:0]						rd				rs			simm10[9:8]		x=0	1	1	1	0	1	rd	simm10	ADD	0	o
	sb rs, simm10(rd)	[rd + simm10] <- rs				simm10[7:0]						rd				rs			simm10[9:8]		0	0	1	1	0	1	rd	simm10	ADD	1	o
	li rd, simm10	rd <- simm10				simm10[7:0]					x=0	x=0	x=0	x=0		rd			simm10[9:8]		1	1	0	1	0	1		simm10	MOV	1	o
	R-Instruction																														
	add rd, rs1, rs2	rd <- rs1 + rs2	x=0	x=0	x=0	x=0			rs2			rs1				rd			x=0	0	0	0	0	0	0	1					o
	sub rd, rs1, rs2	rd <- rs1 - rs2	x=0	x=0	x=0	x=0			rs2			rs1				rd			x=0	0	0	0	1	0	0	1					o
	and rd, rs1, rs2	rd <- rs1 & rs2	x=0	x=0	x=0	x=0			rs2			rs1				rd			x=0	0	0	1	0	0	0	1					o
	xor rd, rs1, rs2	rd <- rs1 ^ rs2	x=0	x=0	x=0	x=0			rs2			rs1				rd			x=0	0	0	1	1	0	0	1					o
	or rd, rs1, rs2	rd <- rs1 | rs2	x=0	x=0	x=0	x=0			rs2			rs1				rd			x=0	0	1	0	0	0	0	1					o
	lsl rd, rs1, rs2	rd <- rs1 << rs2	x=0	x=0	x=0	x=0			rs2			rs1				rd			x=0	0	1	0	1	0	0	1					o
	lsr rd, rs1, rs2	rd <- rs1 >> rs2	x=0	x=0	x=0	x=0			rs2			rs1				rd			x=0	0	1	1	0	0	0	1					o
	asr rd, rs1, rs2	rd <- rs1 >>> rs2	x=0	x=0	x=0	x=0			rs2			rs1				rd			x=0	0	1	1	1	0	0	1					o
	I-Instruction																														
	addi rd, rs1, simm10	rd <- rs1 + simm10				simm10[7:0]						rs1				rd			simm10[9:8]		0	0	0	0	1	1	rs1	simm10		2	o
	andi rd, rs1, simm10	rd <- rs1 & simm10				simm10[7:0]						rs1				rd			simm10[9:8]		0	1	0	0	1	1	rs1	simm10		3	o
	xori rd, rs1, simm10	rd <- rs1 ^ simm10				simm10[7:0]						rs1				rd			simm10[9:8]		0	1	1	0	1	1	rs1	simm10		3	o
	ori rd, rs1, simm10	rd <- rs1 | simm10				simm10[7:0]						rs1				rd			simm10[9:8]		1	0	0	0	1	1	rs1	simm10		3	o
	lsli rd, rs1, uimm4	rd <- rs1 << uimm4	x=0	x=0	x=0	x=0		uimm4[3:0]				rs1				rd			x=0	x=0	1	0	1	0	1	1	rs1	uimm4		3	o
	lsri rd, rs1, uimm4	rd <- rs1 >> uimm4	x=0	x=0	x=0	x=0		uimm4[3:0]				rs1				rd			x=0	x=0	1	1	0	0	1	1	rs1	uimm4		3	o
	asri rd, rs1, uimm4	rd <- rs1 >>> uimm4	x=0	x=0	x=0	x=0		uimm4[3:0]				rs1				rd			x=0	x=0	1	1	1	0	1	1	rs1	uimm4		3	o
	J-Instruction																														
	beq rs1, rs2, simm10	if rs1 == rs2 then PC <- PC + simm10				simm10[7:0]						rs1				rs2			simm10[9:8]		0	0	1	1	1	1					o
	bne rs1, rs2, simm10	if rs1 != rs2 then PC <- PC + simm10				simm10[7:0]						rs1				rs2			simm10[9:8]		1	0	1	1	1	1					o
	blt rs1, rs2, simm10	if rs1 < rs2 then PC <- PC + simm10				simm10[7:0]						rs1				rs2			simm10[9:8]		1	1	0	1	1	1					o
	bltu rs1, rs2, simm10	if rs1 < rs2 then PC <- PC + simm10				simm10[7:0]						rs1				rs2			simm10[9:8]		0	1	0	1	1	1					o
	ble rs1, rs2, simm10	if rs1 <= rs2 then PC <- PC + simm10				simm10[7:0]						rs1				rs2			simm10[9:8]		1	1	1	1	1	1					o
	bleu rs1, rs2, simm10	if rs1 <= rs2 then PC <- PC + simm10				simm10[7:0]						rs1				rs2			simm10[9:8]		0	1	1	1	1	1					o
実装しない	j simm16	PC <- PC + simm16								simm16[15:0]									x=0	x=0	x=0	0	0	1	1	1					
実装しない	jal simm16	RA <- PC + 4, PC <- PC + simm16								simm16[15:0]									x=0	x=0	x=0	0	0	1	1	1					
Notes	**Ops**	**How it works**	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**inA**	**inB**	**op**		
	16bit Length Instruction																														
	M-Instruction																														
	lwsp rd, uimm7(sp)	rd <- [sp + uimm7]										uimm7[4:1]				rd			uimm7[6:5]		0	1	0	1	0	0	sp	uimm7	ADD	4	o
	swsp rs, uimm7(sp)	[sp + uimm7] <- rs										uimm7[4:1]				rs			uimm7[6:5]		0	1	1	1	0	0	sp	uimm7	ADD	4	o
	lsi rd, simm6	rd <- simm6										simm6[3:0]				rd			simm6[5:4]		1	1	0	1	0	0		imm	MOV	5	o
	lui rd, simm6	rd <- (simm6 << 10)										simm6[3:0]				rd			simm6[5:4]		0	0	0	1	0	0		imm	MOV	6	o
	R-Instruction																														
	mov rd, rs	rd <- rs										rs(rs2)				rd(rs1)			1	1	0	0	0	0	0	0					o
	add2 rd, rs	rd <- rd + rs										rs(rs2)				rd(rs1)			1	0	0	0	0	0	0	0					o
	sub2 rd, rs	rd <- rd - rs										rs(rs2)				rd(rs1)			1	0	0	0	1	0	0	0					o
	and2 rd, rs	rd <- rd & rs										rs(rs2)				rd(rs1)			1	0	0	1	0	0	0	0					o
	xor2 rd, rs	rd <- rd ^ rs										rs(rs2)				rd(rs1)			1	0	0	1	1	0	0	0					o
	or2 rd, rs	rd <- rd | rs										rs(rs2)				rd(rs1)			1	0	1	0	0	0	0	0					o
	lsl2 rd, rs	rd <- rd << rs										rs(rs2)				rd(rs1)			1	0	1	0	1	0	0	0					o
	lsr2 rd, rs	rd <- rd >> rs										rs(rs2)				rd(rs1)			1	0	1	1	0	0	0	0					o
	asr2 rd, rs	rd <- rd >>> rs										rs(rs2)				rd(rs1)			1	0	1	1	1	0	0	0					o
	I-Instruction																														

図 13: CAHPv3 の命令リスト

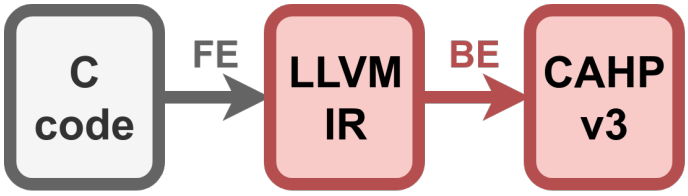

図 14: LLVM におけるコンパイル手順
赤色の変換を担当するバックエンド（BE）を独自 ISA 用に作成した。

4.4.3 開発手順

LLVM バックエンドは LLVM の独自 DSL である TableGen と C++プログラムによって記述される。具体的には、まず TableGen によってターゲットである ISA の枠組みを記述し、これを `llvm-tblgen` に読み込ませて C++プログラムを生成させる。その後 TableGen では表現できない部分について C++プログラムを記述する。結果として、最終的には全てが C++プログラムとして表現され、コンパイルされることになる。

以下に具体的な開発手順と、その詳細を述べる。4.4.4 節・4.4.5 節・4.4.6 節では（多少前後することもあるが基本的に）全てこの手順に従った。

1. アセンブラ・ディスアセンブラの作成
   LLVM バックエンドの MC layer と呼ばれる部分を実装し、アセンブラ・ディスアセンブラを開発した。具体的には、独自 ISA の命令やレジスタ、relocation や fixup などの再配置情報などを TableGen によって記述した。またこの TableGen スクリプトによって生成される C++プログラムを呼び出すための C++プログラムや、その他必要な C++プログラムを記述した。これによって LLVM に独自 ISA の情報を伝えることができ、既存の LLVM 実装と組み合わせた形で独自 ISA 用のアセンブラ・ディスアセンブラを動作させることができた。

2. 基本的なコード生成部の作成
   LLVM IR からアセンブリを生成するためのコード生成部のうち、基本的な部分（算術演算・メモリ操作・条件分岐・関数定義・関数呼び出し・インラインアセンブリなど）を作成した。具体的には、LLVM IR のある並びにマッチしてアセンブリに変換するようなパターンのリストを TableGen を用いて記述し、これを C++プログラムから呼び出した。また関数呼び出しや関数定義時のレジスタ・スタック操作、条件分岐時に使用する命令の選択などは複雑で TableGen では記述できないため、別途 C++プログラムによって記述した。

3. 応用的なコード生成部の作成
   以上で基本的なコード生成は可能になったが、大規模なコードをコンパイルしようとすると機能不足のためにエラーとなる場合があるため、これを解消した。具体的には branch analysis と branch relaxation を実装することによって長距離の（即値 bit 幅に入らないような）分岐命令に対応する他、16bit レジ

スタを用いた32bit整数値の演算などにも対応した。この作業によって、ライブラリ呼び出しを含まないようなおおよそ全てのLLVM IRのコンパイルを行うことができるようになった。

4. Clangにおける独自ISA用ドライバの作成
   以上の開発は全てLLVM IRをアセンブリにコンパイル（アセンブル）するためのものであり、Cプログラムをコンパイルするためには Cコンパイラ（Cプログラムを LLVM IRに変換する）である Clangに手を入れる必要があった。ここでは Clangに独自ISA用ドライバを追加し、整数型のサイズなどの情報を Clangに与えた。これによって独自ISA用の LLVM IRを Clangに生成させることができるようになった。

5. LLDにおける独自ISAドライバの作成
   以上でCプログラムをアセンブリ、またそれと等価なオブジェクトファイルに変換可能になった。しかしこれを実際に実行するためには、標準ライブラリやスタートアップスクリプトなどとまとめてリンクし、1つの実行バイナリを生成する必要があった。これにはリンカが必要である。本プロジェクトではLLVMツールチェインに含まれる LLDというリンカを採用した。具体的には LLDに独自ISA用ドライバを追加し、オブジェクトファイルに含まれるセクションの再配置方法を指定した。

各項目において、その変更が正しいかどうかをチェックするためのテストコードを同時に記述した。これによって回帰バグなど、見つけにくいバグを取り除くことができた。

以上の開発は全てイテレーティブに行われた。すなわちどの時点でもコードはコンパイル可能・動作可能であり、これによって自分が加えた変更点が間違っていないことを保証できた。

4.4.4 RV32Kv1用Cコンパイラの開発

6月から7月にかけて、LLVM v8.0.0上にて RV32Kv1用Cコンパイラの開発を行った。具体的には4.4.3節のうち「アセンブラ・ディスアセンブラの作成」を完了し、「基本的なコード生成部の作成」の途中までを行った。

関数呼び出しを実装しようとした際に、RV32Kv1の枠組みでは関数呼び出しを行うことができないことが分かったため、ISA放棄と同時にこのCコンパイラ開発も放棄された。

4.4.5 RV16Kv2用Cコンパイラの開発

8月から9月にかけて、LLVM v8.0.0上にて RV16Kv2用Cコンパイラの開発を行った[32]。具体的には4.4.3節の全ての項目について開発を完了した他、次世代ISA（CAHPv3）を策定するために、RV16Kv2に変更を加えた際に結果として得られる

ISA	合計コードサイズ（バイト）
CAHPv3	1973
RV16Kv2	2474
RV32IC（LLVM）	2006
RV32EC（GCC[26]）	2224

表 1: ISA ごとのコードサイズの比較

実行バイナリがどのように変化するかを調査した。この結果については 4.4.7 節にて詳解する。

また、おおよそ全ての実装を終えた後に、LLVM v8.0.0 から v9.0.0 への移行を行った。

4.4.6 CAHPv3 用 C コンパイラの開発

10 月から 11 月にかけて、LLVM v9.0.0 上にて CAHPv3 用 C コンパイラの開発を行った [33]。具体的には 4.4.3 節の全ての項目について開発を完了した。

このコンパイラは、本成果報告書執筆時点での KVSP の最新版（v8[21]）に搭載されている。

4.4.7 RV16Kv2 用と CAHPv3 用との比較

RV16Kv2 用 C コンパイラと CAHPv3 用 C コンパイラを、出力されるバイナリサイズの観点から比較する。具体的には、テスト用の C プログラムを 10 個用意し、これを`-Oz`オプションを有効にしてコンパイルした際に得られる実行バイナリ群から合計コードサイズを割り出す。

ISA 毎に計測したコードサイズの合計は表 1 のようになる。RV16Kv2 に比べて CAHPv3 は大きくコードサイズを削減できていることが分かる。また CAHPv3 と同じ個数のレジスタを持つ RV32EC[2]や、2 倍のレジスタを持つ RV32IC と比較してもコードサイズが削減されていることが分かる。以上より、コンパイル後のコードサイズを小さくするという CAHPv3 の目的は、一定程度達成されたということができる。

4.4.8 CompCert 改変の試み

CompCert[23] は証明支援系 Coq[25] を用いて記述された C コンパイラで、形式的証明によってコンパイルが正しく行われる（ミスコンパイルしない）ことが証明されている。すなわち、入力された C プログラムとある意味で等価なアセンブリが出力されることが証明されているため、C コンパイラが正しく動作しないために実行

[2]RV32EC は RV32IC からレジスタの個数を半分に減らした ISA である。RV32EC 用バックエンドは LLVM に存在しなかったため RISC-V 用 GCC[26] を使用した。

バイナリに脆弱性が埋め込まれてしまう Thompson hack[24] などが行われていないことを保証できる。

CompCert に我々の ISA のバックエンドを追加することによって、VSP のセキュリティを強化することができると考え、9 月頃に調査を開始した。しかし CompCert は 32bit 及び 64bit のアーキテクチャのみを対象としていることがわかり、これに 16bit のアーキテクチャである RV16Kv2 を追加することは極めて困難であった。そのためこの改変は断念した。

4.5 プロセッサの開発

4.5.1 概要

VSP では、論理ゲートを準同型暗号ゲート（以後 FHE ゲートと記述）に置き換えることで、あらゆる情報を暗号化した状態で取り扱うことを可能にしている（3 節）。そのため、プロセッサの設計自体は既存のものでも流用可能である。

しかし既存のプロセッサ実装は IO や割り込み機能など VSP には必要の無い機能が多く実装されている。そのため、そのまま VSP 用プロセッサとして用いるのは困難である。また、モジュールごとに実行エンジンと組み合わせて動作確認を行うことも、1 つのプロセッサとして完成された実装においては困難が伴う。

そこで本プロジェクトでは、FHE ゲート上でも稼働可能な独自 ISA 用プロセッサを可能な限りシンプルな設計で実装した。

4.5.2 開発の流れ

以下の流れでプロセッサの開発を行った。

1. ISA の策定
   コンパイラ開発担当者が策定した命令群に対して、即値フィールドや opcode の割付を行う。

2. 全体の構造（マイクロアーキテクチャ）の設計
   策定した ISA を元にマイクロアーキテクチャの設計を行う。

3. 各モジュールの実装
   全体設計から機能ごとにモジュールとして実装を行う。

4. 各モジュールのテスト
   実装したモジュールに対して、設計どおりに動くかテストする。

5. モジュールの連結・テスト
   モジュールごとを連結し、プロセッサとしてまとめ上げる。連結する段階でテストも同時に行う。

6. プロセッサ全体のテスト
   命令ごとのテストと C 言語によって記述されたテストプログラムを実行し、正常に動作することを確認する。

4.5.3 開発手法

開発はハードウェアロジックを記述するための言語である Chisel によって行った。Chisel は Scala の DSL であり、設計した回路は Verilog 形式のソースコードに変換して出力することが可能である。

直接 Verilog を使用しての開発も可能ではあるが、Chisel では原則として 1 つのクロックによる回路の駆動を想定しているため、後述する実行エンジンとの兼ね合いが非常に良い。さらに、テストフレームワークが組み込まれているため、モジュールごとのテストを容易に行うことができる。

このような利点から、全てのプロセッサ設計を Chisel で行った。

4.5.4 テスト・デバッグ手法

Chisel のテストフレームワークを利用したモジュールごとのテストと、テストプログラムを実行バイナリにコンパイルしたものを利用するテストを行った。

モジュールごとのテストでは、入力に対して想定した出力を得られることを確認する基本的なテストを行った。テスト失敗時はテストログを参考にモジュールの修正を行い、実装上のミスを排除することを想定している。

テストプログラムでのテストでは、実際にアプリケーションを実行しながら想定した出力が得られるか確認する。モジュールごとのテストとは異なり、プロセッサ全体の設計に問題がないか確認することを想定している。

基本的にはモジュールごとのテストを軽く行い、テストプログラムによるテストを重点的に行った。

テストプログラムでテストに失敗した場合は、レジスタへの書き込みダンプログをシミュレータのログと比較しながら原因の究明を行った。しかし、数万命令を実行した場合レジスタの書き込みログも数万行に及び、人力では追跡不可能であるため、シミュレータと実装のログ比較を自動的に行うスクリプトを用いて一致しないログの抽出を行った。特定の命令実行順などの限られた状況下でのバグを取り除く際は、問題が起きた前後の命令実行列を追いかけることによって問題の究明を行った。

実装とテスト・デバッグの時間比は約 2:8 となり、テスト・デバッグには非常に労力を要した。

4.5.5 rv32k-garnet

VSP 専用プロセッサの第 1 世代実装である。ISA は RV32Kv1 に準拠する。

RISC-V RV32IC の圧縮命令のみを取り出し、必要な命令を追加する手法によって開発されていたが、ISA 自体の機能不足、開発言語 Chisel での開発経験不足、デバッ

グ手法の整備不足などの要因から開発は中止となり、完動には至っていない。中止時の段階では5段マルチサイクルマシン設計であった。

4.5.6 rv16k-amethyst

VSP 専用プロセッサの第2世代実装である。ISA は RV16Kv2 に準拠する。

5段マルチサイクルマシンで構成され、V2TT を用いた VSP の PoC に成功した初の VSP 専用プロセッサである。特徴としては、マルチサイクルマシンとして構成されているため、各ステージごとの依存関係が存在せず、各ステージを切り離した動作確認等が容易である。性能自体は IPC が 0.200 と実用には程遠いものであり、あくまで VSP の PoC 用として用いられた。

4.5.7 rv16k-aquamarine

VSP 専用プロセッサの第3世代実装である。ISA は RV16Kv2 に準拠する。

rv16k-amethyst で PoC を行った後に全体の構造を流用して開発された5段パイプラインマシンで、rv16k-amethyst のコードを流用したマイナーアップデート版である。パイプラインマシンであるため、IPC は 0.794 と高くなっている。

しかし、rv16k-amethyst での PoC 成功後、RV16Kv2 の実行バイナリが他の ISA よりも小さくなっていないという問題が発覚したため、開発メインラインは CAHPv3 によるものへと移行し、rv16k-aquamarine は開発者の技術向上・経験蓄積のために開発された面が強い。

4.5.8 cahp-diamond

VSP 専用プロセッサの第4世代実装である。ISA は CAHPv3 に準拠する。

5段パイプラインマシン（図 15）によって構成され IPC は 0.788 と高く、ほとんどのコードを書き直されたメジャーアップデート版に位置する。

16bit/24bit 命令長という CAHPv3 の特徴のため、ゲート規模・開発期間は増大し、2度の全面再設計・再実装を経て完成した。

16bit/24bit 命令長は互いに偶数倍長の関係にないため、命令を格納する ROM からの読み出しをブロック（チャンク）単位で行うとブロックをまたいだ命令を実行することが出来ない。そのため、内部に簡単なキャッシュ機構を設けることでアライメントをまたぐ命令フェッチを可能にした（図 16）。

ほとんどの開発期間はこの命令フェッチ機構に費やされ、C 言語によって記述されたテストプログラムをパスするまでに前述したように2度の再設計があった。

最終的な VSP リファレンス実装 KVSPv8 では、cahp-diamond が VSP 専用プロセッサとして利用されている。

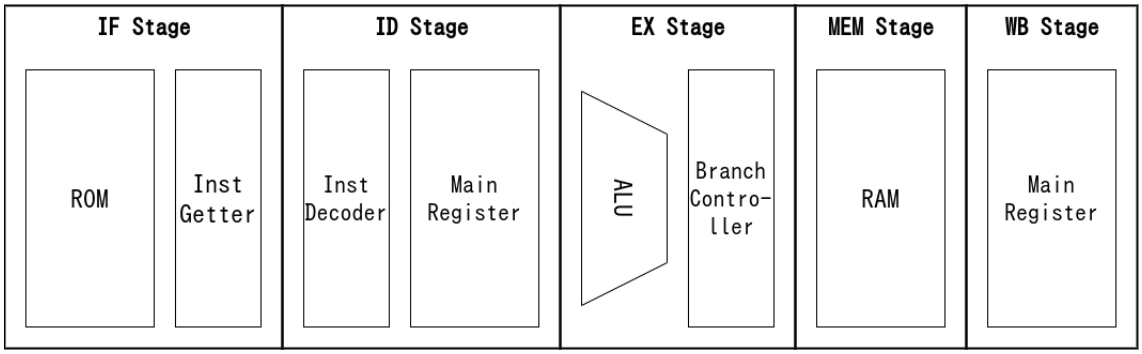

図 15: cahp-diamond マイクロアーキテクチャ

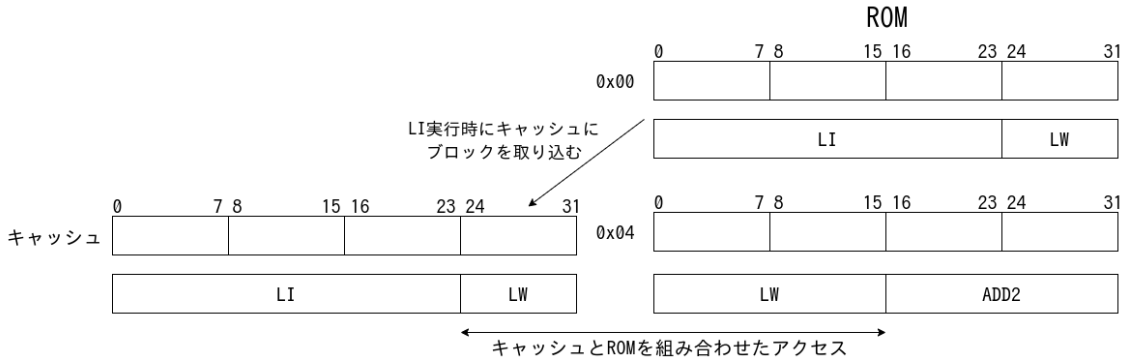

図 16: アライメントを横断する命令のフェッチ機構

4.5.9 cahp-emerald

VSP 専用プロセッサの第 5 世代実装である。ISA は CAHPv3 に準拠する。

5 段パイプラインインラインスーパースカラマシン（図 17）であり、IPC 1.111 という、IPC が 1.0 を超える初の実装である。しかし、ゲート規模が cahp-diamond の倍に膨れ上がり、不採用となった。

当初は、コアのゲート規模が 4000 ゲートと ROM, RAM の 20000 ゲートに比べて小さかったためコアのゲート規模の増大よりも IPC 向上による VSP プロセッサ全体での実行時間短縮の効果があった。しかし、Circuit Bootstrapping の導入によって ROM, RAM の実行時間が短くなり（4.8 項)、コアが占める実行時間の割合が大きくなったため最終的に不採用となった。

構造自体は cahp-diamond とほぼ同一であるが、実装コードはスーパースカラ化に伴い、ほとんど全面的に再実装・整理されたものとなっている。

4.5.10 性能比較

シミュレータ・プロセッサのテストに利用したテストコードでの実行サイクル数を表 2 に示す。シミュレータ（rv16k-sim, cahp-sim）での実行サイクル数は純粋な命令実行数である。

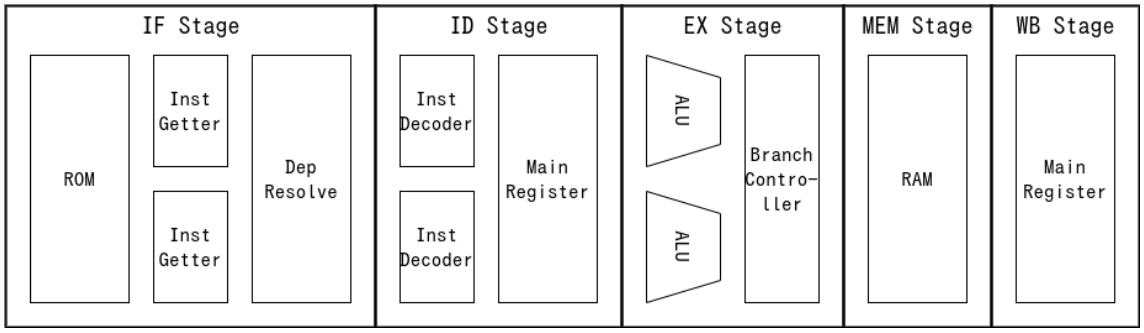

図 17: cahp-emerald マイクロアーキテクチャ

	rv16k-sim	cahp-sim	rv16k-amethyst	rv16k-aquamarine	cahp-diamond	cahp-emerald
0001.bin	5	5	25	29	21	8
0002-fib.bin	272	248	1360	372	482	260
0003-bf.bin	2200	1510	11000	3404	2313	1907
0004-lisp.bin	1075	893	5375	1731	1214	962
0006-editdis.bin	1065	763	5325	1536	1090	867
0007-perceptron.bin	3116	2864	15580	4577	3842	2781
0008-rpn.bin	361	263	1805	484	346	269
0009-gameoflife.bin	13159	9323	65795	14647	10842	7226

表 2: 各テストケースにおける実行サイクル数

表 3 は、RV16Kv2 と CAHPv3 で表 2 のシミュレータ実行サイクル数の合計をとったものである。結果より、RV16Kv2 より CAHPv3 の方が 6000 サイクルほど少なくなり、実行バイナリサイズの縮小が命令実行数の削減にも影響を与えていることが分かる。

ISA	cycle
RV16Kv2	21253
CAHPv3	15869

表 3: ISA ごとの実行サイクル数

表 4 は、RV16Kv2 のプロセッサ設計である rv16k-amethyst と rv16k-aquamarine での実行サイクル数、IPC を比較したものである。rv16k-amethyst はマルチサイクルマシンであるためシミュレータでの 5 倍のサイクル数がかかるが、rv16k-aquamarine はパイプラインマシンであるため必要なサイクル数が大幅に削減され、分岐やジャンプなどによる余分なサイクル数を含めて IPC 0.794 という結果となった。

表 5 は、CAHPv3 のプロセッサ設計である、cahp-diamond と cahp-emerald での実行サイクル数、IPC を比較したものである。cahp-diamond は、rv16k-aquamarine の構造を流用したマルチサイクルマシンであるため、IPC 0.788 とほぼ同じ結果となっている。cahp-emerald は、2 命令同時実行可能なインオーダースーパースカラとなっているため、IPC 1.111 とシミュレータを上回る性能となっている。

表 6 は、各プロセッサ設計のコア単体でのゲート規模である。RV16Kv2 準拠のプロセッサ設計は構造自体がシンプルであるため 4000 ゲート未満の規模で実装で

	cycle	IPC
rv16k-amethyst	106265	0.2
rv16k-aquamarine	26780	0.794

表 4: RV16Kv2 準拠プロセッサ設計の実行サイクル数と IPC

	cycle	IPC
cahp-diamond	20150	0.788
cahp-emerald	14280	1.111

表 5: CAHPv3 準拠プロセッサ設計の実行サイクル数と IPC

きているが、CAHPv3 準拠のプロセッサ設計は、混合命令長対応などの影響によってゲート規模が増大している。さらに、前述したようにスーパースカラを採用した cahp-emerald では、cahp-diamond の 2 倍のゲート規模となった。

	gate
rv16k-amethyst	3869
rv16k-aquamarine	3714
cahp-diamond	4470
cahp-emerald	8930

表 6: 各プロセッサ設計のゲート規模

4.6 準同型暗号ゲート実行エンジンの開発

4.6.1 概要

本節では、Yosys を使用して作成した論理回路を準同型暗号ゲートに置き換えて動作させるための「準同型暗号ゲート実行エンジン」について述べる。

なおこの、準同型暗号上で実行する論理回路を生成し、それを専用の実行エンジンで評価するというアイデアは Cingulata（2.3.5 節）から得たものである。ただし、コードの共通性は全く存在しない。Cingulata はもともとは BFV と呼ばれる種類の準同型暗号のみに対応していたが、本プロジェクト採択後に本プロジェクトで用いている TFHE にも対応した。しかし、Cingulata は論理ゲートの並列実行は現時点では BFV のみで、TFHE には対応していない点で本節で述べる Iyokan と異なる。

4.6.2 V2TT

V2TT（Verilog To TFHE Transpiler）[13] は本プロジェクトの実装上の根幹をなす、論理回路の準同型暗号上でのエミュレーションの PoC 実装である。V2TT は Python3 で書かれており、後述の Iyokan-L1 と同じく、論理合成ソフト Yosys の出力する JSON ネットリストを入力として受け取る。V2TT の時点で組み合わせ回路

だけでなく順序回路も実行可能であり、rv16k-aquamarine までがこの上で動作していた。DFF の入力部分と出力部分を別の変数で保持し、1 クロックごとに入力側を出力側へコピーすることで順序回路を組み合わせ回路として展開して表現する方式は、V2TT の段階で作られ後継の Iyokan-L2 などでも用いられた。

V2TT は論理回路の実行順を静的に解析していた。具体的には、JSON ネットリストをパーズした上で networkx[14] を用いて DAG として表し、各論理ゲートが DFF の出力などの入力ポートから最大で何段の論理ゲートを通って到達するものかを networkx に実装されたベルマン・フォード法によって解析し、同じ段に属するゲートは並列に実行可能であるとみなした。これは、DAG の性質に基づいた解析ではあったが、1 つの段を処理する過程で次の段のゲートの一部の入力が揃ったとしても、それを並列実行可能なものとして扱うことができなかったため、性能の観点で Iyokan に劣るものであった。

また、Iyokan-L2 等と異なり、V2TT 自身は論理ゲートの実行エンジンではなく、論理ゲートの実行を行うための C++プログラムを Jinja2[15] と呼ばれるテンプレートエンジンで出力するものであった。並列実行それ自体は OpenMP[29] に依存していた。

4.6.3 Iyokan-L1・Iyokan-L2

Iyokan-L1・Iyokan-L2 は V2TT による PoC に成功した後に、実用的な準同型暗号ゲート実行基盤として 10 月から 12 月にかけて開発された。Iyokan-L1（Iyokan Layer 1）は、VSP 専用プロセッサを Yosys によって合成した後に得られるネットリスト JSON ファイルを、DAG の構造に変換し JSON として出力する。他方 Iyokan-L2（Iyokan Layer 2）は、Iyokan-L1 が出力した DAG を読み込み、実行順の依存関係を解決しながらこれを並列に CPU や GPU で実行する。

Iyokan-L1 は C#で記述され、主な役割は以下の通りである。

- Yosys 形式の JSON ファイルの読み込み
- ネットリストから DAG への変換
- 実行順の優先度付与
- コアへの ROM 回路の連結
- ROM, RAM 回路に対するアドレス, データビット数のタグ付け

Yosys 形式のファイルは図 18 のようなネットリストになっているため、図 19 のような DAG へと変換している。また、同時に回路の連結や実行順の優先度付、タグ付けを行っている。

Iyokan-L2 は C++で記述され、主な役割は以下の通りである。

- Iyokan-L1 形式の JSON ファイルの読み込み
- TFHE（CPU）を利用した FHE ゲートの並列評価

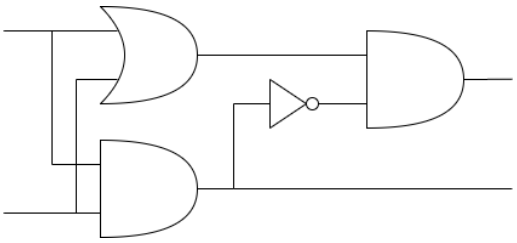

図 18: ネットリスト形式の回路

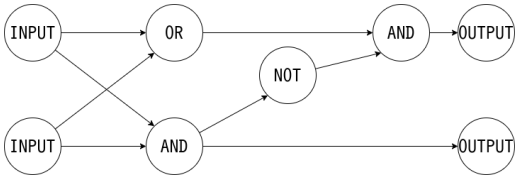

図 19: DAG に変換された回路

- cuFHE（GPU）を利用して FHE ゲートの並列評価
- ROM, RAM に対する値設定・取得

ここで TFHE 用の Iyokan-L2 と cuFHE 用のそれとでは内部構造が異なる。

TFHE 用の Iyokan-L2 は、図 21 にあるようにワーカースレッドが能動的に評価ゲートを取得する。具体的には以下のアルゴリズムで動く。

1. DAG 形式の回路情報を読み込む。
2. 入力端子および DFF のノードを評価したものとして依存解決を行う。
3. 通知を受けたノードで評価可能条件を満たしたものを評価待ちキューに入れる。
4. 評価待ちキューからノードを取得する。
5. ノードを評価する。
6. 評価したノードについて依存解決を行う。
7. 全ノードを評価した場合終了する。そうでなければ 3. のステップに戻る。

ここで依存解決とは、「そのノードが辺で接続される下流のノードに対して評価済みの通知を行うこと」を意味する。図 20 にその様子を示す。また評価可能条件とは、「ノードに対する入力の数と通知の数が等しい場合、評価可能とする」という条件である。

上記アルゴリズムで 1, 2 のステップは初期化時に行い、3, 4, 5, 6, 7 のステップは各ワーカースレッドが行う。エンジンは全ノードの評価が終了した時点で停止し、停止までの一連の処理が 1 クロックの処理となる。

翻って cuFHE 用 Iyokan-L2 では、図 22 にあるようにワーカースレッドとは別にスケジューラースレッドが存在し、このスレッドがそれぞれのワーカースレッドに評価ゲートを割り当てている。

具体的には以下のアルゴリズムで動く。

1. DAG 形式の回路情報を読み込む。
2. 入力端子および DFF のノードを評価したものとして下流のノードに対して通知する。

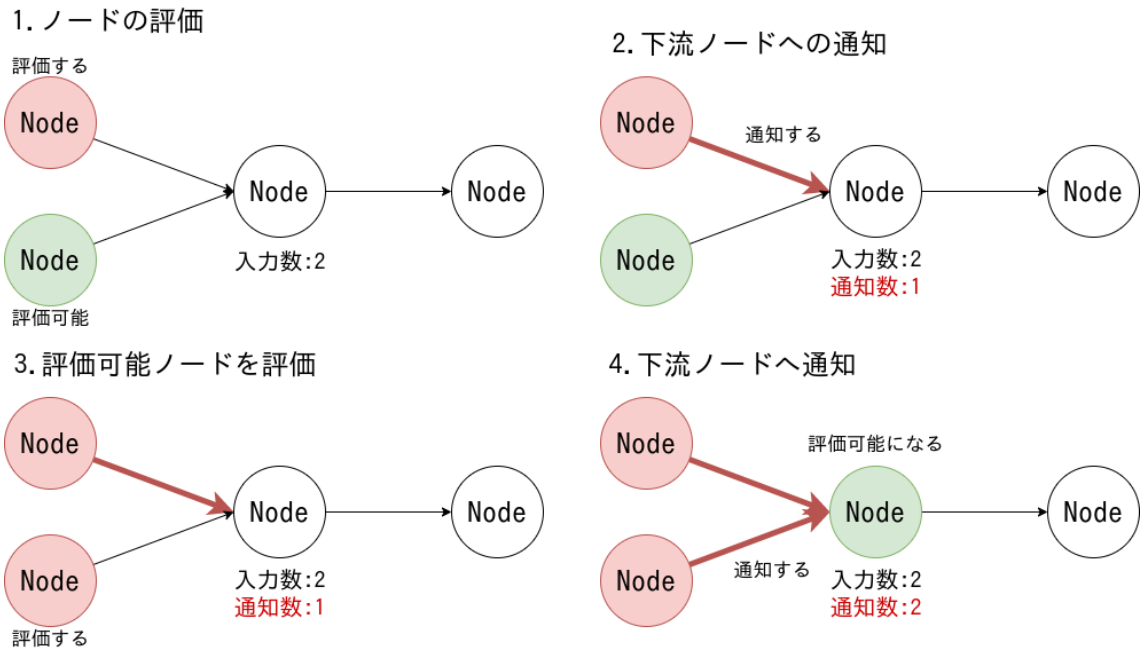

図 20: 依存解決と評価の流れ

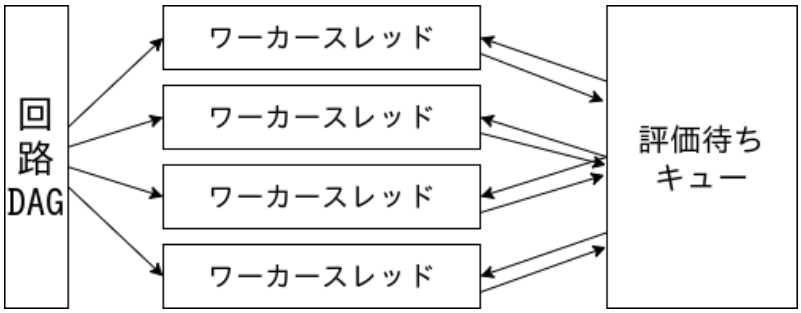

図 21: TFHE 版 Iyokan-L2 のスケジューリング

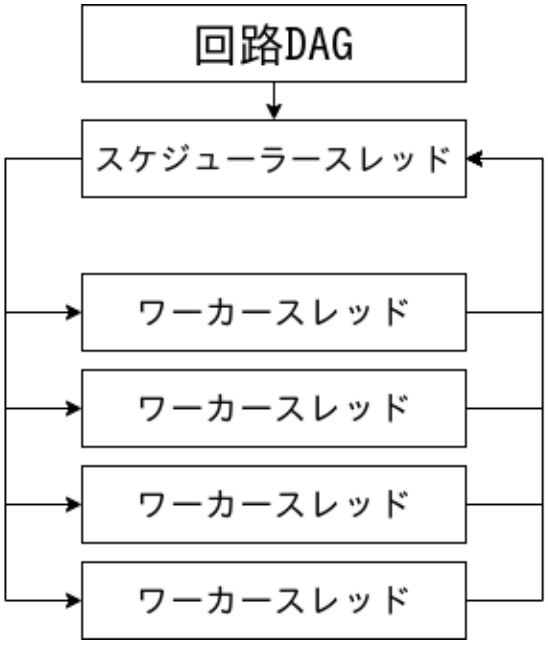

図 22: cuFHE 版 Iyokan-L2 のスケジューリング

3. 通知を受けたノードで評価可能条件を満たしたものをワーカースレッドに割り付ける。
4. ワーカースレッドでノードの評価を行う。
5. ワーカースレッドでの評価が終了したノードについて依存解決を行う。
6. 全ノードを評価した場合終了する。そうでなければ 3. のステップに戻る。

Iyokan-L1・Iyokan-L2 の最大の特徴は、ゲート評価の並列性を最大限に活用することが出来る点にある。TFHE・cuFHE のゲート実行エンジンとして、これは世界初のものである。

前述した依存解決により 1 つのノードを評価した結果、複数のノードが評価可能になる場合がある。これらのノードは互いに依存関係が存在しないため、並列に実行することができる。実際の回路では数百ゲートを並列に実行することが出来る場合が多いため、現状では CPU のコア数に比例して動作速度も増すことになり、スケーラビリティが担保される。

4.6.4 Iyokan

より柔軟な準同型暗号ゲートの実行を行うため、1 月から 2 月にかけて Iyokan を開発した [34]。名前が重複しているためわかりにくいが、Iyokan は Iyokan-L2 の後継にあたるものであり、4.6.3 節にて Iyokan-L1 が担当した回路解析についてはそのまま Iyokan-L1 を使用している。

Iyokan は Iyokan-L2 が担当した役割を担うと同時に、フルスクラッチによってより洗練されたコードベースを実現した。内部構造には Iyokan-L2 の cuFHE 版のエン

ジンを採用し、スケジューラとワーカが別々のスレッドとして動作する。これによってCPUとGPUを組み合わせた柔軟なゲート処理が可能になっている。

またIyokan-L2が持たなかった機能として外部モジュールの接続機能を実装し、複数の回路を複合的に組み合わせて利用することを可能にした。これによって、TFHE上でCircuit Bootstrappingを使用して構成されたROM・RAM（4.8項）をcahp-diamond等のCPUコアと組み合わせて使用でき、結果としてKVSPの実行時間短縮に寄与した。

4.7 kvspコマンドの開発

4.7.1 概要

Cコンパイラや Iyokan、準同型暗号ライブラリなど我々が作成したツールを容易に扱うためのインタフェースとして12月から2月にかけて`kvsp`コマンド[18]を開発した。`kvsp`コマンドはユーザからの入力を受け取り、ユーザの代わりに必要なツールを起動する。また`kvsp`コマンドをビルドすれば必要な全てのツールがビルドされるため、各々のツールを別々に導入する必要がなく、ユーザのインストールが容易になっている。

なおKVSPと表記した場合、このインタフェースとして機能するコマンドを指す場合と、Cコンパイラ・Iyokanなどを含めた全てのツールチェインを総合したVSPのリファレンス実装を指す場合とがある。前者は`kvsp`と小文字で表記され、後者はKVSPと大文字で表記されることが多い。この項では前者の`kvsp`コマンドについて記述している。

4.7.2 機能・構造

`kvsp`コマンドはGo言語[19]によって実装した。内部では`os/exec`[20]パッケージを用いて、外部コマンドを起動している。

`kvsp`コマンドの機能は独立したサブコマンドとして実装されている。具体的には次のようなものを実装した。ただし記述は本成果報告書執筆時点での最新版（v8[21]）による。

`kvsp debug`
: 実行バイナリを平文のまま機能シミュレーションによって実行する。内部ではcahp-simを起動する。

`kvsp emu`
: 実行バイナリを平文のまま回路エミュレーションによって実行する。内部ではIyokanの平文モードを起動する。またIyokan-L1・cahp-diamondの実行結果を使用している。

`kvsp enc`
: 実行バイナリを準同型暗号によって暗号化し、KVSP request packet を作成する。内部では `kvsp-packet` を起動する。

`kvsp run`
: `kvsp enc` によって生成された KVSP request packet を、暗号化したまま実行し、結果を KVSP result packet として出力する。内部では Iyokan・TFHEpp・cuFHE などを起動する。また Iyokan-L1・cahp-diamond の実行結果を使用している。

`kvsp dec`
: KVSP result packet を復号し、結果を表示する。内部では `kvsp-packet` を起動する。

なおここで `kvsp-packet` は Iyokan と同じリポジトリにて開発したツールで、実行バイナリのデータを暗号化して KVSP request packet を作成するほか、KVSP result packet を復号して結果を取り出すために使用できる。

`kvsp` コマンドのビルドの際には、`kvsp` リポジトリに含まれる `Makefile`[22] を使用する。これを実行すると `build/` ディレクトリ以下に kvsp コマンドがビルドされる。このとき `build/` 以外のディレクトリには変更が加わらないため、リポジトリ全体を見通しよく保つことができる。`-j` オプションにも対応しており、指定すると必要に応じて並列にビルドが実行される。

4.8 Circuit Bootstrapping による ROM・RAM の高速化

これは VSP のために開発された一種のアルゴリズムであり、実体としては Iyokan の機能として取り込まれている。

TFHE では論理ゲート 1 つの処理に 10ms 程度の時間がかかる。ROM・RAM を論理回路で実装する場合マルチプレクサと呼ばれる論理ゲートが大量に必要となるため、ナイーブな実装では ROM・RAM の処理に多大な時間を要することになる。本プロジェクトでは、ROM・RAM について CMUX と呼ばれる特殊な演算を論理ゲートの代わりに用いることで高速化を図った。ここで言う特殊の意味は、通常の MUX と比べた場合、セレクタに入る暗号文がそれ以外の入力や出力の暗号文と種類が異なることを指している。

5 開発成果の特徴

本プロジェクトにおいては、標準ライブラリを必要としない範囲の C プログラムを、プログラムを秘匿したまま実行することができる、ソフトウェアのみで実現可能な実行基盤の PoC を与え、これに「バーチャルセキュアプラットフォーム」と名付けた。

本プロジェクトで作成した準同型暗号上の CPU エミュレータは、NVIDIA Tesla V100[30] を 1 基搭載したマシンで 1 クロックあたり約 5.5 秒、8 基では約 1.5 秒で動作する。また Amazon Web Service の c5.metal インスタンス [31] では 1 クロックあたり約 2.5 秒で動作する。したがって数百クロック程度のプログラムであれば現実的な時間で動作させることが可能である。

6 今後の課題、展望

6.1 今後の課題

大きく分けて 4 つある。まず第一に、本プロジェクトの成果は学術的にも新規なものであるため、論文を執筆する予定である。本プロジェクトの学術的な新規性の最も大きな部分は、準同型暗号による CPU エミュレーションによってプログラムの秘匿を達成した点にある。2.3 項でみたように、Garbled Circuit で CPU エミュレーションをしたものはあるが、準同型暗号で行ったものはなく、準同型暗号を用いた秘匿計算の試みはプログラムの秘匿までは実現していない。また、副次的には本プロジェクトではクラウドサーバへの計算処理移譲を想定していたが、VSP の仕組みとしてはクラウドサーバを何らかのサービス提供者に置き換えると、データを秘匿したままでの 2 パーティの秘密計算とすることができる。このような秘密計算自体は既存の仕組みでも可能ではあるが、KVSP は C 言語で書かれたプログラムを利用できる点で新規性がある。

第二に、準同型暗号ゲート実行エンジンにおけるスケジューラの改善が挙げられる。スケジューラは現状シングルスレッドで動作しているため、特にマルチ GPU 下ではこれがボトルネックとなってしまっている。スケジューラをマルチスレッドにするなどしてより高速化することにより、より多くのゲートを並列に実行できる可能性がある。

第三に、KVSP に使用している準同型暗号 TFHE の改善が挙げられる。すでに論文として発表されているもののうち、TFHE のアルゴリズムに改善を与えるもの [ZYL+18][CIM18] がいくつかあるため、これを KVSP に適用することで KVSP の高速化に寄与すると考えられる。

最後に、本プロジェクトは「プログラムの秘匿を可能にする実行基盤」の PoC を作ることが目的であり、その成果物の上に他の人々が新たな成果を付け加えていくことが望ましいため、VSP や KVSP に関するドキュメントも整備したいと考えている。

6.2 展望

5 節で述べたように、我々が作った実装は黎明期のコンピュータの性能に近いものである。しかし、黎明期のコンピュータ同様に性能向上の余地が大きいものでもある。黎明期のコンピュータの性能が向上した大きな要因の一つであったトランジ

スタの集積化は、VSP にとってはCPU やGPU コアのような演算器の集積化に対応する。昨今ムーアの法則の終焉としてトランジスタ集積の限界が言われている一方、演算器の集積化はチップレット化などによってまだ余地がある（AMD の Ryzen が良い例であろう）と考えられている。そうした技術の進化を VSP は受けうるものである。

また、完全準同型暗号は 2009 年に始まった分野であると言え [Gen09]、今後準同型暗号自体のさらなる理論的な高速化も期待される。

理論的な観点で言えば、今回提案した VSP の仕組みは、プログラム提供者とデータ提供者が同一であるような場合に限っている。そこで Multi-Key、つまり複数の秘密鍵を利用可能にすることでプログラム提供者とデータ提供者を別にできるようにすることが考えられる。このアイデアは実装という観点では、現状用いている TFHE を Multi-Key に対応した実装である MK-TFHE に置き換えて多少のインタフェースの変更のみで対応可能である一方で、どのようなユースケースで安全性が確保できるかを理論的に調べる必要がある。

他にも実装の改善として、ハードウェアアクセラレーションの採用が挙げられる。TFHE の論理ゲート演算は条件分岐がほぼ存在せず、整数演算のみで完結できるため、FPGA などで専用のアクセラレータを構成することによる高速化が期待される。

VSP は、ある意味では量子コンピュータに近いものである。量子コンピュータが物理的な量子の性質を用いて「並列性」という古典的コンピュータにない性質を実現するのに対し、VSP は数学的な準同型暗号の性質を用いて「プログラムの秘匿」を実現する。この「プログラムの秘匿」が無二の価値であって人々を惹きつける限りにおいて、VSP 性能向上の試みが続くものと考えている。

黎明期のコンピュータが出来上がってから今日まで数十年の歳月が経過している。同様に我々は、VSP は数十年後に普及しうると考えている。

7　実施計画書内容との相違点

7.1　準同型暗号ライブラリの開発

当初の予定では準同型暗号ライブラリは既存のものをそのまま流用する予定であった。しかし、CPU 設計やコンパイラとの兼ね合いで V2TT の開発が止まるタイミングがあったこと、TFHE の暗号としての原理を理解していなければ安全性を保証できないこと、TFHE の原論文者実装にない Circuit Bootstrapping が実装できれば高速化が見込めたことなどから、pyFHE と TFHEpp をフルスクラッチで実装することとなった。

7.2　速度目標

当初の速度目標は 60Hz であったが、この数字自体は黎明期のコンピュータとして名前が上げられる ABC にならって掲げられたものであって、その達成可能性は十

分に保証されたものではなかった。また、本プロジェクトの新規性は「プログラムを秘匿したまま実行できる仕組みの実装」そのものにあり、60Hz を達成することにはない。そのため、60Hz という速度目標よりもインタフェースを整えるなどの「プログラムを秘匿したまま実行できる仕組みの実装」としての完成度、再現性を優先することとした。

7.3 動的解析を行う準同型暗号ゲート実行エンジンの開発

当初プロセッサの回路は、そのネットリストを静的に解析し、準同型暗号ライブラリを用いた C++コードにトランスパイルされた上で実行されると想定していた。実際本プロジェクト最初の準同型暗号ゲート実行エンジン V2TT はこの手法で実装された（4.6.2 節）。しかしマシン性能を全て使い切ることができないなど V2TT の性能は芳しくなく、我々は静的解析には限界があると結論づけた。

そこで V2TT の後継である Iyokan-L2 や Iyokan では動的解析を採用した。すなわち予めゲートの評価順を決めてしまうのではなく、終了したゲートから次に実行できるゲートを実行時に動的に割り出すことで、マシンパワーをフルに使うことに成功した。

8 開発分担

開発の分担を表 7 に掲げる。大まかに言って、準同型暗号の理論に関わるものは松岡が、コンパイラに関わるものは伴野が、プロセッサに関わるものは松本が担当した。

9 秘匿ノウハウの指定

本プロジェクトの成果物は開発初期の実用に耐えないコンパイラを除いて全てオープンソースとなっており、最も厳しいライセンスでも、商用利用を許す Apache-2.0 となっている。これは、本プロジェクトのような数学的に安全性を保証することを意図した仕組みにおいて秘匿ノウハウを開発者が有していた場合、開発者への信用がセキュリティの保証に必要になってしまうためである。

10 成長の自己分析

10.1 松岡

本プロジェクト開始時点では準同型暗号への理解はほぼ皆無に等しい状態であったが、本プロジェクトを通して TFHE という一方式に限るとは言えフルスクラッチで実装可能なレベルまで理解を深めることができたのは客観的にも明らかな成長であると言える。

開発物		開発者
準同型暗号ライブラリ	pyFHE	松岡・伴野
	TFHEpp	松岡
	cuFHE	松岡・松本
C コンパイラ	RV16Kv1 用 RV32Kv2 用 CAHPv3 用	伴野
プロセッサ	rv32k-garnet rv16k-amethyst rv16k-aquamarine cahp-diamond cahp-emerald	松本
ISA	RV16Kv1 RV32Kv2 CAHPv3	松岡・伴野・松本
準同型暗号ゲート実行エンジン	V2TT	松岡
	Iyokan-L1・Iyokan-L2	松本
	Iyokan	伴野・松本
VSP インタフェース	`kvsp` コマンド	伴野

表 7: 開発分担

プログラミングスキルという点ではプロジェクト開始時点では C と Python をメインとしていたが、TFHEpp の開発を通して C++のテンプレートなどの機能が使えるようになり、また cuFHE の開発によって CUDA の基礎を会得することができた。

10.2 伴野

LLVM バックエンドの開発によって、LLVM core や最新のコンパイラ技術についての理解を深めることができた。また独自 ISA を開発したことにより、ISA にどのような命令が必要か、逆にどのような命令は不要なのか、といった現代では得難い知見を得ることができた。

技術的な観点以外では、小規模なチーム開発を実際に経験でき、メンバー間のコミュニケーションやコードの共有などについて学ぶことができた。自分が理論よりも実装担当に向いていることを再確認し、自分の得意な分野をどのようにチーム内で活かせばよいかを認識した。

10.3 松本

技術的な面においては、プロセッサの開発や実行エンジンの開発を通して初期設計の重要性に関する知見を得た。プロジェクト開始直後は見切り発車で実装に入り、

後になって実装上の困難や大量のバグ・コーナーケースに苦しめられることになった。初期の段階で細部まで設計をすることにより実装が見通しやすい構造になり、バグも減るという経験を得ることができた。また、それに付随して目的に適したデータ構造を用いることで設計・実装がより良いものになるという経験も得た。

技術的な面以外では、チームでの開発を通じて方針や目標などの意思疎通やコード共有など、これまで経験したことがない開発手法であり、プロジェクトの期間を通じて幾度も困難があったものの最後まで完遂することができた。その経験から個人開発とチーム開発で求められるノウハウが全く異なり、コミュニケーションを取り続けることの重要性を知見として得られた。

11 その他

本プロジェクトの開発において、さくらインターネット株式会社様より高火力コンピューティングを無償提供いただいたため、御礼申し上げる。KVSP は原因の断定はできないもののコンシューマ向けの GPU では動作せず、当該サービスの利用は有意に開発を促進したと考えている。

12 付録

12.1 用語説明

ABC
: アタナソフ&ベリー・コンピュータの略で、世界最初のコンピュータと言われる。

Chisel
: HDL（ハードウェア記述言語）の一種で、オープンソースで開発されている [36]。モダンな記法が可能であるという特徴を持つ。

CUDA
: 主に NVIDIA 製の GPU を制御するために用いるコンピューティングプラットフォーム。

DAG
: Directed Acyclic Graph の略で、日本語では有向非巡回グラフと訳される。グラフ理論における閉路を含まないグラフのうち、辺に向きのあるものを指す。

DFF
: Delay Flip Flop の略で、論理回路におけるフリップフロップの一種である。クロックの立ち上がりと同期して入力値を保存し、次のクロックまでその値を出力する。

DSL

Domain-Specific Language の略で、日本語ではドメイン固有言語と訳される。ある目的に特化したようなプログラミング言語のことを指し、通常チューリング完全でない。

IPC

Instruction Per Clock の略で、1 クロックあたりに実行できる命令の数を表す。

ISA

Instruction Set Architecture の略で、日本語では命令セットアーキテクチャと訳される。主にプロセッサに対する命令群のことを指す。

RAM

Random Access Memory の略で、主にプログラムが使用するメモリ領域のことを指す。

ROM

Read Only Memory の略で、主にプログラムが格納されるメモリ領域のことを指す。

TFHE（Torus Fully Homomorphic Encryption）

本プロジェクトが用いている完全準同型暗号である。暗号文に対して論理演算が行える。

Yosys

論理合成を行うためのツールで、オープンソースで公開されている [35]。

n オペランド

プロセッサに対する命令のうち、主に n 個の引数をとる命令のことを n オペランドという。n は 2 や 3 であることが一般的である。

$n$bit アーキテクチャ

命令セットのうち、レジスタやアドレス空間が $n$bit で表現されるものを $n$bit アーキテクチャと呼ぶ。n は $8, 16, 32, 64$ のいずれかであることが一般的である。

高位合成

高位合成とは C/C++などの高級言語に拡張を施したものか、そのサブセット、あるいは専用の DSL などで記述したアルゴリズムを論理回路による表現へ変換する仕組みのことを言う。クロックのタイミングなどを気にせずにアルゴリズムを書くことを可能にする。

準同型暗号

準同型暗号とは、少なくとも 1 つ以上の平文から平文への 2 項以上の算法について、ある暗号文から暗号文への算法があって、暗号化と暗号文から暗号文への算法と復号の合成写像が平文から平文への算法と等しくなるような暗号のことである。

スーパースカラ

図 23 にあるように、複数命令を同時にパイプラインに流して同時実行する。命令間でレジスタ等の依存関係があると実行できないため理論性能を出すことが難しい。

パイプラインマシン

図 24 にあるように、全ステージが常に利用されるように命令を送り込み実行する。実行効率は良いが、分岐命令のストール処理などが必要となる。

マルチサイクルマシン

図 25 にあるように、1 命令を複数ステージに分けて複数サイクルかけて実行する。実行効率は悪いが、実装が非常に簡単である。

論理合成

Chisel や Verilog などのハードウェア記述言語による抽象的な回路の動作に関する記述を、実際の回路の構成要素である論理ゲートなどのネットリストによる表現へ変換する仕組みのことを言う。

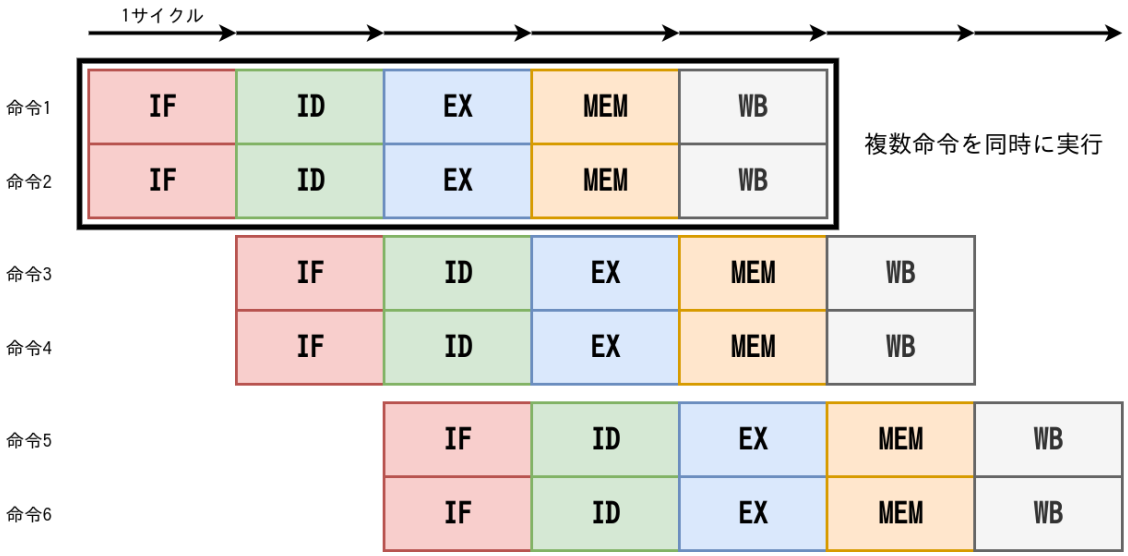

図 23: スーパースカラでの命令実行

12.2 関連 Web サイト

- `https://github.com/virtualsecureplatform/kvsp`
  我々が開発した KVSP が公開されている GitHub リポジトリである。

- `https://anqou.net/poc/2019/10/18/post-3106/`
  VSP という概念について平易に解説した日本語文書である。

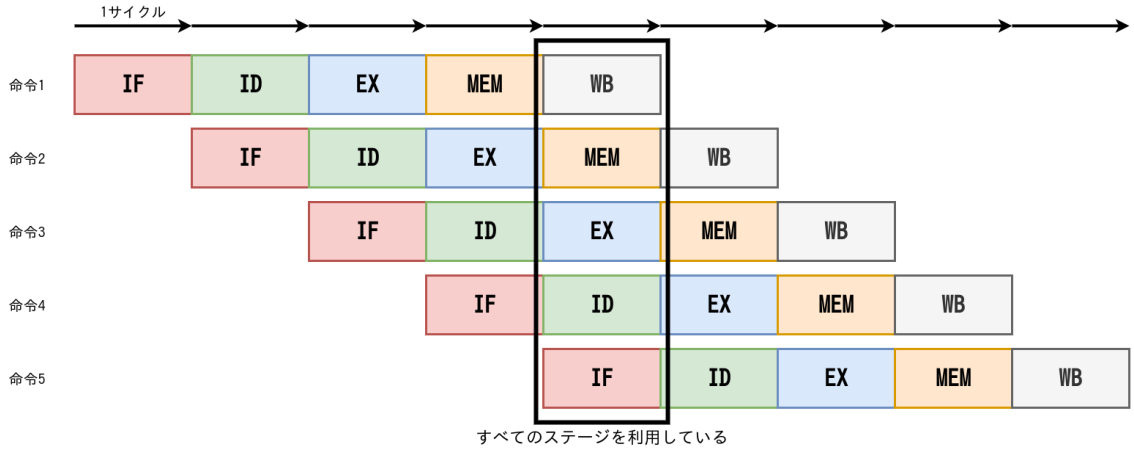

図 24: パイプラインマシンでの命令実行

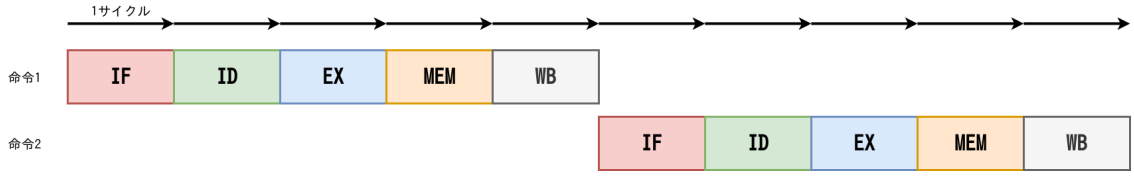

図 25: マルチサイクルマシンでの命令実行

13 参考文献

[1] https://github.com/joeltg/brainfreeze

[2] https://github.com/f-prime/arcanevm

[3] https://qiita.com/Cliffford/items/2f155f40a1c3eec288cf

[4] https://github.com/CEA-LIST/Cingulata

[5] https://github.com/virtualsecureplatform/pyFHE

[6] https://github.com/tfhe/tfhe

[7] https://github.com/tfhe/experimental-tfhe/tree/master/circuit-bootstrapping

[8] https://github.com/virtualsecureplatform/TFHEpp

[9] https://github.com/virtualsecureplatform/cuFHE

[10] https://github.com/vernamlab/cuFHE

[11] https://github.com/virtualsecureplatform/Parameter-Selection

[12] https://bitbucket.org/malb/lwe-estimator/src/master/

[13] https://github.com/virtualsecureplatform/V2TT

[14] https://networkx.github.io/

[15] https://palletsprojects.com/p/jinja/

[16] https://riscv.org/2016/04/risc-v-offers-simple-modular-isa/

[17] https://ja.wikipedia.org/wiki/%E3%82%A2%E3%82%BF%E3%83%8A%E3%82%BD%E3%83%95%26%E3%83%99%E3%83%AA%E3%83%BC%E3%83%BB%E3%82%B3%E3%83%B3%E3%83%94%E3%83%A5%E3%83%BC%E3%82%BF

[18] https://github.com/virtualsecureplatform/kvsp

[19] https://golang.org/

[20] https://golang.org/pkg/os/exec/

[21] https://github.com/virtualsecureplatform/kvsp/releases/tag/v8

[22] https://www.gnu.org/software/make/

[23] http://compcert.inria.fr/

[24] https://www.cs.cmu.edu/~rdriley/487/papers/Thompson_1984_ReflectionsonTrustingTrust.pdf

[25] https://coq.inria.fr/

[26] https://github.com/riscv/riscv-gnu-toolchain

[27] http://llvm.org/

[28] https://riscv.org/

[29] https://www.openmp.org/

[30] https://www.nvidia.com/ja-jp/data-center/v100/

[31] https://aws.amazon.com/jp/ec2/instance-types/c5/

[32] https://github.com/virtualsecureplatform/llvm-rv16k

[33] https://github.com/virtualsecureplatform/llvm-cahp

[34] https://github.com/virtualsecureplatform/Iyokan

[35] http://www.clifford.at/yosys/

[36] https://www.chisel-lang.org/

[ACC+18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[CIM18] Sergiu Carpov, Malika Izabachne, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. Cryptology ePrint Archive, Report 2018/622, 2018. `https://eprint.iacr.org/2018/622`.

[CPS18] Eric Crockett, Chris Peikert, and Chad Sharp. Alchemy: A language and compiler for homomorphic encryption made easy. pages 1020–1037, 10 2018.

[CS15] Ayantika Chatterjee and Indranil Sengupta. Furisc: Fhe encrypted urisc design. *IACR Cryptology ePrint Archive*, 2015:699, 2015.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169178, New York, NY, USA, 2009. Association for Computing Machinery.

[IMP18] F. Irena, D. Murphy, and S. Parameswaran. Cryptoblaze: A partially homomorphic processor with multiple instructions and non-deterministic encryption support. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 702–708, Jan 2018.

[SRH+19] Ebrahim M. Songhori, M. Sadegh Riazi, Siam U. Hussain, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. Arm2gc: Succinct garbled processor for secure computation. *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019.

[SSZ+16] E. M. Songhori, T. Schneider, S. Zeitouni, A. Sadeghi, G. Dessouky, and F. Koushanfar. Garbledcpu: A mips processor for secure computation in hardware. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[TM14] N. G. Tsoutsos and M. Maniatakos. Heroic: Homomorphically encrypted one instruction computer. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[ZYL+18] T. Zhou, X. Yang, L. Liu, W. Zhang, and N. Li. Faster bootstrapping with multiple addends. *IEEE Access*, 6:49868–49876, 2018.