暗号コード自動最適化フレームワーク CryptOpt の拡張

木村 悠人[1,a)]　Chitchanok Chuengsatiansup[2]　佐古 和恵[1]

概要：暗号ソフトウェアを高速化するために最適化は必要である．汎用コンパイラは一定の最適化を行えるものの，ドメイン固有の最適化を十分に活用できず，専門家が手作業で書いたアセンブリほど高性能なコードを生成できない場合がある．これに対し CryptOpt(Kuepper ほか, PLDI 2023) は，セキュリティ要件に起因し，暗号コードに共通して現れる実装パターンを最適化過程に取り込み，専門家が手書きで生成するアセンブリと同程度の性能を達成した．しかし，CryptOpt には入力フォーマットが限定されているという制約がある．本稿では，CryptOpt を拡張し，二つの主要な側面で強化した．第一に，LLVM 中間表現（LLVM IR）を入力として受け取り，CryptOpt の最適化エンジンが扱える形式へ変換するコード変換手法を設計・実装した．LLVM IR は多数のプログラミング言語で広く使用されているため，本拡張により CryptOpt は多様な言語のコードを最適化できるようになる．第二に，既存の CryptOpt の性能評価手法を改良し，測定バイアスを低減した．ベンチマークの結果，LLVM IR を経由して CryptOpt が最適化したコードは，汎用コンパイラが生成したコードに対し最大 33.11% の性能向上を示した．

キーワード：暗号，最適化，コード生成

On Enhancing the Automatic Cryptographic Code Optimization Framework CryptOpt

Haruto Kimura[1,a)]　Chitchanok Chuengsatiansup[2]　Kazue Sako[1]

Abstract: High-speed cryptographic software is greatly desirable. While off-the-shelf compilers can perform decent optimization, they may fail to benefit from domain-specific optimization, thus not producing as performant code as hand-optimized assembly written by experts in the field. To this end, CryptOpt (Kuepper et al., PLDI 2023) tackles this challenge by integrating coding styles, which often appear in cryptographic code due to security requirements, as part of the optimization. As a result, code produced by CryptOpt achieves competitive performance as hand-optimized code. Despite an impressive outcome, CryptOpt has a limitation on its restrictive input format. This paper enhances the automatic cryptographic code optimization CryptOpt by presenting two contributions. First of all, we design and implement code transformations, which, upon receiving LLVM intermediate representation (LLVM-IR) input, convert it into a format that is applicable to CryptOpt's optimizer engine. Since LLVM IR is widely and commonly used by a large number of programming languages, this means that this enhancement enables CryptOpt to optimize code from a broad range. Secondly, we improve the performance evaluation method to reduce potential bias in the measurement. According to our benchmarking results, our LLVM IR CryptOpt-optimized code achieves performance improvements of up to 33.11% over off-the-shelf compiler-generated code.

Keywords: Cryptography, Optimization, Code Generation

[1] 早稲田大学
Waseda University
[2] The University of Klagenfurt

[a)] haruto1999@akane.waseda.jp

1. はじめに

暗号は機微なデータを保護するための仕組みである．暗号ソフトウェアが広範かつ集中的に利用される現代のデジタル環境において，高効率な実装への需要は尽きることがない．暗号プリミティブの多様性，複数の安全性水準の選択肢は，それだけで実装者に困難を課す．加えて，これらの実装は最終的に実機上で動作するため，異なるハードウェアアーキテクチャやデバイス側の制約が，安全かつ高効率な暗号ソフトウェアの実装をさらに難しくする．

可搬性に対処する一つの方法は，C や Rust のような高水準言語で暗号ソフトウェアを実装し，ターゲットプラットフォーム上で動作可能な機械語へコンパイルすることである．この方法は，対象プラットフォームの変化（例えば新世代機の導入）に伴って再実装する必要が生じる問題を回避できる. しかし，そのようなコードの性能は，専門家が直接アセンブリで実装したコードに比べて遅いことが多い [1], [2], [3], [4]．しかも，少数の専門家の手作業による最適化に依拠する開発体制は，暗号ソフトウェア需要の増大に伴ってスケールしない．加えて，手作業の実装はバグを導入する可能性がある．

実装が意図どおりに動作し期待どおりの結果を生成しているかを検証する観点では，さまざまなツールと技法が利用可能である [5], [6], [7]．同様に，暗号用コードの実装および最適化のスケーラビリティに対処する提案も多数存在し，形式的検証と統合されたものもある．

その一つとして CryptOpt [8] がある．これは最適化過程において革新的なアルゴリズムを用いる自動暗号化最適化ツールである．CryptOpt [8] は効率的実装，タイミングサイドチャネル耐性，および実装検証という三つの問題を同時に解決することに成功している．もっとも，CryptOpt の利用可能性には制約があり，比較的限定的な入力空間を要求する．具体的には CryptOpt の入力形式は Fiat Cryptography [9]（関数型プログラムをコードへ翻訳し両者の同値性を証明する枠組み）を前提として設計されており，主に Fiat 由来の入力を受け付ける．したがって，Fiat 形式ではないコードを CryptOpt で最適化することはできない．Fiat Cryptography 以外を最適化できることを示す試みはあるものの，非 Fiat 実装の最適化は，筆者の知る限り，Bitcoin Core の secp256k1 に対する C 実装を対象にしたもののみである [8].

この制約を克服するため，本研究は CryptOpt をより広範な実装に適用可能にする枠組みを導入する．具体的には LLVM ブリッジと名付けた経路を設計・実装し，Rust や C の実装を LLVM 中間表現（LLVM IR）へコンパイルしたのち，それを CryptOpt に適した入力形式である JSON 形式へ変換する．さらに，提案方式が出力したコードと汎用コンパイラが生成したコードの性能を比較した．13 実装・計 25 関数を本 LLVM ブリッジ経由で CryptOpt に入力して評価した結果，CryptOpt 生成コードは汎用コンパイラ（*-O3*）を多くのケースで上回り，最大＋ 33.11% の性能向上を達成した．

本稿の貢献

本稿の貢献は以下のとおりである．

- **汎用 LLVM ブリッジの実装**．*clang* 及び *rustc* が出力する LLVM 中間表現（LLVM IR）を，CryptOpt が期待する JSON 形式へ変換するフロントエンドを実装した．
- **公平な性能評価と改善**．CryptOpt の共有オブジェクト（.so）に対するベンチマークをやめ，アセンブリ対アセンブリの直接比較に置き換え，フォーマットの差異による測定ノイズを低減した．この公平な評価の下，C，Rust 双方における 7 つの実用プリミティブ（Curve25519, P448, secp256k1, Poly1305, BLS12 に加えライブラリ実装 2 種）ならびに OpenSSL 実装に対して，汎用コンパイラが最高レベル最適化（*-O3*）を適用して生成したコードに対し，コンパイラ出力比で最大 33.11%の高速化を達成した．なお，この数値は評価法の変更による見かけの改善ではなく，CryptOpt が生成した命令列（instruction sequence；本稿では LLVM IR/アセンブリの *instruction* を以下 “命令” と呼ぶ）そのものの優位に起因する．

2. 背景

本節では，本研究に関連する先行研究と概念として Fiat Cryptography と CryptOpt そして LLVM について概説する．

2.1 Fiat Cryptography

Fiat Cryptography [9] は，暗号アルゴリズムの数学的仕様から暗号コードを自動生成するためのフレームワークである．具体的に，入力として Coq 定理証明支援系内の関数型プログラムとして記述された有限体演算の高水準形式仕様が定められている．Fiat Cryptography は，これらの仕様から低水準コードを合成すると同時に，生成コードが元の数学的定義を正しく実装していることを証明する．この過程では，演算列を記述する中間表現である Fiat IR を生成するのが通例である．Fiat IR からは，C，Rust，Zig などのコードを生成することができる．

2.2 CryptOpt

CryptOpt [8] は, 暗号プリミティブ向けに高性能な x86-64 アセンブリコードを生成するコード生成ライブラリであり，想定されるプロセッサアーキテクチャに合わせて最適化さ

れる．また，Fiat Cryptography フレームワークを活用しているため，生成されたコードの正当性保証も与えられる．

図 1 は Fiat Cryptography と CryptOpt の連携を概観する．CryptOpt は Fiat Cryptography が生成する低水準中間表現 Fiat IR を受け取る．そこから CryptOpt は二つの中間表現を生成する．(1) 性能評価に用いる Fiat C （Fiat Cryptography の C 実装）と，(2) CryptOpt が最適化の過程で用いる Fiat JSON である．Fiat JSON は Fiat IR の全情報—演算，入出力のビット幅，境界（bounds）—を保持する JSON 構造である．この JSON 入力は前処理を経て CryptOpt の内部モデル（CryptOpt IR ）に変換され，演算列，順序，変異の決定が保持される．

また，CryptOpt には，C で実装された Bitcoin Core の secp256k1 を CryptOpt の入力形式に変換するための“Bitcoin Core ブリッジ”を実装している．図 1 の青矢印にあるように，まず Bitcoin Core は *Clang*（LLVM プロジェクトの C/C++ 用コンパイラフロントエンド）を用いて LLVM IR にコンパイルされる．その後，LLVM IR の各種演算などの関連情報を抽出し，CryptOpt に適した中間 JSON 形式へ変換する．この JSON は Fiat JSON と同趣旨で本質的な演算を記述するが，Coq による形式証明を経ずに LLVM IR から直接得られる点が Fiat Cryptography を入力とした場合と異なる．この中間 JSON はさらに CryptOpt の内部表現へと翻訳され，最適化の入力となる．

最適化戦略として，CryptOpt はランダム探索を用いる．あらかじめ定義されたヒューリスティックや抽象的性能モデルに依存するのではなく，正当な初期実装に対して命令差し替えや順序入れ替えといった変異を反復的に適用する．同時に，元の実装と変異後の実装の実行時間を対象ハードウェア上で直接計測し，速い方を残す．これにより，有限の計算資源内で機械語実装の探索空間を効率的に探索でき，ハードウェア世代やマイクロアーキテクチャの違いにも自動的に対応する．

正当性の保証については，CryptOpt が利用する性能計測ツール MeasureSuite [10] に含まれるプログラム同値性チェッカーを用いる．MeasureSuite は複数の入力プログラムを受け取り，それらの性能を比較するとともに，プログラムの出力がランダムな入力に対し一致するかを確認する．CryptOpt の場合には，性能比較のためのベースライン実装（例．Fiat C ）からコンパイルした共有オブジェクト（`.so`）と，CryptOpt が生成した最適化済みアセンブリ（`.asm`）を MeasureSuite の入力とする．

性能ベンチマーク時には，CryptOpt はベースラインとの比較用に，参照実装である Fiat C や Bitcoin Core からコンパイルした共有オブジェクト（`.so`）を用いる．一方で CryptOpt の出力はアセンブリ（`.asm`）であるため，共有オブジェクト（`.so`）のベースラインと比較するために，MeasureSuite に統合された AssemblyLine [11] を用いて CryptOpt 生成の アセンブリ（`.asm`）をオブジェクト（`.o`）へ変換する．計測対象の関数（乗算や二乗算）について，AssemblyLine が生成したオブジェクト（`.o`）内の該当関数と，ベースラインの共有オブジェクト（`.so`）内の対応関数の実行時間を比較し，性能を評価する．

2.3 LLVM

LLVM プロジェクトは，モジュール式で再利用可能なコンパイラおよびツールチェーン技術の集合体である．本研究に関連する中核要素は LLVM 中間表現（LLVM IR）である．LLVM IR は言語非依存の低水準中間表現であり，コンパイラのフロントエンドが出力する共通対象であると同時に，最適化パスやコード生成バックエンドへの入力として設計されている [12].

LLVM IR は静的単一代入（Static Single Assignment; SSA）形式を用いる．すなわち各レジスタ（仮想レジスタ）は定義されてから使用されるまで，値の代入は必ず一度だけされるため，多様な解析や変換が簡素化される．C/C++ に対する *Clang* や Rust に対する *rustc* を含む各種高水準言語のコンパイラは，ソースコードから LLVM IR を生成できる．その後，LLVM はこの共通 IR 上で動作する最適化パス群を提供し，最終的にバックエンドが最適化済み IR を x86-64 など多様な対象アーキテクチャ向けの機械語へと翻訳する．

3. 本提案のアプローチ

本研究の第一の目的は，Fiat Cryptography [9] 由来の入力仕様に限らず，より広範な暗号コードを最適化できるように CryptOpt [8] の能力を拡張することにある．CryptOpt は主に Fiat の中間表現（Fiat IR ）を入力として用いるため，我々の主要課題は Fiat IR 以外の入力を CryptOpt に受理させることである．この目的のために，我々は変換系列を設計・実装し，最終的に CryptOpt と互換な形式へ到達させる．

我々は，とりわけ各種高水準言語で広く用いられている LLVM 中間表現（LLVM IR）を CryptOpt の入力として受け付けられるようにすることに焦点を当てる．フレームワークの概略は図 2 に示す．以降，図 2 に示した一連の変換手順を “パイプライン” と呼ぶ．本研究の中核的な困難は，汎用で低水準な LLVM IR と，有限体演算に特化した CryptOpt の内部モデルとの間にある意味的ギャップを橋渡しする点にある．

なお，原著の CryptOpt 論文 [8] では，Fiat Cryptography に由来しない入力として Bitcoin Core の secp256k1 曲線のベンチマークも含まれている．しかし，CryptOpt が扱えた非 Fiat 入力はこの一例にとどまる．我々の調査によれば，LLVM IR のようなより汎用的な入力を直接扱う能力には顕著な制約が存在する．

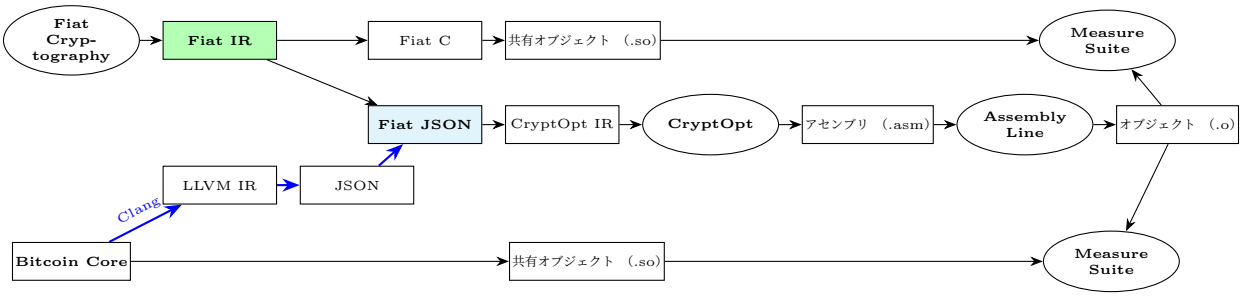

図 1　オリジナルの CryptOpt ワークフローの概観

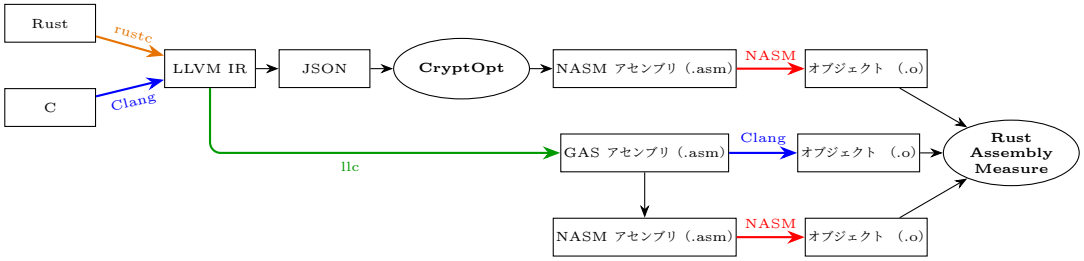

図 2　本研究の概観

3.1　フォーマット変換

入力として LLVM IR を受け付けることを目標とするため，Rust や C といった高水準言語で実装された暗号コードは，まず LLVM IR へ変換する必要がある．この段階は汎用コンパイラを用いればよい．具体的には，Rust 実装には *rustc*，C 実装には *Clang* を用いる．

LLVM IR を取得した後の工程で，しかも最大の難所は，この低水準の LLVM IR を CryptOpt が扱える JSON 形式へ変換することだ．そのために，LLVM IR を解析し，正規表現を用いて各命令の構成要素を抽出する．抽出した要素は `name`，`operation`，`modifiers`，`datatype`，`arguments` の五つに分類する．`name` は変数名，`operation` は演算種別，`modifiers` は最適化や意味上のヒント，`datatype` は符号付き整数の型，`arguments` は被演算子を格納する．

CryptOpt が受理可能な JSON を構成するには，JSON に含まれる各操作が CryptOpt が理解・生成できる命令の集合でなければならない．この要件を満たすため，既存の CryptOpt 命令だけでは不十分となる場合があることを踏まえ（たとえば AUCurves (BLS12) [13] を入力とする場合），CryptOpt の対応命令を拡張した．

この JSON は共通の中間表現として機能し，CryptOpt の Bitcoin Core ブリッジ に接続して，最終的に CryptOpt が受理する中間表現へと変換できる（詳細は appendix A.1）．

3.2　境界チェックと panic ブロック

Rust のような安全志向言語から得られる LLVM IR には，境界チェックや panic ブロック（境界違反などのエラー時に例外処理へ分岐して異常終了するための基本ブロック）が含まれる場合がある．これらは実装中の潜在的なエラーに対処するために重要だが，入力が妥当でエラーが発生しないことを前提とする限り，コードの機能自体には直接影響しない．したがって，本研究のフォーマット変換では，LLVM IR 中の境界チェック分岐と panic ブロックを抽出対象から除外し，後続の変換・最適化でも扱わない．

なお，Fiat を入力源とする元の CryptOpt [8] においても，ソースレベルでは境界チェックを含んでいても，コンパイラ最適化が不要とみなした定義を除去する傾向にあるため，LLVM IR 段階では当該チェックが失われることがある．要するに，LLVM IR に見られる panic ブロックの存在は，ソース言語とコンパイラ設定に起因するアーティファクトである．

3.3　変数名の再命名とアドレス規約の整合

LLVM IR は上述の通り SSA 形式であるが，変数名は必ずしも連番形式でない．Bitcoin Core ブリッジは，`x0`，`x1`，`x2` のような連番の変数名から恩恵を受ける．一方で，Rust 実装から生成される LLVM IR はこの連番規約に必ずしも従わず，`%0.i`，`%0.i14`，$\%_{\mathtt{x}}\mathtt{1}$，$\%_{\mathtt{0}}\mathtt{.i2}$，`%11` のように直感的でない不規則な名前が現れることがある．そこで本パイプラインでは，後段の処理を簡潔にするため，変数名を整理して連番形式へと再命名する．

加えて，LLVM IR と CryptOpt のあいだでアドレス表現の規約不一致も解消する．LLVM IR はオフセットをバイト単位で数えるのに対し，CryptOpt はワード（64 ビット）単位で扱う．この差異を吸収するため，LLVM IR 側のバイトオフセットをワードインデックスへ変換し，バイト基準のメモリ意味論とワード基準のアドレッシングの齟齬を整合させる．

3.4 対応命令集合の拡張

CryptOpt は Fiat IR を主な入力としている．その結果，素の LLVM IR の各種命令は原則としてサポート外である．

本研究のパイプラインでは，LLVM IR を CryptOpt 用の JSON 形式へ変換する過程（section 3.1）で，CryptOpt が未対応の命令である `sext`（符号拡張）と `select` が現れる．この問題に対処するため，これら未対応命令を CryptOpt 互換のプリミティブへ写像する．具体的には，それぞれを CryptOpt がサポートする `cmovznz`（ゼロ/非ゼロ条件付き移動）へ写像する設計とした．以下に写像手順の詳細を述べる．

3.4.1 `select` から `cmovznz` への写像

LLVM IR の `select` は，分岐を用いずに真偽条件に基づいて二つの値のいずれかを選ぶ命令である．CryptOpt の `cmovznz`（ゼロ/非ゼロ条件付き移動）は同等の機能を提供するが，明示的なブールではなく整数フラグに基づいて動作する．`select` の例は次のとおりである．

```
<result> = select i1 <cond>, i64 <val1>, i64 <val2>
```

すなわち，条件 `<cond>` が真（値 1）なら `<result>` に `<val1>` を代入し，そうでなければ `<val2>` を代入する．CryptOpt 側への写像では，`<cond>` を 0 または 1 の整数フラグとして解釈する．`cmovznz` はフラグがゼロか非ゼロかに応じて二つの値から選択する．等価な変換は次のとおりである．

```
<result> = cmovznz i1 <cond>, i64 <val2>, i64 <val1>
```

ここで `<cond>` が真（非ゼロ）なら `<val1>` が選ばれ，偽（ゼロ）なら `<val2>` が選ばれる．`cmovznz` は「フラグが非ゼロなら第 2 オペランド」を選ぶ命令であり，`<val1>` と `<val2>` の位置を入れ替えるため，`select` と同値の処理となる．LLVM の `i1` ブールは 1 ビット整数（0/1）として安全に扱えること，および `cmovznz` が分岐なしに同じ二者択一を行うことから，本変換はプログラムの論理を保ったまま安全に適用できる．

3.4.2 `sext` から `cmovznz` への写像

LLVM IR の `sext` は値のビット幅を増やしつつ符号を保存する命令である．たとえば 1 ビットのブール（`i1`）を 64 ビット整数（`i64`）へ拡張する場合，入力が真（1）なら出力は `-1`（`0xFFFFFFFFFFFFFFFF`）になり，偽（0）なら出力は `0`（`0x0000000000000000`）になる．`sext` の例は次のとおりである．

```
<result> = sext i1 <val3> to i64
```

これは「`<val3>` が真なら上位ビットを 1 で埋めた結果（すなわち `-1`），偽なら `0`」という意味である．この機能は CryptOpt の `cmovznz` でも実現できる．`<val3>` を条件フラグとして用い，ゼロか非ゼロかで二つの定数を選べばよい．すなわち次のとおりに表現する．

```
<result> = cmovznz i1 <val3>, i64 0, i64 -1
```

`<val3>` が真（非ゼロ）なら `-1`（`0xFFFFFFFFFFFFFFFF`），偽（ゼロ）なら `0` が選ばれる．`sext` は符号を保った拡張を行う命令であり，`cmovznz` は条件に応じて二つの値を分岐なしで選択できるため，本写像は語彙の違いによる等価変換である．また，`i64` における `-1` の 2 の補数表現が `0xFFFFFFFFFFFFFFFF` である点とも一致する．さらに `cmovznz` は分岐を導入しないため，`sext` と同じタイミング特性を保つことができる．

4. 評価

本節では本提案方式が出力するコードの性能評価を行った結果について述べる．まず，評価に使った実装群の概要を示す．ついで，何をどのように比較したかという評価方法の詳細を述べる．最後に，実験環境を説明し，結果を提示する．

4.1 対象実装

本研究は 13 実装・計 25 関数（各実装の乗算・二乗算．ただし BLS12 は乗算のみ）を対象とする．概要は以下のとおりである．

- **Fiat Rust/Fiat C**：Fiat Cryptography が生成する Rust/C 実装 [14], [15] を本研究の LLVM ブリッジによる最適化対象とする．これらは Fiat JSON 前提の CryptOpt に取り込みやすく，Fiat Cryptography の形式的検証をそのまま活用し，Fiat Rust/Fiat C と CryptOpt 出力の同値性検査により正当性を継承できる．
- **AUCurves (BLS12)**：Fiat Cryptography と bedrock2 を基盤とする Rust 実装 [9], [13], [16] である．本稿では乗算を評価対象とする．
- **Curve25519-dalek/Rust-EC-secp256k1**：純 Rust 実装 [17], [18] を非 Fiat 系の代表として扱う．
- **OpenSSL**：Curve25519 は可搬 C 版と手書き最適化アセンブリの二系統があり，両者を比較対象とする．P448（素数 $p = 2^{448} - 2^{224} - 1$ 上の有限体を指し，Curve448 で用いられる）は可搬 C 版のみのため，CryptOpt 生成アセンブリと C コンパイル版を比較する [19].

4.2 CryptOpt によるアセンブリ生成と本研究のベースライン生成

既存の CryptOpt 研究 [8] では，性能比較の基準（ベースライン）として，Fiat C を *GCC*/*Clang* でコンパイルして得た共有オブジェクト（`.so`）を用い，これに対して CryptOpt が生成した NASM アセンブリ（`.asm`）の実行速度を比較している（図 1）．しかし，ここには方法論上の問

題がある．CryptOpt の出力は NASM アセンブリ（.asm），計測の際は AssemblyLine によりオブジェクト（.o）にアセンブルされる一方で，Fiat C のベースラインは共有オブジェクト（.so）である．オブジェクト（.o）は典型的に静的リンクされるのに対し，共有オブジェクト（.so）は動的にロードされるため，この形式の不一致は厳密な同条件比較を誤らせる可能性がある．

この問題を解決するため，出力形式をオブジェクト（.o）に統一し，測定条件（リンカ，入力，CPU 設定）を揃えたうえで，コンパイラ生成コードと関数本体の命令列のみが異なる比較を実現する計測パイプラインを構築した．本パイプラインは，Rust または C を LLVM IR へコンパイル後，*llc -O3* で LLVM IR を GAS（GNU Assembler）アセンブリ（.asm）へ変換するところから始まる．これにより最上位最適化を適用したアセンブリを得る．CryptOpt の出力は NASM アセンブリ（.asm）であるため，コンパイラ生成の GAS アセンブリ（.asm）を NASM 形式へ変換するトランスレータも実装した．オペランド順序，レジスタ接頭辞を必要に応じて書き換えるが，機能的意味は変更しない．変換の詳細は appendix A.2 に示す．

最後に，自作の **Rust Assembly Measure** により，各アセンブリ（.asm）を *NASM*，*Clang* でアセンブルしオブジェクト（.o）を生成する．そして完全に同一条件下でそれらのオブジェクト（.o）をベンチマークする．

なお，CryptOpt で NASM アセンブリ（.asm）を生成する際のパラメータは原著 [8] と同一に設定している．

4.3 実験環境

実験は第 13 世代 Intel Core i7-1360P（Raptor Lake-P，8 コア/16 スレッド）上の Ubuntu 22.04.4 LTS で実施した．計測環境の設定について以下の小節で述べる．

4.3.1 ノイズ低減

計測は単一の物理コアにピン留めし，SMT/Turbo Boost/動的周波数制御を無効化して周波数を固定，ASLR を停止し主要デーモンを停止・高頻度 IRQ を迂回させた．これにより，OS スケジューリング・割り込み・周波数変動・アドレス変動といった環境起因による計測値のばらつきを最小化した．

4.3.2 ウォームアップ

コールドキャッシュの影響を避けるため，三段階のウォームアップを行う．まず関数本体を 20 回連続で実行してコードとデータをキャッシュへ載せる．次に一度だけ較正計測を行い，およそ 10,000 サイクルとなるバッチサイズを決定する．最後に，この較正済みバッチサイズで 5 回実行して状態を安定させる．

4.3.3 サイクル計測法

計時は適切なメモリバリア（mfence）とともに RDTSC を用い，バッチサイズは CryptOpt の方法論 [8] に従い動的に較正した．CryptOpt 生成物は Intel 表記（NASM 構文）であり *NASM* によりアセンブルする．一方，ベースラインのアセンブリは AT&T 表記（GAS のデフォルト）で得られるため，自動変換で Intel 表記へ機械的に変換し（詳細は appendix A.2），同一の *NASM* 呼び出しでアセンブルする．これより，被験要因は命令列そのものに限定され，アセンブラ差分は排除される．

4.3.4 統計的安定性のための反復計測

CryptOpt [8] のアルゴリズム単体では稀に周辺値から大きく乖離したサイクル数（外れ値/スパイク）が混入する．このような値を除去するため，CryptOpt のアルゴリズムを 100 回繰り返し，最後にそれらの中央値を最終性能値として採用し，結果を安定化させている．

4.4 結果

本節では，LLVM ブリッジを介して最適化した CryptOpt 生成 x86-64 アセンブリと，二種類のコンパイラ生成アセンブリとの性能比較を 表 1 に示す．指標は，ベースラインの中央値に対する CryptOpt 生成物の性能改善率である．

第 1 列「実装」は最適化対象の実装を示す．第 2 列「言語」は実装言語（Rust か C）を示す．第 3・4 列（および第 5・6 列）「GAS」「NASM」は，乗算（および二乗算）関数について，LLVM ブリッジ経由の CryptOpt 生成コードと汎用コンパイラ生成コード（ベースライン）を，それぞれ GAS 形式と NASM 形式で速度を比較した結果である．表中の数値は，対応するベースラインに対する速度向上率を表す．正の値は本手法が速いこと，負の値は低下を意味する．

また「—」は本パイプラインで未対応であることを示す．AUCurves (BLS12) 実装で観測された大幅な低下を含む性能差の要因については section 5 で議論する．

さらに，表 1 の OpenSSL の部分では，Curve25519 実装について，汎用コンパイラ生成コード（vs.OTS）および専門家による手書きアセンブリ（vs.Hand）との比較を示す．P448 には OpenSSL から手書きアセンブリが提供されていないため，比較は Curve25519 のみである．Curve25519 の乗算では，OpenSSL の専門家による手書きアセンブリ（vs.Hand）に対しても優位であり，GAS/NASM でそれぞれ 18.70%，21.77% 高速である．

5. 議論

本節では，性能低下の潜在的要因ついて議論する．

5.1 レジスタ割り当てと BLS12 の性能

表 1 に示した性能比較によれば，我々の LLVM ブリッジを経て CryptOpt が最適化したコードは，AUCurves (BLS12) を例外として，通常のコンパイラ生成コードをほとんど同等かそれを上回った．生成されたアセンブリコードを詳細

表 1　LLVM ブリッジ経由で CryptOpt により最適化したコードとコンパイラ生成コードの性能比較（速度向上率，%）

		乗算 (%)		二乗算 (%)	
実装	言語	GAS	NASM	GAS	NASM
Fiat-NIST-P448	Rust	+32.66%	+33.11%	+0.19%	+0.34%
Fiat-Curve25519	Rust	+26.23%	+22.97%	+13.93%	+13.76%
Curve25519-Dalek	Rust	+21.78%	+22.38%	-0.03%	+0.35%
Fiat-Secp256k1	Rust	+7.56%	+13.96%	+3.12%	+3.43%
Fiat-Poly1305	Rust	+2.51%	+2.66%	+1.90%	+2.34%
EC-Secp256k1	Rust	+2.13%	+5.79%	-7.24%	-9.83%
BLS12	Rust	-45.64%	-36.12%	—	—
Fiat-Curve25519	C	+28.29%	+25.49%	+9.70%	+9.25%
Fiat-NIST-P448	C	+27.86%	+27.84%	-10.55%	-10.79%
Fiat-Secp256k1	C	+5.23%	+5.23%	+2.40%	+2.38%
Fiat-Poly1305	C	+2.51%	+2.66%	+1.90%	+2.34%
OpenSSL					
Curve25519(vs.OTS)	C	+18.86%	+21.47%	+7.72%	+6.84%
Curve25519(vs.Hand)	C	+18.70%	+21.77%	+11.45%	+11.61%
P448	C	+0.34%	+0.43%	-0.41%	+0.93%

注: —は本ワークフローがサポートしていない実装を示す．正の値は CryptOpt の方が高速であることを，負の値はベースラインの方が高速であることを示す．

に分析したところ，特に AUCurves (BLS12) では，レジスタを空けるためレジスタからメモリへ値を退避させる mov 命令，いわゆるレジスタスピルが多数存在し，汎用コンパイラが生成するコードと比べてコードサイズが約 2 倍に膨らんでいることを見出した．

現代的なコンパイラは詳細な生存区間解析を行い，干渉グラフを構築して変数の生存範囲を正確に捉え，スピルを最小化する．これに対して，我々の手法では LLVM IR から JSON への変換の段階で正確な生存範囲情報を失う．CryptOpt はこれを依存グラフのみに基づいて近似しながら扱うが，ある変数の生存範囲が不明確な場合，CryptOpt は後で必要になる可能性を潰さないよう，値を上書きしない保険としてレジスタをメモリにスピルする方を選ぶ．したがって，BLS12 のように規模が大きく複雑なルーチンでは，性能を大きく劣化させる．

5.2　LLVM IR に起因する制約

本研究のパイプラインでは，Rust/C 及び CryptOpt で実装されているが，LLVM プロジェクトが直接サポートしていない演算，具体的には addcarryx と subborrow が問題となる．addcarryx については，LLVM IR 上で 2 つの add128 へと分解することが可能である．2 つの add128 へ分解した後は，あらかじめ用意したテンプレートに従ってアセンブリへと変換する．しかしこの過程では，下位 64 ビット加算で生じたキャリービットを上位 64 ビット加算へ正しく伝搬させる処理を適切に扱えない．その結果，CryptOpt は addcarryx を正しく処理できず，これらの演算を含む Rust/C 実装は本研究のパイプラインを用いて CryptOpt で最適化することができない．

一方で subborrow については，128 ビット減算へ置き換える手法を用いることができない．理由は CryptOpt が符号なし 128 ビットの sub 演算をサポートしていないためであり，subborrow を sub128 に分解して CryptOpt で扱うことができない．

ここでの制約をうける暗号プリミティブの具体例としては，NIST 曲線群（P224，P256，P384，P521）などが挙げられる．

6.　まとめ

本論文は，暗号コードの自動最適化をより広い範囲へ拡張するフレームワークを提示する．CryptOpt エンジンを統合し，二つの主要な側面で強化した．

第一に，LLVM IR から CryptOpt の内部表現への変換を開発した．このブリッジにより，CryptOpt は既存の C および Rust 実装の広範な集合を最適化可能となり，従来の Fiat Cryptography への依存から解放される．

第二に，性能評価手法を改良した．新しい手法は，より正確で統計的に頑健な比較を提供し，外部ツールへの依存を排して，信頼性の高い結果を得る．

今後の課題として，汎用コンパイラに依存しない本ブリッジ向けのコンパイル経路を整備，CryptOpt がサポートする演算集合を拡張し，現代の暗号に必要なプリミティブを網羅することである．

参考文献

[1] Tung Chou. Sandy2x: new Curve25519 speed records. In *SAC*, pages 145–160, 2015.

[2] Tung Chou. QcBits: constant-time small-key code-based cryptography. In *CHES*, volume 9813, pages 280–300, 2016.

[3] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Peter Schwabe. Kummer strikes back: New DH speed records. In *ASIACRYPT*, pages 317–337, 2014.

[4] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. Faster multiplication in $Z_{2^m}[x]$ on Cortex-M4 to speed up NIST PQC candidates. In *ACNS*, pages 281–301, 2019.

[5] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, pages 125–128, 2013.

[6] Yu-Fang Chen, Chang-Hong Hsu, Hsin-Hung Lin, Peter Schwabe, Ming-Hsien Tsai, Bow-Yaw Wang, Bo-Yin Yang, and Shang-Yi Yang. Verifying Curve25519 software. In *CCS*, pages 299–309, 2014.

[7] Andy Polyakov, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verifying arithmetic assembly programs in cryptographic primitives (invited talk). In *CONCUR*, volume 118 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[8] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam

Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom. CryptOpt: verified compilation with randomized program search for cryptographic primitives. *Proc. ACM Program. Lang.*, 7(PLDI):1268–1292, 2023.

[9] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE, 2019.

[10] 0xADE1A1DE. Measuresuite v2.2.2. `https://github.com/0xADE1A1DE/MeasureSuite`, 2023. Accessed: 2025-02-01.

[11] 0xADE1A1DE. Assemblyline v1.3.2. `https://github.com/0xADE1A1DE/AssemblyLine`, 2023. Accessed: 2025-02-01.

[12] llvm-admin team. The LLVM Compiler Infrastructure Project. `https://llvm.org/`, 2025. Accessed: 2025-05-07.

[13] AU-COBRA. Aucurves - high assurance cryptography by means of code synthesis. `https://github.com/AU-COBRA/AUCurves`, 2022. Accessed: 2025-02-01.

[14] The Fiat Cryptography Contributors. Fiat-Crypto rust bindings. `https://github.com/mit-plv/fiat-crypto/tree/main/rust`, 2023. commit `Commit 4e7dde9` (includes Curve25519, P448, Secp256k1, Poly1305).

[15] The Fiat Cryptography Contributors. Fiat-Crypto c bindings. `https://github.com/mit-plv/fiat-crypto/tree/main/c`, 2023. commit `Commit c79cf60` (includes Curve25519, P448, Secp256k1, Poly1305).

[16] mit plv. A work-in-progress language and compiler for verified low-level programming. `https://github.com/mit-plv/bedrock2`, 2018. Accessed: 2025-04-05.

[17] dalek-cryptography Developers. Dalek elliptic curve cryptography. `https://github.com/dalek-cryptography/curve25519-dalek`, 2023. Accessed: 2025-04-05.

[18] RustCrypto Developers. Rustcrypto. `https://github.com/RustCrypto`, 2020. Accessed: 2025-04-05.

[19] OpenSSL: Cryptography and SSL/TLS Toolkit. `https://github.com/openssl/openssl`. Commit snapshot accessed 2 July 2025. The evaluation uses the files `crypto/ec/curve25519.c` (portable C) and `crypto/ec/asm/x25519_x86_64.pl` (hand-optimised x86−64).

付　　録

A.1 JSON から Fiat JSON 類似 IR へ

第二段階では，Fiat JSON 類似 IR のための前処理が，フィルタ済みの JSON を形式的な Fiat JSON 類似 IR へ変換する．主眼は，定時間的なデータフローを保ったままメモリアクセスを抽象化する点にある．

A.1.1 ポインタ参照の配列アクセスへの解決

このスクリプトは低水準の `load`, `store`, `getelementptr` を取り込み，高水準の配列アクセスへ解決する．たとえば「引数配列の 5 番目の要素へのポインタを取り，その値をロードする」という意味の LLVM 命令列は，Fiat JSON 類似 IR の単一命令 `xN = arg1[4]` に変換される．これらの配列インデックスは元の getelementptr 命令中の定数整数から導かれるため，メモリアクセスの定時間性は保持される．この変換は新たなデータ依存を導入しない．ポインタ中心の表現を配列中心の表現へ置き換えるだけである．

A.1.2 算術の一対一対応

算術および論理の LLVM 命令（`add`, `mul`, `and`, `lshr` など）は，Fiat JSON 類似 IR の同等な意味を持つ命令へと一対一で写像する．これは直接変換であり，新たな可変時間動作を導入しない．

定時間性を侵すおそれのある構成要素をすべて除去し，意味保持的な抽象化を施すことで，最終的な Fiat JSON 類似 IR は定時間アルゴリズムの忠実な高水準表現となることをこのパイプラインは保証する．Fiat JSON 類似 IR を生成した後は，既存の CryptOpt [8] パイプラインがこれを CryptOpt IR へ変換し，それが CryptOpt への最終的な IR 入力となる．

A.2 ベースラインとアセンブリ形式

CryptOpt の出力とコンパイラのベースラインを公正に比較するために，コンパイラの GAS 構文を NASM 形式へ体系的に変換するスクリプトを実装する必要がある．このスクリプトは次のような主要な相違を処理する．

- **オペランド順序と接頭辞:** オペランドの入れ替え（AT&T `src,dst` → Intel `dst,src`），レジスタ接頭辞「`%`」の除去（`%rax` → `rax`），即値接頭辞「`$`」の除去（`$42` → `42`）
- **ニーモニックとアドレッシング:** 命令名の正規化（`movabs` → `mov`），メモリオペランドの変換（`8(%rax)` → `QWORD[rax+8]`）
- **アセンブラ疑似命令:** 主要ディレクティブの写像（`.globl` → `global`, `.align N` → `ALIGN N`）、GNU 固有メタデータの除去（`.type`, `.size`, `# %bb.*`）

変換の正当性は差分テストにより検証した．まず LLVM IR から *llc* を用いて基準となるオブジェクト（`.o`）を直接生成する．次に，この基準オブジェクト（`.o`），変換後の NASM オブジェクト（`.o`），および CryptOpt が生成したオブジェクト（`.o`）を，同一のランダム入力を多数回にわたり与えて実行し，それらの出力が完全に一致することを検証する．不一致は一切観測されず，GAS と NASM のベースライン間で見られる計時差（通常 5% 未満）は，アセンブラ固有のレイアウト最適化に起因すると考えられる．