ChaCha20-Poly1305の実装性能調査

株式会社レピダム

2017年10月

目 次

第 1 章　エグゼクティブサマリー　3

第 2 章　ChaCha20-Poly1305 について　4

2.1 概要 4
2.2 歴史的背景 5
2.3 アルゴリズムの構造 5
2.3.1 ChaCha20 5
2.3.2 Poly1305 7
2.3.3 ChaCha20-Poly1305 8

第 3 章　ChaCha20-Poly1305 の採用状況　11

3.1 標準化動向 11
3.1.1 IETF 11
3.1.2 その他の標準化団体 14
3.2 OSS コミュニティ動向 14

第 4 章　ChaCha20-Poly1305 の実装性能評価に関する既存文献調査　17

4.1 論文調査 17
4.2 ベンチマーク実装評価調査 18
4.2.1 Cloudflare 版 OpenSSL 19
4.2.2 Edge Security の WireGuard 20
4.2.3 Barco Silex の BA417 と Inside Secure の ChaCha-IP-13 / EIP-13, POLY-IP-53 / EIP-53 22

第 5 章　認証暗号方式の性能比較調査　24

5.1 性能比較調査方法 24
5.1.1 比較対象アルゴリズム 24
5.1.2 実行環境 32
5.1.3 暗号ライブラリ 34
5.2 実行コマンド 35
5.3 性能比較結果 37
5.3.1 AES-NI 有効の場合 37

5.3.2 AES-NI 無効の場合 46
5.3.3 性能測定における注意点 51
5.4 考察 52
5.4.1 Poly1305 によるオーバーヘッド 52
5.4.2 AES-NI の影響 53
5.4.3 ChaCha20-Poly1305 と AES-GCM の比較 53

第 6 章 まとめ 54

Arm, Cortex, mbed は Arm Limited の登録商標です。
Cloudflare は Cloudflare, Inc の登録商標です。
Google は Google LLC の登録商標です。
Inside Secure は Inside Secure またはその子会社の登録商標です。
Intel は Intel Corporation またはその子会社の登録商標です。
WireGuard は Jason A. Donenfeld の登録商標です。
AVR は Microchip Technology Incorporated の登録商標です。
OpenSSL は OpenSSL Software Foundation の登録商標です。
OpenVPN は OpenVPN Inc. の登録商標です。

第1章　エグゼクティブサマリー

本調査では、認証暗号 (Authentication Encryption: AE または Authenticated Encryption with Associated Data: AEAD) の一種である ChaCha20-Poly1305 について、実際に利用する観点で、標準化状況や OSS コミュニティでの採用状況を踏まえて、実装性能評価を実施した。

ChaCha20-Poly1305 は特に IETF (Internet Engineering Task Force) で積極的に標準化が進められており、今後も幅広いプロトコルへの適用が予想される。また各種 OSS での採用も広まっており、ChaCha20-Poly1305 の利用環境は整いつつあると言える。

ChaCha20-Poly1305 の実装性能を評価した論文はまだ少ない。ベンチマークとしてソフトウェア実装やハードウェア実装が公開されており、ベンチマーク結果において今後組み込み環境において ChaCha20-Poly1305 の性能が向上することが示唆されている。

世界中で広く利用されている OpenSSL を用いて ChaCha20-Poly1305 と他の認証暗号との性能比較を行った。AES-NI を有効にした Intel Core i7 において、ChaCha20-Poly1305 のスループットは、鍵長以下のデータサイズでは AES-NI 無効時より小さく、鍵長を超えるデータサイズでは大きくなることがわかった。また AES-NI を有効にした Intel Xeon においては、ChaCha20-Poly1305 のスループットはデータサイズによらずほとんど変わらないことがわかった。AES-NI 有効時および AES-NI 無効時において AES-GCM と ChaCha20-Poly1305 のスループットを比較すると、AES-NI 無効時に ChaCha20-Poly1305 の実行速度が AES-GCM を上回ることがわかった。

第2章　ChaCha20-Poly1305について

2.1　概要

ChaCha20-Poly1305は、ストリーム暗号であるChaCha20とメッセージ認証コード (MAC) であるPoly1305を組み合わせ、認証暗号機能を実現した暗号アルゴリズムである。ChaCha20-Poly1305の安全性は暗号学的に解析されており、ChaCha20およびPoly1305が安全なアルゴリズムであるとの仮定において、ChaCha20-Poly1305は安全な認証暗号アルゴリズムであることが示されている [1]。また、ChaCha20に対する効率的な攻撃（差分攻撃、線形攻撃等）が存在しないことも示されており、ChaCha20-Poly1305に脆弱性は存在せず、安全なアルゴリズムであると結論づけられている [1]。ChaCha20-Poly1305は特に、AES-NI (Advanced Encryption Standard New Instructions) [2] による高速化が有効でない環境において、AES-GCM等のAESを用いた認証暗号アルゴリズムよりも高速に実行可能であると言われている。

従来、Rivestによって提案されたRC4がストリーム暗号として広く用いられて、SSL/TLSにも採用されていた。しかし、RC4に対する攻撃が次々と発見されたことによりRC4の安全性が低下し、RC4の使用が非推奨となるとともに新たなストリーム暗号の提案が望まれた。

他方、2013年のEdward Snowdenによる内部告発により、アメリカ国家安全保障局 (NSA) による通信監視プログラムPRISMの存在が暴露された [3], [4]。この過程で、アメリカ国立標準技術研究所 (National Institute of Standards and Technology: NIST) によるNIST SP 800-90Aで標準化された乱数生成器であるDual_EC_DRBGにバックドアが仕掛けられていたことが判明した [5]。

これらの事象により、暗号アルゴリズムの利用に関する2つの動きが生じた。ひとつは、政府機関による監視への技術的対抗策を講じることである。この流れとしては、128ビット以上の安全性を持つ暗号アルゴリズムの利用、認証暗号の利用、forward secrecyを持つ暗号アルゴリズムの利用等が挙げられる。もうひとつは、政府機関により策定された暗号アルゴリズムの利用を避けることである。インターネット技術の標準化を推進

する IETF (Internet Engineering Task Force) では、中立性を重要視する文化も相まって、2013 年 11 月に開催された IETF 88 以来、IETF として NIST で採用されていない暗号アルゴリズムを利用する動きが活発化した。NIST で標準化されていない暗号アルゴリズムには例えば、楕円曲線暗号におけるエドワーズ曲線 (Ed25519) やこれを用いた署名アルゴリズム EdDSA が挙げられる。このような流れの中、認証暗号機能を実現する ChaCha20-Poly1305 に対する注目が集まっている。

2.2 歴史的背景

ChaCha20 および Poly1305 はいずれも D. J. Bernstein によって考案された暗号アルゴリズムである。

ChaCha20 は、2007 年に SalSa20 として公開されたストリーム暗号 [6] の改良版として、2008 年に発表された [7]。

一方、Poly1305 は 2005 年にメッセージ認証コード (MAC) として発表された [8]。発表された論文において、Poly1305 はブロック暗号 AES と組み合わせた方式として示された (Poly1305-AES)。

その後、2.1 章に示した NIST で採用されていない暗号アルゴリズムを利用する流れを受けて、ストリーム暗号 ChaCha20 とメッセージ認証コード (MAC) Poly1305 を組み合わせて利用する動きが進み、2015 年 5 月には IETF から ChaCha20-Poly1305 の仕様を規定した RFC が発行された [9]。

2.3 アルゴリズムの構造

本項では、IETF で標準化された ChaCha20 と Poly1305、およびその組み合わせである ChaCha20-Poly1305 の構造を記述する [9]。

2.3.1 ChaCha20

ChaCha は、256 ビットの秘密鍵 $K = (k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7)$ と 32 ビットのカウンター $C = (c_0)$ を入力とし、512 ビットの鍵ストリームを生成する。鍵ストリームに用いる行列 X を以下の通り定義する。

$$
\begin{aligned}
X &= \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} \\
&= \begin{pmatrix} \sigma_0 & \sigma_1 & \sigma_2 & \sigma_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ c_0 & n_0 & n_1 & n_2 \end{pmatrix}
\end{aligned}
$$

ここで、

$$(\sigma_0, \sigma_1, \sigma_2, \sigma_3) = (\texttt{0x61707876}, \texttt{0x3320646E}, \texttt{0x79622D32}, \texttt{0x6B206574})$$

である。

鍵ストリームブロック Z は $Z = X + X^{(r)}$ で定義される。ここで $X^{(r)}$ はラウンド関数を用いて $X^{(r)}$ =$\mathsf{Round}^r(X)$ と表され、$+$ は 2^{32} の剰余によるワードごとの加算を表す。$Z = X + X^{(r)}$ を「r ラウンド ChaCha」または「ChaCha$r$」と呼ぶ。

ラウンド関数は “quarter round function” と呼ばれる次の非線形演算で構成される。4 ワードから成るベクトル (a, b, c, d) は以下のように変換される。

$$
\begin{aligned}
a &= a + b \\
d &= d \oplus a \\
d &= (d)_{<<<16} \\
c &= c + d \\
b &= b \oplus c \\
b &= (b)_{<<<12} \\
a &= a + b \\
d &= d \oplus a \\
d &= (d)_{<<<8} \\
c &= c + d \\
b &= b \oplus c \\
b &= (b)_{<<<7}
\end{aligned}
$$

quarter round function は、奇数ラウンドには列方向に、偶数ラウンドには対角線方向に作用する。Algorithm 1 において、ChaCha の詳細な手順を示す。

Algorithm 1 ChaCha

Input: Key K, Counter C, and Nonce N
Output: Keystream Z
  Generate initial matrix X using K, C, and N
  $y \leftarrow X$
  for $i \leftarrow 0$ **to** 9 **do**
    `/* Column Round */`
    $(x_0, x_4, x_8, x_{12}) \leftarrow quarterround(x_0, x_4, x_8, x_{12})$
    $(x_5, x_9, x_{13}, x_1) \leftarrow quarterround(x_5, x_9, x_{13}, x_1)$
    $(x_{10}, x_{14}, x_2, x_6) \leftarrow quarterround(x_{10}, x_{14}, x_2, x_6)$
    $(x_{15}, x_3, x_7, x_{11}) \leftarrow quarterround(x_{15}, x_3, x_7, x_{11})$
    `/* Diagonal Round */`
    $(x_0, x_5, x_{10}, x_{15}) \leftarrow quarterround(x_0, x_5, x_{10}, x_{15})$
    $(x_1, x_6, x_{11}, x_{12}) \leftarrow quarterround(x_1, x_6, x_{11}, x_{12})$
    $(x_2, x_7, x_8, x_{13}) \leftarrow quarterround(x_2, x_7, x_8, x_{13})$
    $(x_3, x_4, x_9, x_{14}) \leftarrow quarterround(x_3, x_4, x_9, x_{14})$
  end for
  $Z \leftarrow X + y$
  return Z

2.3.2 Poly1305

Poly1305 は、256 ビットの鍵と任意長のメッセージを入力とし、128 ビットのタグを出力する。256 ビットの入力鍵はそれぞれ 128 ビットの鍵 r と s に分割される。このアルゴリズムでは、r の 22 ビットを固定する。出力タグは $((m[0]r^n + m[1]r^{n-1} + \cdots + m[n]) \bmod (2^{130} - 5) \bmod 2^{128}$ で表される。ここで、$m[i]$ は入力メッセージ M を 16 ビットに分割したブロックの i 番目のブロックである。Algorithm 2 において、Poly1305 の詳細な手順を示す。

Algorithm 2 Poly1305

Input: Key K and Message M
Output: Tag T
$(m[0], m[1], \ldots, m[d-1]) \overset{16}{\leftarrow} M$
$d \leftarrow \lceil len(M)/16 \rceil$
$(r, s) \overset{16}{\leftarrow} K$
$r \leftarrow r\&$ `0x0FFFFFFC0FFFFFFC0FFFFFFC0FFFFFFF`
for $i \leftarrow 0$ **to** $d-1$ **do**
  $m[i] \leftarrow m[i] + 2^{8len(m[i])}$
end for
$T \leftarrow m[0]$
for $i \leftarrow 1$ **to** $d-1$ **do**
  $T \leftarrow (r \cdot T + m[i]) \bmod (2^{130} - 5)$
end for
$T \leftarrow (T + s) \bmod 2^{128}$
return T

Algorithm 3 ChaCha20-Poly1305

Input: Key K, Nonce N, Authentication data A, and Message M
Output: Ciphertext C and Tag T
$z \leftarrow \text{CC-Poly-KS}(K, N, len(M))$
$C \leftarrow M \oplus z$
$T \leftarrow \text{CC-Poly-T}(K, N, A, C)$
return (C, T)

2.3.3 ChaCha20-Poly1305

ChaCha20-Poly1305 は、256 ビットの鍵 K、96 ビットのナンス N、任意長の認証対象データ A、任意長のメッセージ M を入力とし、暗号文 C と認証タグ T を出力する。Algorithm 3, 4 および 5 において、ChaCha20-Poly1305 の詳細な手順を示す。図 2.1 に ChaCha20-Poly1305 の全体構成図を示す。

Algorithm 4 CC-Poly-KS

Input: Key K, Nonce N and Input Length L
Output: Keystream Z

$b \leftarrow \lceil L/64 \rceil$
for $i \leftarrow 0$ **to** $b-1$ **do**
  $z[i] \leftarrow \mathsf{ChaCha}(K, i+1, N)$
end for
$z \leftarrow \sum_{i=0}^{b-1} z[i] \cdot 2^{512i}$
$Z \leftarrow trancate(l, z)$
return Z

Algorithm 5 CC-Poly-T

Input: Key K, Nonce N, Authentication data A, and Message M
Output: Tag T

$k \leftarrow trancate(32, \mathsf{ChaCha}(K, 0, N))$
$y \leftarrow A$
$y \leftarrow y + M \cdot 2^{128\lceil len(A)/16 \rceil}$
$y \leftarrow y + len(A) \cdot 2^{128(\lceil len(A)/16 \rceil + \lceil len(M)/16 \rceil)}$
$y \leftarrow y + len(M) \cdot 2^{128(\lceil len(A)/16 \rceil + \lceil len(M)/16 \rceil + 1/4)}$
$T \leftarrow \mathsf{Poly1305}(k, y)$
return T

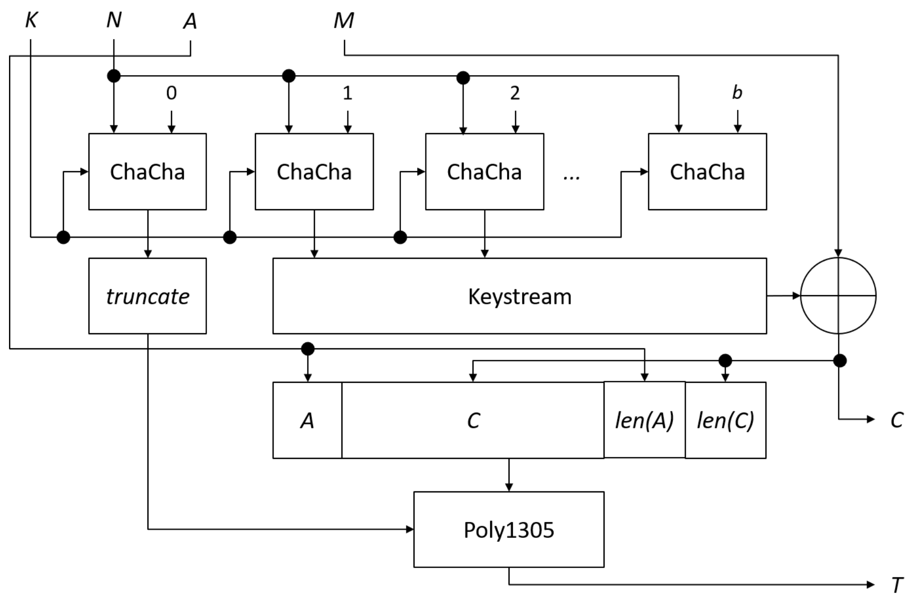

図 2.1: ChaCha20-Poly1305 全体構成図

第3章　ChaCha20-Poly1305の採用状況

新しい暗号アルゴリズムを実社会で広く利用するためには、国際標準化団体におけるにSSL/TLSのような一般的な通信プロトコルでの標準化仕様策定と実際に動作環境となるオープンソース（OSS）による実装実績が左右すると言える。ChaCha20-Poly1305については標準化仕様策定の前から、Googleによって自社で利用するGoogleサーバとGoogle Chromeで実装することによって、実環境であるインターネット上で動作実績を重ねて問題なく動作することを示した積極的な活動の結果だと言える。

3.1　標準化動向

3.1.1　IETF

IETFにおいて、ChaCha20-Poly1305を利用するために標準化されたドキュメントは、4つのRFC（Request For Comments）が存在している。それらのRFCについては、以下に列挙したとおりである。それぞれのRFCとしては、ChaCha20-Poly1305自体のアルゴリズムを記述したもの、ChaCha20-Poly1305をIKEおよびIPsecで利用するために規定したもの、ChaCha20-Poly1305をTLSで利用するために規定したもの、ChaCha20-Poly1305をCMS（暗号メッセージ構文）で利用するために規定したものとなる。

RFC 7539

このドキュメントは2015年5月に発行されたInformationalなRFCであり、タイトルは「ChaCha20 and Poly1305 for IETF Protocols」である。このドキュメントの内容は、ChaCha20-Poly1305のアルゴリズム自体の記述および実装者が自分の実装が正しく実装されているのかを確かめるために必要となる独立した2実装により確認されたテストベクターを示している。このRFCで記述されているアルゴリズムが、IETFとして標準化されたChaCha20-Poly1305となるため、IETFで標準化される様々な通信プロトコルから参照されることになる。

表 3.1: IETF で策定された ChaCha20-Poly1305 に関連する RFC

番号	タイトル	発行日
RFC 7539 [9]	ChaCha20 and Poly1305 for IETF Protocols	2015/5/13
RFC 7634 [10]	ChaCha20, Poly1305, and Their Use in the Internet Key Exchange Protocol (IKE) and IPsec	2015/8/20
RFC 7905 [11]	ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)	2016/6/22
RFC 8103 [12]	Using ChaCha20-Poly1305 Authenticated Encryption in the Cryptographic Message Syntax (CMS)	2017/2/28

しかしながら、RFC 7539 の記述内容には 7 つの誤りが指摘されており、その正誤表で示された内容を踏まえて指摘箇所をアップデートし、かつ Security Considerations を加筆した rfc7634bis [13] が 2016 年 11 月に投稿され、2 回のコメント吸収を経て 2017 年 10 月現在 IRSG での投票を行なっているステータスである。

RFC 7634

このドキュメントは 2015 年 8 月に発行された Proposed Standard な RFC であり、タイトルは「ChaCha20, Poly1305, and Their Use in the Internet Key Exchange Protocol (IKE) and IPsec」である。このドキュメントの内容は、Internet Key Exchange Protocol version 2 (IKEv2) および IPsec において、ChaCha20-Poly1305 を利用するために発行された RFC である。この RFC の中において、IANA より Transform Type 1 - Encryption Algorithm Transform IDs [14] として、28 がアサインされたことにより、Encapsulated Security Protocol（ESP）と IKEv2 で利用できるようになった。

RFC 7905

このドキュメントは 2016 年 6 月に発行された Proposed Standard な RFC であり、タイトルは「ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)」である。このドキュメントの内容は、TLS と DTLS において利用できるよう 7 つの暗号スイートを規定したものである。

```
TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256   = {0xCC, 0xA8}
```

```
TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 = {0xCC, 0xA9}
TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256     = {0xCC, 0xAA}
TLS_PSK_WITH_CHACHA20_POLY1305_SHA256         = {0xCC, 0xAB}
TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256   = {0xCC, 0xAC}
TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256     = {0xCC, 0xAD}
TLS_RSA_PSK_WITH_CHACHA20_POLY1305_SHA256     = {0xCC, 0xAE}
```

なお、RFC 7905 が発行されるまでに、Google 社において様々な暗号スイートに関する値を勝手に取得して実装している状況となっていた。

RFC 8103

このドキュメントは 2016 年 6 月に発行された Proposed Standard な RFC であり、タイトルは「Using ChaCha20-Poly1305 Authenticated Encryption in the Cryptographic Message Syntax (CMS)」である。このドキュメントの内容は、CMS（暗号メッセージ構文）において、ChaCha20-Poly1305 を利用できるよう必要な情報を規定したものである。

この RFC で規定された内容としては、IANA の Structure of Management Information (SMI) Numbers (MIB Module Registrations) [15] において、

- ”SMI Security for S/MIME Algorithms (1.2.840.113549.1.9.16.3)” として id-alg-AEADChaCha20Poly1305（18）
- ”SMI Security for S/MIME Module Identifier (1.2.840.113549.1.9.16.0)” として id-mod-CMS-AEADChaCha20Poly1305（66）

を規定している。

なお、ChaCha20-Poly1305 の OID は、以下のものとなっている。

```
id-alg-AEADChaCha20Poly1305 OBJECT IDENTIFIER ::=
       { iso(1) member-body(2) us(840) rsadsi(113549)
pkcs(1) pkcs9(9) smime(16) alg(3) 18 }
```

IETF におけるその他動向

IETF には IRTF を含むと 8 つの Area が存在しており、これらの Area に対して 10 以上の WG が紐づいている。とても多くの WG があるような状況から IETF での ChaCha20-Poly1305 に関する標準化動向を監視することは難しい。2017 年 10 月現在、IETF Security

Area において CURves, Deprecating and a Little more Encryption (curdle) [16] という IETF における既存プロトコルに対して古くなった暗号アルゴリズムを廃止して、新しい暗号アルゴリズムを追加することを検討する WG が存在している。この WG での活動を追うことで、IETF における ChaCha20-Poly1305 の標準化動向を把握することができる。

3.1.2 その他の標準化団体

IETF 以外での標準化団体として、ISO/IEC JTC 1/SC 27/WG 2 [17]、ITU-T SG17 [18]、W3C [19] に対して、ChaCha20-Poly1305 の標準化動向について調査を実施したが、2017 年 10 月現在標準化された仕様の中に ChaCha20-Poly1305 に関する情報がなかった。

3.2 OSS コミュニティ動向

本章では、IETF で標準化されている ChaCha20-Poly1305 が、OSS コミュニティにおいてどこまで実装されているのかを調査する。調査対象となるコミュニティを以下に示す。

OpenSSL [20] The OpenSSL Project によるオープンソースプロジェクトである。1995 年に世界初の SSL ライブラリとして SSLeay がリリースされ、その後 1998 年に SSLeay の後継として OpenSSL 0.9.1c がリリースされた。

The Sodium crypto library (libsodium) [21] D. J. Bernstein および T. Lange により開発されたオープンソースライブラリである NaCl からフォークされたライブラリである。現在は Frank Denis を中心に開発されている。

LibreSSL [22] The OpenBSD Project によるオープンソースプロジェクトである。The OpenBSD Project はセキュリティを重要視しており、2014 年の Heartbleed 脆弱性の発生をきっかけに、OpenSSL 1.0.1 からフォークして独自の OSS の開発を開始した。

GnuTLS [23] TLS のようなプロトコルを GNU プロジェクトのアプリケーションで扱えるようにすることを目的として作成された。

BoringSSL [24] Google が開発主体であるオープンソースプロジェクトである。2014 年 6 月に OpenSSL からフォークすることが発表された。

BouncyCastle [25] Java SE で作業を行うジョブを変更する際の暗号化ライブラリの開発の効率化のために生まれたライブラリである。

Linux Kernel [26] Unix 系オペレーティングシステムである Linux のカーネルであり、The Linux Kernel Organization によって配布されている。

NSS [27] Mozilla Foundation によるオープンソースライブラリである。Netscape 社が SSL プロトコルのために開発したライブラリを起源としている。

wolfSSL [28] 2004 年に商用ライセンスと GPL でのデュアルライセンスにおいて利用可能な OpenSSL の代替として開発された。

上記に示した OSS コミュニティにおける実装状況を表 3.2 に示す。表内において、認証暗号への対応状況として、ChaCha20-Poly1305、AES-GCM、AES-CCM への対応状況をそれぞれ CP, AG, AC で表した。表からも分かるように、ChaCha20-Poly1305 を実装する OSS では必ず AES-GCM および AES-CCM も実装されている。

ここで、IETF における ChaCha20-Poly1305 の利用を規定した RFC 7539 の発行が 2015 年 5 月 13 日、ChaCha20-Poly1305 の TLS への適用を規定した RFC 7905 の発行が 2016 年 6 月 22 日である。つまり、BoringSSL、libsodium、GnuTLS は RFC の発行に先立って ChaCha20-Poly1305 の実装を公開していることが分かる。また、OpenSSL は RFC 7905 の発行後に実装を公開していることが分かる。

表 3.2: OSS コミュニティごとの ChaCha20-Poly1305 採用状況

OSS コミュニティ	対応状況	ChaCha20-Poly1305 採用	
		バージョン	リリース日
BoringSSL	CP, AG, AC	de0b202	2014/6/20
libsodium	CP, AG, AC	0.6.0	2014/7/2
GnuTLS	CP, AG, AC	3.4.0	2015/4/8
Linux Kernel	CP, AG, AC	4.2	2015/8/30
NSS	CP, AG, AC	3.23	2016/3/3
wolfSSL	CP, AG, AC	3.9.0	2016/3/18
LibreSSL	CP, AG, AC	2.4	2016/5/31
OpenSSL	CP, AG, AC	1.1.0	2016/8/25
BouncyCastle	CP, AG, AC	1.56	2016/12/23

(凡例) CP: ChaCha20-Poly1305, AG: AES-GCM, AC: AES-CCM

第4章　ChaCha20-Poly1305の実装性能評価に関する既存文献調査

4.1　論文調査

主要な国際会議および論文誌において、ChaCha20 および Poly1305 の実装性能に関する論文はまだ少なく、2017 年 10 月時点で以下の件数が発表されているのみである。

- ChaCha20：4 件
- Poly1305：2 件
- ChaCha20-Poly1305：1 件

本調査ではこのうち、ChaCha20-Poly1305 の実装性能に関する研究として、De Santis らによる論文 [29] の内容を解説する。

当該論文では、ChaCha20-Poly1305 の高速かつ軽量な IoT アプリケーションでの利用を目的として、ChaCha20-Poly1305 の ARM Cortex-M4 プロセッサへの実装とその性能実験結果を示している。

表 4.1, 4.1 および 4.1 に、当該論文での性能実験結果および既存研究との比較結果を示す。当該論文にて示された実験結果を太字で示す。性能実験は、64 バイト（512 ビット）入力の暗号アルゴリズム、128 バイト（1024 ビット）入力の MAC、16 バイト（128 ビット）入力および 16 バイト（128 ビット）補助入力の認証暗号に対して測定した。実験結果のうち速度 [Cycles] は、ARM mbed library の内部クロックサイクルカウンター (CYCCNT) を用いて測定したものである。実行時間 [Cycles/Byte] は、メッセージ長が十分大きいときの漸近的な値を示す。スタック [Byte] は、関数呼び出しによりメモリ上で書き換えられたデータ量を示す。

当該論文の性能実験結果によると、当該論文で示された ChaCha20 の高速実装によって、既存研究と比較して約 2 倍の速度を達成している。また、ChaCha20 および Poly1305 の高速実装によって、既存研究で示された他の認証暗号（AES128-GCM や AES128-CCM）と比較しても高速な実装を達成している。

表 4.1: ARM プロセッサによる ChaCha20-Poly1305 性能実験結果（暗号化） [29]

プラットフォーム	アルゴリズム	速度 [Cycles]	実行時間 [Cycles/Byte]	コードサイズ [Byte]	スタック [Byte]
8-bit AVR ATmega	Salsa20	17,787	268.0	-	268
32-bit ARM Cortex-M4	Salsa20	3,311	-	1,272	552
32-bit ARM Cortex-M0	ChaCha20	-	39.9	-	-
32-bit ARM Cortex-M4	ChaCha20	3,468	-	1,328	544
32-bit ARM Cortex-M4	**ChaCha20**	**1,584**	**22.0**	**696**	**256**

表 4.2: ARM プロセッサによる ChaCha20-Poly1305 性能実験結果（MAC） [29]

プラットフォーム	アルゴリズム	速度 [Cycles]	実行時間 [Cycles/Byte]	コードサイズ [Byte]	スタック [Byte]
32-bit ARM Cortex-M0	Chaskey	-	18.3	1,308	-
32-bit ARM Cortex-M4	Chaskey	-	7.0	908	-
8-bit AVR ATmega	Poly1305	-	195.0	-	148
32-bit ARM Cortex-M4	**Poly1305**	**733**	**3.6**	**648**	**112**

4.2 ベンチマーク実装評価調査

ChaCha20-Poly1305 のベンチマーク実装評価として、

- Cloudflare 社が独自に実装した ChaCha20-Poly1305 が組み込まれた OpenSSL [30]
- Edge Security 社が独自に実装した ChaCha20-Poly1305 が組み込まれた VPN である WireGuard [31]

についての性能比較を示す。これらはいずれもソフトウェアとして実装されたものである。

一方、ChaCha20-Poly1305 は ASIC や FPGA 向け IP (Intellectual Property) コアとしても提供されている。これらの例として、

- Barco Silex 社が IP コアとして提供する ChaCha20-Poly1305 [32]
- Inside Secure 社が IP コアとして提供する ChaCha20 と Poly1305 [33, 34]

についての性能比較を示す。

表 4.3: ARM プロセッサによる ChaCha20-Poly1305 性能実験結果（認証暗号） [29]

プラットフォーム	アルゴリズム	速度 [Cycles]	実行時間 [Cycles/Byte]	コードサイズ [Byte]	スタック [Byte]
32-bit ARM Cortex-M4	AES128-GCM	43,657	-	2,644	812
32-bit ARM Cortex-M4	AES128-EAX	32,159	-	2,780	932
32-bit ARM Cortex-M4	AES128-CCM	23,949	-	2,256	780
8-bit AVR ATmega	NORX32	-	146.0	-	-
32-bit ARM Cortex-M4	NORX32	6,855	-	1,820	320
32-bit ARM Cortex-M4	**ChaCha20 -Poly1305**	**4,235**	**33.6**	**1,668**	**520**

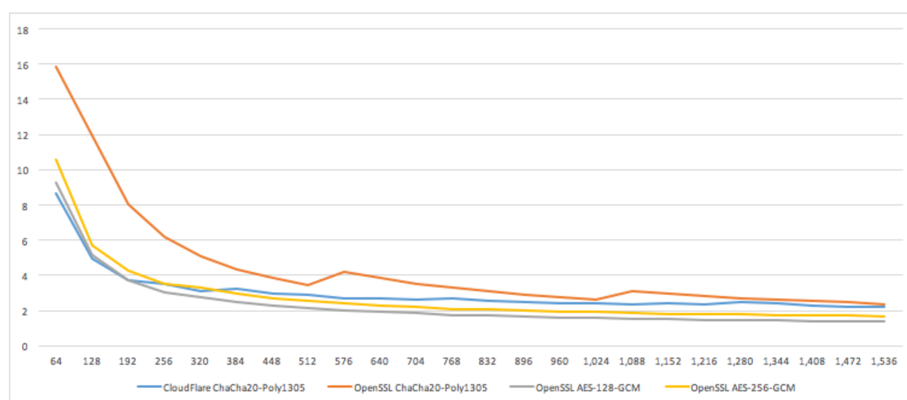

図 4.1: Cloudflare 版 ChaCha20-Poly1305 と OpenSSL 版認証暗号の性能比較 [30]
縦軸：CPU サイクル数 [Cycle/Byte]　横軸：平文サイズ [Byte]

4.2.1 Cloudflare 版 OpenSSL

ChaCha20-Poly1305 のベンチマーク実装評価として、Cloudflare 社が独自に実装した ChaCha20-Poly1305 と、OpenSSL 1.1.0 pre に組み込まれている ChaCha20-Poly1305 および AES-GCM (128bit/256bit) の性能比較の結果を示す。図 4.1 および図 4.2 にグラフを示す。

この性能比較は Intel Haswell プロセッサ上で行なわれた。このプロセッサでは 256-bit SIMD 拡張命令の AVX2 が利用可能であり、これを利用することで ChaCha20-Poly1305 を高速に実行できる。

それぞれの図は、所定のレコード長の平文に対して暗号化処理を実行した場合の 1 バイトあたりの CPU サイクル数を示したものであり、図 4.1 は平文のレコード長が 64–1,536[Byte], 図 4.2 は 1,536–16K[Byte] の場合のものである。暗号化においては、TLS と同様に、与えられた平文

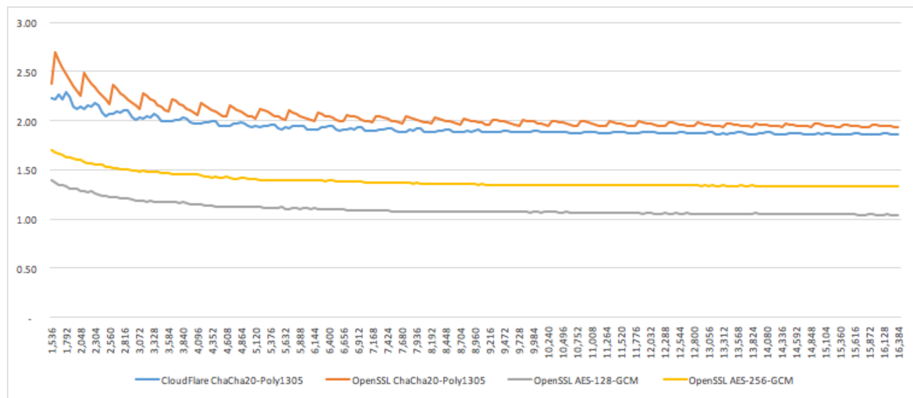

図 4.2: Cloudflare 版 ChaCha20-Poly1305 と OpenSSL 版認証暗号の性能比較 [30]
縦軸：CPU サイクル数 [Cycle/Byte]　横軸：平文サイズ [Byte]

に 13 バイトの AD(Additional Data) を追加したものに対して処理を行っている。

Cloudflare の ChaCha20-Poly1305 は、OpenSSL のものに比べ常に高速で、平均で 7%パフォーマンスが良く、特にレコード長が短い場合は優れている。

OpenSSL の AES-128-GCM, AES-256-GCM はいずれもレコードサイズが 320 バイトを超えると Cloudflare の ChaCha20-Poly1305 の効率を上回るが、その差は 1 バイトあたり 2 サイクル未満となっている。その他の多くのモードはこの性能を上回ることは無かった。

ここで、AES-GCM は AES アルゴリズム向け拡張命令 (AESENC, CLMULQDQ) を使用するのに対し、ChaCha20-Poly1305 は一般的な SIMD 命令を使用して実現されている点に注意されたい。

4.2.2 Edge Security の WireGuard

ChaCha20-Poly1305 のベンチマーク実装評価として、Edge Security 社が独自に実装した ChaCha20-Poly1305 が組み込まれた VPN である WireGuard の ChaCha20-Poly1305 コンフィグレーションと、IPsec の ChaCha20-Poly1305, AES-GCM コンフィグレーション, OpenVPN (AES with HMAC-SHA2 相当) コンフィグレーションの性能比較の結果を示す。いずれも鍵長は 256bit となっている。

この性能比較は Intel Core i7 プロセッサで動作する Linux 4.6.1 上で行われた。図 4.3 は単位時間あたりのデータ転送量 (Mbps) を、Figure

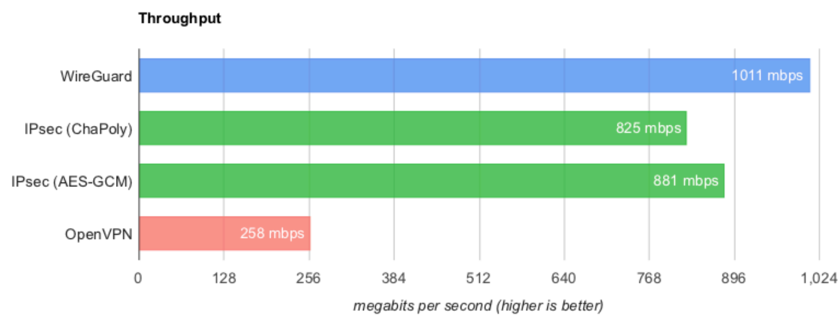

図 4.3: 単位時間あたりのデータ転送量 (Mbps) [35]

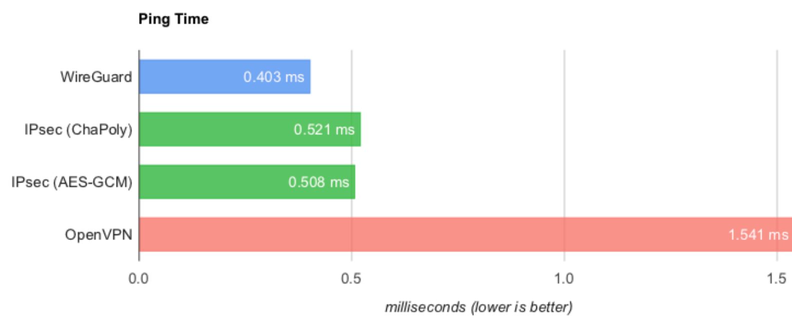

図 4.4: ping にかかる時間 (msec) [35]

表 4.4: Barco Silex 社の認証暗号 IP コアの性能比較

IP Core	throughput
BA417(ChaCha20-Poly1305)	multi-Gbps
BA415(Scalable AES-GCM/GMAC/CTR)	10Gbps to 100Gbps

4.4 は ping にかかる時間 (msec) を示している。いずれの結果においても、WireGuard は OpenVPN、IPsec の性能を上回っている。

OpenVPN および IPsec の性能評価時、CPU の利用率は 100%となっていたが、WireGuard の評価時は全てを使い切ってはいなかった。これは、 Gigabit Ethernet Link がボトルネックになっていたためではないかと考えられる。

AES-NI 拡張命令を使用している IPsec(AES-GCM) は、AVX2 拡張命令を使用している IPsec (ChaCha20-Poly1305) より性能が上回っているが、今後登場する CPU でベクトル命令の幅が広がると (例えば AVX512 拡張命令)、やがて ChaCha20-Poly1305 が AES-GCM の性能を上回ることが期待される[1]。

ChaCha20-Poly1305 は AES と比べ、ソフトウェア実装に向いており、サイドチャネル攻撃を受けにくく、効率が良い。そのため、 AES 計算のための拡張命令が利用できない組み込みプラットフォームでは、 ChaCha20-Poly1305 が最も性能が良いものとなる。また、 WireGuard が IPsec のいずれのコンフィグレーションより効率が良くなっているが、これは、実装のシンプルさとオーバーヘッドが無いことに起因する。

OpenVPN と WireGuard の性能の間にデータ伝送量と ping 時間のどちらにも大きな差があるが、これは OpenVPN がユーザー空間で動作するアプリケーションであり、タスクスケジューリングによる遅延やオーバーヘッド、パケット情報がユーザ空間とカーネル空間の間で何度もコピーされることによるオーバーヘッドが加わることによる。

4.2.3 Barco Silex の BA417 と Inside Secure の ChaCha-IP-13 / EIP-13, POLY-IP-53 / EIP-53

Barco Silex 社による BA417 は、ASIC や FPGA 向け IP コアとして提供される ChaCha20-Poly1305 である [32]。同社は AES-GCM の IP コアである BA415 を提供しており [36]、表 4.4 において、公開されているカタログ情報を元に両者の比較を示す。

[1] D. J. Bernstein による moderncrypto.org ML への投稿を参照 (https://moderncrypto.org/mail-archive/noise/2016/000699.html)

表 4.5: Inside Secure 社の認証暗号 IP コアの性能比較

IP Core	throughput	gates
ChaCha-IP-13 / EIP-13 (ChaCha20 accelerators)	2 – 12.8Gbps	30K
POLY-IP-53 / EIP-53 (Poly1305-based MAC accelerators)	1 – 6.4Gbps	50K
AES-IP-38 / EIP-38 (AES XTS/GCM accelerators)	6 – 100Gbps+	90 – 650K
AES-IP-39 / EIP-39 (AES “all modes” accelerators)	1 – 6.4Gbps	27 – 45K
AES-IP-61 / EIP-61 (High speed low latency AES-GCM pipeline)	134Gbps	890K

一方、Inside Secure 社は IP コアとしての ChaCha20 および Poly1305 をそれぞれ ChaCha-IP-13 / EIP-13 [33], POLY-IP-53 / EIP-53 [34] として提供する。同社は、AES の IP である AES-IP-38 / EIP-38 [37], AES-IP-39 / EIP-39 [38], AES-IP-61 / EIP-61 [39] を提供しており、表 4.5 において公開されているカタログ情報を元にこれらの比較を示す。

いずれもカタログで公開されている情報が少なく、詳しく比較することは難しいが、BA417(ChaCha20-Poly1305) は BA415(Scalable AES-GCM/GMAC/CTR) に比べ効率が悪く、また、ChaCha-IP-13 / EIP-13, POLY-IP-53 / EIP-53 の効率は、AES-IP-38 / EIP-38、AES-IP-39 / EIP-39, AES-IP-61 / EIP-61 を上回ることはない。この結果は、ソフトウェア実装である Cloudflare 版 OpenSSL および Edge Securty の WireGuard の性能比較結果と異なる。これは、AES が選定されてからの歴史が長く、IP コアとしての効率的な実現方法が確立しているのに対し、ChaCha20 および Poly1305 は標準化されてからの歴史が浅く、実現方法が十分に効率的なものになっていないのではないかと考えられる。今後、実現方法が改良され、AES の実装よりも ChaCha20 および Poly1365 の実装の性能が上回ることも十分考えられる。

第5章　認証暗号方式の性能比較調査

本章では、ChaCha20-Poly1305 および他の認証暗号の実行速度を計測し、性能を比較する。ChaCha20-Poly1305 は AES-NI が利用できない環境において高速に動作することが知られているため、AES-NI を有効にした場合と AES-NI を無効にした場合について性能を計測する。

5.1　性能比較調査方法

本項では、性能比較調査における比較対象アルゴリズム、性能比較調査の実行環境、および使用した暗号ライブラリについて説明する。

5.1.1　比較対象アルゴリズム

本調査では ChaCha20-Poly1305 の比較対象として、ChaCha20-Poly1305 と同等の機能を持つブロック暗号の認証暗号モードである CCM モードと GCM モードを採用する。次項において、CCM モードおよび GCM モードについて解説する。

CCM モード

ここでは、NIST SP 800-38C [40] で規定されている守秘・認証用暗号利用モードである CCM (Counter with Cipher Block Chaining-Message Authentication Code) の仕様について記述する。CCM は、Whiting らによって提案されたブロック暗号ベースの認証暗号方式であり、守秘用のカウンタモードと認証用の CBC-MAC を組み合わせた暗号利用モードである。

CCM モードの仕様は、認証タグ生成・暗号化アルゴリズム Generation-Encryption と、復号・検証アルゴリズム Decryption-Verification から構成される。CCM モードではブロック長 128 ビットのブロック暗号を用いることが想定されている。また、CCM モードではブロック暗号の暗号化関数のみが利用され、復号関数は利用しない仕様となっている。

秘密鍵 K によるブロック暗号 E_K、フォーマット関数、カウンタ生成関数、MAC のビット長 $len(T)$ に対し、Generation-Encryption はナンス N、Associated data A、および平文 P を入力、暗号文 C を出力として、以下のように定義される。ここで、フォーマット関数は入力 (N, A, P) に対し $(B_0, \ldots, B_r)$ を出力する関数であり、以下の性質を満たす。

- B_0 は N から一意に決定される。
- $B_1, \ldots, B_r$ は P, A から一意に決定し、$(N, P, A) \neq (N, P', A')$ となる入力に対する出力が $B_0, B_1, \ldots, B_r$ および $B'_0, B'_1, \ldots, B'_{r'}$ となるとき、$i \leq r$ かつ $i \leq r'$ となる i に対し $B_i \neq B'_i$ を満たす。
- B_0 は同じ秘密鍵 K を用いた CCM で使用されるどのカウンタブロックとも異なる。

フォーマット関数およびカウンタ生成関数の具体的な構成例は NIST SP 800-38C [40] の Appendix A に示されている。

Generation-Encryption

- 入力：(N, A, P)
- 出力：C

1. (N, A, P) にフォーマット関数を用いて、B_0, B_1, ... , B_r を計算する。
2. $Y_0 = E_K(B_0)$
3. $Y_i = E_K(B_i \oplus Y_{i-1})$ $(1 \leq i \leq r)$
4. $T = msb_{len(T)}(Y_r)$
5. カウンタ生成関数を用いて、カウンタブロック $ctr_0, ctr_1, ..., ctr_m$ を計算する。ただし、$m = \lceil len(P)/128 \rceil$
6. $S_j = E_K(ctr_j)$ $(0 \leq j \leq m)$
7. $S = S_1 || S_2 || \ldots || S_m$
8. $C = (P \oplus msb_{len(P)}(S)) || (T \oplus msb_{len(T)}(S_0))$

以下に、Generation-Encryption に関する処理手順を図示する。（図.5.1）

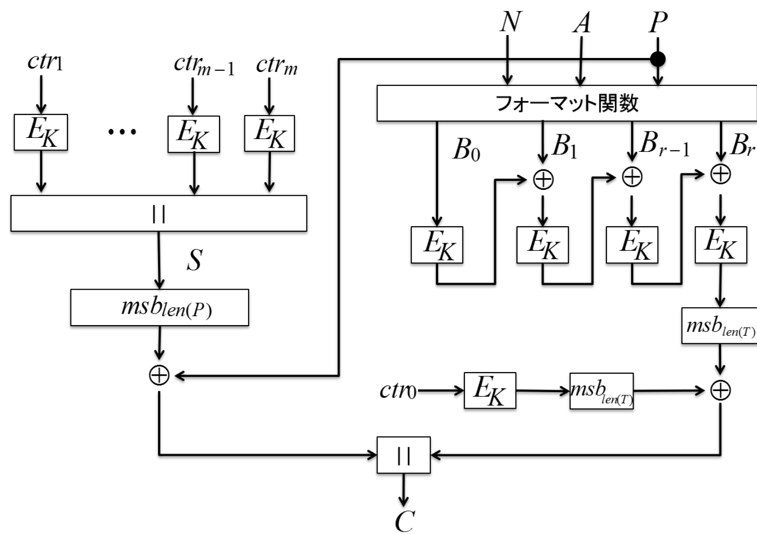

図 5.1: Generation-Encryption

ナンス N、associated data A、および暗号文 C に対し、Decryption-Verification は以下のように定義される。

Decryption-Verification

- 入力：(N, A, C)
- 出力：P

1. $len(C) \leq len(T)$ の時、INVALID を返却する
2. カウンタ生成関数を用いて、カウンタブロック $ctr_0, ctr_1, ..., ctr_m$ を計算する。ただし、$m = \lceil len(C) - len(T)/128 \rceil$
3. $S_j = E_K(ctr_j)$ $(0 \leq j \leq m)$
4. $S = S_1||S_2|| \dots ||S_m$
5. $P = msb_{len(C)-len(T)}(C) \oplus msb_{len(C)-len(T)}(S)$
6. $T = lsb_{len(T)}(C) \oplus msb_{len(T)}(S_0)$
7. (N, A, P) に不備がある場合には INVALID を返却する。それ以外の場合は (N, A, P) にフォーマット関数を適用して B_0, B_1, ... , B_r を計算する。
8. $Y_0 = E_K(B_0)$
9. $Y_i = E_K(B_i \oplus Y_{i-1})$ $(1 \leq i \leq r)$
10. $T \neq msb_{len(T)}(Y_r)$ の場合、INVALID を返却する。$T = msb_{len(T)}(Y_r)$ であれば P を返却する。

以下に、Decryption-Verification に関する処理手順を図示する。（図.5.2）

GCM モード

ここでは、NIST SP800-38D [41] で規定されている Galois/Counter Mode (GCM) および Galois Message Authentication Code (GMAC) に関する仕様について記述する。なお、GCM は CCM と同様ブロック暗号ベースの認証暗号方式であり、GCM はブロック長 128 ビットのブロック暗号を用いるよう設計されている。

GCM モードは、ブロック暗号のカウンタモードによる暗号化関数とユニバーサルハッシュ関数を利用した MAC から構成される。ブロック H、非負整数 m に対して $len(X) = 128m$ となるビット列 X に対し、GCM

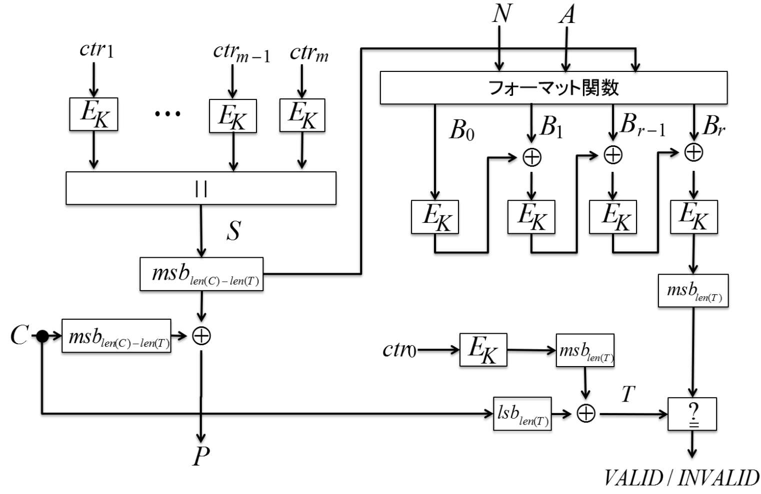

図 5.2: Decryption-Verification

で用いられるユニバーサルハッシュ関数 $GHASH_H(X)$ は以下のように構成される。なお、$X = X_1||X_2||\dots||X_m$ $(1 \leq i \leq m)$ としたとき、X_i の長さは 128 ビットとなる。加算と乗算は $GF(2^{128})$ 上で演算される。また、GCM モードでの法は $u^{128} + u^7 + u^2 + u + 1$ を用いる。

1. $Y_0 = 0$
2. $i = 1, \dots, m$ に対し、$Y_i = (Y_{i-1} \oplus X_i) \cdot H$ とする。
3. Y_m を出力する。

これより、$GHASH_H(X) = X_1 \cdot H^m \oplus X_2 \cdot H^{m-1} \oplus \cdots \oplus X_{m-1} \cdot H^2 \oplus X_m \cdot H$ となる。図 5.3 に $GHASH_H(X)$ を示す。

また、GCM で用いられるブロック暗号を E_K とし（K は秘密鍵）、E_K のブロック長を 128 とする。GCM のカウンタモードによる暗号化アルゴリズム $GCTR_K(ICB, X)$ は以下の通りである。なお、準備として、$X = X_1||X_2||\cdots||X_{n-1}||X_n^*$ とし、$len(X_i) = 128$ $(1 \leq i \leq n-1)$ および $1 \leq len(X_n^*) \leq 128$ とする。$n = \lceil len(X)/128 \rceil$ であり、$ICB \in \{0,1\}^{128}$ である。$GCTR_K(ICB, X)$ の構成を 図 5.4 に示す。

1. X が空列であれば、空列 Y を出力する。
2. $CB_1 = ICB$ とし、$CB_i = inc_{32}(CB_{i-1})$ $(2 \leq i \leq n)$ とする。ここで $inc_s(X) = msb_{len(X)-s}(X)||[int(lsb_s(X)) + 1 \bmod 2^s]_s$ とする。
3. $Y_i = X_i \oplus E_K(CB_i)$ $(1 \leq i \leq n-1)$

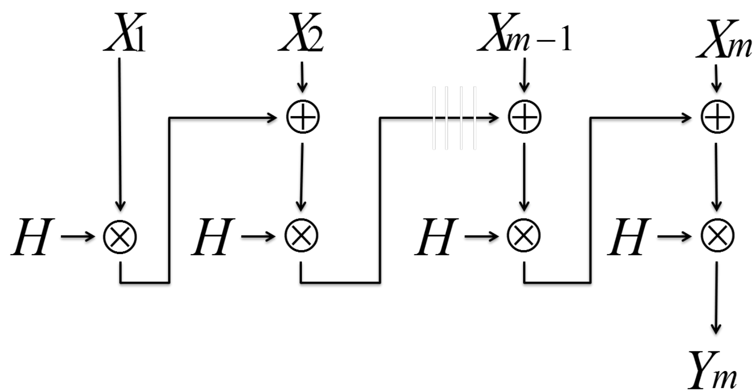

図 5.3: $GHASH_H(X)$

4. $Y_n^* = X_n^* \oplus msb_{len(X_n^*)}(E_K(CB_n))$

5. $Y = Y_1||Y_2||\cdots||Y_{n-1}||Y_n^*$ を出力する。

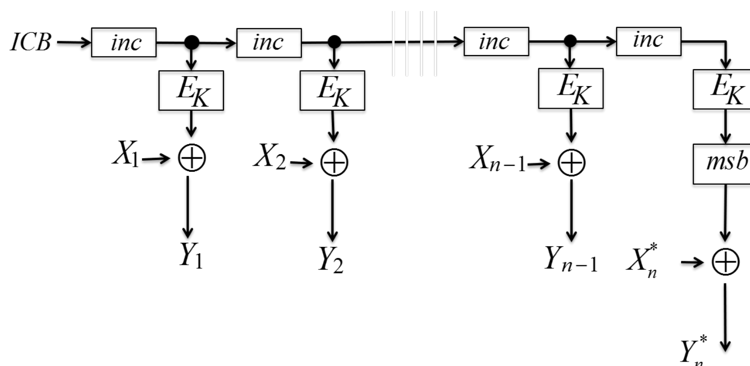

図 5.4: $GCTR_K(ICB, X)$

以上に示した準備を踏まえて、GCM モードの認証暗号化および認証復号のアルゴリズムを示す。なお、これらのアルゴリズムでは、平文 P、暗号文 C が空列の場合、GCM の MAC 生成と検証のアルゴリズムになる。

認証暗号化

GCM モードにおける認証暗号化アルゴリズム $GCM\!-\!AK_K(IV,P,A)$ は、$GCTR, GHASH$ を用いることで、以下のように構成される。なお、IV $(1 \le |IV| \le 2^{64}-1)$ は初期値であり、P $(0 \le len(P) \le (2^{32}-2)\cdot 128$ である。A は associated data$(0 \le len(A) \le 2^{64}-1)$ である。

1. $H = E_K(0^{128})$
2. J_0 を以下のように定める。$s = 128\lceil len(IV)/128 \rceil - len(IV)$ である。

$$J_0 = \begin{cases} IV||0^{31}1 & (len(IV)=96) \\ GHASH_H(IV||0^{s+64}||[len(IV)]_{64}) & (len(IV) \neq 96) \end{cases}$$

3. $C = GCTR_K(inc_{32}(J_0), P)$
4. $S = GHASH_H(A||0^v||C||C||0^u||[len(A)]_{64}||[len(C)]_{64}$ のように定める。ここで、以下のような u, v とする。

$$\begin{cases} u = 128\lceil len(C)/128 \rceil - len(C) \\ v = 128\lceil len(A)/128 \rceil - len(A) \end{cases}$$

5. $T = msb_r(GCTR_K(J_0, S))$
6. (C, T) を出力する。

以下に、$GCM - AE_K(IV, P, A)$ を図示する。（図.5.5）

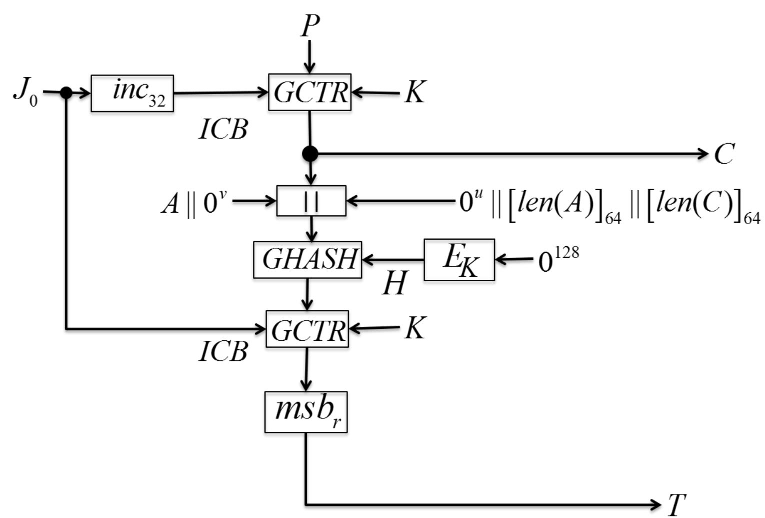

図 5.5: $GCM - AE_K(IV, P, A)$

認証復号

GCM における認証復号アルゴリズム $GCM - AD_K(IV, C, A, T)$ は以下のように定義される。

1. IV, C, A のビット長が定義の範囲外であるか、もしくは、$len(T) \neq r$ の時、INVALID を返却する
2. $H = E_K(0^{128})$
3. J_0 を以下のように定める。$s = 128\lceil len(IV)/128\rceil - len(IV)$ である。

$$J_0 = \begin{cases} IV||0^{31}1 (len(IV) = 96) \\ GHASH_H(IV||0^{s+64}||[len(IV)]_{64})(len(IV) \neq 96) \end{cases}$$

4. $P = GCTR_K(inc_{32}(J_0), C)$
5. $S = GHASH_H(A||0^v||C||C||0^u||[len(A)]_{64}||[len(C)]_{64}$ のように定める。ここで、以下のような u, v とする。

$$\begin{cases} u = 128\lceil len(C)/128\rceil - len(C) \\ v = 128\lceil len(A)/128\rceil - len(A) \end{cases}$$

6. $T' = msb_r(GCTR_K(J_0, S))$
7. $T = T'$ ならば、P を出力する。$T \neq T'$ ならば、INVALID を返却する。

以下に、$GCM - AD_K(IV, C, A, T)$ における P, T' をの生成手順を図示する。（図.5.6）

5.1.2 実行環境

本調査における性能比較調査では、プラットフォームとして Windows、MacOS、Linux および Amazon Linux を選定した。

Amazon Linux は Amazon Web Service (AWS) において提供されている Linux ディストリビューションのひとつである。RedHat 系 Linux をベースとしたディストリビューションであり、AWS のツールが追加されているため、Amazon EC2 で Linux サーバを構築するするのに適した OS である。今日のクラウド化の流れにより、Amazon Linux がサーバ

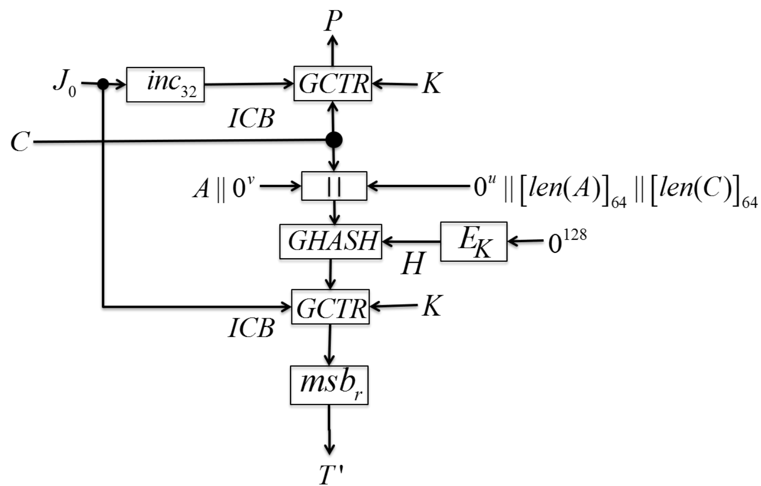

図 5.6: $GCM - AD_K(IV, C, A, T)$ における P, T'

OS として大きなシェアを持つと考えたため、本調査ではサーバ系プラットフォームとして Amazon Linux を選定し、性能測定を行った。

Windows、MacOS、Linux、および Amazon Linux それぞれの実行環境を表 5.1 に示す。

表 5.1: 実行環境の詳細 (Windows)

OS バージョン	Windows 10
CPU	Intel Core i7
クロック数	2.5 GHz
メモリ	8GB RAM

表 5.2: 実行環境の詳細 (MacOS)

OS バージョン	MacOS 10.12
CPU	Intel Core i7
クロック数	3.3 GHz
メモリ	16GB RAM

表 5.3: 実行環境の詳細 (Linux)

OS バージョン	Ubuntu 17.10
CPU	Intel Core i7
クロック数	3.3 GHz
メモリ	16GB RAM

表 5.4: 実行環境の詳細 (Amazon Linux)

OS バージョン	Amazon Linux AMI 2017.03.01 (HVM)
CPU	Intel Xeon E5-2670
クロック数	2.5 GHz
メモリ	3.75GB RAM

5.1.3 暗号ライブラリ

本調査における性能比較調査では、暗号ライブラリとして OpenSSL 1.1.0f を採用した。

各プラットフォームにおける OpenSSL 1.1.0f のビルドオプションを示す。

Windows のビルドオプションは以下の通りである。

```
$openssl version -a
OpenSSL 1.1.0f  25 May 2017
built on: reproducible build, date unspecified
platform:
compiler: cl " "VC-WIN64A
```

ここで、インストールログから抽出したビルドオプションは以下の通りである。

```
cl -DOPENSSL_USE_APPLINK -DDSO_WIN32 -DNDEBUG -DOPENSSL_
     THREADS  -DOPENSSL_NO_STATIC_ENGINE -DOPENSSL_PIC
     -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT
     -DOPENSSL_BN_ASM_MONT5 -DOPENSSL_BN_ASM_GF2m
     -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DRC4_ASM
     -DMD5_ASM -DAES_ASM -DVPAES_ASM -DBSAES_ASM
     -DGHASH_ASM -DECP_NISTZ256_ASM -DPADLOCK_ASM
     -DPOLY1305_ASM
```

MacOS のビルドオプションは以下の通りである。

```
$ openssl version -a
OpenSSL 1.1.0f  25 May 2017
built on: reproducible build, date unspecified
platform: darwin64-x86_64-cc
compiler: cc -DDSO_DLFCN -DHAVE_DLFCN_H -DNDEBUG
     -DOPENSSL_THREADS -DOPENSSL_NO_STATIC_ENGINE
     -DOPENSSL_PIC -DOPENSSL_IA32_SSE2
     -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5
     -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM
     -DSHA512_ASM -DRC4_ASM -DMD5_ASM -DAES_ASM
     -DVPAES_ASM -DBSAES_ASM -DGHASH_ASM
     -DECP_NISTZ256_ASM -DPADLOCK_ASM -DPOLY1305_ASM
```

Amazon Linux のビルドオプションは以下の通りである。

```
$ openssl version -a
OpenSSL 1.1.0f  25 May 2017
built on: reproducible build, date unspecified
platform: linux-x86_64
compiler: gcc -DDSO_DLFCN -DHAVE_DLFCN_H -DNDEBUG
     -DOPENSSL_THREADS -DOPENSSL_NO_STATIC_ENGINE
     -DOPENSSL_PIC -DOPENSSL_IA32_SSE2
     -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_MONT5
     -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM
     -DSHA512_ASM -DRC4_ASM -DMD5_ASM -DAES_ASM
     -DVPAES_ASM -DBSAES_ASM -DGHASH_ASM
     -DECP_NISTZ256_ASM -DPADLOCK_ASM
     -DPOLY1305_ASM -Wa,--noexecstack
```

5.2 実行コマンド

OpenSSL を用いて、認証暗号アルゴリズムである AES-GCM (128 ビット/256 ビット)、AES-CCM (128 ビット/256 ビット) および ChaCha20-Poly1305 (256 ビット) のスループットを測定した。また、認証機能の追加によるオーバーヘッドを測定するため、暗号化アルゴリズムである AES-ECB (128 ビット/256 ビット) および ChaCha20 (256 ビット) のスループットも測定した。

スループットの測定には OpenSSL の speed コマンドを利用した。各測定では、Intel 製 CPU の AES による暗号化・復号処理を高速に実行するための拡張命令セット (AES-NI) を有効にした場合と無効にした場合のそれぞれについて測定を実施した。これにより、AES-NI による高速化の影響を測ることが可能となる。

speed コマンドにおいて、AES-NI はデフォルトで有効になっている。AES-NI を無効にするためには、speed コマンド実行の前に環境変数 `OPENSSL_ia32cap` に 値 `~0x200000200000000` を設定する必要がある。ここで設定した値はビットパターンになっており、

bit #57 AES-NI 拡張命令を使用

bit #33 キャリーなし乗算命令 (PCLMULQDQ) を使用

をクリアする (~) という指示になっている [1]。

各アルゴリズムの具体的な実行コマンドは以下の通りである。

AES-NI 有効時

認証暗号

```
openssl speed -elapsed -evp aes-128-gcm
```

```
openssl speed -elapsed -evp aes-256-gcm
```

```
openssl speed -elapsed -evp aes-128-ccm
```

```
openssl speed -elapsed -evp aes-256-ccm
```

```
openssl speed -elapsed -evp chacha20-poly1305
```

暗号化

```
openssl speed -elapsed -evp aes-128-ecb
```

```
openssl speed -elapsed -evp aes-256-ecb
```

```
openssl speed -elapsed -evp chacha20
```

AES-NI 無効時

[1] https://wiki.openssl.org/index.php/Manual:OPENSSL_ia32cap(3)

認証暗号

```
OPENSSL_ia32cap=‘‘~0x200000200000000’’
openssl speed -elapsed -evp aes-128-gcm
```

```
OPENSSL_ia32cap=‘‘~0x200000200000000’’
openssl speed -elapsed -evp aes-256-gcm
```

```
OPENSSL_ia32cap=‘‘~0x200000200000000’’
openssl speed -elapsed -evp aes-128-ccm
```

```
OPENSSL_ia32cap=‘‘~0x200000200000000’’
openssl speed -elapsed -evp aes-256-ccm
```

```
OPENSSL_ia32cap=‘‘~0x200000200000000’’
openssl speed -elapsed -evp chacha20-poly1305
```

暗号化

```
OPENSSL_ia32cap=‘‘~0x200000200000000’’
openssl speed -elapsed -evp aes-128-ecb
```

```
OPENSSL_ia32cap=‘‘~0x200000200000000’’
openssl speed -elapsed -evp aes-256-ecb
```

```
OPENSSL_ia32cap=‘‘~0x200000200000000’’
openssl speed -elapsed -evp chacha20
```

5.3 性能比較結果

OpenSSL による性能調査の結果を示す。

5.3.1 AES-NI 有効の場合

AES-NI を有効にした場合の性能比較結果のグラフを図 5.7, 5.8, 5.9, 5.10 に示す。

グラフより、AES-NI が有効の環境では、どのプラットフォームにおいても AES-GCM のスループットが一番大きいことが分かる。

また、暗号化アルゴリズムである AES-ECB (128 ビット/256 ビット) および ChaCha20 の性能を表 5.9, 5.10, 5.11, 5.12 に示す。

表 5.5: Windows における認証暗号の性能比較結果（AES-NI 有効時）

データサイズ	AES-GCM 128bit	AES-CCM 128bit	AES-GCM 256bit	AES-CCM 256bit	ChaCha20 -Poly1305
16 bytes	343776.66	337223.00	318975.12	322720.64	201638.25
64 bytes	924426.26	1382766.92	818621.04	1369247.84	396466.16
256 bytes	2052004.95	5704795.96	1620242.36	5448530.26	789687.83
1024 bytes	3510987.43	22822030.06	2593168.25	21660819.80	1584114.35
8192 bytes	4584017.72	178479587.17	3279489.71	139663011.53	1501168.99
16384 bytes	4650636.63	357049445.03	3298878.27	315931907.67	1516885.79

表 5.6: MacOS における認証暗号の性能比較結果（AES-NI 有効時）

データサイズ	AES-GCM 128bit	AES-CCM 128bit	AES-GCM 256bit	AES-CCM 256bit	ChaCha20 -Poly1305
16 bytes	484429.89	613620.05	283171.66	466186.58	249966.15
64 bytes	1279874.05	2528114.43	838202.03	1715477.01	493434.09
256 bytes	2557165.37	10134563.21	1454444.63	6611662.08	1017337.62
1024 bytes	4040038.40	40809770.33	2459613.98	29209715.71	1838556.16
8192 bytes	5133650.60	317973793.45	2510285.48	223001859.23	1922640.63
16384 bytes	5194721.96	640941424.64	2528247.81	388032670.38	2008623.79

表 5.7: Linux における認証暗号の性能比較結果（AES-NI 有効時）

データサイズ	AES-GCM 128bit	AES-CCM 128bit	AES-GCM 256bit	AES-CCM 256bit	ChaCha20 -Poly1305
16 bytes	510175.13	608111.52	428784.76	535849.70	241889.67
64 bytes	1188073.26	2438522.67	1131148.69	2213954.71	471620.46
256 bytes	2373195.95	9757960.96	2004236.03	8849291.52	916063.15
1024 bytes	3767685.80	39013930.33	2931220.82	35412411.05	1716506.28
8192 bytes	4724222.63	310425097.56	3431729.83	266724133.55	1825169.41
16384 bytes	4795012.44	623660083.88	3480622.42	566605561.86	1851042.47

表 5.8: Amazon Linux における認証暗号の性能比較結果（AES-NI 有効時）

データサイズ	AES-GCM 128bit	AES-CCM 128bit	AES-GCM 256bit	AES-CCM 256bit	ChaCha20 -Poly1305
16 bytes	126038.59	177002.47	107726.93	141741.60	95781.94
64 bytes	327848.09	710136.00	294067.24	566400.45	182294.44
256 bytes	460328.02	2848054.36	426075.99	2258598.95	357304.56
1024 bytes	527513.94	11375318.36	476462.08	9041514.14	393295.19
8192 bytes	548782.08	90982375.42	490982.06	72403197.95	404491.57
16384 bytes	536155.48	181951129.43	489576.78	145233810.77	407076.86

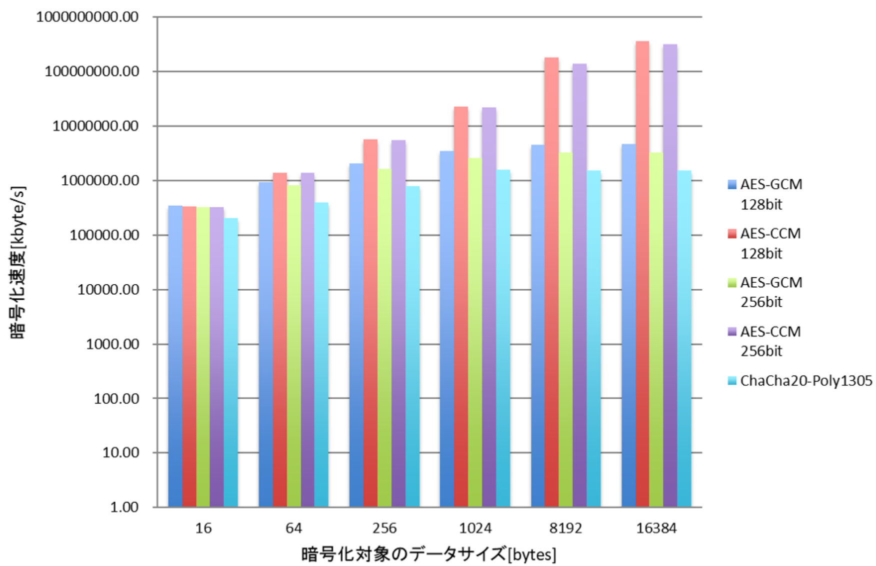

図 5.7: AES-NI 有効時の性能比較結果 (Windows)

表 5.9: Windows における暗号化アルゴリズムの性能比較結果（AES-NI 有効時）

データサイズ	AES-ECB 128bit	AES-ECB 256bit	ChaCha20
16 bytes	721452.09	670536.90	306495.98
64 bytes	2577843.65	2354187.41	553130.61
256 bytes	4150158.40	2955164.04	1198468.01
1024 bytes	4661619.22	3197009.24	2499813.41
8192 bytes	4771662.74	3120849.30	2530989.40
16384 bytes	4842673.49	3188932.61	2582094.40

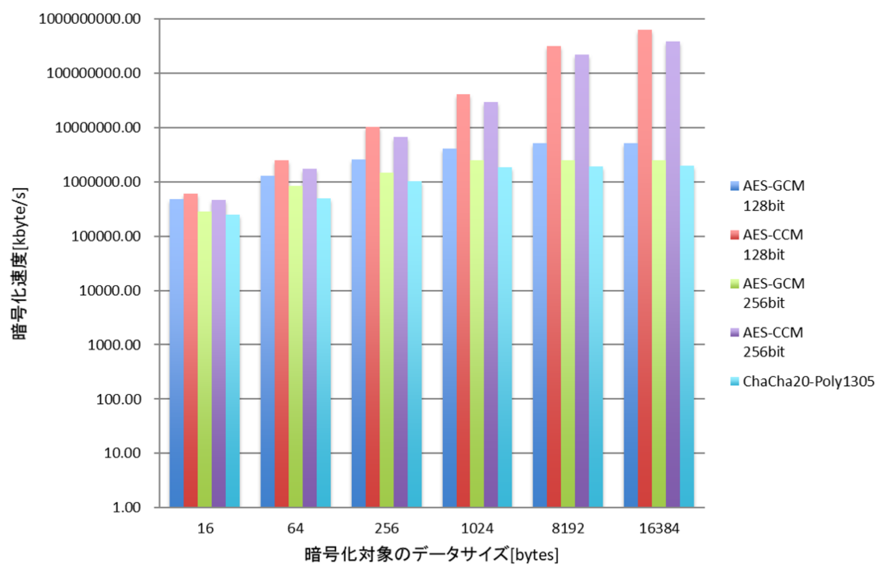

図 5.8: AES-NI 有効時の性能比較結果 (MacOS)

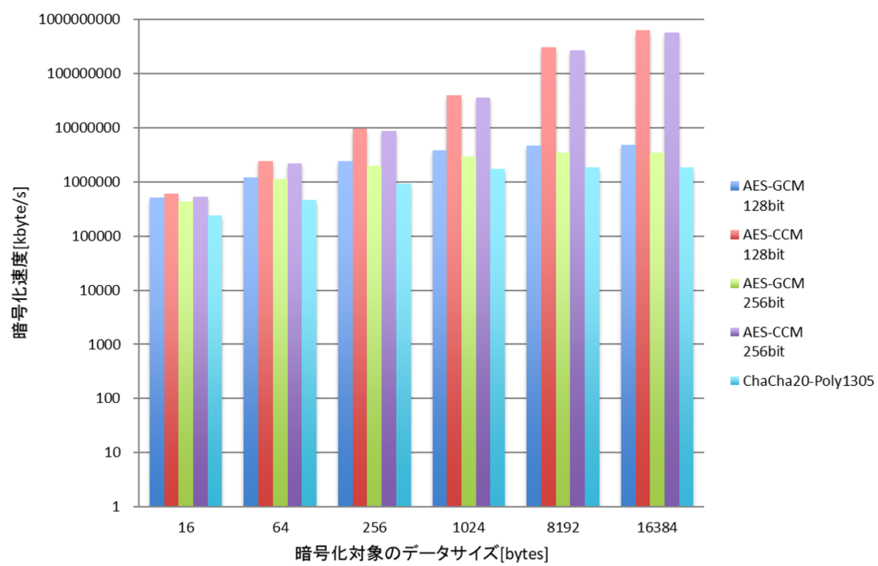

図 5.9: AES-NI 有効時の性能比較結果 (Linux)

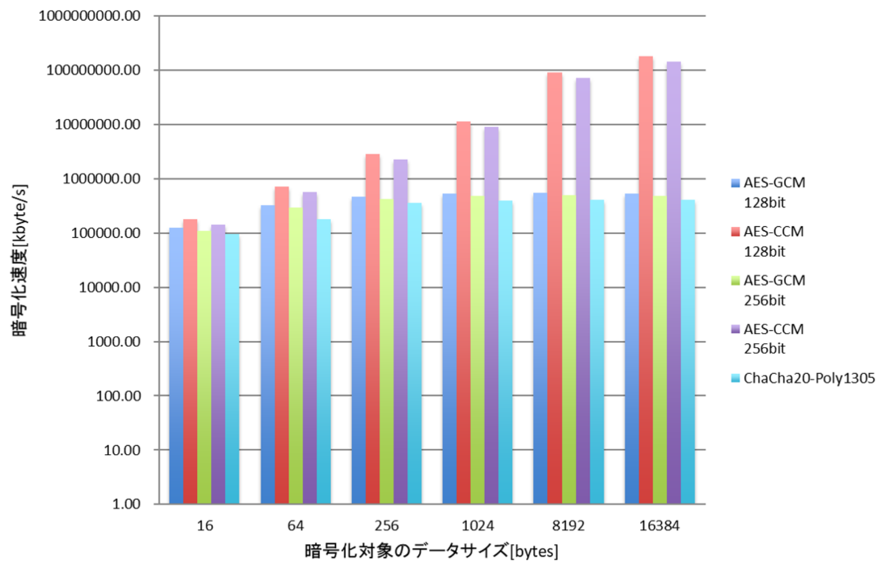

図 5.10: AES-NI 有効時の性能比較結果 (Amazon Linux)

表 5.10: MacOS における暗号化アルゴリズムの性能比較結果（AES-NI 有効時）

データサイズ	AES-ECB 128bit	AES-ECB 256bit	ChaCha20
16 bytes	718493.38	625663.43	341538.18
64 bytes	3247742.44	2796769.56	604936.65
256 bytes	4752086.87	3420986.37	1346618.88
1024 bytes	5192655.87	3704925.55	2657458.26
8192 bytes	5343029.93	3847618.56	2872923.48
16384 bytes	5265643.42	3847787.86	2860145.63

表 5.11: Linux における暗号化アルゴリズムの性能比較結果（AES-NI 有効時）

データサイズ	AES-ECB 128bit	AES-ECB 256bit	ChaCha20
16 bytes	808106.49	715061.93	330001.09
64 bytes	3403812.95	2940045.16	596987.80
256 bytes	4619562.75	3369008.13	1248808.19
1024 bytes	4859779.41	3488602.79	2559582.55
8192 bytes	4914921.47	3515992.75	2656531.80
16384 bytes	4929344.85	3523969.02	2666708.99

表 5.12: Amazon Linux における暗号化アルゴリズムの性能比較結果（AES-NI 有効時）

データサイズ	AES-ECB 128bit	AES-ECB 256bit	ChaCha20
16 bytes	259012.21	191405.61	120775.73
64 bytes	765501.19	606232.62	240767.64
256 bytes	1377031.08	1061144.15	545402.54
1024 bytes	1546464.94	1151635.11	580840.45
8192 bytes	1583736.77	1165607.67	590610.54
16384 bytes	1596866.56	1182924.80	591522.28

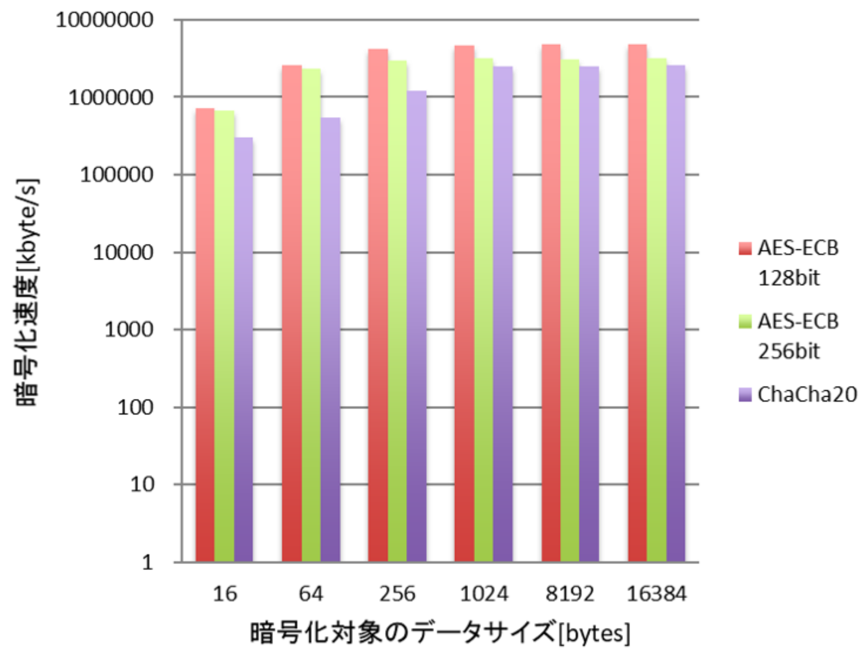

図 5.11: AES-NI 有効時の性能比較結果 (Windows)

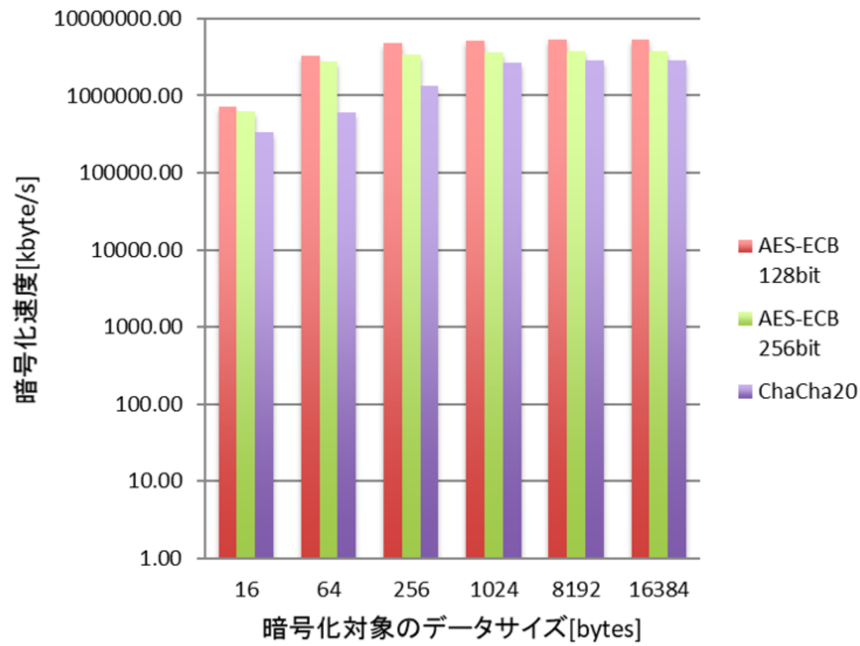

図 5.12: AES-NI 有効時の性能比較結果 (MacOS)

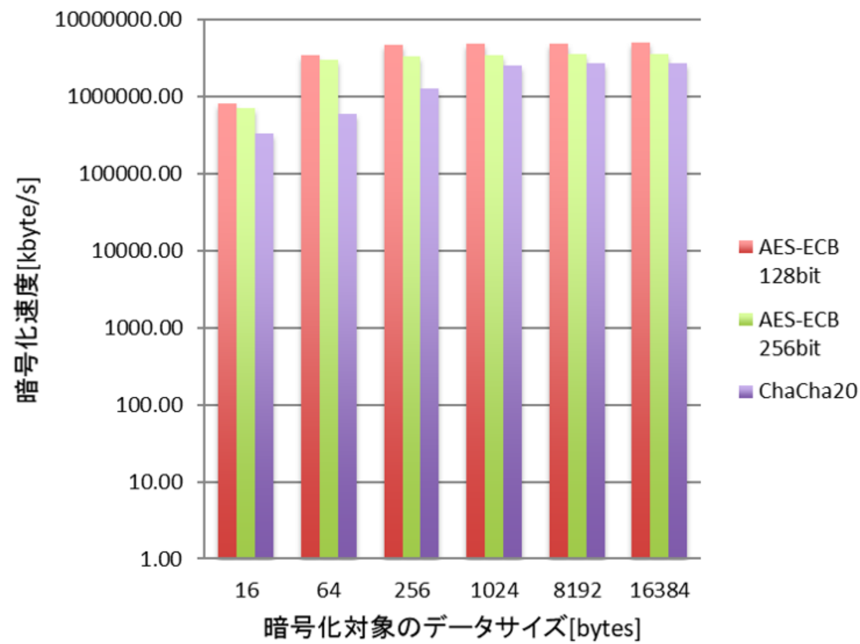

図 5.13: AES-NI 有効時の性能比較結果 (Linux)

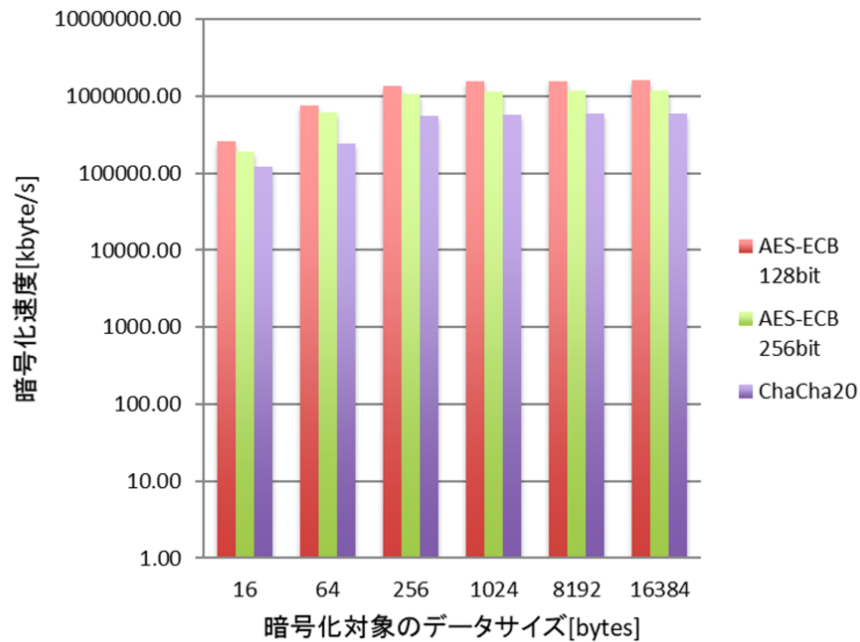

図 5.14: AES-NI 有効時の性能比較結果 (Amazon Linux)

表 5.13: Windows における認証暗号の性能比較結果（AES-NI 無効時）

データサイズ	AES-GCM 128bit	AES-CCM 128bit	AES-GCM 256bit	AES-CCM 256bit	ChaCha20 -Poly1305
16 bytes	74992.24	184308.92	62806.59	151690.23	207645.95
64 bytes	88992.29	736739.64	75086.25	608918.54	412952.27
256 bytes	205170.69	2958477.31	183246.51	2456274.94	804276.48
1024 bytes	224038.06	11827934.68	194865.39	9885726.38	913482.04
8192 bytes	234830.05	92428320.77	197692.07	78478601.13	950241.96
16384 bytes	238660.27	186632039.08	200856.92	154689901.91	944756.05

表 5.14: MacOS における認証暗号の性能比較結果（AES-NI 無効時）

データサイズ	AES-GCM 128bit	AES-CCM 128bit	AES-GCM 256bit	AES-CCM 256bit	ChaCha20 -Poly1305
16 bytes	82281.26	234343.81	48848.18	151794.47	238310.98
64 bytes	97496.19	940038.78	59834.20	667101.87	459198.06
256 bytes	219546.45	3710785.11	154244.34	2566293.69	851702.22
1024 bytes	231688.68	14482764.37	184312.15	10861402.11	929472.85
8192 bytes	230962.52	119366934.53	184287.23	77965429.42	944772.44
16384 bytes	240162.13	238576222.21	162141.53	138127780.52	963521.19

表 5.15: Linux における認証暗号の性能比較結果（AES-NI 無効時）

データサイズ	AES-GCM 128bit	AES-CCM 128bit	AES-GCM 256bit	AES-CCM 256bit	ChaCha20 -Poly1305
16 bytes	84327.32	221654.70	68747.66	173721.25	241264.39
64 bytes	98523.65	887972.20	76813.31	697759.83	460033.79
256 bytes	221492.39	3554475.52	187173.03	2798027.18	845216.94
1024 bytes	235878.06	14200349.01	200182.44	11171555.33	929916.93
8192 bytes	237016.41	113009104.21	203634.01	88966176.77	953292.12
16384 bytes	239697.92	227576113.83	203696.81	178625265.66	944701.44

表 5.16: Amazon Linux における認証暗号の性能比較結果（AES-NI 無効時）

データサイズ	AES-GCM 128bit	AES-CCM 128bit	AES-GCM 256bit	AES-CCM 256bit	ChaCha20 -Poly1305
16 bytes	34329.14	99995.46	27475.16	79140.51	95851.95
64 bytes	40436.57	401854.59	31699.75	317728.62	181693.00
256 bytes	96728.32	1597726.89	81618.16	1269617.49	359539.46
1024 bytes	103351.54	6417883.14	88686.25	5077132.63	395694.76
8192 bytes	105526.61	51319529.47	90537.98	40686116.86	405796.18
16384 bytes	105103.36	102071737.58	90734.59	81282542.65	407213.40

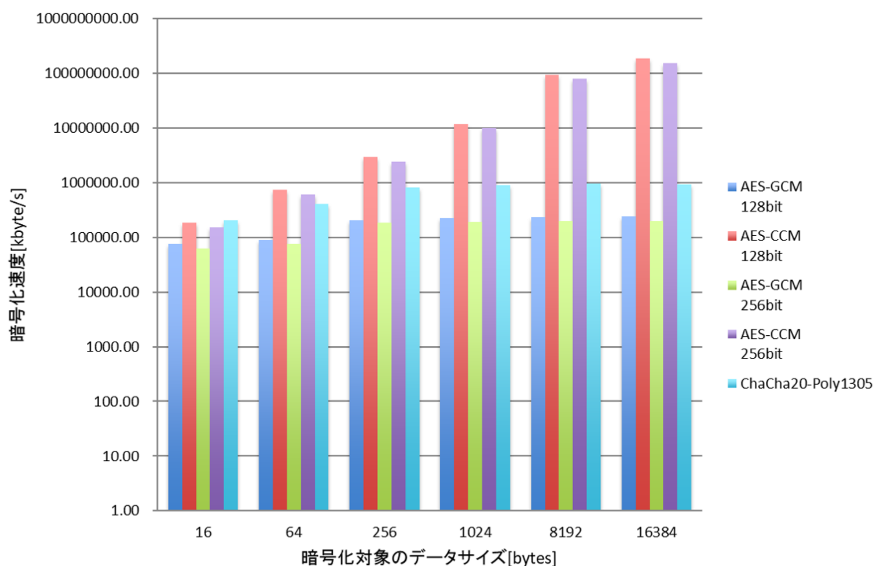

図 5.15: AES-NI 無効時の性能比較結果 (Windows)

5.3.2 AES-NI 無効の場合

AES-NI を無効にした場合の性能比較結果のグラフを図 5.15, 5.16, 5.17, 5.18 に示す。

グラフより、AES-NI が無効の環境では、どのプラットフォームにおいても ChaCha20-Poly1305 のスループットが AES-GCM のスループットより大きいことが分かる。これにより、AES-NI が利用できない環境において ChaCha20-Poly1305 が高速に実行可能であることが確認できた。

また、暗号化アルゴリズムである AES-ECB (128 ビット/256 ビット) および ChaCha20 の性能を表 5.17, 5.18, 5.19, 5.20 に示す。

表 5.17: Windows における暗号化アルゴリズムの性能比較結果（AES-NI 無効時）

データサイズ	AES-ECB 128bit	AES-ECB 256bit	ChaCha20
16 bytes	241695.97	184421.71	305343.17
64 bytes	290278.41	217672.88	575569.24
256 bytes	305992.86	226239.91	1212559.00
1024 bytes	318974.98	228838.60	1310936.53
8192 bytes	322458.97	230300.46	1338326.74
16384 bytes	323518.46	227472.53	1340947.63

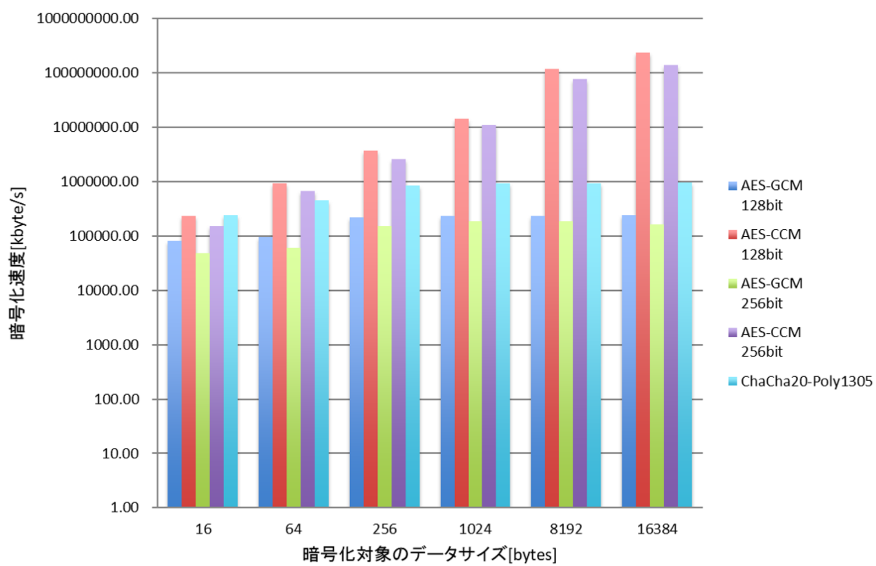

図 5.16: AES-NI 無効時の性能比較結果 (MacOS)

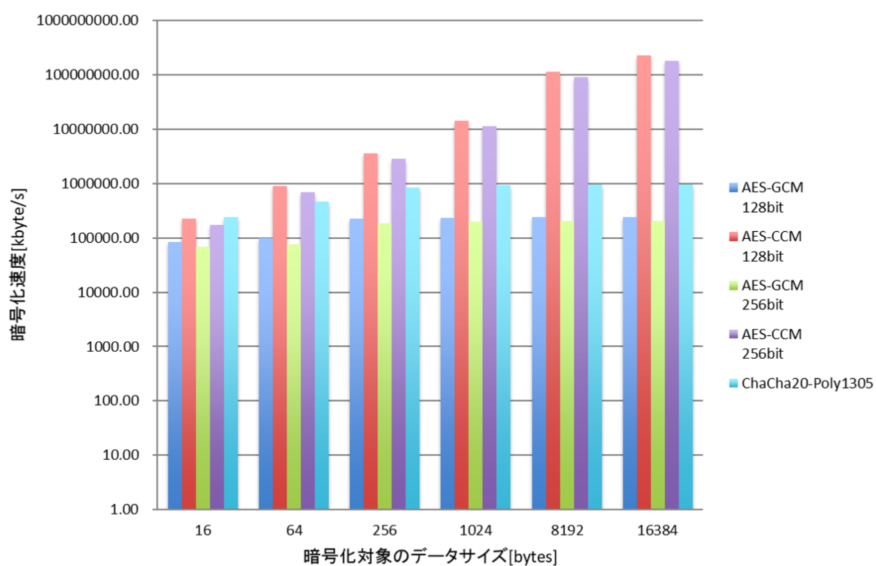

図 5.17: AES-NI 無効時の性能比較結果 (Linux)

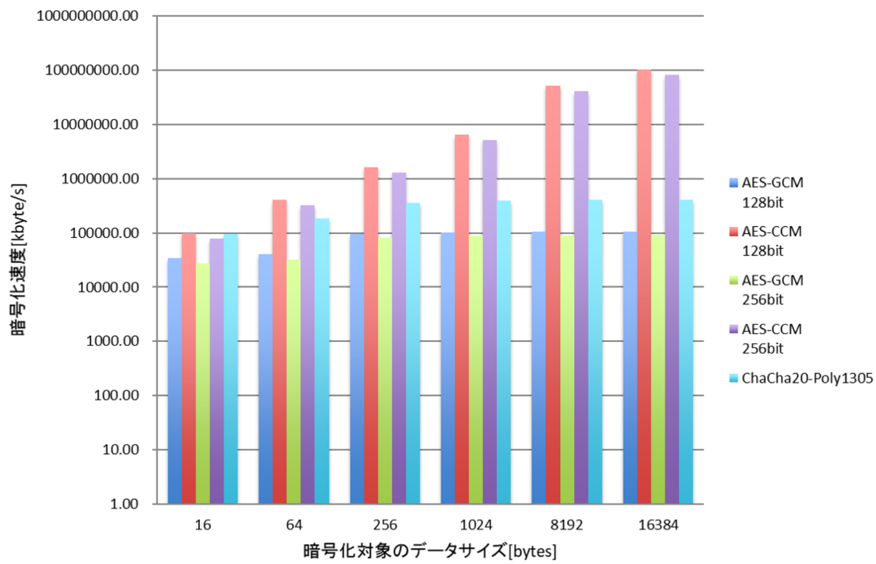

図 5.18: AES-NI 無効時の性能比較結果 (Amazon Linux)

表 5.18: MacOS における暗号化アルゴリズムの性能比較結果（AES-NI 無効時）

データサイズ	AES-ECB 128bit	AES-ECB 256bit	ChaCha20
16 bytes	246222.38	180807.01	339680.79
64 bytes	317403.30	212682.20	630984.13
256 bytes	354566.31	251819.95	1374155.78
1024 bytes	360030.92	253260.37	1456034.10
8192 bytes	359661.57	255306.41	1488565.59
16384 bytes	357373.27	255791.80	1500747.09

表 5.19: Linux における暗号化アルゴリズムの性能比較結果（AES-NI 無効時）

データサイズ	AES-ECB 128bit	AES-ECB 256bit	ChaCha20
16 bytes	284800.45	211349.85	330452.10
64 bytes	321965.70	231066.03	598775.25
256 bytes	329079.55	220908.80	1254947.58
1024 bytes	334208.68	226405.72	1326326.10
8192 bytes	335527.94	237177.51	1356013.57
16384 bytes	335473.32	236754.26	1357916.84

表 5.20: Amazon Linux における暗号化アルゴリズムの性能比較結果（AES-NI 無効時）

データサイズ	AES-ECB 128bit	AES-ECB 256bit	ChaCha20
16 bytes	120089.54	89603.25	120495.06
64 bytes	141207.15	102201.79	240215.04
256 bytes	145135.96	104110.95	543566.76
1024 bytes	148468.39	106945.54	580691.35
8192 bytes	148933.29	106870.10	590919.00
16384 bytes	149007.31	107255.13	591238.49

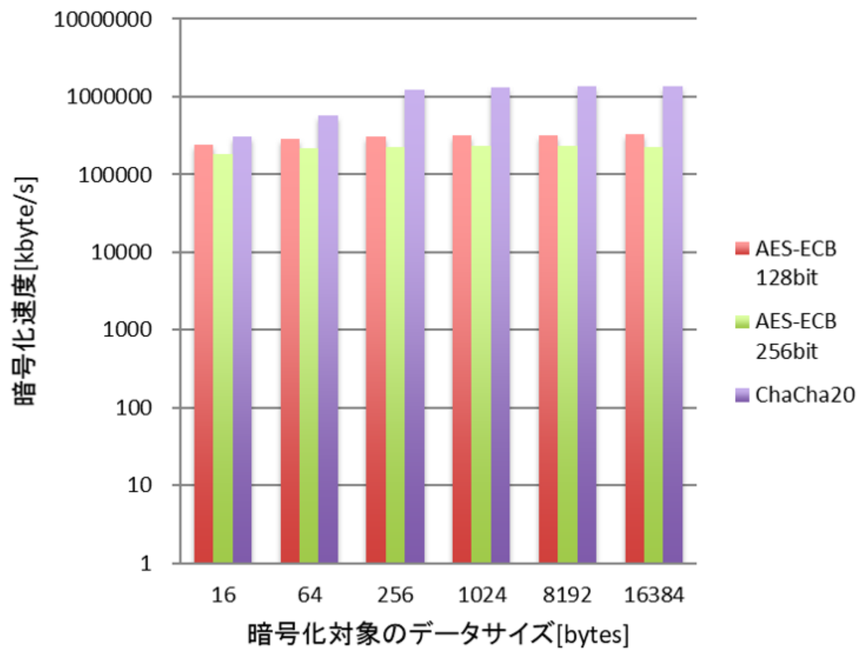

図 5.19: AES-NI 無効時の性能比較結果 (Windows)

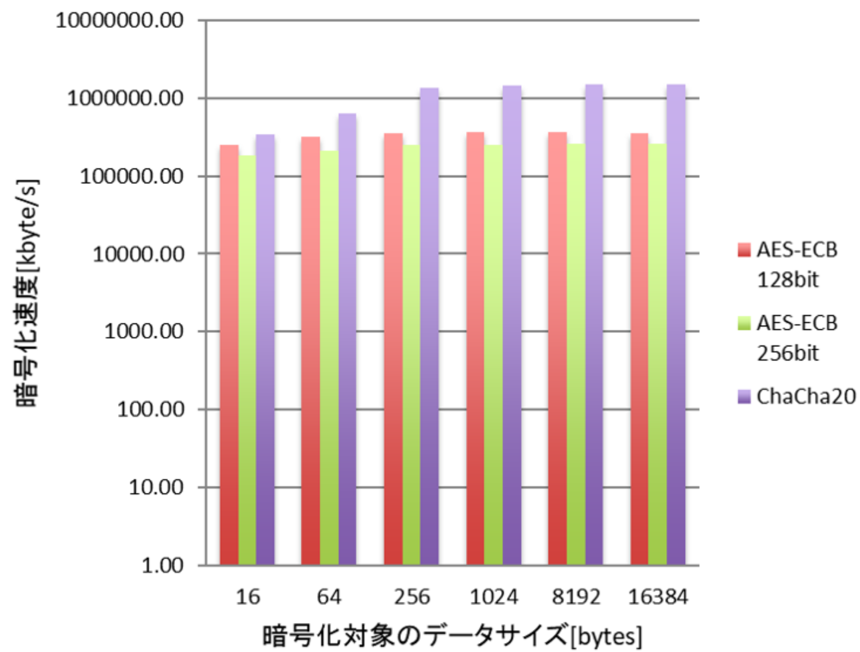

図 5.20: AES-NI 無効時の性能比較結果 (MacOS)

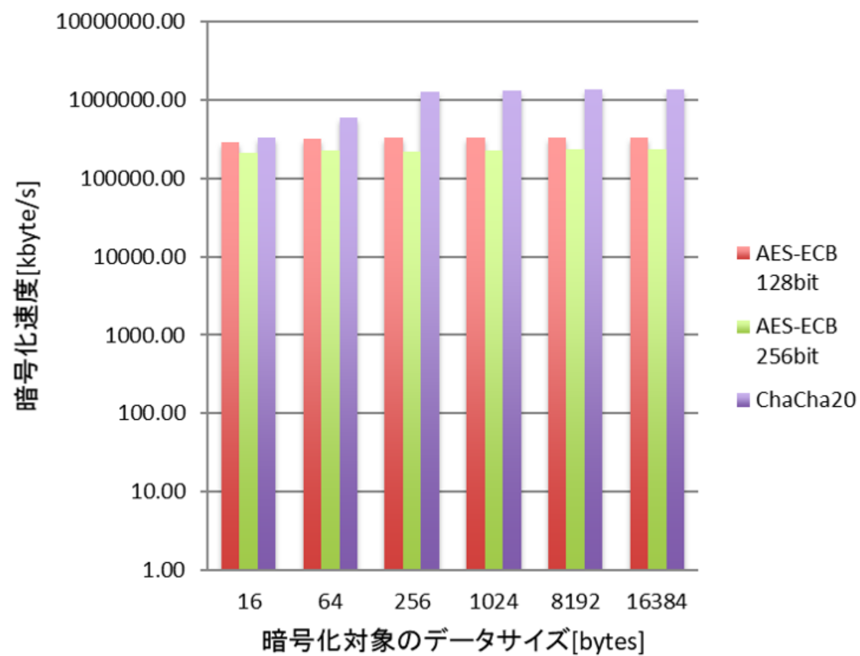

図 5.21: AES-NI 無効時の性能比較結果 (Linux)

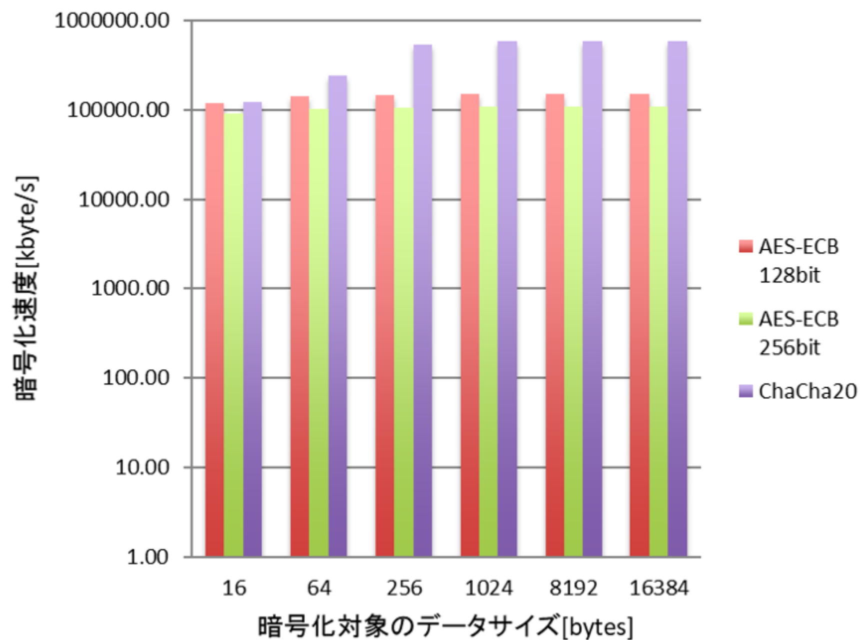

図 5.22: AES-NI 無効時の性能比較結果 (Amazon Linux)

5.3.3 性能測定における注意点

5.3.1 節および 5.3.2 節に示した性能測定結果において、ChaCha20-Poly1305 と AES-GCM が処理対象のサイズを増加させるとスループットが頭打ちとなるのに対し、AES-CCM のスループットは処理対象のサイズに対して単調増加する傾向が観測された。

この現象の原因について OpenSSL 1.1.0f のソースコードを調査した結果、以下のことが判明した。

- OpenSSL の暗号化処理は一般的に、初期化 (Init)、暗号化 (Update)、終期化 (Final) で行われる。
- OpenSSL の speed コマンドでは、指定された暗号アルゴリズムに対してまず初期化 (Init) を行い、一定時間内に暗号化 (Update) を繰り返し実行し、最後に終期化 (Final) を実行している。
- しかし、OpenSSL における CCM モードの実装では、初期化 (Init) に対して 暗号化 (Update) が 1 回しか実行できないようになっていた。そのため、2 回目以降の暗号化 (Update) が実行されないため、スループットの増加を招く。

表 5.21: MacOS における認証暗号の性能比較結果（AES-NI 有効時）

データサイズ	AES-GCM 128bit	AES-CCM 128bit	AES-GCM 256bit	AES-CCM 256bit	ChaCha20 -Poly1305
16 bytes	465485.51	117009.03	413062.50	112885.9	245685.01
64 bytes	1172012.91	385554.24	1124296.98	330171.75	471882.18
256 bytes	2385178.11	761944.32	2046490.03	623611.24	909457.95
1024 bytes	3844752.38	1051821.74	3020038.49	799936.17	1666071.21
8192 bytes	4744915.63	1196976.81	3559775.09	869780.14	1887357.61
16384 bytes	4861878.52	1198019.93	3586643.29	869433.34	1934491.65

これに対し 2017 年 10 月 7 日、OpenSSL の GitHub において、speed コマンドの実装である speed.c の修正が提案され、2017 年 10 月 10 日に OpenSSL の master ブランチに修正が反映された[2]。この修正は、CCM モードの実行時のみ初期化 (Init) - 暗号化 (Update) - 終期化 (Final) のサイクルで実行するようにしたものである。この修正により、CCM モードにおいても、処理対象のサイズを増加させるとスループットが頭打ちとなる結果が得られるようになった。修正された speed コマンドを用いて AES-CCM の性能を測定した結果を表 5.21 に示す（測定は MacOS 上で行った）。

ただし、この修正では CCM モードと他の暗号アルゴリズムの実行方法が異なっており、暗号アルゴリズムの性能比較という意味では公平でない実装となっている。そのため、CCM モードを含む暗号アルゴリズムの性能比較を公平に行うためには、CCM モードの実装そのものを修正する必要がある。

本調査における性能測定では、上記の speed.c に対する修正は採用せず、speed コマンドによる測定を実施した他の文献と結果を比較できるようにした。

5.4 考察

本章では、5.3 章の結果に対する考察を与える。

5.4.1 Poly1305 によるオーバーヘッド

ChaCha20-Poly1305 のスループットは、ChaCha20 単体のスループットの約 0.7 倍である。これにより、Poly1305 を加えることによりスルー

[2] OpenSSL GitHub に Pull request が送信された。https://github.com/openssl/openssl/pull/4480

プットが約 3 割減少することがわかる。

5.4.2 AES-NI の影響

AES-NI 有効時、Intel Core i7 で実行した AES-GCM のスループットは AES-NI 無効時の 12〜15 倍になる。ChaCha20-Poly1305 は、256 バイト以下のデータサイズでは AES-NI 有効時にスループットが低下するが、256 バイトを超えるデータサイズでは AES-NI 有効時のスループットが 1.6〜2.0 倍になる。

他方、AES-NI 有効時、Intel Xeon で実行した AES-GCM のスループットは AES-NI 無効時の約 5 倍になる。ただし、ChaCha20-Poly1305 のスループットはデータサイズによらずほとんど変わらない。

5.4.3 ChaCha20-Poly1305 と AES-GCM の比較

AES-NI 有効時 AES-GCM 128bit は、Windows, MacOS および Linux において ChaCha20-Poly1305 の 2.5〜3.0 倍、Amazon Linux において 1.4 倍のスループットを示す。AES-GCM 256bit は、Windows, MacOS および Linux において ChaCha20-Poly1305 の 1.7〜2.3 倍、Amazon Linux において 1.3 倍のスループットを示す。AES-NI 有効時のスループットに差が出た理由は、Windows, MacOS および Linux がデュアルコア CPU を使用しているのに対し、Amazon Linux ではシングルコア CPU を使用しているためと考えられる。

また AES-NI 無効時、いずれのプラットフォームにおいても、AES-GCM 128bit は ChaCha20-Poly1305 の約 0.3 倍、AES-GCM 256bit は ChaCha20-Poly1305 の約 0.2 倍のスループットを示す。

以上より、AES-NI 無効時には AES-GCM より ChaCha20-Poly1305 の方が高速に動作することが分かる。

第6章　まとめ

本調査では、認証暗号の一種である ChaCha20-Poly1305 について、実際に利用する観点で、標準化状況や OSS コミュニティでの採用状況を踏まえて、実装性能評価を実施した。

ChaCha20-Poly1305 は特に IETF で積極的に標準化が進められており、今後も幅広いプロトコルへの適用が予想される。また各種 OSS での採用も広まっており、ChaCha20-Poly1305 の利用環境は整いつつあると言える。

ChaCha20-Poly1305 の実装性能を評価した論文はまだ少ない。ベンチマークとしてソフトウェア実装やハードウェア実装が公開されており、ベンチマーク結果において今後組み込み環境において ChaCha20-Poly1305 の性能が向上することが示唆されている。

世界中で広く利用されている OpenSSL を用いて ChaCha20-Poly1305 と他の認証暗号との性能比較を行った。AES-NI を有効にした Intel Core i7 において、ChaCha20-Poly1305 のスループットは、鍵長以下のデータサイズでは AES-NI 無効時より小さく、鍵長を超えるデータサイズでは大きくなることがわかった。また AES-NI を有効にした Intel Xeon においては、ChaCha20-Poly1305 のスループットはデータサイズによらずほとんど変わらないことがわかった。AES-NI 有効時および AES-NI 無効時において AES-GCM と ChaCha20-Poly1305 のスループットを比較すると、AES-NI 無効時に ChaCha20-Poly1305 の実行速度が AES-GCM を上回ることがわかった。

参考文献

[1] KDDI Research, Inc. Security Analysis of ChaCha20-Poly1305 AEAD. Technical report, CRYPTREC, 2016.

[2] Intel®Advanced Encryption Standard Instructions (AES-NI). https://software.intel.com/en-us/articles/intel-advanced-encryption-standard-instructions-aes-ni/.

[3] Barton Gellman and Laura Poitras. U.S., British intelligence mining data from nine U.S. Internet companies in broad secret program, 6 2013.

[4] James Ball and Dominic Rushe. NSA Prism program taps in to user data of Apple, Google and others. https://www.theguardian.com/world/2013/jun/06/us-tech-giants-nsa-data, 6 2013.

[5] Nicole Perlroth. Government Announces Steps to Restore Confidence on Encryption Standards, 9 2013.

[6] D. J. Bernstein. The Salsa20 family of stream ciphers, 2007.

[7] D. J. Bernstein. ChaCha, a variant of Salsa20, 2008.

[8] D. J. Bernstein. The Poly1305-AES Message-Authentication Code. In *Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*, pp. 32–49, 2005.

[9] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. RFC 7539, May 2015.

[10] Yoav Nir. ChaCha20, Poly1305, and Their Use in the Internet Key Exchange Protocol (IKE) and IPsec. RFC 7634, August 2015.

[11] Adam Langley, Wan-Teh Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS). RFC 7905, June 2016.

[12] Russ Housley. Using ChaCha20-Poly1305 Authenticated Encryption in the Cryptographic Message Syntax (CMS). RFC 8103, February 2017.

[13] Yoav Nir and Adam Langley. ChaCha20 and Poly1305 for IETF Protocols. Internet-Draft draft-nir-cfrg-rfc7539bis-02, Internet Engineering Task Force, 2017. Work in Progress.

[14] Internet Key Exchange Version 2 (IKEv2) Parameters. `https://www.iana.org/assignments/ikev2-parameters/ikev2-parameters.xhtml`, 2017.

[15] Structure of Management Information (SMI) Numbers (MIB Module Registrations). `https://www.iana.org/assignments/smi-numbers/smi-numbers.xhtml`, 2017.

[16] CURves, Deprecating and a Little more Encryption. `https://datatracker.ietf.org/wg/curdle/about/`.

[17] ISO/IEC JTC 1/SC 27 - IT Security techniques. `https://www.iso.org/committee/45306.html`, 2007.

[18] ITU-T SG17: Security. `http://www.itu.int/en/ITU-T/studygroups/2013-2016/17/Pages/default.aspx`.

[19] World Wide Web Consortium (W3C). `https://www.w3.org/`.

[20] OpenSSL. `https://www.openssl.org/`.

[21] The Sodium crypt library (libsodium). `https://www.gitbook.com/book/jedisct1/libsodium/details`.

[22] LibreSSL. `https://www.libressl.org/`.

[23] GnuTLS. `http://www.gnutls.org/`.

[24] BoringSSL. `https://boringssl.googlesource.com/boringssl/`.

[25] BouncyCastle. `https://www.bouncycastle.org/`.

[26] Linux Kernel. `https://www.kernel.org/`.

[27] NSS. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS`.

[28] wolfSSL. `https://www.wolfssl.com/`.

[29] Fabrizio De Santis, Andreas Schauer, and Georg Sigl. ChaCha20-Poly1305 authenticated encryption for high-speed embedded IoT applications. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pp. 692–697, 2017.

[30] Vlad Krasnov. It takes two to ChaCha (Poly). `https://blog.cloudflare.com/it-takes-two-to-chacha-poly/`.

[31] Jason A. Donenfeld. WireGuard: Next Generation Kernel Network Tunnel. Technical report.

[32] Cryptography IP cores – Chacha20-Poly1305. `https://www.barco-silex.com/products/security/chacha20-poly1305-ip-core/`.

[33] ChaCha-IP-13 / EIP-13 ChaCha20 accelerators. `https://www.insidesecure.com/jp/Products/Silicon-IP/Cipher-Accelerators/ChaCha-IP-13`.

[34] POLY-IP-53 / EIP-53 Poly1305-based MAC accelerators. `https://www.insidesecure.com/jp/Products/Silicon-IP/Hash-and-HMAC-Accelerators/POLY-IP-53`.

[35] WireGuard Benchmarks. `https://www.wireguard.com/performance/`.

[36] Cryptography IP cores – Scalable ARS-GCM/GMAC/CTR. `https://www.barco-silex.com/products/security/high-speed-aes/`.

[37] AES-IP-38 / EIP-38 AES XTS/GCM accelerators. `https://www.insidesecure.com/jp/Products/Silicon-IP/Cipher-Accelerators/AES-IP-38`.

[38] AES-IP-39 / EIP-39 AES "all modes" accelerators. `https://www.insidesecure.com/jp/Products/Silicon-IP/Cipher-Accelerators/AES-IP-39`.

[39] AES-IP-61 / EIP-61 High speed low latency AES-GCM pipeline, 100Gbps. `https://www.insidesecure.com/jp/Products/Silicon-IP/Complex-Cryptographic-Accelerators/AES-IP-61`.

[40] Morris J. Dworkin. SP 800-38C. Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality. Technical report, Gaithersburg, MD, United States, 2004.

[41] Morris J. Dworkin. SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC. Technical report, Gaithersburg, MD, United States, 2007.