

デジタル署名アルゴリズム EdDSA の実装性能調査

株式会社インフォーズ
2021 年 12 月

エグゼクティブサマリー

本報告書はデジタル署名 Edwards-curve Digital Signature Algorithm (EdDSA) の実装性能に関する評価結果を報告するものである。

EdDSA とは、2011 年 9 月に Daniel J. Bernstein らによって “High-speed high-security signatures” [1] で発表され、Internet Research Task Force (IRTF) の Request for Comments (RFC) 8032 [2] で規定されるデジタル署名のことであり、有限体上の twisted Edwards 曲線 [3] [4] といわれる楕円曲線上の Schnorr 署名 [5] [6] の署名内部乱数（ノンス）を署名者の秘密情報と署名されるメッセージのハッシュ値に置き換えた確定的（deterministic）版の Schnorr 署名である。EdDSA で推奨される twisted Edwards 曲線は IRTF の RFC7748 [7] で規定されるものであり edwards25519, edwards448 と記述される。

以下、評価結果の概要を述べる。

- EdDSA の技術的な特徴として以下を確認できた。
 - EdDSA は ECDSA と同様に射影座標系を使用可能であるため、有限体上の乗法逆元の演算を用いずに演算可能であることから、高速な実装が可能である。
 - EdDSA で推奨される twisted-Edwards 曲線は無限遠点を例外的に扱うための分岐が不要なため、ECDSA で使われる Weierstrass 形式の楕円曲線と比較して実装しやすい。
 - EdDSA で推奨される Edwards 楕円曲線は、異なる点の加算と同一の点の加算 (2 倍算) を同じ処理で計算できるため、実装するアルゴリズムが 1 つでよく、点が異なるかどうかの条件分岐を考慮する必要がないため、Weierstrass 形式の楕円曲線と比較して Side Channel Attack (SCA) 対策に有利である。
- EdDSA の標準化での採用状況としては、RFC8032 [2] として標準化され、Draft 版を含め参照する標準も増えつつあることが分かった。Open Source Software (OSS) での採用状況としては、少なくとも 20 件の OSS の暗号ライブラリが採用していることが分かった。ただし、224 ビットセキュリティの EdDSA である Ed448 は 128 ビットセキュリティの EdDSA である Ed25519 よりも採用数は少なかった。今後、EdDSA の実装の品質や高速化技法が洗練され、安全性の高い Ed448 も性能が高まると採用が進むことが期待できる。
- 公開されている第三者の測定結果および本調査での実装評価として、4 つの C 言語実装および 2 つの Java 言語実装の OSS に対して、Windows, Linux, Mac の環境でそれぞれでの性能測定を実施した。その結果は以下のとおりである。
 - Ed25519

- 鍵ペア生成
 - ほぼ全ての環境において ECDSA P-256 より高スループットであった.
- 署名生成
 - 半数以上の環境で ECDSA P-256 より高スループットであり, それ以外の環境において半分以下のスループットとなることはなかった.
- 署名検証
 - ほぼ全ての ECDSA P-256 より高スループットであった.

➤ Ed448

- 鍵ペア生成
 - ほぼ全ての環境で ECDSA P-384 より高スループットであり, 全環境で ECDSA P-521 より高スループットであった.
- 署名生成
 - ほぼ全ての環境で ECDSA P-384 より高スループットであり, 全環境で ECDSA P-521 より高スループットであった.
- 署名検証
 - ほぼ全ての環境で ECDSA P-384 より高スループットであり, 全環境で ECDSA P-521 より高スループットであった.

以上から, EdDSA は, 方式そのものに安全性担保, 高速化, 実装がしやすい特性があると評価でき, 実際に OSS として実装された暗号ライブラリの性能は, 同程度の安全性の ECDSA や高い安全性の ECDSA と比べても遜色のないものであった. 実装された OSS の品質, 高速化技法は洗練されているとはいき切れない部分もあるが, 歴史のある ECDSA と比べても性能で劣ることはないと考えられる.

目次

1. はじめに	1
2. 数学的記法	3
3. EdDSA の技術的解説.....	4
3.1. EdDSA の処理内容の解説.....	4
3.1.1. データ変換.....	5
3.1.2. Ed25519, Ed25519ctx, Ed25519ph 共通パラメータ	7
3.1.3. Ed25519, Ed25519ctx, Ed25519ph 共通処理	7
3.1.4. Ed25519.....	9
3.1.5. Ed25519ctx.....	10
3.1.6. Ed25519ph.....	11
3.1.7. Ed448, Ed448ph 共通パラメータ	12
3.1.8. Ed448, Ed448ph 共通処理	12
3.1.9. Ed448.....	14
3.1.10. Ed448ph.....	15
3.2. 実装の解説	16
3.2.1. Edwards 曲線の特徴	16
3.2.2. EdDSA の特徴	17
3.2.3. RFC8032 に記載されている注意点	20
3.2.4. SCA 対策	21
3.2.5. 長いメッセージに対する配慮.....	22
3.2.6. パラメータ n の選択.....	22
3.3. Ed25519 実装の解説.....	23
3.4. Ed448 実装の解説.....	25
4. EdDSA の標準化動向や OSS に関する採用状況など.....	28
4.1. EdDSA に関する標準化動向調査.....	28
4.2. 標準仕様における EdDSA に関する採用状況	30
4.3. EdDSA に関する OSS 採用状況.....	36
4.4. EdDSA 実装の脆弱性に関する状況.....	40
5. 現在の電子政府推奨暗号リストに含まれる同技術分類の方式 (ECDSA) との性能比較調査	43
5.1. 性能比較調査について	43
5.2. ECDSA および EdDSA が実装されている OSS の選定.....	43
5.3. 第三者による性能比較	44
5.3.1. eBATS による測定結果	44
5.3.2. 論文 “Is there a case to prefer Ed25519 over ECDSA P-256 for DNSSEC?” での測定結果	55
5.3.3. 論文 “Size, Speed, and Security: An Ed25519 Case Study” での測定結果	57

5. 4.	ECDSA および EdDSA に関する性能評価.....	58
5. 4. 1.	性能評価対象.....	58
5. 4. 2.	性能測定方法.....	58
5. 4. 3.	性能測定結果と評価.....	94
6.	EdDSA に関する実装評価まとめ.....	106
7.	参考文献.....	108

1. はじめに

デジタル署名方式である Edwards-curve Digital Signature Algorithm (EdDSA) とは、2011 年に Daniel J. Bernstein らによって “High-speed high-security signatures” として発表され [1], Internet Research Task Force (IRTF) の Request for Comments (RFC) 8032 “Edwards-Curve Digital Signature Algorithm (EdDSA)” [2] で標準化されたアルゴリズムである。

EdDSA については、現在、CRYPTREC において 2020 年度 CRYPTREC 外部評価報告として、「デジタル署名 EdDSA の構成の安全性に関する調査および評価」 [8], 「デジタル署名 EdDSA で使われている曲線の安全性に関する調査及び評価」 [9] および 「CRYPTREC Review of EdDSA」 [10] として理論的な側面に関する評価結果が公開されている。

[8] は、EdDSA の構成の安全性について「証明可能安全性という枠組みでは不十分であったり十分なビット安全性が保証されなかったりするが、Schnorr 署名という成熟した方式をもとにノンスの生成で既存の攻撃を注意深く回避する配慮がされており現実的には安全である」と評価している。

[9] は、EdDSA に対する現時点で最良の攻撃は ρ 法 [11] としており、EdDSA で使われている曲線の安全性について、Curve25519 は「 $2^{125.8257}$ 回の点の加算が必要」、Curve448 は「 $2^{222.8257}$ 回の点の加算が必要」であり、「Curve25519 における ECDLP を攻撃するには、少なくとも $2^{125.8257}$ 回の楕円加算が必要であり、ほぼ 128 ビットのセキュリティレベルを持つと結論付けられる」と評価している。

[10] は、EdDSA について「I am confident that EdDSA using Curve25519 is a good signature scheme for use up to 2030.」として評価している。

本調査報告書では、理論的な面での評価が行われている EdDSA に対して、実社会で利用するために必要とされる標準化動向や手軽に利用するための環境としての OSS に関する状況を把握するための実装性能評価を行いその結果を報告する。

本報告書で対象となっている EdDSA は、IETF において RFC8032 “Edwards-Curve Digital Signature Algorithm (EdDSA)” [2] として 2021 年 1 月に RFC として発行されたデジタル署名方式である。これは、twisted Edwards 曲線 [7] と言われる有限体上の楕円曲線表現を用いた Schnorr 署名 [5] の署名内部乱数 (ノンス) を署名者の秘密情報と署名される平文のハッシュ値に置き換えた確定的 (Deterministic) な Schnorr 署名である。また、EdDSA で使用される (twisted) Edwards 曲線については、RFC8032 に先駆けて 2015 年 10 月に発行された RFC7748 “Elliptic Curves for Security” [7] において、edwards25519 および edwards448 として規定されている¹。

¹ [6] は、Montgomery 型曲線 “curve25519” とその双有理的同値な twisted Edwards 曲線 “ed25519”, Montgomery 型曲線 “curve448” とその双有理的同値な Edwards 曲線 “ed448” を定義している。

2019 年 11 月に National Institute of Standards and Technology (NIST) は米国の連邦情報処理標準におけるデジタル署名の改定版 Federal Information Processing Standards (FIPS) Pub 186-5 (Draft) Digital Signature Standard (DSS) [12] で EdDSA を追加した。また、EdDSA は RFC8446 で TLS1.3 に採用された。

現在広く利用されている ECDSA に対して、今後、EdDSA の利用が進むと予想される状況の中、CRYPTOREC では上記の外部評価を受けて、“暗号技術検討会 2020 年度報告書”において、Curve25519 の場合はほぼ 128 ビットセキュリティ、Curve448 の場合はほぼ 224 ビットセキュリティの安全性を持ち、現実的な脅威に結びつくような脆弱性、安全性に問題を与える点はないと判断し、CRYPTREC 暗号リストへの追加を視野に入れた検討として、実装性能評価を行うことを報告している。

本報告書の構成について以下のとおりである。2 章で必要な記法の準備を行い、3 章で EdDSA の仕様および実装について記述する。4 章で EdDSA が標準化やオープンソースソフトウェアでどのように採用されているかを記述し、5 章で EdDSA と ECDSA の実測した性能を比較した結果を記述し、6 章で 5 章までの調査結果を踏まえて、EdDSA の実装性能についてまとめを記述する。

なお、本報告書は、2021 年 12 月時点での情報に基づき記載している。

2. 数学的記法

本報告書で用いる数学的記法を準備する.

オクテット列	0 から 255 までの数値の並び.
p	基礎となる体を定義する素数.
$\text{GF}(p)$	p 個の元の有限体.
x^y	x の y 乗.
B	楕円曲線上の点による群または部分群の生成元.
$[n]X$	X を n 回加算したもの.
$h[i]$	オクテット列 h の i 番目のオクテット.
h_i	h の i 番目のビット.
$a \parallel b$	(ビット) 列 a の後ろに (ビット) 列 b を連結したもの.
$a \leq b$	a は b 以下.
$a \geq b$	a は b 以上.
$i + j$	i と j の和.
$i * j$	i と j の積.
$i - j$	i から j を引いた差.
i/j	i を j で割る除算.
$i \times j$	i と j の直積.
(u, v)	x 座標が u , y 座標が v であるような楕円曲線上の点.
$\text{SHAKE256}(x, y)$	入力 x に対する SHAKE256 [13]の出力の先頭 y オクテット.
$\text{SHA512}(x)$	入力 x に対する SHA-512 [14]の出力 (64 オクテット) .
$\text{OCTET}(x)$	値 x のオクテット列.
$\text{OLEN}(x)$	オクテット列 x のオクテット数.

3. EdDSA の技術的解説

RFC8032 で定められた EdDSA が次世代の署名アルゴリズムとして注目される理由として、安全性だけでなく性能も考慮されたアルゴリズムであることが挙げられる。

本章ではアルゴリズムの持つ性能面の優位性を紐解くため、EdDSA の処理内容の解説、実装面での特徴を解説する。

3.1. EdDSA の処理内容の解説

RFC8032 [2]および RFC7748 [7]をもとに整理した処理内容の解説を行う。なお、解説を行う上で説明のために RFC8032 [2]および [7]と記述が異なる部分が存在している。

RFC8032 [2]では汎用的な EdDSA だけでなく、以下のような選択肢を持つアルゴリズムも記載されている。以下に EdDSA における選択可能な項目を示す。

- Edward 曲線の種類
 - RFC7748 [7]で規定された `edwards25519/edwards448` のどちらの曲線
- ハッシュ値の状態
 - 署名生成、署名検証の処理の初期段階でメッセージのハッシュ値を保持するかどうか
- コンテキストの状態
 - 署名者と検証者があらかじめ合意するバイト列の状態

RFC8032 [2]では選択肢を組み合わせた以下のアルゴリズムが記載されている。

- RFC7748 [7]の `edwards25519` のための EdDSA (128 ビットセキュリティ²)
 - ハッシュ値を保持しない
 - ✧ `Ed25519:0` バイトのコンテキストを入力する
 - ✧ `Ed25519ctx:1` バイト以上のコンテキストを入力する
 - ハッシュ値を保持する
 - ✧ `Ed25519ph:0` バイト以上のコンテキストを入力する
- RFC7748 [7]の `edwards448` のための EdDSA (224 ビットセキュリティ)
 - ハッシュ値を保持しない
 - ✧ `Ed448:0` バイト以上のコンテキストを入力する
 - ハッシュ値を保持する
 - ✧ `Ed448ph:0` バイト以上のコンテキストを入力する

以下では、これらの名前の付けられた 5 つのアルゴリズムについて解説する。

² [8]によると RFC8032 [1]に記載された通りではないと評価されているが、本書では同程度の強度のアルゴリズムと比較する目安として、RFC8032 [1]に記載された通りの暗号強度を記載している。

3.1.1. データ変換

5つのアルゴリズム共通で使用するデータとデータ間の変換について解説する。
[2]では、下記の種類のデータを扱う。

- オクテット列
- 整数
- 元 ($\text{GF}(p)$ の要素)
- 点

EdDSA で必要な変換は“オクテットと整数”，“オクテット列と元” および “オクテット列と点” である。オクテット列への変換をエンコード，オクテット列からの変換をデコードと呼ぶ。

以下で b は5つのアルゴリズムで定義されるパラメータである。具体的には，
Ed25519, Ed25519ctx および Ed25519ph の場合は $b = 256$ であり，Ed448, Ed448ph の場合は $b = 456$ である。

① オクテット

8 ビットのビット列

$$b_0b_1b_2b_3b_4b_5b_6b_7$$

を1つのオクテットとする。 b_0 を最下位ビット， b_7 を最上位ビットと呼ぶ。

例えば，値が128のオクテットに対応するビット列は最下位ビットから7ビットが0で最上位ビットが1の00000001である。

② 整数

整数のエンコードを $\text{ENCI}(s)$ と表記する。

s は変換元の整数である。変換は，整数を256進数表現したときの各桁をオクテットとみなすことに基づき，リトルエンディアンの並び順で $\frac{b}{8}$ オクテットに格納する処理である。

例えば，整数 s が以下のように256進数表現できる場合

$$s = s_2 * 256^2 + s_1 * 256 + s_0$$

$\text{ENCI}(s)$ は次のオクテット列を表す。

$$\underbrace{s_0s_1s_20\dots0}_{\frac{b}{8}\text{オクテット}}$$

整数のデコードを $\text{DECI}(t)$ と表記する。

t は変換元の $\frac{b}{8}$ オクテットのオクテット列である。変換は，前項の $\text{ENCI}(s)$ の逆変換であり， t の各オクテットを256進数表現の各桁とみなすことに基づく。

例えば $\frac{b}{8}$ オクテットのオクテット列 t が以下の場合

$$t = \underbrace{t_0 t_1 t_2 \dots 0}_{\frac{b}{8} \text{オクテット}}$$

DECI(t)は次の整数を表す.

$$0 * 256^3 + t_2 * 256^2 + t_1 * 256 + t_0$$

③ 元

元のエンコードをENCF(x)と表記する.

x は変換元の元である. 変換は, x を $\{0, 1, \dots, p-1\}$ における範囲の整数としたときの ENCI(x)である. ENCF(x)のビット列は最上位ビットが必ず 0 になる.

元のデコードをDECF(t)と表記する.

t は変換元の $\frac{b}{8}$ オクテットのオクテット列である. 変換は, 前項のENCF(x)の逆変換であり, DECI(t) ($\{0, 1, \dots, p-1\}$ でなければならない) をGF(p)の元とみなすことである.

④ 点

ENCE(A):

点のエンコードをENCE(A)と表記する.

A は変換元の点である. 変換は, A の y 座標をENCFでオクテット列に変換し, その最終オクテットの最上位ビットに A における x 座標の符号 (正または 0 : 0, 負 : 1) を格納する.

ENCF(A の x 座標)のビット列がENCF($-(A$ の x 座標))のビット列より辞書順に大きい場合, A の x 座標の符号を負と定義する. ここで「辞書順に」とは, 先頭ビット (最下位ビット) から順に比較していくことを意味している.

A の x 座標と, $-(A$ の x 座標) = $p - (A$ の x 座標)は, p が奇数であるため必ず偶奇が異なる. つまり, ENCF(A の x 座標)の最下位ビットとENCF($-(A$ の x 座標))の最下位ビットは必ず異なり, ENCF(A の x 座標)の最下位ビットが 1 であることで辞書順に大きい, すなわち負と分かる.

したがって, 点のエンコード処理は以下となる.

1. $t = \text{ENCF}(A$ の y 座標)
2. $t \left[\frac{b}{8} - 1 \right]_7 = \text{ENCF}(A$ の x 座標)₀
3. t を出力する.

DECE(t):

元のデコードをDECF(t)と表記する.

t は変換元の $\frac{b}{8}$ オクテットのオクテット列である.

変換は以下の手順で行う.

1. $x_0 = t \left\lfloor \frac{b}{8} - 1 \right\rfloor_7$
2. $t \left\lfloor \frac{b}{8} - 1 \right\rfloor_7 = 0$
3. $y = \text{DECF}(t)$
4. $y \geq p$ ならばデコード失敗とする.
5. Ed25519, Ed25519ctx, Ed25519ph の場合
 - (ア) $u = y^2 - 1, v = dy^2 + 1$ として $x = uv^3(uv^7)^{\frac{p-5}{8}} \bmod p$
 - (イ) $vx^2 \neq \pm u \bmod p$ ならばデコード失敗とする.
 - (ウ) $vx^2 = -u \bmod p$ ならば, $x = 2^{\frac{p-1}{4}} x$
6. Ed448, Ed448ph の場合
 - (ア) $u = y^2 - 1, v = dy^2 - 1$ として $x = u^3 v(u^5 v^3)^{\frac{p-3}{4}} \bmod p$
 - (イ) $vx^2 \neq u \bmod p$ ならばデコード失敗とする.
7. $x = 0$ かつ $x_0 = 1$ ならばデコード失敗とする.
8. x_0 が $x \bmod 2$ と異なるならば $x = p - x$
9. 点 (x, y) を出力する.

3.1.2. Ed25519, Ed25519ctx, Ed25519ph 共通パラメータ

Ed25519, Ed25519ctx, Ed25519ph では以下の共通パラメータを使用する.

- ① $p = 2^{255} - 19$ (edwards25519 [7] の p)
- ② $b = 256$
- ③ $c = 3$ (edwards25519 [7] の補因子の 2 を底とする対数)
- ④ $n = 254$
- ⑤ $d = -\frac{121665}{121666} = 37095705934669439343138083508754565189542113879$
 $843219016388785533085940283555$ (edwards25519 [7] の d)
- ⑥ $a = -1$
- ⑦ $B = (151122213495354007725011514095885315114540126930418572$
 $06046113283949847762202, 46316835694926478169428394003475163$
 $141307993866256225615783033603165251855960960)$
 (edwards25519 [7] の $(X(P), Y(P))$)
- ⑧ $L = 2^{252} + 27742317777372353535851937790883648493$
 (edwards25519 [7] の位数)

3.1.3. Ed25519, Ed25519ctx, Ed25519ph 共通処理

以下の処理は Ed25519, Ed25519ctx, Ed25519ph で共通である.

① 点の加算, 2 倍算 (同一の点の加算), 無限遠点との加算

楕円点の加算, 2 倍算 (同一の点の加算), 無限遠点との加算は以下の 1 つの式で計算できる.

$$x_3 = \frac{x_1 y_2 + x_2 y_1}{1 + dx_1 x_2 y_1 y_2}, y_3 = \frac{y_1 y_2 + x_1 x_2}{1 - dx_1 x_2 y_1 y_2}$$

これを RFC8032 [2] で推奨される拡張射影座標系 (extended twisted Edwards coordinates) (3.2.1 参照) で計算するアルゴリズムは以下ようになる.

入力: 加算する点 $P_1 = (X_1, Y_1, Z_1, T_1)$, 点 $P_2 = (X_2, Y_2, Z_2, T_2)$

出力: 加算結果の点 $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3, T_3)$

1. $A = (Y_1 - X_1)(Y_2 - X_2)$
2. $B = (Y_1 + X_1)(Y_2 + X_2)$
3. $C = T_1 \cdot 2d \cdot T_2$
4. $D = Z_1 \cdot 2 \cdot Z_2$
5. $E = B - A$
6. $F = D - C$
7. $G = D + C$
8. $H = B + A$
9. $X_3 = E \cdot F$
10. $Y_3 = G \cdot H$
11. $T_3 = E \cdot H$
12. $Z_3 = F \cdot G$

② 2 倍算の高速化

楕円点の 2 倍算は①で計算できるが, 入力が同一の点かどうか条件分岐してもよい場合は高速なアルゴリズムを使うこともできる. 拡張射影座標系 (3.2.1 参照) で計算するアルゴリズムは以下ようになる.

入力: 2 倍算する点 $P_1 = (X_1, Y_1, Z_1, T_1)$

出力: 2 倍算結果の点 $P_3 = P_1 + P_1 = (X_3, Y_3, Z_3, T_3)$

1. $A = X_1^2$
2. $B = Y_1^2$
3. $C = 2Z_1^2$
4. $H = A + B$
5. $E = H - (X_1 + Y_1)^2$
6. $G = A - B$
7. $F = C + G$
8. $X_3 = E \cdot F$
9. $Y_3 = G \cdot H$

$$10. T_3 = E \cdot H$$

$$11. Z_3 = F \cdot G$$

③ dom2(x, y)

バイト列 "SigEd25519 no Ed25519 collisions" || OCTET(x) || OCTET(OLEN(y)) || y .

x は 0~255 の値.

y は最小で 1 バイト, 最大で 255 バイトのデータ列.

"SigEd25519 no Ed25519 collisions" は ASCII 文字列 (32 バイト) .

④ 鍵生成

入力 : なし

出力 : 秘密鍵 k , 公開鍵 A

1. k = 暗号的に安全なランダムデータの 32 オクテット
2. $h = \text{SHA512}(k)$ (64 オクテット)
3. $h[0]_0 = 0, h[0]_1 = 0, h[0]_2 = 0, h[31]_7 = 0, h[31]_6 = 1$
4. $s = \text{DECI}(h) \bmod L$
5. $A = \text{ENCE}([s]B)$ (32 オクテット)

秘密のスカラー値 s は 8 の倍数で, 正確に 255 ビットとなる.

3.1.4. Ed25519

Ed25519 は, 楕円曲線として Edwards25519 を選択し, 設定可能な項目としてコンテキストを入力せず, ハッシュ値を保持しない EdDSA のことである.

① 署名生成

入力 : 秘密鍵 k (32 オクテット) , メッセージ M (任意のサイズのオクテット列)

出力 : 署名 σ

1. $h = \text{SHA512}(k)$
2. $h[0]_0 = h[0]_1 = h[0]_2 = h[31]_7 = 0, h[31]_6 = 1$
3. $s = \text{DECI}(h[0] || \dots || h[31])$
4. $A = \text{ENCE}([s]B)$ (32 オクテット)
5. $r = \text{DECI}(\text{SHA512}(h[32] || \dots || h[63] || M)) \bmod L$
6. $R = \text{ENCE}([r]B)$ (32 オクテット)
7. $k = \text{DECI}(\text{SHA512}(R || A || M)) \bmod L$
8. $S = (r + k * s) \bmod L$
9. $\sigma = R || \text{ENCO}(S)$ (64 オクテット)

秘密のスカラー値 s は 8 の倍数で, 正確に 255 ビットとなる.

② 署名検証

入力：署名 σ (64 オクテット) , 公開鍵 A (32 オクテット) , メッセージ M (任意のサイズのオクテット列)

出力：署名検証の結果 (有効または無効)

1. $R = \text{DECE}(\sigma[0] || \dots || \sigma[31])$
2. $S = \text{DECI}(\sigma[32] || \dots || \sigma[63])$
3. $A' = \text{DECE}(A)$
4. 1. ~3. でデコードに失敗した場合 ($0 \leq S < L$ の範囲外であることを含む) , 無効を出力して終了.
5. $k = \text{DECI}(\text{SHA512}(R || A || M)) \bmod L$
6. $[8][S]B = [8]R + [8][k]A'$ が成り立たない場合は無効を出力し, 成り立つ場合は有効を出力する.

3.1.5. Ed25519ctx

1 バイト以上のコンテキストを入力し, ハッシュ値を保持しない, Edwards25519 を使った EdDSA である.

① 署名生成

入力：秘密鍵 k (32 オクテット) , メッセージ M (任意のサイズのオクテット列) , 最小 1 オクテット, 最大 255 オクテットのコンテキスト C , フラグ $F = 0$ (既定値)

出力：署名 σ

1. $h = \text{SHA512}(k)$
2. $h[0]_0 = h[0]_1 = h[0]_2 = h[31]_7 = 0, h[31]_6 = 1$
3. $s = \text{DECI}(h[0] || \dots || h[31])$
4. $A = \text{ENCE}([s]B)$ (32 オクテット)
5. $r = \text{DECI}(\text{SHA512}(\text{dom2}(F, C) || h[32] || \dots || h[63] || M)) \bmod L$
6. $R = \text{ENCE}([r]B)$ (32 オクテット)
7. $k = \text{DECI}(\text{SHA512}(\text{dom2}(F, C) || R || A || M)) \bmod L$
8. $S = (r + k * s) \bmod L$
9. $\sigma = R || \text{ENCO}(S)$ (64 オクテット)

秘密のスカラー値 s は8の倍数で, 正確に 255 ビットとなる.

② 署名検証

入力：署名 σ (64 オクテット) , 公開鍵 A (32 オクテット) , メッセージ M (任意のサイズのオクテット列) , 最小 1 オクテット, 最大 255 オクテットのコンテキスト C , フラグ $F = 0$ (既定値)

出力：署名検証の結果（有効または無効）

1. $R = \text{DECE}(\sigma[0] || \dots || \sigma[31])$
2. $S = \text{DECI}(\sigma[32] || \dots || \sigma[63])$
3. $A' = \text{DECE}(A)$
4. 1. ～3. でデコードに失敗した場合（ $0 \leq S < L$ の範囲外であることを含む），無効を出力して終了.
5. $k = \text{DECI}(\text{SHA512}(\text{dom2}(F, C) || R || A || M)) \bmod L$
6. $[8][S]B = [8]R + [8][k]A'$ が成り立たない場合は無効を出力し，成り立つ場合は有効を出力する.

3.1.6. Ed25519ph

0 バイト以上のコンテキストを入力し，ハッシュ値を保持する，Edwards25519 を使った EdDSA である.

① 署名生成

入力：秘密鍵 k (32 オクテット)，メッセージ M (任意のサイズのオクテット列)，最小 0 オクテット，最大 255 オクテットのコンテキスト C ，フラグ $F = 1$

出力：署名 σ

1. $h = \text{SHA512}(k)$
2. $h[0]_0 = h[0]_1 = h[0]_2 = h[31]_7 = 0, h[31]_6 = 1$
3. $s = \text{DECI}(h[0] || \dots || h[31])$
4. $A = \text{ENCE}([s]B)$ (32 オクテット)
5. $r = \text{DECI}(\text{SHA512}(\text{dom2}(F, C) || h[32] || \dots || h[63] || \text{SHA512}(M))) \bmod L$
6. $R = \text{ENCE}([r]B)$ (32 オクテット)
7. $k = \text{DECI}(\text{SHA512}(\text{dom2}(F, C) || R || A || \text{SHA512}(M))) \bmod L$
8. $S = (r + k * s) \bmod L$
9. $\sigma = R || \text{ENCO}(S)$ (64 オクテット)

秘密のスカラー値 s は 8 の倍数で，正確に 255 ビットとなる.

② 署名検証

入力：署名 σ (64 オクテット)，公開鍵 A (32 オクテット)，メッセージ M (任意のサイズのオクテット列)，最小 0 オクテット，最大 255 オクテットのコンテキスト C ，フラグ $F = 1$

出力：署名検証の結果（有効または無効）

1. $R = \text{DECE}(\sigma[0] || \dots || \sigma[31])$
2. $S = \text{DECI}(\sigma[32] || \dots || \sigma[63])$
3. $A' = \text{DECE}(A)$

4. 1. ～3. でデコードに失敗した場合（ S が範囲外であることを含む），無効を出力して終了.
5. $k = \text{DECI}(\text{SHA512}(\text{dom2}(F, C) || R || A || \text{SHA512}(M))) \bmod L$
6. $[8][S]B = [8]R + [8][k]A'$ が成り立たない場合は無効を出力し，成り立つ場合は有効を出力する.

3. 1. 7. Ed448, Ed448ph 共通パラメータ

Ed448, Ed448ph では以下の共通パラメータを使用する.

- ① $p = 2^{448} - 2^{224} - 1$ (edwards448 [7]の p)
- ② $b = 456$
- ③ $c = 2$ (edwards448 [7]の補因子の2を底とする対数)
- ④ $n = 447$
- ⑤ $d = -39081$ (edwards448 [7]の d)
- ⑥ $a = 1$
- ⑦ $B = (224580040295924300187604334099896036246789641632564134246125461686950415467406032909029192869357953282578032075146446173674602635247710, 298819210078481492676017930443930673437544040154080242095928241372331506189835876003536878655418784733982303233503462500531545062832660)$
(edwards448 [7]の $(X(P), Y(P))$)
- ⑧ $L = 2^{446} + 13818066809895115352007386748515426880336692474882178609894547503885$ (edwards448 [7]の位数)

3. 1. 8. Ed448, Ed448ph 共通処理

以下の処理は EdDSA448, EdDSA448ph で共通である.

① 点の加算, 2 倍算 (同一の点の加算), 無限遠点との加算

楕円点の加算, 2 倍算 (同一の点の加算), 無限遠点との加算は以下の1つの式で計算できる.

$$x_3 = \frac{x_1 y_2 + x_2 y_1}{1 + dx_1 x_2 y_1 y_2}, y_3 = \frac{y_1 y_2 - x_1 x_2}{1 - dx_1 x_2 y_1 y_2}$$

これを RFC8032 [2]で推奨される射影座標系 (projective coordinates) (3.2.1 参照) で計算するアルゴリズムは以下ようになる.

入力: 加算する点 $P_1 = (X_1, Y_1, Z_1)$, 点 $P_2 = (X_2, Y_2, Z_2)$

出力: 加算結果の点 $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$

1. $A = Z_1 \cdot Z_2$

2. $B = A^2$
3. $C = X_1 \cdot X_2$
4. $D = Y_1 \cdot Y_2$
5. $E = d \cdot C \cdot D$
6. $F = B - E$
7. $G = B + E$
8. $H = (X_1 + Y_1)(X_2 + Y_2)$
9. $X_3 = A \cdot F \cdot (H - C - D)$
10. $Y_3 = A \cdot G \cdot (D - C)$
11. $Z_3 = F \cdot G$

② 2 倍算の高速化

楕円点の 2 倍算は①で計算できるが，入力が同一の点かどうか条件分岐してもよい場合は高速なアルゴリズムを使うこともできる．射影座標系（3.2.1 参照）で計算するアルゴリズムは以下のようになる．

入力：2 倍算する点 $P_1 = (X_1, Y_1, Z_1)$

出力：2 倍算結果の点 $P_3 = P_1 + P_1 = (X_3, Y_3, Z_3)$

1. $B = (X_1 + Y_1)^2$
2. $C = X_1^2$
3. $D = Y_1^2$
4. $E = C + D$
5. $H = Z_1^2$
6. $J = E - 2H$
7. $X_3 = (B - E) \cdot J$
8. $Y_3 = E \cdot (C - D)$
9. $Z_3 = E \cdot J$

③ dom4(x, y)

入力：0～255 の値 x ，最大で 255 バイトのデータ列 y

出力：バイト列

1. バイト列 "SigEd448" || OCTET(x) || OCTET(OLEN(y)) || y を出力する．
"SigEd448" は ASCII 文字列 (8 バイト) とする．

④ 鍵生成

入力：なし

出力：秘密鍵 k ，公開鍵 A

1. k = 暗号的に安全なランダムデータの 57 オクテット

2. $h = \text{SHAKE256}(k, 114)$ (114 オクテット)
 3. $h[0]_0 = h[0]_1 = 0, h[55]_7 = 1, h[56] = 0$
 4. $s = \text{DECI}(h[0] || \dots || h[56]) \bmod L$
 5. $A = \text{ENCE}([s]B)$ (32 オクテット)
- 秘密のスカラー値 s は4の倍数で、正確に448ビットとなる。

3.1.9. Ed448

0 バイト以上のコンテキストを入力し、ハッシュ値を保持しない、Edwards448 を使った EdDSA である。

① 署名生成

入力：秘密鍵 k (57 オクテット)，メッセージ M (任意のサイズのオクテット列)，最小0 オクテット，最大255 オクテットのコンテキスト C ，フラグ $F = 0$ (既定値)

出力：署名 σ

1. $h = \text{SHAKE256}(k, 114)$ (114 オクテット)
 2. $h[0]_0 = h[0]_1 = 0, h[55]_7 = 1, h[56] = 0$
 3. $s = \text{DECI}(h[0] || \dots || h[56]) \bmod L$
 4. $A = \text{ENCE}([s]B)$ (57 オクテット)
 5. $r = \text{DECI}(\text{SHAKE256}(\text{dom4}(F, C) || h[57] || \dots || h[113] || M, 114)) \bmod L$
 6. $R = \text{ENCE}([r]B)$ (57 オクテット)
 7. $k = \text{DECI}(\text{SHAKE256}(\text{dom4}(F, C) || R || A || M, 114)) \bmod L$
 8. $S = (r + k * s) \bmod L$
 9. $\sigma = R || \text{ENCO}(S)$ (114 オクテット)
- 秘密のスカラー値 s は4の倍数で、正確に448ビットとなる。

② 署名検証

入力：署名 σ (114 オクテット)，公開鍵 A (57 オクテット)，メッセージ M (任意のサイズのオクテット列)，最小0 オクテット，最大255 オクテットのコンテキスト C ，フラグ $F = 0$ (既定値)

出力：署名検証の結果 (有効または無効)

1. $R = \text{DECE}(\sigma[0] || \dots || \sigma[56])$
2. $S = \text{DECI}(\sigma[57] || \dots || \sigma[113])$
3. $A' = \text{DECE}(A)$
4. 1. ~3. でデコードに失敗した場合 ($0 \leq S < L$ の範囲外であることを含む)，無効を出力して終了。
5. $k = \text{DECI}(\text{SHAKE256}(\text{dom4}(F, C) || R || A || M, 114)) \bmod L$

6. $[4][S]B = [4]R + [4][k]A'$ が成り立たない場合は**無効**を出力し，成り立つ場合は**有効**を出力する．

3.1.10. Ed448ph

0 バイト以上のコンテキストを入力し，ハッシュ値を保持する，Edwards448 を使った EdDSA である．

① 署名生成

入力：秘密鍵 k (57 オクテット)，メッセージ M (任意のサイズのオクテット列)，最小 0 オクテット，最大 255 オクテットのコンテキスト C ，フラグ $F = 1$

出力：署名 σ

1. $h = \text{SHAKE256}(k, 114)$ (114 オクテット)
2. $h[0]_0 = h[0]_1 = 0$, $h[55]_7 = 1$, $h[56] = 0$
3. $s = \text{DECI}(h[0] || \dots || h[56]) \bmod L$
4. $A = \text{ENCE}([s]B)$ (57 オクテット)
5. $r =$

$$\text{DECI}(\text{SHAKE256}(\text{dom4}(F, C) || h[57] || \dots || h[113] || \text{SHAKE256}(M, 64), 114)) \bmod L$$
6. $R = \text{ENCE}([r]B)$ (57 オクテット)
7. $k = \text{DECI}(\text{SHAKE256}(\text{dom4}(F, C) || R || A || \text{SHAKE256}(M, 64), 114)) \bmod L$
8. $S = (r + k * s) \bmod L$
9. $\sigma = R || \text{ENCO}(S)$ (114 オクテット)

秘密のスカラー値 s は 4 の倍数で，正確に 448 ビットとなる．

② 署名検証

入力：署名 σ (114 オクテット)，公開鍵 A (57 オクテット)，メッセージ M (任意のサイズのオクテット列)，最小 0 オクテット，最大 255 オクテットのコンテキスト C ，フラグ $F = 1$

出力：署名検証の結果 (**有効**または**無効**)

1. $R = \text{DECE}(\sigma[0] || \dots || \sigma[56])$
2. $S = \text{DECI}(\sigma[57] || \dots || \sigma[113])$
3. $A' = \text{DECE}(A)$
4. 1. ～3. でデコードに失敗した場合 ($0 \leq S < L$ の範囲外であることを含む)，**無効**を出力して終了．
5. $k = \text{DECI}(\text{SHAKE256}(\text{dom4}(F, C) || R || A || \text{SHAKE256}(M, 64), 114)) \bmod L$
6. $[4][S]B = [4]R + [4][k]A'$ が成り立たない場合は**無効**を出力し，成り立つ場合は**有効**を出力する．

3.2. 実装の解説

EdDSA のアルゴリズム特性を考慮した実装をする上での実現方法を検討可能な項目について解説する.

3.2.1. Edwards 曲線の特徴

Edwards 曲線の特徴を実装面の観点で説明する.

- 加算と 2 倍算が同一式で演算可能
点の加算式

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1 y_2 + x_2 y_1}{1 + d x_1 x_2 y_1 y_2}, \frac{y_1 y_2 - a x_1 x_2}{1 - d x_1 x_2 y_1 y_2} \right) \quad (1)$$

では同一の点同士の加算を特別扱いせず、分岐なしで演算可能である. 例えば、ECDSA でよく使われる曲線では同一の点の場合、同一かどうかの判定と 2 倍専用の演算が必要である.

演算処理中に、秘密情報に依存した分岐が存在すると、Side Channel 情報³の分析によって、真の処理が行われたか、偽の処理が行われたかを推測し、分岐の条件となった秘密の情報を推測する Side Channel Attack (SCA) を行うことができる.

分岐なしで処理を行うことができれば、Side Channel 情報は秘密の情報によらず一定となるため、SCA への対策となる.

なお、署名検証処理では秘密情報を処理しないため、SCA に対する対策は不要であり、2 倍算専用のアルゴリズムで高速化が可能である.

- 無限遠点を座標(0,1)で計算可能
無限遠点が座標(0,1)であることにより式(1)で無限遠点を特別扱いせず、分岐なしで演算可能となる. 例えば、ECDSA でよく使われる曲線では加算のオペランドの片方または両方が無限遠点の場合、何もしないで無限遠点でないオペランドを出力する、という分岐が必要である.

これらの 2 つの特徴から次のことが言える.

- 点の加算、2 倍算がすべての点を例外なく、式(1)で演算可能
計算式のレベルで分岐や例外処理がないため、入力値によらず一定の処理時間や一定の消費電力等で演算するような実装が可能であり、数式レベルで SCA に対して耐性を持っていると言える.
また、式が 1 つなので実装が簡素化できるため、省メモリを実現することにも寄与する.

³ 暗号装置の処理中に入出力とは別に暗号装置の外部に伝わる情報. 処理時間、CPU の消費電力、電圧、電磁波、音、熱など.

また、座標系を適切に選択することで高速化が可能である。

- 拡張射影座標

Ed25519 では拡張した射影座標系 ε^e [15]で計算することが RFC8032 [2]で推奨されている。座標 (x, y) はこの座標系では $x = \frac{X}{Z}, y = \frac{Y}{Z}, xy = \frac{T}{Z}$ としたときの (X, Y, Z, T) で表現される。この表現により、加算の式(1)から乗法逆元の計算が消え、処理コストの高い乗法逆元の計算がなくなることによって以下のように式(1)をそのまま計算するより高速化できる。

$$\begin{aligned} (X_1, Y_1, Z_1, T_1) + (X_2, Y_2, Z_2, T_2) \\ = ((X_1 Y_2 + X_2 Y_1)(Z_1 Z_2 - d T_1 T_2), (Y_1 Y_2 + X_1 X_2)(Z_1 Z_2 + d T_1 T_2), (Z_1 Z_2 \\ + d T_1 T_2)(Z_1 Z_2 - d T_1 T_2), (X_1 Y_2 + X_2 Y_1)(Y_1 Y_2 + X_1 X_2)) \end{aligned}$$

拡張射影座標系では無限遠点 $(0,1)$ は、 Z を非ゼロの任意の元として $(0, Z, Z, 0)$ となる。

- 射影座標

Ed448 では射影座標系 [4]で計算することが RFC8032 [2]で推奨されている。座標 (x, y) はこの座標系では $x = \frac{X}{Z}, y = \frac{Y}{Z}$ としたときの (X, Y, Z) で表現される。この表現により、加算の式(1)から乗法逆元の計算が消え、処理コストの高い乗法逆元の計算がなくなることによって以下のように式(1)をそのまま計算するより高速化できる。

$$\begin{aligned} (X_1, Y_1, Z_1) + (X_2, Y_2, Z_2) \\ = ((X_1 Y_2 Z_1 Z_2 + X_2 Y_1 Z_1 Z_2)(Z_1 Z_2 Z_1 Z_2 - d X_1 X_2 Y_1 Y_2), (Y_1 Y_2 Z_1 Z_2 \\ - X_1 X_2 Z_1 Z_2)(Z_1 Z_2 Z_1 Z_2 + d X_1 X_2 Y_1 Y_2), (Z_1 Z_2 Z_1 Z_2 + d X_1 X_2 Y_1 Y_2)(Z_1 Z_2 \\ - d X_1 X_2 Y_1 Y_2)) \end{aligned}$$

射影座標系では無限遠点 $(0,1)$ は、 Z を非ゼロの任意の元として $(0, Z, Z)$ となる。

3. 2. 2. EdDSA の特徴

EdDSA の特徴を実装面の観点で説明する。

- 推奨パラメータが固定

RFC8032 [2]ではパラメータに依存しない汎用的な EdDSA についても説明されているが、Ed25519 と Ed448 を使用した特定のパラメータを使用した実装を推奨している。これにより扱う素体、曲線およびハッシュアルゴリズム等が固定化され、実装では使用するメモリ量をあらかじめ計画することが可能となっている。これにより動的にメモリを確保する必要がなく、あらかじめ必要な固定量のメモリを確保しておき、そのメモリ上ですべて処理をすることが可能である。

EdDSA の考案者である Daniel J. Bernstein が公開しているリファレンス実装の署名生成関数 `crypto_sign` (図 4) を例に挙げると、`unsigned char` の配列 (バイト列) を変数 `pk`, `az`, `nonce` および `hram` として固定の要素数でスタック

領域に確保している。また、sc25519 構造体（図 1）の変数 sck, scs および scsk, ge25519 構造体（図 3）の変数 ger, ここでは使用していないが fe25519 構造体（図 2）も固定長であり、スタック上に確保している。

このようなスタック上の固定長のメモリ確保は、コンパイル時に必要な総量が計算される。コンパイルで生成される命令はスタックポインタを固定バイト進める命令のみとなるため高速化が期待できる。

```
25  typedef struct
26  {
27      |  crypto_uint32 v[32];
28  }
29  sc25519;
```

図 1 sc25519 構造体

```
25  typedef struct
26  {
27      |  crypto_uint32 v[32];
28  }
29  fe25519;
```

図 2 fe25519 構造体

```
15  typedef struct
16  {
17      |  fe25519 x;
18      |  fe25519 y;
19      |  fe25519 z;
20      |  fe25519 t;
21  } ge25519;
```

図 3 ge25519 構造体

```

2  #include "crypto_sign.h"
3  #include "crypto_hash_sha512.h"
4  #include "ge25519.h"
5
6  int crypto_sign(
7      unsigned char *sm,unsigned long long *smlen,
8      const unsigned char *m,unsigned long long mlen,
9      const unsigned char *sk
10 )
11 {
12     unsigned char pk[32];
13     unsigned char az[64];
14     unsigned char nonce[64];
15     unsigned char hram[64];
16     sc25519 sck, scs, scsk;
17     ge25519 ger;
18
19     memmove(pk,sk + 32,32);
20     /* pk: 32-byte public key A */
21
22     crypto_hash_sha512(az,sk,32);
23     az[0] &= 248;
24     az[31] &= 127;
25     az[31] |= 64;
26     /* az: 32-byte scalar a, 32-byte randomizer z */
27
28     *smlen = mlen + 64;
29     memmove(sm + 64,m,mlen);

```

図 4 署名生成関数 crypto_sign (sign.c)

- 分岐のないアルゴリズム

入力によって分岐することがないアルゴリズムとなっており、また、呼び出す Edwards 曲線上の演算、素体の演算等も分岐のない実装とすることが可能である。これにより前項のメモリに対して「どこをいつ書き換えるか」や「どこをいつ参照するか」が入力によらず固定化できる。

これは更新/参照するメモリアドレスの計算を入力値によらず固定化できることになり、高速化、および SCA への耐性になる。

- 乱数生成は秘密鍵生成時のみでの使用

EdDSA では乱数を鍵生成でのみ使用している。この乱数は秘密鍵に直接影響を与えるため、暗号的に安全である必要がある。一方で、署名生成、署名検証では乱数を使用しない。一般的に、暗号的に安全な乱数は生成の時間がランダムで処

理コストが高い。EdDSA では署名生成、署名検証を一定時間で行い、乱数を使用するアルゴリズムに比べて高速に処理することができる。

3.2.3. RFC8032 に記載されている注意点

RFC8032 [2]において、Security Considerations として安全性の観点からいくつかの実装上の注意点を挙げられている。実装時に参考すべき項目について、参考情報として以下に概要を示す。

- ① Edwards 曲線上の点の加算は SCA に耐性があるが、 $\text{GF}(p)$ の演算ではキャリーの伝搬等に依存した分岐をしてはならない。スカラー倍ではバイナリ法を使って 1 ビットずつ処理し、SCA 耐性のある条件付き代入を行うと単純である。
- ② 複数の署名を同時検証する高速化手法の中には、乱数が必要なものがある。
- ③ プロトコルが指定した定数をコンテキストに使用すること。メッセージに依存したコンテキストは使用するべきでなく、状況によってコンテキストを変えるべきではない。
- ④ Ed25519ph と Ed448ph はハッシュ関数の強度に依存するため、既存 API（分割データ入力等）との互換性を考慮しても使用すべきではない。Ed25519 は品質の高い実装が多いため、128 ビットの安全性で十分な場合は Ed25519 を、それ以外の場合は Ed448 を推奨する。
- ⑤ Ed25519, Ed25519ctx, および Ed25519ph に同じ鍵ペアを使用できる。Ed448 および Ed448ph も同様。
- ⑥ 少なくとも 2 回の操作が必要なため、一度に大量のデータに署名することは避ける。一定量のバッファのみを備えた Init/Update/Finalize インターフェースを使用して Ed25519/Ed448 署名を作成できるように、Ed25519/Ed448 の署名生成機能を変更することは勧めない。
- ⑦ 署名検証の等式でコファクタ倍することは安全性の面で必須ではないが、一部のアプリケーションでは正確な署名を定めないと指紋攻撃等を引き起こすことがある。

3.2.4. SCA 対策

3.2.1 および 3.2.2 で示したように EdDSA は SCA への耐性をもったアルゴリズムであるが、3.2.3 の①でも言及したように、SCA 耐性を下げるとなような実装も可能であることに注意する。なお、署名検証でのスカラー倍は秘密にするべき情報が何もないのでここに書いたような注意は不要である。

- 条件付きの代入

ビット b が 1 の場合だけ既存の変数 r に x を代入する処理を考える。

条件により代入する／しないが変わる処理を CPU 命令の用語で Conditional Move, CMOV と呼び、専用の命令が用意されている。アセンブラ言語や機械語が利用できる場合は利用してもよいが、C 言語や Java 等でこの処理を実装する場合は注意が必要となる。

単純な実装は

if $b = 1$ then $r = x$ endif

であるが、代入処理の有無の差のサイドチャネル情報によってビットの値が漏れる可能性がある。このような分岐を避けるため、以下のような演算を行う。

$M = bbb...bbb$
 $r = r \oplus (M \wedge (x \oplus r))$

ここで M は x と同じビット数のビット列で各ビットは b とし、 $\oplus$ はビットごとの排他的論理和、 $\wedge$ はビットごとの論理積とする。

$b = 0$ の場合は $M = 000...000$ となり、 $M \wedge (x \oplus r)$ も $000...000$ となるため、 $r = r \oplus 000...000$ が行われ、結果的に代入を行わないのと同じである。

$b = 1$ の場合は $M = 111...111$ となり、 $M \wedge (x \oplus r)$ は $x \oplus r$ となるため、 $r = r \oplus x \oplus r$ が行われ、結果的に $r = x$ と同じである。

どちらの場合も b の値によらず同じ演算を行っている点に注意する。

Ed25519 のリファレンスコードでは (図 5) のような実装が使われている。

```
131 void fe25519_cmov(fe25519 *r, const fe25519 *x, unsigned char b)↓
132 {↓
133     int i;↓
134     crypto_uint32 mask = b;↓
135     mask = -mask;↓
136     for(i=0; i<32; i++) r->v[i] ^= mask & (x->v[i] ^ r->v[i]);↓
137 }↓
```

図 5 リファレンスコードにおける条件付き代入の例

ここでは上記の数式を 32 ビットの crypto_uint32 単位で行っている。

mask の符号反転は、 b が 1 の場合は $\text{mask}=1 \rightarrow -1=111\cdots111$ を、 b が 0 の場合は $\text{mask}=0 \rightarrow 000\cdots000$ を作成している。

Ed25519 のリファレンスコードではこの fe25519_cmov を使い、スカラー値によらない一定の命令群で事前計算値を用いたスカラー倍を実現している。

- 点の2倍算

点の加算 $R = R + Q$ を行う場合に、少しでも高速にしようと

if $R = Q$ then $R = 2R$ else $R = R + Q$ endif

とすると、加算と2倍算の差のサイドチャンネル情報により、 $R = Q$ となったタイミングが推測しやすくなり、そのタイミングをヒントにスカラー値が推測される可能性がある。

Edwards 曲線の加算と2倍算を同じ式で演算できる特性を活かし、分岐をせずに加算を行うほうがよい。

3.2.5. 長いメッセージに対する配慮

3.2.3の④および⑥で言及されたように、RFC8032 [2]は巨大なメッセージや断続的なメッセージに対する署名を推奨しておらず、このような署名を可能にする IUF (Init, Update, Finalize) インターフェースも推奨していない。

一方で、既存のハッシュ関数や署名アルゴリズムのような実装の多くで、IUF インターフェースが採用されており EdDSA を採用するにあたってインターフェースの互換性が問題になることがある。

推奨はされていないが、あくまでも互換性を保つための方法として、Ed25519ph を例に IUF インターフェースを簡単に紹介する。

① Ed25519ph 署名生成

<Init> SHA512(M)のコンテキスト ctx に対して Init を行う。

<Update> ctx に対して、メッセージ M で Update を行う。

<Finalize> ctx に対して Finalize を行い、出力 H をメッセージとして、秘密鍵 k 、コンテキスト C 、フラグ $F = 1$ を用いて Ed25519ctx の署名生成を行う。

② Ed25519ph 署名検証

<Init> SHA512(M)のコンテキスト ctx に対して Init を行う。

<Update> ctx に対して、メッセージ M で Update を行う。

<Finalize> ctx に対して Finalize を行い、出力 H をメッセージとして、署名 σ 、公開鍵 A 、コンテキスト C 、フラグ $F = 1$ を用いて Ed25519ctx の署名検証を行う。

3.2.6. パラメータ n の選択

[8]の p. 3 の脚注において「RFC8032 の記述に従ったが $c = n$ は安全性の観点からありえない。」との指摘がある。RFC8032 [2]の汎用的なアルゴリズム解説ではこの指摘通りの記述が発見された。

ただし、Ed25519 等の具体的なアルゴリズムのパラメータの説明においては Ed25519, Ed25519ctx, Ed25519ph では $n = 254$ と定義され、Ed448, Ed448ph では $n = 447$ と定義されており、これらの具体的なアルゴリズムの記述を忠実に実装していれば問題のある実装とはならない。

3.3. Ed25519 実装の解説

EdDSA の考案者である Daniel J. Bernstein によってリファレンス実装が公開されている。⁴

プログラムは SUPERCOP [16] の一部として公開されている。⁵
ルートディレクトリから見て supercop-20210604/crypto_sign/ed25519 のディレクトリにソースが格納されている。

このリファレンス実装には 4 つの版があり、以下のような特徴を持っている。

- ref 版. 性能よりも移植性を考慮して C 言語で記述されている。
汎用性を考慮してデータに 32 ビットレジスタを使用している。
構造も単純なため EdDSA のアルゴリズムを理解することが容易になっている。
射影座標系を使う点以外は、ほぼ RFC8032 [2] と RFC7748 [7] のアルゴリズムのままである。
- ref10 版. python で計算した事前計算値を使い、ref 版より高速になっている。
C 言語で記載されているが、Bernstein によって開発された qhasm（共通の記述から CPU ごとに最適化されたアセンブラソースを生成するツール）によって CPU に最適化されたアセンブラコードも含んでいる。
- amd64-51-30k 版. 51 ビットベースの多倍長整数を使い、30kb の事前計算値を使う。
ref10 版をさらに高速化した版. C 言語、qhasm によるアセンブラ言語で実装されている。
- amd64-64-24k 版. 64 ビットベースの多倍長整数を使い、24kb の事前計算値を使う。ref10 版をさらに高速化した版. C 言語、qhasm によるアセンブラ言語で実装されている。

⁴ <https://ed25519.cr.yp.to/software.html>

⁵ <https://bench.cr.yp.to/supercop/supercop-20210604.tar.xz>

ここでは、アルゴリズムの特徴が把握しやすい ref 版について説明する．ref 版のファイル構成は図 6 のようになっている．

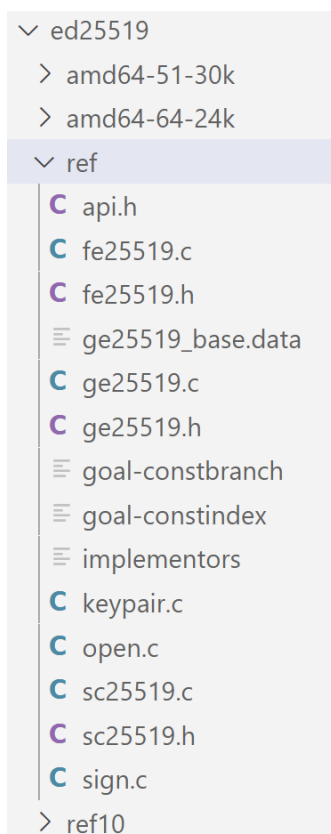


図 6 リファレンス実装 ref 版ファイル構成

各ソースファイルは以下のような内容となっている．

- fe25519.c $\text{GF}(p)$ 上の有限体演算を行う．乗算は教科書法． $2^{255} = 19 \bmod p$, $2^{256} = 38 \bmod p$ を活用している．有限体の元（255 ビット）同士を乗算して 510 ビットにした後，上位 $(510 - 256 = 254)$ ビットは 2^{256} の倍数とみなせる．これを $2^{256} = 38 \bmod p$ により，38 の倍数として下位 256 ビットに加算する．この 38 の倍数を計算する関数が times38 として定義されている．有限体の元は 8 ビットごとに uint32 で保管するが，uint32 の 32 ビットのうち上位 24 ビットは計算途中の値を格納するためのバッファとなっている．計算の途中で 8 ビットを超えた分は，後でまとめて処理される． p による剰余算も加減算のたびに行わずに，いくつかの計算をしたのちにまとめて剰余算を行う．
(lazy reduction)
- ge25519_base.data ベースポイント B のための事前計算値． $0, B, 2B, 3B, 4B, 0, (2^8 + 1)B, (2^8 + 2)B, (2^8 + 3)B, (2^8 + 4)B, 0, \dots$ を格納．
- ge25519.c 楕円曲線上の点の群演算．幅 3 の符号付 Window 法．

- `keypair.c` 公開鍵と秘密鍵の鍵ペアの生成.
- `open.c` 署名検証.
- `sc25519.c` 有限体演算以外の多倍長整数演算. 多倍長整数は 32 ビット整数 (のうち 8 ビットのみ使用) × 32 個の配列.
- `sign.c` 署名生成.

トップレベルの関数は

- `crypto_sign_keypair` 鍵ペア生成.
- `crypto_sign` 署名生成.
- `crypto_sign_open` 署名検証.

3.4. Ed448 実装の解説

Curve448 の考案者である Mike Hamburg によってリファレンス実装が公開されている. (SourceForge.net: Ed448-Goldilocks⁶)

プログラムは以下の Git リポジトリで公開されている.

`git://git.code.sf.net/p/ed448goldilocks/code ed448goldilocks-code`

このプロジェクトでは, ビルド時に Python で相対パス `build/src/GENERATED` に C 言語ソースを生成する仕組みになっている.

ここでは生成後の C 言語ソースについて解説する.

生成された C 言語ソースは図 7 のような構成になっている.

⁶ <http://ed448goldilocks.sourceforge.net/>

```

  ▾ c
    ▾ curve25519
      C decaf.c
      C eddsa.c
      C elligator.c
      C scalar.c
    ▾ decaf
      G+ crypto.hxx
    ▾ ed448goldilocks
      C decaf.c
      C eddsa.c
      C elligator.c
      C scalar.c
    ▾ p448
      C f_field.h
      C f_generic.c
    ▾ p25519
      C f_field.h
      C f_generic.c

```

図 7 生成された C 言語ソースファイル構成

各ソースファイルは以下のような内容となっている。

- curve25519 ディレクトリ配下 Ed25519 実装
 - decaf.c 楕円曲線上の点の群演算. “Decaf: Eliminating cofactors through point compression”⁷で示される高速化技法を実装.
 - eddsa.c Ed25519 の実装.
 - elligator.c “Elligator: Elliptic-curve points indistinguishable from uniform random strings”⁸で示される技法の実装
 - scalar.c スカラー値の多倍長整数演算.
- decaf ディレクトリ配下 C++インタフェースヘッダ
- ed448goldilocks ディレクトリ配下 Ed448 実装
 - decaf.c 楕円曲線上の点の群演算. “Decaf: Eliminating cofactors through point compression”⁷で示される高速化技法を実装.
 - eddsa.c Ed448 の実装.
 - elligator.c “Elligator: Elliptic-curve points indistinguishable from uniform random strings”⁸で示される技法の実装

⁷ <https://www.shiftright.org/papers/decaf/decaf.pdf>

⁸ <https://elligator.cr.yp.to/elligator-20130828.pdf>

- scalar.c スカラー値の多倍長整数演算.
- p448 ディレクトリ配下 Ed448 の $\text{GF}(p)$ 上の有限体演算
 - f_field.h 有限体演算のヘッダファイル.
 - f_generic.c 有限体演算の実装.
- p25519 ディレクトリ配下 Ed25519 の $\text{GF}(p)$ 上の有限体演算
 - f_field.h 有限体演算のヘッダファイル.
 - f_generic.c 有限体演算の実装.

トップレベルの関数は、以下に示す通りである.

- decaf_ed448_derive_public_key (秘密鍵から公開鍵を生成する)
- decaf_ed448_sign (Ed448 の署名生成の実装, 秘密鍵と公開鍵が別引数)
- decaf_ed448_sign_prehash (Ed448ph の署名生成の実装, 秘密鍵と公開鍵が別引数)
- decaf_ed448_keypair_sign (Ed448 の署名生成の実装, 鍵ペアが引数)
- decaf_ed448_keypair_sign_prehash (Ed448ph の署名生成の実装, 鍵ペアが引数)
- decaf_ed448_verify (Ed448 の署名検証の実装)
- decaf_ed448_verify_prehash (Ed448ph の署名検証の実装)

4. EdDSA の標準化動向や OSS に関する採用状況など

EdDSA のアルゴリズムは 2011 年に発明された、近年標準化が進んでいるアルゴリズムであるため、EdDSA のアルゴリズムが記載された標準や EdDSA を利用する標準について改めて採用状況を調査することで、EdDSA の標準化動向の全体像を把握することにつながると考えられる。

ただし、EdDSA が標準化されているだけでは実社会で利用されるための状況には遠いと考えられ、OSS など実際のシステムやサービスなどで利用可能にする環境が実社会での利用について大きく寄与すると考えられる。このため、OSS で実装されているかどうかという観点が、広く利用されているかを把握するために有益であると考えられる。

本調査では、ハードウェアやソースコードの調査ができない商用ソフトウェアなどについては、EdDSA を実装しているかの判断をすることが困難であると考えられるため、本調査対象から除外することとした。

これらのことを踏まえて、EdDSA アルゴリズムに関して標準化されているかどうかの調査結果を 4.1 節にまとめ、さらに、4.1 節で抽出された標準仕様を参照する形式で標準仕様において採用されているかの調査結果を 4.2 節にまとめる。また、企業・団体が実装し、ソースコードの調査が可能な OSS に焦点を絞った実装の調査結果を 4.3 にまとめ、EdDSA の実装に対して発見されている脆弱性について調査結果を 4.4 節にまとめた。

4.1. EdDSA に関する標準化動向調査

本調査対象である EdDSA に関する標準化動向を調査する。標準化動向を調査するために調査効率化を考慮して、インターネット上で公開されている情報に対して検索エンジンを用いて、以下に示す(A)群と(B)群の検索キーワードの組み合わせを使用して検索し、各検索結果のサイトやそのリンク先を調査することとした。

(A) "EdDSA", "25519", "448"

(B) "specification" "standard"

調査した範囲では表 1 の 3 件が該当した。

表 1 EdDSA に関する標準化動向調査

項番	発行組織	ドキュメント名	概要	発行日
1	IETF	RFC8032 “Edwards-Curve Digital Signature Algorithm (EdDSA)” ⁹	デジタル署名方式である Edwards-curve Digital Signature Algorithm (EdDSA) を説明するドキュメント。推奨する edwards25519, edwards448 曲線で実体化されたアルゴリズム, サンプル実装およびテストベクトルを提供している。	2017 年 1 月
2	NIST	FIPS Pub 186-5 (Draft) “Digital Signature Standard (DSS)” ¹⁰	デジタル署名を生成するのに利用可能なアルゴリズム群を規定する標準。 RFC8032 を参照しているが, Ed25519ctx は規定されず, Ed25519, Ed448, Ed25519ph, Ed448ph が規定されている。	2019 年 10 月
3	NIST	SP800-186 (Draft) “Recommendations for Discrete Logarithm-Based Cryptography: Elliptic Curve Domain Parameters” ¹¹	米国政府での利用で推奨される曲線を規定する。素体と 2 の拡大体上の Weierstrass 曲線および, 効率, side-channel 耐性および, 実装の簡素化を主張する新しい Montgomery 曲線とその代替表現を規定する。新しい曲線は, IETF の CFRG で規定された曲線と相互運用可能である。	2019 年 10 月

⁹ 表記上の問題が 5 件報告されているが, 処理内容についての指摘はない。

¹⁰ <https://csrc.nist.gov/publications/detail/fips/186/5/draft>

¹¹ <https://csrc.nist.gov/publications/detail/sp/800-186/draft>

EdDSA のアルゴリズムを標準仕様として規定しているのは RFC8032 のみという結果となった。また、Draft 版ではあるが、NIST の FIPS Pub 186-5 (Draft) [12], SP800-186 にも RFC8032 [2] と相互運用可能なアルゴリズム (Ed25519, Ed448, Ed25519ph, Ed448ph) が記載されていた。なお、Ed25519ctx は規定されていないという状況となっていた。

現在、Draft である FIPS Pub 186-5 (Draft) [12] および SP800-186 について、正式な文書として公開されるかどうかの情報について、今回調査した範囲では発見することはできなかった。しかし、2020 年 1 月 29 日に締め切られた Public comment では、Draft 版の記述の誤りを指摘するコメントが多かったが、RFC8032 [2] との互換性を保つためのコメントや、RFC8032 [2] との互換性を保ちつつ、更に安全性を高められるオプションを望むコメントなど、すでに RFC8032 [2] が受け入れられている状況を考慮したと思われるコメントもあった。以上のことから、EdDSA は近い将来、NIST によって承認されたアルゴリズムに含まれると考える。

4.2. 標準仕様における EdDSA に関する採用状況

EdDSA の標準化動向の全体像を把握するため、4.1 で抽出されたリストにある EdDSA アルゴリズムに関する標準化仕様を参照する形式で EdDSA がどのような標準仕様 (Draft も含む) で利用されているかどうかを調査する。具体的には、4.1 で抽出されたリストのドキュメントを参照する標準化仕様にはどのようなものがあるかを以下のように調査した。

調査効率化を考慮して、インターネット上で公開されている情報に対して検索エンジンで以下の (A) 群と (B) 群の検索キーワードの組み合わせを使用して検索し、各検索結果のサイトやそのリンク先を調査した。

(A) "RFC8032", "FIP186-5", "SP800-186"

(B) "specification", "standard"

また、4.1 の調査中に発見した情報についても RFC8032, FIP180-5, SP800-186 を参照している標準も結果に含めた。

さらに、IETF 標準化された仕様である RFC8032 については、Data Tracker というサービスを用いて RFC8032 を参照している RFC や Internet Draft (I-D) の一覧を抽出した。¹²

比較対象としての参照、例として挙げているだけのものは除外した。

調査した結果を表 2 に示す。

¹² <https://datatracker.ietf.org/doc/rfc8032/referencedby/>

表 2 標準仕様における EdDSA に関する採用状況

項番	発効組織	ドキュメント名	公開年月	Ed25519	Ed448	参照している文献
1	IETF	RFC7479: "Using Ed25519 in SSHFP Resource Records"	2015 年 3 月	○	×	[1]
2	IETF	RFC8037: "CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE)"	2021 年 1 月	○	○	RFC8032 [2]
3	IETF	RFC8080: "Edwards-Curve Digital Security Algorithm (EdDSA) for DNSSEC"	2021 年 2 月	○	○	RFC8032 [2]
4	IETF	RFC8152: "CBOR Object Signing and Encryption (COSE)"	2021 年 7 月	○	○	RFC8032 [2]
5	IETF	RFC8410: "Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the Internet X.509 Public Key Infrastructure"	2021 年 8 月	○	○	RFC8032 [2]
6	IETF	RFC8411: "Additional XML Security Uniform Resource Identifiers (URIs)"	2021 年 8 月	○	○	RFC8032 [2]
7	IETF	RFC8419: "Use of Edwards-Curve Digital Signature Algorithm (EdDSA) Signatures in the Cryptographic Message Syntax (CMS)"	2021 年 8 月	○	○	RFC8032 [2]
8	IETF	RFC8420: "Using the Edwards-Curve Digital Signature Algorithm (EdDSA) in the Internet Key Exchange Protocol Version 2 (IKEv2)"	2021 年 8 月	○	○	RFC8032 [2]
9	IETF	RFC8422: "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS) Versions 1.2 and Earlier"	2021 年 8 月	○	○	RFC8032 [2]

項番	発効組織	ドキュメント名	公開年月	Ed25519	Ed448	参照している文献
10	IETF	RFC8446: "The Transport Layer Security (TLS) Protocol Version 1.3"	2021 年 8 月	○	○	RFC8032 [2]
11	IETF	RFC8463: "A New Cryptographic Signature Method for DomainKeys Identified Mail (DKIM)"	2021 年 9 月	○	×	RFC8032 [2]
12	IETF	RFC8550: "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Certificate Handling"	2021 年 4 月	○	×	RFC8032 [2]
13	IETF	RFC8551: "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification"	2021 年 4 月	○	×	RFC8419
14	IETF	RFC8591: "SIP-Based Messaging with S/MIME"	2021 年 4 月	○	×	RFC8032 [2]
15	IETF	RFC8624 "Algorithm Implementation Requirements and Usage Guidance for DNS-SEC"	2021 年 6 月	○	○	RFC8032 [2]
16	IETF	RFC8709: "Ed25519 and Ed448 Public Key Algorithms for the Secure Shell (SSH) Protocol"	2021 年 2 月	○	○	RFC8032 [2]
17	IETF	RFC8928: "Address-Protected Neighbor Discovery for Low-Power and Lossy Networks"	2021 年 11 月	○	×	RFC8032 [2]
18	IETF	Draft: "Additional XML Security Uniform Resource Identifiers (URIs)"	2021 年 11 月	○	○	RFC8032 [2]
19	IETF	Draft: "Alternative Elliptic Curve Representations"	2021 年 10 月	○	○	RFC8032 [2]

項番	発効組織	ドキュメント名	公開年月	Ed25519	Ed448	参照している文献
20	IETF	Draft: "Certificate Management Protocol (CMP) Algorithms"	2021 年 11 月	○	○	RFC8032 [2] FIPS Pub 186-5 (Draft) [12]
21	IETF	Draft: "Certificate Transparency Version 2.0"	2021 年 8 月	○	×	RFC8032 [2]
22	IETF	Draft: "CBOR Object Signing and Encryption (COSE): Initial Algorithms"	2020 年 9 月	○	○	RFC8032 [2]
23	IETF	Draft: "DRIP Entity Tag (DET) for Unmanned Aircraft System Remote Identification (UAS RID)"	2021 年 12 月	○	○	RFC8032 [2]
24	IETF	Expired Draft: "EdDSA for OpenPGP"	2016 年 8 月	○	×	[1]
25	IETF	Draft: "Group OSCORE - Secure Group Communication for CoAP"	2021 年 10 月	○	×	RFC8032 [2]
26	IETF	Draft: "Key Management for OSCORE Groups in ACE"	2021 年 10 月	○	○	RFC8032 [2]
27	IETF	Draft: "License Activation Protocol"	2021 年 9 月	○	×	RFC8032 [2]
28	IETF	Draft: "Mathematical Mesh 3.0 Part II: Uniform Data Fingerprint."	2021 年 10 月	○	○	RFC8032 [2]
29	IETF	Draft: "Mathematical Mesh 3.0 Part VIII: Cryptographic Algorithms"	2021 年 8 月	○	○	RFC8032 [2]
30	IETF	Draft: "Message Queuing Telemetry Transport (MQTT)-TLS profile of Authentication and Authorization for Constrained Environments (ACE) Framework"	2021 年 10 月	○	×	RFC8032 [2]
31	IETF	Draft: "New Cryptographic Algorithms for HIP"	2021 年 8 月	○	○	RFC8032 [2]
32	IETF	Draft: "OpenPGP Message Format"	2021 年 10 月	○	○	RFC8032 [2]

項番	発効組織	ドキュメント名	公開年月	Ed25519	Ed448	参照している文献
33	IETF	Draft: "S/MIME Example Keys and Certificates"	2021 年 8 月	○	×	RFC8032 [2]
34	IETF	Draft: "The GNU Name System"	2021 年 11 月	○	×	RFC8032 [2]
35	IETF	Draft: "The Messaging Layer Security (MLS) Protocol"	2021 年 10 月	○	○	RFC8032 [2]
36	IETF	Draft: "The Transport Layer Security (TLS) Protocol Version 1.3"	2021 年 10 月	○	○	RFC8032 [2]
37	IETF	Draft: "Threshold Modes in Elliptic Curves"	2021 年 8 月	○	○	RFC8032 [2]
38	DMTF	DSP0274 1.2.0WIP.90 "Security Protocol and Data Model (SPDM) Specification"(Work in Progress Version)	2021 年 8 月	○	○	RFC8032 [2]
39	ETSI	ETSI TS 102 809 V1.3.1 (2017-06) "Digital Video Broadcasting (DVB); Signalling and carriage of interactive applications and services in Hybrid broadcast/broadband environments"	2017 年 6 月	○	×	RFC8032 [2]
40	FIDO Alliance	"FIDO Authenticator Allowed Cryptography List"	2021 年 10 月	○	×	RFC8032 [2]
41	FIDO Alliance	"FIDO Registry of Predefined Values" Review Draft 02	2018 年 7 月	○	×	RFC8032 [2]
42	JCP	"Java SE 15 (JSR 390)Final Release Specification"	2020 年 9 月	○	○	RFC8032 [2]
43	NIST	Draft: "ACVP EdDSA Algorithm JSON Specification"	2021 年 11 月	○	○	FIPS Pub 186-5 (Draft) [12]

項番	発効組織	ドキュメント名	公開年月	Ed25519	Ed448	参照している文献
44	OASIS	"PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0"	2020 年 6 月	○	○	RFC8032 [2]
45	TCG	"TCG Algorithm Registry Family "2.0" Level 00 Revision 01.32"	2021 年 6 月	○	×	RFC8032 [2]
46	W3C	"Web Authentication: An API for accessing Public Key Credentials Level 2"	2021 年 4 月	○	×	RFC8152

ここで発見された 46 件のうち 37 件が IETF の RFC または I-D であり、IETF では EdDSA を採用した標準化が活発であることが分かった。

また、参照している文献は、41 件が RFC8032 [2] を、2 件が [12] を、2 件が [1] を、1 件が RFC8419 を、1 件が RFC8152 を参照して [12] おり、正式に標準化された唯一の標準である RFC8032 [2] を採用する標準化仕様が多かった。

標準化の公開年月では、36 件が 2020 年と 2021 年に公開されており、2017 年の RFC8032 [2] の公開から 4 年を経て、急速に標準化仕様への採用が進んだと考えられる。

今回調査した範囲では、その他の団体の標準化もここ 2 年で行われており、全体的にここ 2 年で採用が進みつつあることが分かった。2022 年以降、NIST の FIPS Pub 186-5 が正式に発効した後はさらに採用が進み、2021 年末時点で Draft 版が多いことも考慮すると、2022 年以降はさらに採用が進むと予想される。

アルゴリズム別に調査結果を見ると、46 件のうち、Ed25519 を採用していたものが 46 件だったのに対し、Ed448 を採用しているものは 28 件という結果となった。今回、実施した調査の範囲では Ed25519 と Ed448 における採用実績の差に関する差については判明できなかった。

以上から、標準化された EdDSA（主に Ed25519）については、2020 年から標準化仕様での採用によって広く利用される状況が整いつつあり、今後、採用する標準が増えるにつれ、さらに一般に広く受け入れられていくことが予想される。

4.3. EdDSA に関する OSS 採用状況

EdDSA が標準化された上で、実社会での利用に寄与する OSS など実際のシステムやサービスなどで利用可能な環境が整ってきているかどうかという観点が、EdDSA が広く利用されているかを把握するために有益であると考ええる。

このため、実際のシステムやサービスなどで利用可能な環境として利用される暗号ライブラリに焦点を当て、広く知られているメジャーな暗号ライブラリについて EdDSA が実装されているかを調査した。

調査対象の暗号ライブラリを調査するために調査効率化を考慮して、インターネット上で公開されている情報に対して検索エンジンで

“eddsa”, “library”, “open source”

の検索キーワードを使用して検索し、検索結果のサイトやそのリンク先を調査した。なお、ハードウェアやソースコードの調査ができない商用ソフトウェアなどについては、EdDSA を実装しているかの判断をすることが困難であるため OSS に限定した。

実装の確認は、EdDSA の処理手順が実装されているかソースコードを確認した。Ed25519 と Ed448 のどちらが実装されているかは関数名やクラス名などの識別子で判断した。

表 3 の OSS が EdDSA を実装していた。

表 3 EdDSA の OSS による採用状況

項番	OSS 名	開発者	URL	実装箇所 ¹³	公開/更新日 ¹⁴
1	Apple Swift Crypto	Apple	https://github.com/apple/swift-crypto	/Sources/CCryptoBoringSSL/ crypto/curve25519/curve25519.c の ED25519_sign 関数	2021 年 8 月 17 日
2	BoringSSL	Google	https://boringssl. googlesource.com/boringssl/	/crypto/curve25519/ curve25519.c の ED25519_sign 関数	2021 年 5 月 28 日
3	Botan	randombit	https://github.com/random- bit/botan	/src/lib/pubkey/ed25519/ ed25519.cpp の ed25519_sign 関数	2020 年 11 月 5 日
4	Bouncy Castle	Legion of the Bouncy Castle Inc.	https://github.com/bcgit/bc- java	/core/src/main/java/org/ bouncycastle/math/ec/ rfc8032/Ed25519.java の Ed25519 ク ラス	2021 年 2 月 11 日
5	blockchain-crypto-mpc	Unbound Security	https://github.com/ unboundsecurity/blockchain- crypto-mpc	src/crypto_utils/ ec25519_core.cpp の ED25519_sign 関数	2019 年 2 月 14 日
6	CIRCL	CloudFlare	https://github.com/ cloudflare/circl	/sign/ed25519/ed25519.go の sig- nAll 関数	2020 年 11 月 20 日
7	cothority	Decentralized Distributed Systems Laboratory	https://github.com/dedis/ cothority	/external/java/src/main/java /ch/epfl/dedis/lib/crypto/ SchnorrSig.java の SchnorrSig クラ ス	2019 年 1 月 17 日

¹³ 複数のソースファイルに実装されている場合は代表として Ed25519 の署名生成の実装箇所とした。

¹⁴ 実装箇所に記載したソースコードの公開/更新日とした。

項番	OSS 名	開発者	URL	実装箇所 ¹³	公開/更新日 ¹⁴
8	eddsa	Invisible Internet Project (I2P)	https://github.com/str4d/ed25519-java	/src/net/i2p/crypto/eddsa/ EdDSAEngine.java の EdDSAEngine クラス	2018 年 5 月 6 日
9	go	Google	https://cs.open- source.google/go/go/+/-/mas- ter:	src/crypto/ed25519/ed25519. go の sign 関数	2021 年 12 月 2 日
10	Hyperledger Iroha/iroha-ed25519	Linux Foundation	https://github.com/ hyperledger/iroha-ed25519	/lib/ed25519/ref10/ed25519.c の ed25519_sign 関数	2019 年 2 月 15 日
11	Libgcrypt	The GnuPG project	https://git.gnupg.org/cgi- bin/gitweb.cgi?p=libgcrypt. git;a=tree	/cipher/ecc-eddsa.c の _gcry_ecc_eddsa_sign 関数	2021 年 5 月 17 日
12	libsodium	libsodium project	https://github.com/jedisct1/ libsodium	/src/libsodium/crypto_sign/ ed25519/ref10/sign.c の crypto_sign_ed25519 関数	2019 年 5 月 21 日
13	Monocypher	Monocypher	https://github.com/ LoupVaillant/Monocypher	/src/optional/ monocypher-ed25519.c の crypto_ed25519_sign 関数	2020 年 5 月 3 日
14	NaCl	欧州委員会 CASE プロジェクト	https://nacl.cr.yp.to/ install.html	crypto_sign/ edwards25519sha512batch/ref/ sign.c の crypto_sign 関数	2011 年 2 月 21 日
15	Nettle	Niels Möller	https://git.lysator.liu.se/ nettle/nettle	/eddsa-sign.c の _eddsa_sign 関数	2020 年 11 月 9 日
16	Olm	Matrix.org Founda- tion	https://gitlab.matrix.org/ matrix-org/olm	/lib/ed25519/src/sign.c の ed25519_sign 関数	2015 年 5 月 3 日

項番	OSS 名	開発者	URL	実装箇所 ¹³	公開/更新日 ¹⁴
17	OpenJDK	Oracle	https://github.com/openjdk/jdk	/src/jdk.crypto.ec/share/classes/sun/security/ec/ed/EddSAOperations.java の EddSAOperations クラス	2020 年 5 月 19 日
18	OpenSSL	OpenSSL	https://github.com/openssl/openssl	/crypto/ec/curve25519.c の openssl_ed25519_sign 関数	2021 年 4 月 8 日
19	SUPERCOP	VAMPIRE lab	http://bench.cr.yp.to/supercop.html	crypto_sign/ed25519/amd64-64-24k/sign.c の crypto_sign 関数	2021 年 11 月 8 日
20	wolfSSL	wolfSSL Inc.	https://github.com/wolfssl/wolfssl	/wolfcrypt/src/ed25519.c の wc_ed25519_sign_msg_ex 関数	2021 年 11 月 24 日

Ed25519 についての採用は多く発見された一方で、Ed448 についての採用はほとんど発見されなかった。

また、多くの OSS がソース管理で利用している GitHub で EdDSA を検索すると 82 件のプロジェクトが検索条件に当てはまった。同様に Ed25519 を検索すると 673 件のプロジェクトが検索条件に当てはまった一方で、Ed448 を検索すると 38 件のプロジェクトが検索条件に当てはまった。

以上のことから、Ed25519 に比べて Ed448 の OSS 実装が少ないと言え、これは標準化仕様での採用状況にも関係していることが予想される。

4.4. EdDSA 実装の脆弱性に関する状況

OSS が EdDSA を採用しつつある中、基幹系システムや政府系システムなど、ミッションクリティカルなシステムでの利用が進むためには、動作が安定し、脆弱性がないことが重要となる。そのような成熟した実装とするためには、多くの人による様々な環境での動作確認に加え、専門家による実装の分析や脆弱性の指摘が行われ、正しく修正されたことの確認まで行われる必要がある。

ここでは、専門家による脆弱性の指摘の状況を把握するため、EdDSA に関する脆弱性について NIST の脆弱性データベースに関する検索サイト¹⁵で調査を行った。

検索キーワードとして“eddsa”を用いて検索したところ、2 件の脆弱性に関する情報が発見された。

¹⁵ <https://nvd.nist.gov/vuln/search>

NVD - CVE-2017-9526 (nist.gov)¹⁶

CVE-2017-9526 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Current Description

In Libcrypt before 1.7.7, an attacker who learns the EdDSA session key (from side-channel observation during the signing process) can easily recover the long-term secret key. 1.7.7 makes a cipher/ecc-eddsa.c change to store this session key in secure memory, to ensure that constant-time point operations are used in the MPI library.

[+View Analysis Description](#)

Severity

CVSS Version 3.x

CVSS Version 2.0

CVSS 3.x Severity and Metrics:



NIST: NVD

Base Score: **5.9 MEDIUM**

Vector: CVSS:3.0/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:N/A:N

NVD Analysts use publicly available information to associate vector strings and CVSS scores. We also display any CVSS information provided within the CVE List from the CNA.

Note: NVD Analysts have published a CVSS score for this CVE based on publicly available information at the time of analysis. The CNA has not provided a score within the CVE List.

8 CVE-2017-9526

2017 年 6 月 10 日に公開された脆弱性である。

直接的には、Libcrypt の脆弱性であるが、OS 等の多くの製品に影響する脆弱性である。

修正前は、セッション鍵を通常のメモリに保管したまま、そのセッション鍵で楕円曲線上の点の演算を行っていた。通常のメモリ上のセッション鍵で楕円曲線上の点の演算を行うと処理時間がセッション鍵の影響を受けるため、署名生成処理中の SCA によって攻撃者にセッション鍵が知られてしまい、長期秘密鍵も簡単に知られてしまう。

修正後は、セッション鍵は安全なメモリ上に保管され、楕円曲線上の点の演算の処理時間はセッション鍵に依存せず一定時間となった。

Libcrypt 1.7.7 以降では解消されている。

¹⁶ <https://nvd.nist.gov/vuln/detail/CVE-2017-9526>

✖ CVE-2021-20305 Detail

MODIFIED

This vulnerability has been modified since it was last analyzed by the NVD. It is awaiting reanalysis which may result in further changes to the information provided.

Current Description

A flaw was found in Nettle in versions before 3.7.2, where several Nettle signature verification functions (GOST DSA, EDDSA & ECDSA) result in the Elliptic Curve Cryptography point (ECC) multiply function being called with out-of-range scalars, possibly resulting in incorrect results. This flaw allows an attacker to force an invalid signature, causing an assertion failure or possible validation. The highest threat to this vulnerability is to confidentiality, integrity, as well as system availability.

[+View Analysis Description](#)

Severity

CVSS Version 3.x

CVSS Version 2.0

CVSS 3.x Severity and Metrics:



NIST: NVD

Base Score: **8.1 HIGH**

Vector: CVSS:3.1/AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:H

NVD Analysts use publicly available information to associate vector strings and CVSS scores. We also display any CVSS information provided within the CVE List from the CNA.

Note: NVD Analysts have published a CVSS score for this CVE based on publicly available information at the time of analysis. The CNA has not provided a score within the CVE List.

図 9 CVE-2021-20305

2021 年 4 月 5 日に公開された脆弱性である。

直接的には、Nettle の脆弱性であるが、OS 等多くの製品に影響する脆弱性である。

修正前は、GOST DSA, ECDSA, EdDSA の署名検証関数で、特定の署名に対して、範囲外のスカラー値で楕円曲線上の点のスカラー倍を実行し、誤った結果を生じる可能性があった。これにより攻撃者は不正な署名の検証を強制して、アサーションの失敗（クラッシュ）や不正な署名の検証成功を引き起こすことができた。最悪の場合、機密性、整合性、システムの可用性が脅かされる。

修正後は、範囲外のスカラー値はスカラー倍の前に修正されるようになった。これにより、不正な署名の検証を強制したとしても、アサーションの失敗（クラッシュ）や不正な署名の検証成功を引き起こすことはできなくなった。

Nettle 3.7.2 以降では解消されている。

¹⁷ <https://nvd.nist.gov/vuln/detail/CVE-2021-20305>

5. 現在の電子政府推奨暗号リストに含まれる同技術分類の方式（ECDSA）との性能比較調査

現在，CRYPTREC では理論上の安全性等に関する外部評価を行ってきた．また，4 章の調査によって，標準仕様や OSS などにおいて EdDSA が世間で広く受け入れられつつあることが分かった．ここでは，実際にソフトウェアとして実装された EdDSA と，CRYPTREC 暗号リストで電子政府推奨暗号リストに選択されている ECDSA の実装性能を比較調査する．

5.1. 性能比較調査について

性能比較調査では EdDSA と ECDSA を実測で比較する．

OSS により高速化技術にばらつきがあると予想されるため，1 つの OSS で ECDSA と EdDSA 両方が実装されたものを比較することが望ましい．

このため，広く知られているメジャーな暗号ライブラリから，Ed25519，Ed448 が実装され，安全性の近い ECDSA と EdDSA が両方実装されているものを選定して測定した．

また，本調査の実測は CPU やアーキテクチャが限定的であるが，第三者が行った多くの環境での性能比較結果を分析することで，CPU やアーキテクチャの違いによって EdDSA と ECDSA の性能比較に影響があるかどうか補助的に確認し，5.3 節に記述した．

実測した測定結果を比較し，EdDSA の性能が ECDSA に比べてどのような特性を示すかを調査した．

5.2. ECDSA および EdDSA が実装されている OSS の選定

4.3 節で調査した広く知られているメジャーな暗号ライブラリから以下の条件を満たす OSS を選定した．（表 4）

- Ed25519 および Ed448 が実装されている．
 - 安全性の観点から評価対象である EdDSA と比較対象と妥当と想定される ECDSA (P-256, P-384 および P-521) がすべて実装されている．
- ボールド体のライブラリが条件を満たしていた．

表 4 ECDSA および EdDSA が実装されている選定した OSS 一覧

項番	暗号ライブラリ	EdDSA		ECDSA		
		Ed25519	Ed448	P-256	P-384	P-521
1	Apple Swift Crypto	○	×	○	○	○
2	BoringSSL	○	×	○	○	○
3	Botan	○	×	○	○	○
4	Bouncy Castle	○	○	○	○	○

項番	暗号ライブラリ	EdDSA		ECDSA		
		Ed25519	Ed448	P-256	P-384	P-521
5	blockchain-crypto-mpc	○	×	○	○	○
6	CIRCL	○	○	×	×	×
7	cothority	○	×	×	×	×
8	eddsa	○	×	×	×	×
9	go	○	×	○	○	○
10	Hyperledger Iroha/iroha-ed25519	○	×	×	×	×
11	Libgcrypt	○	○	○	○	○
12	libsodium	○	×	×	×	×
13	Monocypher	○	×	×	×	×
14	NaCl	○	×	×	×	×
15	Nettle	○	○	○	○	○
16	01m	○	×	×	×	×
17	OpenJDK	○	○	○	○	○
18	OpenSSL	○	○	○	○	○
19	SUPERCOP	○	○	○	○	○
20	wolfSSL	○	○	○	○	○

5.3. 第三者による性能比較

本調査の実測はCPUやアーキテクチャが限定的であるが、4章の調査中に発見された第三者が行った性能比較結果を分析することで、補助的にCPUやアーキテクチャの違いによってEdDSAとECDSAの性能比較に影響があるかどうか確認した。

5.3.1. eBATSによる測定結果

eBATSは最も高速な公開鍵システムを探索するプロジェクトである。

European Network of Excellence for Cryptology II (ECRYPT II)¹⁸の一部であるVirtual Applications and Implementations Research Lab (VAMPIRE)¹⁹が構築し、現在も計測と記録が継続されている。

eBATSにはSUPERCOPで計測したECDSAおよびEdDSAの計測結果が記録されている。

なお、本サイトで、ECDSA P-XXXはecdondpXXX、Ed25519はed25519、Ed448はed448goldilocksという名前で記載されているが²⁰、本報告書ではECDSA P-XXX、Ed25519、Ed448という表記に統一する。

¹⁸ <https://www.ecrypt.eu.org/ecrypt2/>

¹⁹ <http://hyperelliptic.org/ECRYPTII/vampire/>

²⁰ <https://bench.cr.yp.to/impl-sign/ecdondp256.html>

Implementation comparison: crypto_sign/ecdonaldp256

Operation	crypto_sign	
Primitive	ecdonaldp256	
Description	ECDSA signatures using the standard NIST P-256 elliptic curve, a curve modulo the prime $2^{256}-2^{224}+2^{192}+2^{96}-1$	
Designers	Example for eBATS	
Implementations	openssl	Daniel J. Bernstein (wrapper around OpenSSL)
	openssl new	Daniel J. Bernstein (wrapper around OpenSSL)

図 10 Implementation comparison: crypto_sign/ecdonaldp256

ここでは、計測結果のサイト²¹から ECDSA P-256, ECDSA P-384, ECDSA P-521, Ed25519, Ed448 の結果がそろっている、以下の 12 個の測定環境と各計測結果を抜き出した。

表 5 Implementation comparison: 測定環境

環境番号	詳細
[環境 1]	amd64; TigerLake (806c1); 2020 Intel Core i7-1165G7; 4 x 2800MHz; unstable; pascalinspiron75062n1, supercop-20210125
[環境 2]	amd64; IceLake (706e5); 2020 Intel Core i5-1030NG7; 4 x 1100MHz; icelake, supercop-20200826
[環境 3]	amd64; Zen2 (830f10); 2019 AMD EPYC 7742; 64 x 2250MHz; unstable; colossus6, supercop-20210125
[環境 4]	amd64; CometLake (806ec); 2019 Intel Core i3-10110U; 2 x 2100MHz; comet, supercop-20211108
[環境 5]	amd64; CascadeLake (50657); 2019 Intel Core i9-10980XE; 18 x 3000MHz; avx512imath, supercop-20210125
[環境 6]	amd64; CoffeeLake (906ea); 2018 Intel Xeon E-2124; 4 x 3300MHz; r24000, supercop-20211108
[環境 7]	amd64; Zen (800f12); 2017 AMD EPYC 7501; 32 x 2000MHz; unstable; colossus5, supercop-20210125
[環境 8]	amd64; Zen (800f12); 2017 AMD EPYC 7451; 48 x 2300MHz; unstable; firefly, supercop-20201130
[環境 9]	amd64; Zen (800f11); 2017 AMD Ryzen 7 1700; 8 x 3000MHz; rumba7, supercop-20211108
[環境 10]	amd64; Zen (800f11); 2017 AMD Ryzen 5 1600; 6 x 3200MHz; rumba5, supercop-20211108
[環境 11]	amd64; Zen (800f11); 2017 AMD Ryzen 3 1200; 4 x 3100MHz; rumba3, supercop-20200906
[環境 12]	amd64; KabyLake (906e9); 2017 Intel Xeon E3-1220 v6; 4 x 3000MHz; kizomba, supercop-20211108

ここでは、計測結果のうち、鍵ペア生成 (Cycles to generate a key pair) , 署名生成 (Cycles to sign 59 bytes) , 署名検証 (Cycles to verify 59 bytes) の処理にかかったサイクル数の中央値 (四分位数の 50%) を抜粋した。

²¹ <https://bench.cr.yp.to/results-sign.html>

サイクル数が大きいと処理に時間が長く，サイクル数が小さいと処理の時間が短いことを示す．

また，本節のグラフでは縦軸はサイクル数を表している．

① 鍵ペア生成

表 6 鍵ペア生成におけるサイクル数計測結果の中央値

	[環境1]	[環境2]	[環境3]	[環境4]	[環境5]	[環境6]	[環境7]	[環境8]	[環境9]	[環境10]	[環境11]	[環境12]
ECDSA P-256	146,136	146,442	76,523	106,062	95,752	97,081	117,020	148,166	117,481	117,843	120,094	108,705
ECDSA P-384	5,599,777	3,280,399	2,434,410	3,379,855	3,281,797	3,286,632	3,561,460	4,499,559	3,553,883	3,549,587	3,573,060	3,340,724
ECDSA P-521	1,211,113	7,502,147	511,852	717,799	710,068	707,880	838,820	984,653	841,169	840,831	864,838	711,153
Ed25519	80,195	114,320	33,593	45,611	46,905	45,665	48,200	61,663	48,442	48,303	49,104	45,764
Ed448	315,396	139,697	121,388	153,872	182,441	153,474	237,300	330,694	200,014	199,763	199,702	153,332

128 ビットセキュリティの ECDSA P-256 と Ed25519 を比較したグラフを図 11 に示す。

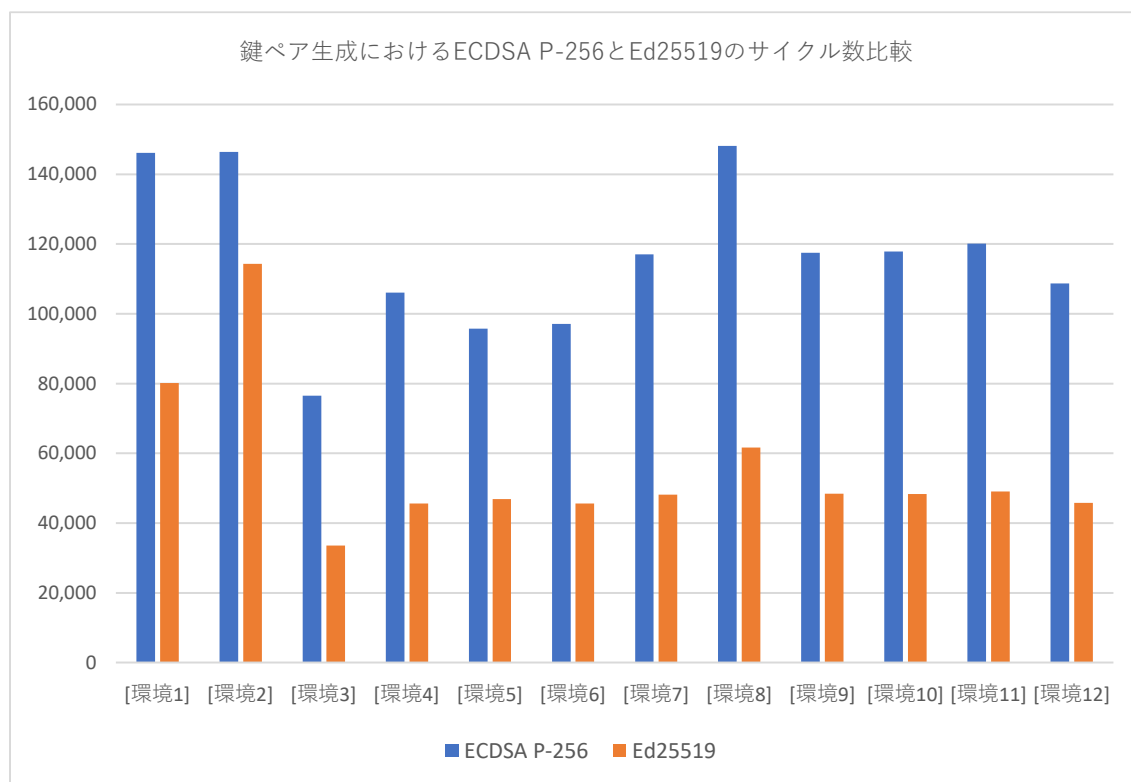


図 11 鍵ペア生成における ECDSA P-256 と Ed25519 のサイクル数比較

図 11 から、12 環境中 12 環境で ECDSA P-256 より Ed25519 が少ないサイクル数で鍵ペアを生成できており、CPU 等の環境によらず EdDSA の鍵ペア生成のほうが高速であることが分かった。

192 ビットセキュリティの ECDSA P-384, 256 ビットセキュリティの ECDSA P-521, 224 ビットセキュリティの Ed448 を比較したグラフを図 12 に示す。

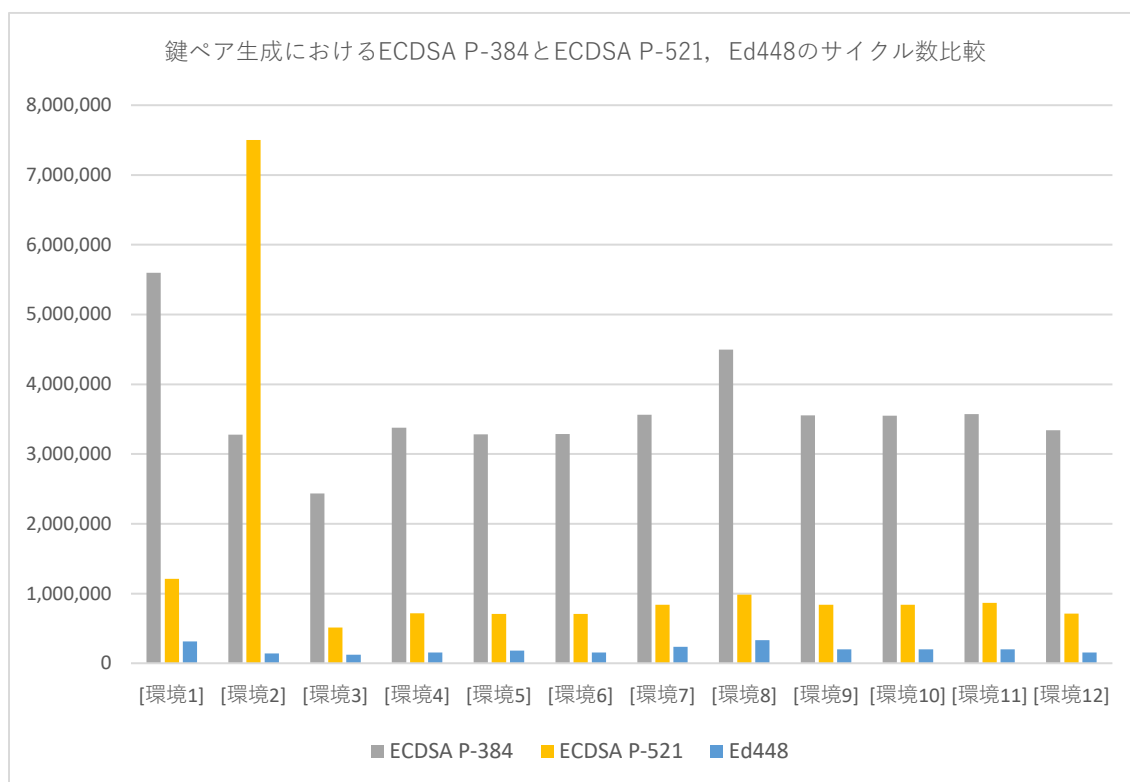


図 12 鍵ペア生成における ECDSA P-384 と ECDSA P-521, Ed448 のサイクル数比較

図 12 から、12 環境中 11 環境で ECDSA P-384 より ECDSA P-521 が少ないサイクル数で鍵ペアを生成できており、[環境 2]だけ ECDSA P-384 より ECDSA P-521 が遅いことが分かる。[環境 2]は SUPERCOP のバージョンが最も古いことから、それ以降の SUPERCOP では ECDSA P-521 の実装が高速化されたと推測される。

ECDSA と EdDSA の比較という観点では、12 環境中 12 環境で ECDSA P-384, ECDSA P-521 より Ed448 が少ないサイクル数で鍵ペアを生成できており、CPU 等の環境によらず EdDSA の鍵ペア生成のほうが高速であることが分かった。

② 署名生成

表 7 署名生成におけるサイクル数計測結果の中央値

	[環境1]	[環境2]	[環境3]	[環境4]	[環境5]	[環境6]	[環境7]	[環境8]	[環境9]	[環境10]	[環境11]	[環境12]
ECDSA P-256	196,904	201,443	98,978	144,568	130,219	133,710	151,300	192,717	151,779	151,768	154,814	141,272
ECDSA P-384	5,986,180	3,468,754	2,536,267	3,494,435	3,416,282	3,415,800	3,707,800	4,636,777	3,702,111	3,696,594	3,725,332	3,432,919
ECDSA P-521	1,839,049	8,064,420	782,370	1,094,418	1,078,906	1,073,777	1,284,780	1,481,982	1,283,307	1,284,174	1,284,299	1,081,181
Ed25519	85,034	117,932	35,460	48,796	50,023	48,744	51,060	67,183	51,300	51,194	52,514	48,740
Ed448	330,815	146,873	126,653	161,783	189,019	161,084	246,280	342,907	206,307	206,964	206,677	160,529

128 ビットセキュリティの ECDSA P-256 と Ed25519 を比較したグラフを図 13 に示す。

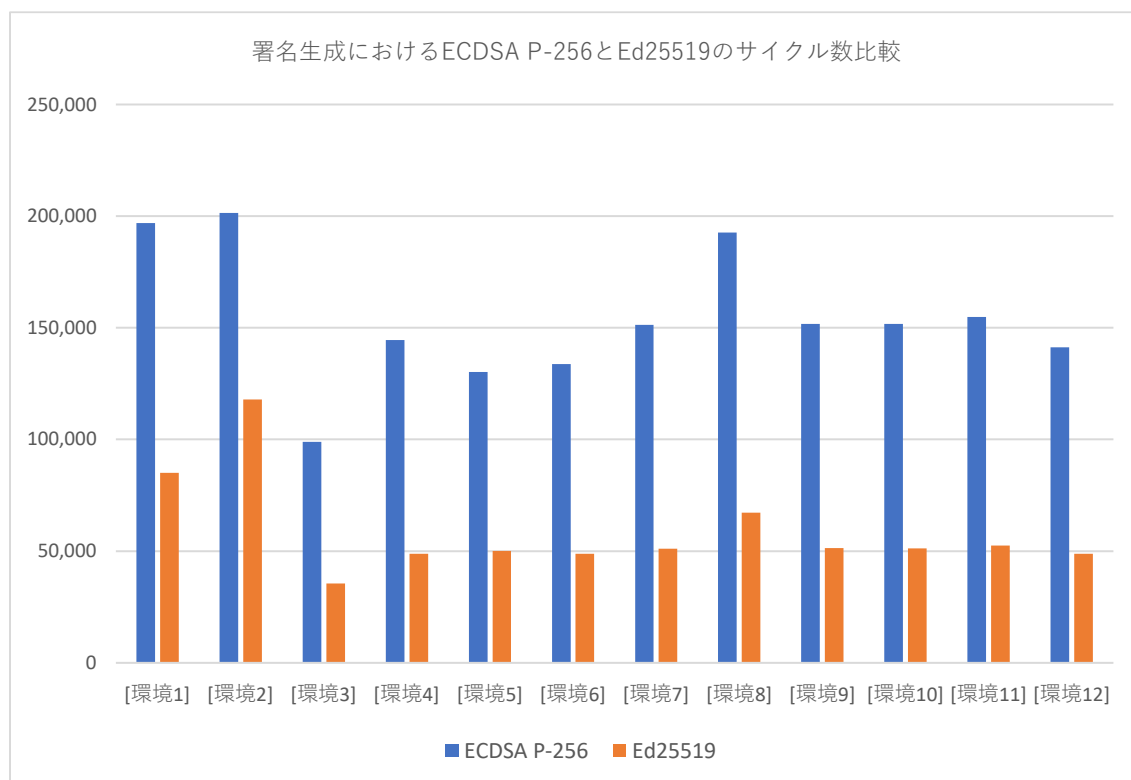


図 13 署名生成における ECDSA P-256 と Ed25519 のサイクル数比較

図 13 から、12 環境中 12 環境で ECDSA P-256 より Ed25519 が少ないサイクル数で署名を生成できており、CPU 等の環境によらず EdDSA の署名生成のほうが高速であることが分かった。

192 ビットセキュリティの ECDSA P-384、256 ビットセキュリティの ECDSA P-521、224 ビットセキュリティの Ed448 を比較したグラフを図 14 に示す。

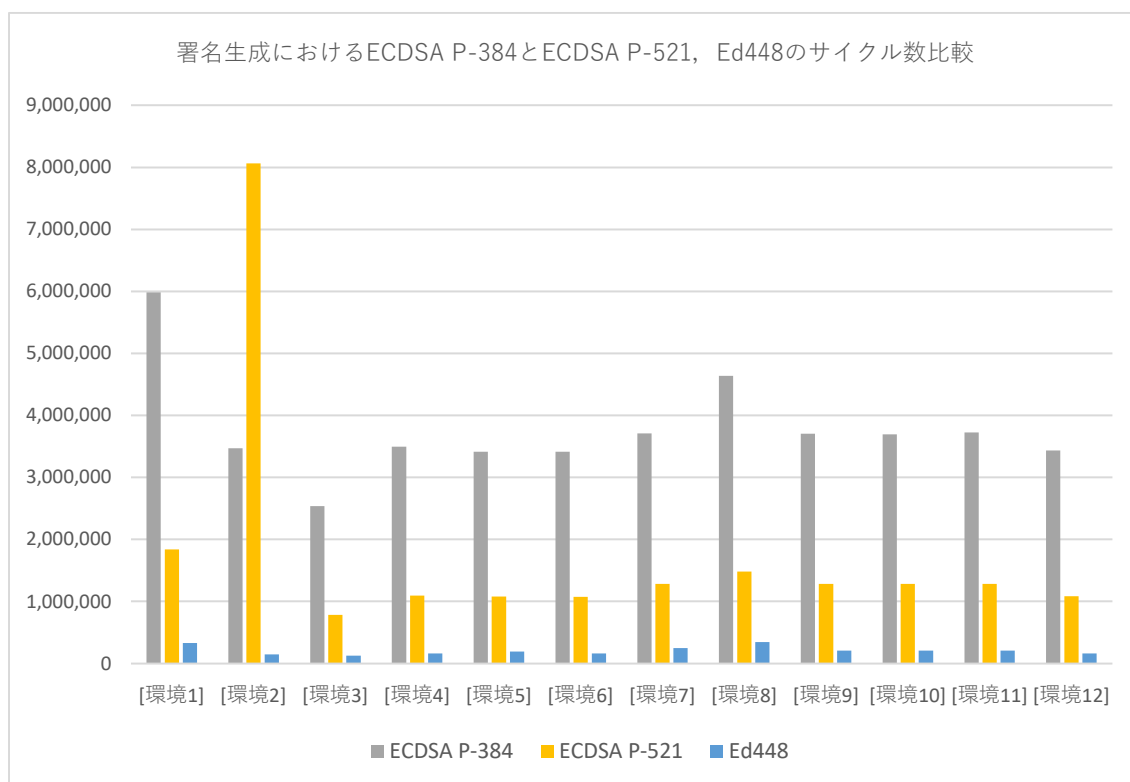


図 14 署名生成における ECDSA P-384 と ECDSA P-521, Ed448 のサイクル数比較

図 14 から、12 環境中 11 環境で ECDSA P-384 より ECDSA P-521 が少ないサイクル数で署名を生成できており、[環境 2]だけ ECDSA P-384 より ECDSA P-521 が遅いことが分かる。[環境 2]は SUPERCOP のバージョンが最も古いことから、それ以降の SUPERCOP では ECDSA P-521 の実装が高速化されたと推測される。

ECDSA と EdDSA の比較という観点では、12 環境中 12 環境で ECDSA P-384, ECDSA P-521 より Ed448 が少ないサイクル数で署名を生成できており、CPU 等の環境によらず EdDSA の署名生成のほうが高速であることが分かった。

③ 署名検証

表 8 署名検証におけるサイクル数計測結果の中央値

	[環境1]	[環境2]	[環境3]	[環境4]	[環境5]	[環境6]	[環境7]	[環境8]	[環境9]	[環境10]	[環境11]	[環境12]
ECDSA P-256	469,712	397,681	217,260	294,201	282,072	283,645	335,040	382,674	336,801	336,971	339,388	279,030
ECDSA P-384	4,607,722	2,829,899	1,949,625	2,737,884	2,658,879	2,662,617	2,852,800	3,930,033	2,827,335	2,833,127	2,912,605	2,598,478
ECDSA P-521	3,520,035	6,299,129	1,480,545	2,053,835	2,037,256	2,028,732	2,433,560	2,816,051	2,430,981	2,433,120	2,436,910	2,029,163
Ed25519	277,176	389,141	123,998	166,294	167,960	164,961	182,480	222,019	182,148	182,150	182,962	165,246
Ed448	1,050,136	509,133	404,280	510,254	576,488	506,935	754,040	975,200	637,525	640,953	637,360	507,970

128 ビットセキュリティの ECDSA P-256 と Ed25519 を比較したグラフを図 15 に示す。

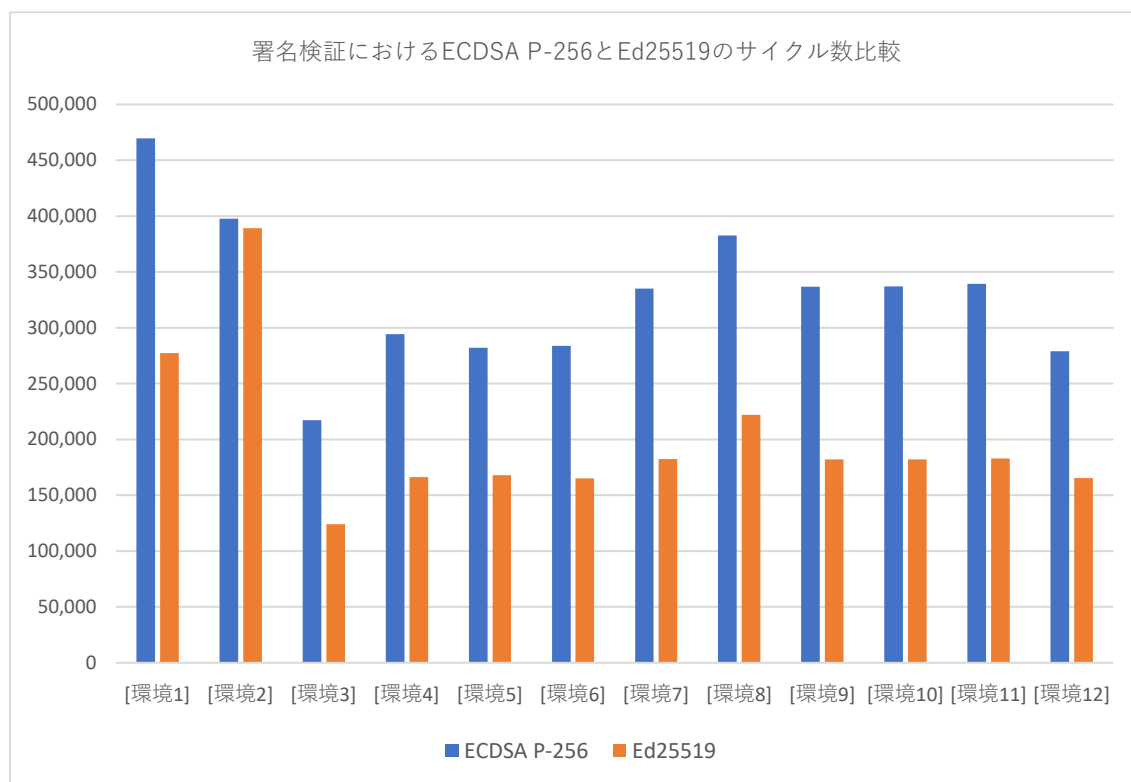


図 15 署名検証における ECDSA P-256 と Ed25519 のサイクル数比較

図 15 から、12 環境中 12 環境で ECDSA P-256 より Ed25519 が少ないサイクル数で署名を検証できており、CPU 等の環境によらず EdDSA の署名検証のほうが高速であることが分かった。

192 ビットセキュリティの ECDSA P-384、256 ビットセキュリティの ECDSA P-521、224 ビットセキュリティの Ed448 を比較したグラフを図 16 に示す。

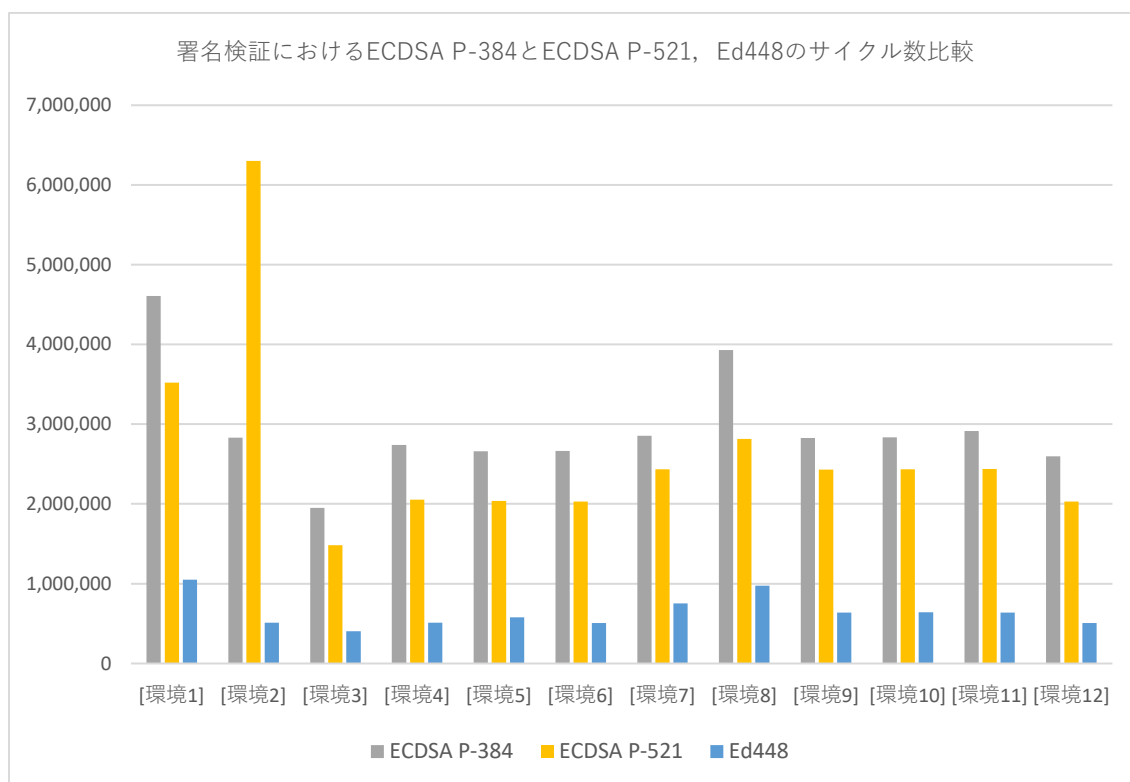


図 16 署名検証における ECDSA P-384 と ECDSA P-521, Ed448 のサイクル数比較

図 16 から、12 環境中 11 環境で ECDSA P-384 より ECDSA P-521 が少ないサイクル数で署名を検証できており、[環境 2]だけ ECDSA P-384 より ECDSA P-521 が遅いことが分かる。[環境 2]は SUPERCOP のバージョンが最も古いことから、それ以降の SUPERCOP では ECDSA P-521 の実装が高速化されたと推測される。

ECDSA と EdDSA の比較という観点では、12 環境中 12 環境で ECDSA P-384, ECDSA P-521 より Ed448 が少ないサイクル数で署名を検証できており、CPU 等の環境によらず EdDSA の署名検証のほうが高速であることが分かった。

以上の整理した eBATS における測定結果から、EdDSA は、同等または高い安全性の ECDSA に比べて、CPU 等の環境によらず高速に動作することが分かった。

5.3.2. 論文 “Is there a case to prefer Ed25519 over ECDSA P-256 for DNSSEC?” での測定結果

University of Twente の J. J. Yu による “Is there a case to prefer Ed25519 over ECDSA P-256 for DNSSEC?”²²によると、図 17, 図 18 のような測定結果となったと報告している。

²² <http://essay.utwente.nl/75354/1/DNSSEC%20curves.pdf>

6.1 Intel

Benchmarking results on the Intel Xeon E3-12xx V2 @2.4Ghz for signature verification and signature generation have been shown in table 1V and 1S and the speed improvements for Ed25519 are shown in table 1CV and 1CS.

6.1.1 Intel: Verifying

Table 1V: Verifications/second of P-256 and Ed25519 on a 2.4Ghz Intel Xeon E3-12xx v2 (Ivy Bridge) running Ubuntu 16.04.4 LTS

	P-256 in OpenSSL 0.98x	P-256 in OpenSSL 1.0.1f	P-256 in OpenSSL 1.0.2e	P-256 in OpenSSL 1.1.0f	Ed25519 in ed25519-donna	Ed25519 in Libsodium 1.0.12	Ed25519 in OpenSSL 1.1.1- dev
Verifications/ second	1968.6	1944.0	6579.4	6345.8	12345.0	5502.3	6043.9
Standard deviation	28.6	31.0	78.1	83.8	139.1	16.3	43.1

Table 1CV: Signature verification speed improvement of Ed25519 compared to P-256 on a 2.4Ghz Intel Xeon E3-12xx v2 (Ivy Bridge) running Ubuntu 16.04.4 LTS

Signature verification speed improvements for Ed25519	Ed25519 in ed25519-donna (=optimized for x86/x64)	Ed25519 in Libsodium 1.0.12	Ed25519 in OpenSSL 1.1.1-dev
P-256 in OpenSSL 0.98x	~6.3X	~2.8X	~3.1X
P-256 in OpenSSL 1.0.1f	~6.4X	~2.8X	~3.1X
P-256 in OpenSSL 1.0.2e (=optimized for x86/x64)	~1.9X	~0.8X	~0.9X
P-256 in OpenSSL 1.1.0f (=optimized for x86/x64)	~1.9X	~0.9X	~1X

As shown in table 1CV, Ed25519 verifies signatures faster when comparing the non-optimized versions of Ed25519 to the non-optimized versions of P-256. E.g. the non-optimized version of Ed25519 in OpenSSL 1.1.1-dev is ~3.1X as fast as the non-optimized version of P-256 in OpenSSL 1.0.1f. Ed25519 is also faster when comparing the optimized version of Ed25519 to the optimized version of P-256. E.g. Ed25519 in ed25519-donna is ~1.9X as fast as P-256 in OpenSSL 1.1.0f. When comparing the non-optimized version of Ed25519 to an optimized version of P-256 in OpenSSL the signature verification speeds are similar to each other. As explained before, such a comparison would make sense, considering that DNSSEC software could rely on cryptographic implementations that are available in OpenSSL.

6.1.2 Intel: Signing

Table 1S: Signatures/second of P-256 and Ed25519 on a 2.4Mhz Intel Xeon E3-12xx v2 (Ivy Bridge) running Ubuntu 16.04.4 LTS

	P-256 in OpenSSL 0.98x	P-256 in OpenSSL 1.0.1f	P-256 in OpenSSL 1.0.2e	P-256 in OpenSSL 1.1.0f	Ed25519 in ed25519-donna	Ed25519 in Libsodium 1.0.12	Ed25519 in OpenSSL 1.1.1-dev
Signatures/second	7720.2	7329.9	15909.8	14905.0	40872.0	14463.6	20123.2
Standard deviation	57.0	71.7	361.3	556.2	294.2	237.9	146.7

図 17 論文 “Is there a case to prefer Ed25519 over ECDSA P-256 for DNSSEC?” での測定結果 (Table IV, ICV, IS)

Table 1CS: Signature generation speed improvement of Ed25519 compared to P-256 on a 2.4Ghz Intel Xeon E3-12xx v2 (Ivy Bridge) running Ubuntu 16.04.4 LTS

Signature generation speed improvements for Ed25519	Ed25519 in ed25519-donna (=optimized for x86/x64)	Ed25519 in Libsodium 1.0.12	Ed25519 in OpenSSL 1.1.1-dev
P-256 in OpenSSL 0.98x	~5.3X	~1.9X	~2.6X
P-256 in OpenSSL 1.0.1f	~5.6X	~2.0X	~2.7X
P-256 in OpenSSL 1.0.2e (=optimized for x86/x64)	~2.6X	~0.9X	~1.3X
P-256 in OpenSSL 1.1.0f (=optimized for x86/x64)	~2.7X	~1X	~1.4X

As shown in table 1CS, the non-optimized versions of Ed25519 generates signatures faster than the non-optimized versions of P-256. E.g. the non-optimized version of Ed25519 in OpenSSL 1.1.1-dev is ~2.7X as fast as the non-optimized version of P-256 in OpenSSL 1.0.1f. Ed25519 is also faster when comparing the optimized version of Ed25519 to the optimized version of P-256. E.g. Ed25519 in ed25519-donna is ~2.7X as fast as P-256 in OpenSSL 1.1.0f. When comparing the non-optimized version of Ed25519 to an optimized version of P-256 the performance is similar to each other. In which the OpenSSL implementation of Ed25519 is slightly faster.

図 18 論文 “Is there a case to prefer Ed25519 over ECDSA P-256 for DNSSEC?” での測定結果 (Table ICS)

ここでは Intel の結果のみ引用したが、結論としては、ECDSA P-256 より最適化された Ed25519 のほうが性能がよいという結果を報告している。

5.3.3. 論文 “Size, Speed, and Security: An Ed25519 Case Study” での測定結果

Tampere University の Cesar Pereida García¹ と Huawei Technologies の Sampo Sovio による “Size, Speed, and Security: An Ed25519 Case Study”²³によると図 19 のような測定結果となったと報告している。

Architecture	Implementation	Sign	Verify	KeyGen
x86_64	ref10	140 (□ base)	455 (□ base)	135 (□ base)
	ref	1560 (▽11.1x)	5218 (▽11.4x)	1531 (▽11.3x)
	amd64-64-24k	64 (▲2.18x)	225 (▲2.02x)	60 (▲2.25x)
	amd64-51-30k	66 (▲2.12x)	210 (▲2.16x)	62 (▲2.17x)
	donna	64 (▲2.18x)	217 (▲2.09x)	59 (▲2.28x)
	monocypher	230 (▽1.64x)	525 (▽1.15x)	210 (▽1.55x)
	overoptimized	264 (▽1.88x)	455 (▽1.00x)	227 (▽1.68x)
	ecckila-precomp	101 (▲1.38x)	280 (▲1.62x)	96 (▲1.4x)
x86	ref10	399 (□ base)	1155 (□ base)	374 (□ base)
	ref	4137 (▽10.3x)	14105 (▽12.2x)	4086 (▽10.9x)
	amd64-64-24k	—	—	—
	amd64-51-30k	—	—	—
	donna	310 (▲1.28x)	962 (▲1.20x)	291 (▲1.28x)
	monocypher	533 (▽1.33x)	1347 (▽1.16x)	471 (▽1.25x)
	overoptimized	958 (▽2.40x)	1155 (▽1.00x)	914 (▽2.44x)
	ecckila-no-precomp	1133 (▽2.83x)	1231 (▽1.06x)	1075 (▽2.87x)
	ecckila-precomp	427 (▽1.07x)	1228 (▽1.06x)	368 (▲1.01x)

Table 2. Comparison of timings on Intel architecture. □ is the baseline. ▲ means a speedup (better) w.r.t. baseline. ▽ means a slowdown (worst) w.r.t. baseline. Timings are given in clock cycles (thousands).

Architecture	Implementation	Sign	Verify	KeyGen
aarch64	ref10	245 (□ base)	688 (□ base)	238 (□ base)
	ref	2924 (▽11.9x)	9579 (▽13.9x)	2425 (▽10.1x)
	amd64-64-24k	—	—	—
	amd64-51-30k	—	—	—
	donna	196 (▲1.25x)	638 (▲1.07x)	162 (▲1.46x)
	monocypher	422 (▽1.72x)	812 (▽1.18x)	366 (▽1.53x)
	overoptimized	726 (▽2.96x)	688 (▽1.00x)	635 (▽2.66x)
	ecckila-precomp	270 (▽1.10x)	808 (▽1.17x)	261 (▽1.09x)
armv7l	ref10	597 (□ base)	1755 (□ base)	582 (□ base)
	ref	9933 (▽16.6x)	28642 (▽16.3x)	8442 (▽14.5x)
	amd64-64-24k	—	—	—
	amd64-51-30k	—	—	—
	donna	508 (▲1.17x)	1508 (▲1.16x)	495 (▲1.17x)
	monocypher	983 (▽1.64x)	2505 (▽1.42x)	987 (▽1.69x)
	overoptimized	1622 (▽2.71x)	1800 (▽1.02x)	1534 (▽2.63x)
	ecckila-no-precomp	2134 (▽3.57x)	2237 (▽1.27x)	2050 (▽3.52x)
	ecckila-precomp	815 (▽1.36x)	2213 (▽1.26x)	732 (▽1.25x)

Table 3. Comparison of timings on ARM architecture. □ is the baseline. ▲ means a speedup (better) w.r.t. baseline. ▽ means a slowdown (worst) w.r.t. baseline. Timings are given in clock cycles (thousands).

図 19 論文 “Size, Speed, and Security: An Ed25519 Case Study”
での測定結果 (Table 2, 3)

これは Ed25519 のみ実装性能を測定したものであるが、最適化された高速な Ed25519 では鍵生成が 60000 クロック、署名生成が 64000 クロック、署名検証が 210000 クロックで処理が可能であることを報告している。

²³ <https://eprint.iacr.org/2021/471.pdf>

CPU の種類等によって、どの程度の性能差が出るかは分かったが、OS、言語の違いでどのような特性を示すかは分からなかった。

5.4. ECDSA および EdDSA に関する性能評価

本章では ECDSA と EdDSA の実測した性能比較結果について報告する。

5.4.1. 性能評価対象

ここでは、5.2 で選定した OSS について性能評価対象とする。
性能評価対象の OSS の詳細を表 9 に示す。

表 9 性能評価対象 OSS の詳細

項番	OSS 名	バージョン	言語
1	Bouncy Castle	1.69	Java
2	OpenJDK	JDK 17	Java
3	Libgcrypt	1.9.4	C
4	Nettle (GMP を使用)	3.7.3 (GMP は 6.2.1)	C
5	OpenSSL	3.0.0	C
6	wolfSSL	5.0.0	C

5.4.2. 性能測定方法

実機の Windows マシン、Linux マシン、Mac マシンを使い、実際の性能を測定した。
以下では、Windows のための記述には〈Win〉、Linux のための記述には〈Linux〉、
MacOS のための記述には〈Mac〉と記載する。

測定対象の OSS は、以下の方針で入手した。

- Java のライブラリは各 OS 向けにビルド済みの配布されているバイナリを入手した。
- C 言語のライブラリは、各 OS の一般的な開発環境でビルドする方法が開発者によって示されている場合、その開発環境でビルドする。各 OS の一般的な開発環境として以下を想定した。
 - 〈Win〉 Microsoft Visual Studio Professional 2022 (64 ビット) Version 17.0.2 (VS)
 - 〈Linux〉 GNU Compiler Collection (gcc) (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0
 - 〈Mac〉 Apple Xcode 13.2
- 上記以外の場合、OS が当該バージョンをディストリビューションで配布している場合は配布されているものを利用する。

- そのようなパッケージが存在しない場合、開発者が推奨する方法でビルドする。

OSS の個別の入手方法については、測定手順の中で後述する。

測定に使用した環境は以下のとおりである。

(1) 〈Win〉 計測マシン

CPU : Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz

搭載メモリ : 32.0 GB

OS : Windows 10 Pro 21H1

シェル : コマンド プロンプト

Java : JDK 17.0.1 General-Availability Release²⁴の Windows / x64 版を任意のディレクトリに展開し、Windows の環境変数 PATH に展開した jdk-17.0.1 の bin ディレクトリの絶対パスを追加することでインストールした。

- OpenJDK Runtime Environment (build 17.0.1+12-39)
- OpenJDK 64-Bit Server VM (build 17.0.1+12-39, mixed mode, sharing)

(2) 〈Linux〉 計測マシン

CPU : Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz

搭載メモリ : 32.0 GB

OS : Ubuntu 18.04.6 LTS

シェル : ターミナル (bash)

Java : コマンド

```
sudo apt install openjdk-17-jdk
```

を実行することで Ubuntu パッケージをインストールした。

- OpenJDK Runtime Environment (build 17.0.1+12-Ubuntu-118.04)
- OpenJDK 64-Bit Server VM (build 17.0.1+12-Ubuntu-118.04, mixed mode, sharing)

(3) 〈Mac〉 計測マシン

CPU : 2.3 GHz 8 コア Intel Core i9

搭載メモリ : 64 GB 2667 MHz DDR4

OS : macOS Big Sur 11.4 (20F71)

シェル : ターミナル (zsh)

²⁴ <https://jdk.java.net/17/>

Java : JDK 17.0.1 General-Availability Release²⁴ の macOS /x64 版をスーパーユーザで/Library/Java/JavaVirtualMachines/に展開し, コマンド

```
export JAVA_HOME=`/usr/libexec/java_home -v17`
```

を実行することでインストールした.

- Java(TM) SE Runtime Environment (build 17.0.1+12-LTS-39)
- Java HotSpot(TM) 64-Bit Server VM (build 17.0.1+12-LTS-39, mixed mode, sharing)

以下では, 各 OSS について, 各実行環境の入手方法, 計測用プログラムの準備, 計測用プログラムの実行方法について説明する.

① Bouncy Castle

【OSS の入手】

〈Win〉 〈Linux〉 〈Mac〉 Bouncy Castle 1.69 を以下からダウンロードした.

<https://www.bouncycastle.org/download/bcprov-jdk15on-169.jar>

【計測用プログラムの準備】

計測用プログラムは, ECDSA P-256, ECDSA P-384, ECDSA P-521, Ed25519, Ed448 について

- KeyPairGenerator クラスの generateKeyPair メソッドでの鍵生成
- Signature クラスの initSign, update, sign メソッドを一連の処理としたブロックでの署名生成
- Signature クラスの initVerify, update, verify メソッドを一連の処理としたブロックでの署名検証

を, それぞれ 10^3 回実行し, それぞれの 10^3 回分の処理時間 (ミリ秒) t を測定する.

t を 10^3 で割ることでミリ秒から秒に変換し, 得られた秒数で 10^3 [回]を割ることで, スループット (1 秒あたりの実行回数) を求めた. 計算式は

$$\frac{10^3}{(t/10^3)} = \frac{10^6}{t} [\text{回/秒}]$$

となる.

Libgcrypt 等, サポートしていない他のライブラリと合わせるため, Ed25519ph, Ed25519ctx, Ed448ph は計測していない.

計測用プログラム EcDsaP256.java (図 20) , EcDsaP384.java (図 21) , EcDsaP521.java (図 22) , EdDsaEd25519.java (図 23) および EdDsaEd448.java (図 24) を作成した.

```

1 package bcBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Security;
7 import java.security.Signature;
8 import java.security.spec.ECGenParameterSpec;
9
10 import org.bouncycastle.util.Strings;
11
12 public class EcDsaP256 {
13 {
14     static final int LOOP = 1000;
15     static final String CURVE = "P-256";
16     static final String ALG = "SHA256withECDSA";
17
18     public static void main(String[] args)
19     throws GeneralSecurityException {
20 {
21     Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
22
23     /* ECDSA Key Pair generator */
24     KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance("EC", "BC");
25     kpGenerator.initialize(new ECGenParameterSpec(CURVE));
26
27     KeyPair[] kp = new KeyPair[LOOP];
28
29     /* ECDSA Key Pair generate */
30     long start = System.currentTimeMillis();
31     for (int i = 0; i < LOOP; ++i) {
32         kp[i] = kpGenerator.generateKeyPair();
33     }
34     long end = System.currentTimeMillis();
35     System.out.println(ALG + " on " + CURVE + " keygen: " + 1000.0 * 1000 / (end - start) + " ops/sec");
36
37     /* Message */
38     byte[] msg = Strings.toByteArray("hello, world!");
39
40     Signature signature = Signature.getInstance(ALG, "BC");
41
42     /* ECDSA sign */
43     byte[][] sig = new byte[LOOP][];
44     start = System.currentTimeMillis();
45     for (int i = 0; i < LOOP; ++i) {
46         signature.initSign(kp[i].getPrivate());
47         signature.update(msg);
48         sig[i] = signature.sign();
49     }
50     end = System.currentTimeMillis();
51     System.out.println(ALG + " on " + CURVE + " sign: " + 1000.0 * 1000 / (end - start) + " ops/sec");
52
53     /* ECDSA verify */
54     boolean[] verify = new boolean[LOOP];
55     start = System.currentTimeMillis();
56     for (int i = 0; i < LOOP; ++i) {
57         signature.initVerify(kp[i].getPublic());
58         signature.update(msg);
59         verify[i] = signature.verify(sig[i]);
60     }
61     end = System.currentTimeMillis();
62     System.out.println(ALG + " on " + CURVE + " verify: " + 1000.0 * 1000 / (end - start) + " ops/sec");
63
64     /* Ensure results */
65     String result = "OK";
66     for (int i = 0; i < LOOP; ++i) {
67         if (!verify[i]) {
68             System.out.println(ALG + " on " + CURVE + " verified failed at " + i);
69             result = "NG";
70         }
71     }
72     System.out.println(ALG + " on " + CURVE + " keygen/sign/verified : " + result);
73 }
74 }

```

図 20 Bouncy Castle EcDsaP256 計測用プログラム EcDsaP256.java

```

1 package bcBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Security;
7 import java.security.Signature;
8 import java.security.spec.ECGenParameterSpec;
9
10 import org.bouncycastle.util.Strings;
11
12 public class EcDsaP384 {
13 {
14     static final int LOOP = 1000;
15     static final String CURVE = "P-384";
16     static final String ALG = "SHA512withECDSA";
17
18     public static void main(String[] args)
19     throws GeneralSecurityException {
20     {
21         Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
22
23         /* ECDSA Key Pair generator */
24         KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance("EC", "BC");
25         kpGenerator.initialize(new ECGenParameterSpec(CURVE));
26
27         KeyPair[] kp = new KeyPair[LOOP];
28
29         /* ECDSA Key Pair generate */
30         long start = System.currentTimeMillis();
31         for (int i = 0; i < LOOP; ++i) {
32             kp[i] = kpGenerator.generateKeyPair();
33         }
34         long end = System.currentTimeMillis();
35         System.out.println(ALG + " on " + CURVE + " keygen: " + 1000.0 * 1000 / (end - start) + " ops/sec");
36
37         /* Message */
38         byte[] msg = Strings.toByteArray("hello, world!");
39
40         Signature signature = Signature.getInstance(ALG, "BC");
41
42         /* ECDSA sign */
43         byte[][] sig = new byte[LOOP][];
44         start = System.currentTimeMillis();
45         for (int i = 0; i < LOOP; ++i) {
46             signature.initSign(kp[i].getPrivate());
47             signature.update(msg);
48             sig[i] = signature.sign();
49         }
50         end = System.currentTimeMillis();
51         System.out.println(ALG + " on " + CURVE + " sign: " + 1000.0 * 1000 / (end - start) + " ops/sec");
52
53         /* ECDSA verify */
54         boolean[] verify = new boolean[LOOP];
55         start = System.currentTimeMillis();
56         for (int i = 0; i < LOOP; ++i) {
57             signature.initVerify(kp[i].getPublic());
58             signature.update(msg);
59             verify[i] = signature.verify(sig[i]);
60         }
61         end = System.currentTimeMillis();
62         System.out.println(ALG + " on " + CURVE + " verify: " + 1000.0 * 1000 / (end - start) + " ops/sec");
63
64         /* Ensure results */
65         String result = "OK";
66         for (int i = 0; i < LOOP; ++i) {
67             if (!verify[i]) {
68                 System.out.println(ALG + " on " + CURVE + " verified failed at " + i);
69                 result = "NG";
70             }
71         }
72         System.out.println(ALG + " on " + CURVE + " keygen/sign/verified : " + result);
73     }
74 }

```

図 21 Bouncy Castle EcDsaP384 計測用プログラム EcDsaP384.java

```

1 package bcBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Security;
7 import java.security.Signature;
8 import java.security.spec.ECGenParameterSpec;
9
10 import org.bouncycastle.util.Strings;
11
12 public class EcDsaP521
13 {
14     static final int LOOP = 1000;
15     static final String CURVE = "P-521";
16     static final String ALG = "SHA512withECDSA";
17
18     public static void main(String[] args)
19     throws GeneralSecurityException
20     {
21         Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
22
23         /* ECDSA Key Pair generator */
24         KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance("EC", "BC");
25         kpGenerator.initialize(new ECGenParameterSpec(CURVE));
26
27         KeyPair[] kp = new KeyPair[LOOP];
28
29         /* ECDSA Key Pair generate */
30         long start = System.currentTimeMillis();
31         for (int i = 0; i < LOOP; ++i) {
32             kp[i] = kpGenerator.generateKeyPair();
33         }
34         long end = System.currentTimeMillis();
35         System.out.println(ALG + " on " + CURVE + " keygen: " + 1000.0 * 1000 / (end - start) + " ops/sec");
36
37         /* Message */
38         byte[] msg = Strings.toByteArray("hello, world!");
39
40         Signature signature = Signature.getInstance(ALG, "BC");
41
42         /* ECDSA sign */
43         byte[][] sig = new byte[LOOP][];
44         start = System.currentTimeMillis();
45         for (int i = 0; i < LOOP; ++i) {
46             signature.initSign(kp[i].getPrivate());
47             signature.update(msg);
48             sig[i] = signature.sign();
49         }
50         end = System.currentTimeMillis();
51         System.out.println(ALG + " on " + CURVE + " sign: " + 1000.0 * 1000 / (end - start) + " ops/sec");
52
53         /* ECDSA verify */
54         boolean[] verify = new boolean[LOOP];
55         start = System.currentTimeMillis();
56         for (int i = 0; i < LOOP; ++i) {
57             signature.initVerify(kp[i].getPublic());
58             signature.update(msg);
59             verify[i] = signature.verify(sig[i]);
60         }
61         end = System.currentTimeMillis();
62         System.out.println(ALG + " on " + CURVE + " verify: " + 1000.0 * 1000 / (end - start) + " ops/sec");
63
64         /* Ensure results */
65         String result = "OK";
66         for (int i = 0; i < LOOP; ++i) {
67             if (!verify[i]) {
68                 System.out.println(ALG + " on " + CURVE + " verified failed at " + i);
69                 result = "NG";
70             }
71         }
72         System.out.println(ALG + " on " + CURVE + " keygen/sign/verified : " + result);
73     }
74 }

```

図 22 Bouncy Castle EcDsaP521 計測用プログラム EcDsaP521.java

```

1 package bcBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Security;
7 import java.security.Signature;
8
9 import org.bouncycastle.util.Strings;
10
11 public class EdDsaEd25519 {
12 {
13     static final int LOOP = 1000;
14     static final String ALG = "Ed25519";
15
16 public static void main(String[] args)
17     throws GeneralSecurityException {
18     {
19         Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
20
21         /* EdDSA Key Pair generator */
22         KeyPairGenerator keyPair = KeyPairGenerator.getInstance(ALG, "BC");
23
24         KeyPair[] kp = new KeyPair[LOOP];
25
26         /* EdDSA Key Pair generate */
27         long start = System.currentTimeMillis();
28         for (int i = 0; i < LOOP; ++i) {
29             kp[i] = keyPair.generateKeyPair();
30         }
31         long end = System.currentTimeMillis();
32         System.out.println(ALG + " keygen: " + 1000.0 * 1000 / (end - start) + " ops/sec");
33
34         /* Message */
35         byte[] msg = Strings.toByteArray("hello, world!");
36
37         Signature signature = Signature.getInstance("EdDSA", "BC");
38
39         /* EdDSA sign */
40         byte[][] sig = new byte[LOOP][];
41         start = System.currentTimeMillis();
42         for (int i = 0; i < LOOP; ++i) {
43             signature.initSign(kp[i].getPrivate());
44             signature.update(msg);
45             sig[i] = signature.sign();
46         }
47         end = System.currentTimeMillis();
48         System.out.println(ALG + " sign: " + 1000.0 * 1000 / (end - start) + " ops/sec");
49
50         /* EdDSA verify */
51         boolean[] verify = new boolean[LOOP];
52         start = System.currentTimeMillis();
53         for (int i = 0; i < LOOP; ++i) {
54             signature.initVerify(kp[i].getPublic());
55             signature.update(msg);
56             verify[i] = signature.verify(sig[i]);
57         }
58         end = System.currentTimeMillis();
59         System.out.println(ALG + " verify: " + 1000.0 * 1000 / (end - start) + " ops/sec");
60
61         /* Ensure results */
62         String result = "OK";
63         for (int i = 0; i < LOOP; ++i) {
64             if (!verify[i]) {
65                 System.out.println(ALG + " verified failed at " + i);
66                 result = "NG";
67             }
68         }
69         System.out.println(ALG + " keygen/sign/verified : " + result);
70     }
71 }
72

```

図 23 Bouncy Castle Ed25519 計測用プログラム EdDsaEd25519.java

```

1 package bcBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Security;
7 import java.security.Signature;
8
9 import org.bouncycastle.util.Strings;
10
11 public class EdDsaEd448
12 {
13     static final int LOOP = 1000;
14     static final String ALG = "Ed448";
15
16     public static void main(String[] args)
17     throws GeneralSecurityException
18     {
19         Security.addProvider(new org.bouncycastle.jce.provider.BouncyCastleProvider());
20
21         /* EdDSA Key Pair generator */
22         KeyPairGenerator keyPair = KeyPairGenerator.getInstance(ALG, "BC");
23
24         KeyPair[] kp = new KeyPair[LOOP];
25
26         /* EdDSA Key Pair generate */
27         long start = System.currentTimeMillis();
28         for (int i = 0; i < LOOP; ++i) {
29             kp[i] = keyPair.generateKeyPair();
30         }
31         long end = System.currentTimeMillis();
32         System.out.println(ALG + " keygen: " + 1000.0 * 1000 / (end - start) + " ops/sec");
33
34         /* Message */
35         byte[] msg = Strings.toByteArray("hello, world!");
36
37         Signature signature = Signature.getInstance("EdDSA", "BC");
38
39         /* EdDSA sign */
40         byte[][] sig = new byte[LOOP][];
41         start = System.currentTimeMillis();
42         for (int i = 0; i < LOOP; ++i) {
43             signature.initSign(kp[i].getPrivate());
44             signature.update(msg);
45             sig[i] = signature.sign();
46         }
47         end = System.currentTimeMillis();
48         System.out.println(ALG + " sign: " + 1000.0 * 1000 / (end - start) + " ops/sec");
49
50         /* EdDSA verify */
51         boolean[] verify = new boolean[LOOP];
52         start = System.currentTimeMillis();
53         for (int i = 0; i < LOOP; ++i) {
54             signature.initVerify(kp[i].getPublic());
55             signature.update(msg);
56             verify[i] = signature.verify(sig[i]);
57         }
58         end = System.currentTimeMillis();
59         System.out.println(ALG + " verify: " + 1000.0 * 1000 / (end - start) + " ops/sec");
60
61         /* Ensure results */
62         String result = "OK";
63         for (int i = 0; i < LOOP; ++i) {
64             if (!verify[i]) {
65                 System.out.println(ALG + " verified failed at " + i);
66                 result = "NG";
67             }
68         }
69         System.out.println(ALG + " keygen/sign/verified : " + result);
70     }
71 }
72

```

図 24 Bouncy Castle Ed448 計測用プログラム EdDsaEd448.java

bcprov-jdk15on-169.jar, EcDsaP256.java, EcDsaP384.java, EcDsaP521.java, EdDsaEd25519.java, EdDsaEd448.java を同一のディレクトリに配置し、そのディレクトリ内で以下の内容のコマンドを実行して計測用プログラムをビルドした。

```
javac -classpath bcprov-jdk15on-169.jar -d . EcDsaP256.java
javac -classpath bcprov-jdk15on-169.jar -d . EcDsaP384.java
javac -classpath bcprov-jdk15on-169.jar -d . EcDsaP521.java
javac -classpath bcprov-jdk15on-169.jar -d . EdDsaEd25519.java
javac -classpath bcprov-jdk15on-169.jar -d . EdDsaEd448.java
```

【計測用プログラムの実行方法】

〈Linux〉 〈Mac〉 以下のコマンドを実行した.

```
java -classpath $PWD/bcprov-jdk15on-169.jar:$PWD bcBench.EcDsaP256
java -classpath $PWD/bcprov-jdk15on-169.jar:$PWD bcBench.EcDsaP384
java -classpath $PWD/bcprov-jdk15on-169.jar:$PWD bcBench.EcDsaP521
java -classpath $PWD/bcprov-jdk15on-169.jar:$PWD bcBench.EdDsaEd25519
java -classpath $PWD/bcprov-jdk15on-169.jar:$PWD bcBench.EdDsaEd448
```

〈Win〉 以下のコマンドを実行した.

```
java -classpath %CD%/bcprov-jdk15on-169.jar;%CD% bcBench.EcDsaP256
java -classpath %CD%/bcprov-jdk15on-169.jar;%CD% bcBench.EcDsaP384
java -classpath %CD%/bcprov-jdk15on-169.jar;%CD% bcBench.EcDsaP521
java -classpath %CD%/bcprov-jdk15on-169.jar;%CD% bcBench.EdDsaEd25519
java -classpath %CD%/bcprov-jdk15on-169.jar;%CD% bcBench.EdDsaEd448
```

② OpenJDK

【OSS の入手】

〈Win〉 〈Linux〉 〈Mac〉 Java 実行環境の機能を計測するため、別途の入手は不要である.

【計測用プログラムの準備】

計測用プログラムは、ECDSA P-256, ECDSA P-384, ECDSA P-521, Ed25519, Ed448 について

- KeyPairGenerator クラスの generateKeyPair メソッドでの鍵生成
- Signature クラスの initSign, update, sign メソッドを一連の処理としたブロックでの署名生成
- Signature クラスの initVerify, update, verify メソッドを一連の処理としたブロックでの署名検証

を、それぞれ 10^3 回実行し、それぞれの 10^3 回分の処理時間（ミリ秒） t を測定する.

t を 10^3 で割ることでミリ秒から秒に変換し、得られた秒数で 10^3 [回]を割ることで、スループット（1秒あたりの実行回数）を求めた。計算式は

$$\frac{10^3}{(t/10^3)} = \frac{10^6}{t} [\text{回/秒}]$$

となる。

Libgcrypt 等他のライブラリでサポートしていないため、Ed25519ph, Ed25519ctx, Ed448ph は計測していない。

計測用プログラム EcDsaP256.java（図 25）, EcDsaP384.java（図 26）, EcDsaP521.java（図 27）, EdDsaEd25519.java（図 28）および EdDsaEd448.java（図 29）を作成した。

```

1 package openJdkBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Signature;
7 import java.security.spec.ECGenParameterSpec;
8
9 public class EcDsaP256
10 {
11     static final int LOOP = 1000;
12     static final String CURVE = "secp256r1";
13     static final String ALG = "SHA256withECDSA";
14
15     public static void main(String[] args)
16     throws GeneralSecurityException
17     {
18         /* ECDSA Key Pair generator */
19         KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance("EC");
20         kpGenerator.initialize(new ECGenParameterSpec(CURVE));
21
22         KeyPair[] kp = new KeyPair[LOOP];
23
24         /* ECDSA Key Pair generate */
25         long start = System.currentTimeMillis();
26         for (int i = 0; i < LOOP; ++i) {
27             kp[i] = kpGenerator.generateKeyPair();
28         }
29         long end = System.currentTimeMillis();
30         System.out.println(ALG + " on " + CURVE + " keygen: " + 1000.0 * 1000 / (end - start) + " ops/sec");
31
32         /* Message */
33         byte[] msg = "hello, world!".getBytes();
34
35         Signature signature = Signature.getInstance(ALG);
36
37         /* ECDSA sign */
38         byte[][] sig = new byte[LOOP][];
39         start = System.currentTimeMillis();
40         for (int i = 0; i < LOOP; ++i) {
41             signature.initSign(kp[i].getPrivate());
42             signature.update(msg);
43             sig[i] = signature.sign();
44         }
45         end = System.currentTimeMillis();
46         System.out.println(ALG + " on " + CURVE + " sign: " + 1000.0 * 1000 / (end - start) + " ops/sec");
47
48         /* ECDSA verify */
49         boolean[] verify = new boolean[LOOP];
50         start = System.currentTimeMillis();
51         for (int i = 0; i < LOOP; ++i) {
52             signature.initVerify(kp[i].getPublic());
53             signature.update(msg);
54             verify[i] = signature.verify(sig[i]);
55         }
56         end = System.currentTimeMillis();
57         System.out.println(ALG + " on " + CURVE + " verify: " + 1000.0 * 1000 / (end - start) + " ops/sec");
58
59         /* Ensure results */
60         String result = "OK";
61         for (int i = 0; i < LOOP; ++i) {
62             if (!verify[i]) {
63                 System.out.println(ALG + " on " + CURVE + " verified failed at " + i);
64                 result = "NG";
65             }
66         }
67         System.out.println(ALG + " on " + CURVE + " keygen/sign/verified : " + result);
68     }
69 }
70

```

図 25 OpenJDK EcDsaP256 計測用プログラム EcDsaP256. java

```

1 package openJdkBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Signature;
7 import java.security.spec.ECGenParameterSpec;
8
9 public class EcDsaP384
10 {
11     static final int LOOP = 1000;
12     static final String CURVE = "secp384r1";
13     static final String ALG = "SHA512withECDSA";
14
15     public static void main(String[] args)
16     throws GeneralSecurityException
17     {
18         /* ECDSA Key Pair generator */
19         KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance("EC");
20         kpGenerator.initialize(new ECGenParameterSpec(CURVE));
21
22         KeyPair[] kp = new KeyPair[LOOP];
23
24         /* ECDSA Key Pair generate */
25         long start = System.currentTimeMillis();
26         for (int i = 0; i < LOOP; ++i) {
27             kp[i] = kpGenerator.generateKeyPair();
28         }
29         long end = System.currentTimeMillis();
30         System.out.println(ALG + " on " + CURVE + " keygen: " + 1000.0 * 1000 / (end - start) + " ops/sec");
31
32         /* Message */
33         byte[] msg = "hello, world!".getBytes();
34
35         Signature signature = Signature.getInstance(ALG);
36
37         /* ECDSA sign */
38         byte[][] sig = new byte[LOOP][];
39         start = System.currentTimeMillis();
40         for (int i = 0; i < LOOP; ++i) {
41             signature.initSign(kp[i].getPrivate());
42             signature.update(msg);
43             sig[i] = signature.sign();
44         }
45         end = System.currentTimeMillis();
46         System.out.println(ALG + " on " + CURVE + " sign: " + 1000.0 * 1000 / (end - start) + " ops/sec");
47
48         /* ECDSA verify */
49         boolean[] verify = new boolean[LOOP];
50         start = System.currentTimeMillis();
51         for (int i = 0; i < LOOP; ++i) {
52             signature.initVerify(kp[i].getPublic());
53             signature.update(msg);
54             verify[i] = signature.verify(sig[i]);
55         }
56         end = System.currentTimeMillis();
57         System.out.println(ALG + " on " + CURVE + " verify: " + 1000.0 * 1000 / (end - start) + " ops/sec");
58
59         /* Ensure results */
60         String result = "OK";
61         for (int i = 0; i < LOOP; ++i) {
62             if (!verify[i]) {
63                 System.out.println(ALG + " on " + CURVE + " verified failed at " + i);
64                 result = "NG";
65             }
66         }
67         System.out.println(ALG + " on " + CURVE + " keygen/sign/verified : " + result);
68     }
69 }
70

```

図 26 OpenJDK EcDsaP384 計測用プログラム EcDsaP384. java

```

1 package openJdkBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Signature;
7 import java.security.spec.ECGenParameterSpec;
8
9 public class EcDsaP521
10 {
11     static final int LOOP = 1000;
12     static final String CURVE = "secp521r1";
13     static final String ALG = "SHA512withECDSA";
14
15     public static void main(String[] args)
16     throws GeneralSecurityException
17     {
18         /* ECDSA Key Pair generator */
19         KeyPairGenerator kpGenerator = KeyPairGenerator.getInstance("EC");
20         kpGenerator.initialize(new ECGenParameterSpec(CURVE));
21
22         KeyPair[] kp = new KeyPair[LOOP];
23
24         /* ECDSA Key Pair generate */
25         long start = System.currentTimeMillis();
26         for (int i = 0; i < LOOP; ++i) {
27             kp[i] = kpGenerator.generateKeyPair();
28         }
29         long end = System.currentTimeMillis();
30         System.out.println(ALG + " on " + CURVE + " keygen: " + 1000.0 * 1000 / (end - start) + " ops/sec");
31
32         /* Message */
33         byte[] msg = "hello, world!".getBytes();
34
35         Signature signature = Signature.getInstance(ALG);
36
37         /* ECDSA sign */
38         byte[][] sig = new byte[LOOP][];
39         start = System.currentTimeMillis();
40         for (int i = 0; i < LOOP; ++i) {
41             signature.initSign(kp[i].getPrivate());
42             signature.update(msg);
43             sig[i] = signature.sign();
44         }
45         end = System.currentTimeMillis();
46         System.out.println(ALG + " on " + CURVE + " sign: " + 1000.0 * 1000 / (end - start) + " ops/sec");
47
48         /* ECDSA verify */
49         boolean[] verify = new boolean[LOOP];
50         start = System.currentTimeMillis();
51         for (int i = 0; i < LOOP; ++i) {
52             signature.initVerify(kp[i].getPublic());
53             signature.update(msg);
54             verify[i] = signature.verify(sig[i]);
55         }
56         end = System.currentTimeMillis();
57         System.out.println(ALG + " on " + CURVE + " verify: " + 1000.0 * 1000 / (end - start) + " ops/sec");
58
59         /* Ensure results */
60         String result = "OK";
61         for (int i = 0; i < LOOP; ++i) {
62             if (!verify[i]) {
63                 System.out.println(ALG + " on " + CURVE + " verified failed at " + i);
64                 result = "NG";
65             }
66         }
67         System.out.println(ALG + " on " + CURVE + " keygen/sign/verified : " + result);
68     }
69 }
70

```

図 27 OpenJDK EcDsaP521 計測用プログラム EcDsaP521. java

```

1 package openJdkBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Signature;
7
8 public class EdDsaEd25519
9 {
10     static final int LOOP = 1000;
11     static final String ALG = "Ed25519";
12
13     public static void main(String[] args)
14     throws GeneralSecurityException
15     {
16         /* EdDSA Key Pair generator */
17         KeyPairGenerator keyPair = KeyPairGenerator.getInstance(ALG);
18
19         KeyPair[] kp = new KeyPair[LOOP];
20
21         /* EdDSA Key Pair generate */
22         long start = System.currentTimeMillis();
23         for (int i = 0; i < LOOP; ++i) {
24             kp[i] = keyPair.generateKeyPair();
25         }
26         long end = System.currentTimeMillis();
27         System.out.println(ALG + " keygen: " + 1000.0 * 1000 / (end - start) + " msec");
28
29         /* Message */
30         byte[] msg = "hello, world!".getBytes();
31
32         Signature signature = Signature.getInstance(ALG);
33
34         /* EdDSA sign */
35         byte[][] sig = new byte[LOOP][];
36         start = System.currentTimeMillis();
37         for (int i = 0; i < LOOP; ++i) {
38             signature.initSign(kp[i].getPrivate());
39             signature.update(msg);
40             sig[i] = signature.sign();
41         }
42         end = System.currentTimeMillis();
43         System.out.println(ALG + " sign: " + 1000.0 * 1000 / (end - start) + " msec");
44
45         /* EdDSA verify */
46         boolean[] verify = new boolean[LOOP];
47         start = System.currentTimeMillis();
48         for (int i = 0; i < LOOP; ++i) {
49             signature.initVerify(kp[i].getPublic());
50             signature.update(msg);
51             verify[i] = signature.verify(sig[i]);
52         }
53         end = System.currentTimeMillis();
54         System.out.println(ALG + " verify: " + 1000.0 * 1000 / (end - start) + " msec");
55
56         /* Ensure results */
57         String result = "OK";
58         for (int i = 0; i < LOOP; ++i) {
59             if (!verify[i]) {
60                 System.out.println(ALG + " verified failed at " + i);
61                 result = "NG";
62             }
63         }
64         System.out.println(ALG + " keygen/sign/verified : " + result);
65     }
66 }

```

図 28 OpenJDK Ed25519 計測用プログラム EdDsaEd25519. java

```

1 package openJdkBench;
2
3 import java.security.GeneralSecurityException;
4 import java.security.KeyPair;
5 import java.security.KeyPairGenerator;
6 import java.security.Signature;
7
8 public class EdDsaEd448
9 {
10     static final int LOOP = 1000;
11     static final String ALG = "Ed448";
12
13     public static void main(String[] args)
14     throws GeneralSecurityException
15     {
16         /* EdDSA Key Pair generator */
17         KeyPairGenerator keyPair = KeyPairGenerator.getInstance(ALG);
18
19         KeyPair[] kp = new KeyPair[LOOP];
20
21         /* EdDSA Key Pair generate */
22         long start = System.currentTimeMillis();
23         for (int i = 0; i < LOOP; ++i) {
24             kp[i] = keyPair.generateKeyPair();
25         }
26         long end = System.currentTimeMillis();
27         System.out.println(ALG + " keygen: " + 1000.0 * 1000 / (end - start) + " ops/sec");
28
29         /* Message */
30         byte[] msg = "hello, world!".getBytes();
31
32         Signature signature = Signature.getInstance(ALG);
33
34         /* EdDSA sign */
35         byte[][] sig = new byte[LOOP][];
36         start = System.currentTimeMillis();
37         for (int i = 0; i < LOOP; ++i) {
38             signature.initSign(kp[i].getPrivate());
39             signature.update(msg);
40             sig[i] = signature.sign();
41         }
42         end = System.currentTimeMillis();
43         System.out.println(ALG + " sign: " + 1000.0 * 1000 / (end - start) + " ops/sec");
44
45         /* EdDSA verify */
46         boolean[] verify = new boolean[LOOP];
47         start = System.currentTimeMillis();
48         for (int i = 0; i < LOOP; ++i) {
49             signature.initVerify(kp[i].getPublic());
50             signature.update(msg);
51             verify[i] = signature.verify(sig[i]);
52         }
53         end = System.currentTimeMillis();
54         System.out.println(ALG + " verify: " + 1000.0 * 1000 / (end - start) + " ops/sec");
55
56         /* Ensure results */
57         String result = "OK";
58         for (int i = 0; i < LOOP; ++i) {
59             if (!verify[i]) {
60                 System.out.println(ALG + " verified failed at " + i);
61                 result = "NG";
62             }
63         }
64         System.out.println(ALG + " keygen/sign/verified : " + result);
65     }
66 }
67

```

図 29 OpenJDK Ed448 計測用プログラム EdDsaEd448. java

〈Linux〉 〈Mac〉 EcDsaP256. java, EcDsaP384. java, EcDsaP521. java, EdDsaEd25519. java, EdDsaEd448. java を同一のディレクトリに配置し、そのディレクトリ内で以下のコマンドを実行して測定用プログラムをビルドした。

```
javac -d . EcDsaP256. java
javac -d . EcDsaP384. java
javac -d . EcDsaP521. java
javac -d . EdDsaEd25519. java
javac -d . EdDsaEd448. java
```

〈Win〉 EcDsaP256. java, EcDsaP384. java, EcDsaP521. java, EdDsaEd25519. java, EdDsaEd448. java を同一のディレクトリに配置し、そのディレクトリ内で以下のコマンドを実行して測定用プログラムをビルドした。

```
javac -d . EcDsaP256. java
javac -d . EcDsaP384. java
javac -d . EcDsaP521. java
javac -d . EdDsaEd25519. java
javac -d . EdDsaEd448. java
```

【計測用プログラムの実行方法】

〈Linux〉 〈Mac〉 以下のコマンドを実行した。

```
java openJdkBench. EcDsaP256
java openJdkBench. EcDsaP384
java openJdkBench. EcDsaP521
java openJdkBench. EdDsaEd25519
java openJdkBench. EdDsaEd448
```

〈Win〉 以下のコマンドを実行した。

```
java openJdkBench. EcDsaP256
java openJdkBench. EcDsaP384
java openJdkBench. EcDsaP521
java openJdkBench. EdDsaEd25519
java openJdkBench. EdDsaEd448
```

③ Libgcrypt

【OSS の入手】

〈Linux〉 〈Mac〉 libgpg-error 1.43 と Libgcrypt 1.9.4 を以下からダウンロードした。

<https://gnupg.org/ftp/gcrypt/libgpg-error/libgpg-error-1.43.tar.bz2>

<https://gnupg.org/ftp/gcrypt/libgcrypt/libgcrypt-1.9.4.tar.bz2>

以下の内容のコマンドを実行してライブラリをビルドし、benchlib にインストールした。

```
mkdir benchlib
tar -xjf libgpg-error-1.43.tar.bz2
cd libgpg-error-1.43
./configure --prefix=$PWD/../benchlib; make; make install
cd ..
tar -xjf libgcrypt-1.9.4.tar.bz2
cd libgcrypt-1.9.4
./configure --prefix=$PWD/../benchlib; make; make install
cd ..
```

〈Win〉 Libgcrypt は Visual Studio でビルドできないため、以下のコマンドで MSYS2 MinGW 64-bit のパッケージおよび依存するライブラリをインストールした。

```
pacman -S mingw-w64-x86_64-libgcrypt-1.9.4-1
```

依存するライブラリは Linux, Mac と同じバージョンであった。

【計測用プログラムの準備】

計測用プログラムは、ECDSA P-256, ECDSA P-384, ECDSA P-521, Ed25519, Ed448 について

- gcry_pk_genkey 関数, gcry_sexp_find_token 関数での公開鍵取り出し, gcry_sexp_find_token 関数での秘密鍵取り出しを一連の処理としたブロックでの鍵生成
- gcry_pk_sign 関数での署名生成
- gcry_pk_verify 関数での署名検証

を、それぞれ 10^4 回実行し、それぞれの 10^4 回分の処理時間（ナノ秒） t を測定する。

t を 10^9 で割ることでナノ秒から秒に変換し、得られた秒数で 10^4 [回]を割ることで、スループット（1秒あたりの実行回数）を求めた。計算式は

$$\frac{10^4}{(t/10^9)} = \frac{10^{13}}{t} [\text{回/秒}]$$

となる。

Libgcrypt では Ed25519ph, Ed25519ctx, Ed448ph をサポートしていないため計測していない。

libgcrypt-1.9.4/tests/benchmark.c を参考に計測用プログラム libgcrypt_benchmark.c (図 30) を作成した。

```

1 #include <time.h>
2 #include <gcrypt.h>
3
4 #define LOOP 10000
5 #define SEC(spec) ((double)spec.tv_sec + (double)spec.tv_nsec / 1000000000.0)
6 #define SEC_DIFF(st, ed) (SEC(ed) - SEC(st))
7
8 int main(int argc, const char *argv[]) {
9     /* Initialize libgcrypt library */
10    if (gcry_check_version("1.9.4") == NULL) {
11        return -1;
12    }
13    gcry_control(GCRYCTL_DISABLE_SECMEM, 0);
14    const struct {
15        char *alg;
16        int p;
17        char *key_s;
18        char *data_s;
19        int is_ecdsa;
20    } list[] = {
21        {"ECDSA P-256", 256, "(genkey(ecdsa(curve ¥"nistp256¥")))",
22         "(data (flags raw) (value %m))", 1},
23        {"ECDSA P-384", 384, "(genkey(ecdsa(curve ¥"nistp384¥")))",
24         "(data (flags raw) (value %m))", 1},
25        {"ECDSA P-521", 521, "(genkey(ecdsa(curve ¥"nistp521¥")))",
26         "(data (flags raw) (value %m))", 1},
27        {"Ed25519", 256, "(genkey(ecdsa(curve ¥"Ed25519¥") (flags eddsa)))",
28         "(data (flags eddsa) (hash-algo sha512) (value %m))", 0},
29        {"Ed448", 256, "(genkey(ecdsa(curve ¥"Ed448¥") (flags eddsa)))",
30         "(data (flags eddsa) (hash-algo shake256) (value %m))", 0}
31    };
32
33    struct timespec st, ed;
34
35    printf("Algorithm      KeyGen      Sign      Verify¥n");
36    printf("-----¥n");
37
38    for (int testno=0; testno < sizeof(list)/sizeof((list)[0]); testno++) {
39        printf("%-11s ", list[testno].alg);
40
41        /* Prepare key spec */
42        gcry_sexp_t key_spec;
43        gcry_sexp_build(&key_spec, NULL, list[testno].key_s);
44
45        /* KeyGen */
46        gcry_sexp_t key_pair[LOOP];
47        gcry_sexp_t pub_key[LOOP];
48        gcry_sexp_t pri_key[LOOP];
49
50        clock_gettime(CLOCK_MONOTONIC, &st);
51
52        for (int i = 0; i < LOOP; ++i) {
53            gcry_pk_genkey(&key_pair[i], key_spec);
54            pub_key[i] = gcry_sexp_find_token(key_pair[i], "public-key", 0);
55            pri_key[i] = gcry_sexp_find_token(key_pair[i], "private-key", 0);
56        }

```

```

57 ~ clock_gettime(CLOCK_MONOTONIC, &ed);
58 ~ printf(" %11.06f", (double)LOOP / SEC_DIFF(st, ed));
59 ~
60 ~
61 ~ /* Message */
62 ~ gcry_mpi_t x;
63 ~ if (list[testno].is_ecdsa) {
64 ~ ~ unsigned char digest[32];
65 ~ ~ gcry_md_hash_buffer(GCRY_MD_SHA256, digest, "Hello World!", 12);
66 ~ ~ gcry_mpi_scan(&x, GCRYMPI_FMT_USG, digest, 32, NULL);
67 ~ ~ } else {
68 ~ ~ gcry_mpi_scan(&x, GCRYMPI_FMT_USG, "Hello World!", 12, NULL);
69 ~ ~ }
70 ~ gcry_sexp_t data;
71 ~ gcry_sexp_build(&data, NULL, list[testno].data_s, x);
72 ~ gcry_mpi_release(x);
73 ~
74 ~ /* Sign */
75 ~ gcry_sexp_t sig[LOOP];
76 ~
77 ~ clock_gettime(CLOCK_MONOTONIC, &st);
78 ~
79 ~ for (int i = 0; i < LOOP; ++i) {
80 ~ ~ gcry_pk_sign(&sig[i], data, pri_key[i]);
81 ~ }
82 ~
83 ~ clock_gettime(CLOCK_MONOTONIC, &ed);
84 ~ printf(" %11.06f", (double)LOOP / SEC_DIFF(st, ed));
85 ~
86 ~ /* Verify */
87 ~ clock_gettime(CLOCK_MONOTONIC, &st);
88 ~
89 ~ for (int i = 0; i < LOOP; ++i) {
90 ~ ~ gcry_error_t result = gcry_pk_verify(sig[i], data, pub_key[i]);
91 ~ ~ if (result != 0) {
92 ~ ~ ~ printf("verify failed\n");
93 ~ ~ ~ break;
94 ~ ~ }
95 ~ }
96 ~
97 ~ clock_gettime(CLOCK_MONOTONIC, &ed);
98 ~ printf(" %11.06f\n", (double)LOOP / SEC_DIFF(st, ed));
99 ~
100 ~ gcry_sexp_release(key_spec);
101 ~ for (int i = 0; i < LOOP; ++i) {
102 ~ ~ gcry_sexp_release(key_pair[i]);
103 ~ ~ gcry_sexp_release(pri_key[i]);
104 ~ ~ gcry_sexp_release(pub_key[i]);
105 ~ ~ gcry_sexp_release(sig[i]);
106 ~ }
107 ~ gcry_sexp_release(data);
108 ~ }
109 ~

```

図 30 libgcrypt_benchmark.c

〈Linux〉 〈Mac〉 libgcrypt_benchmark.c を libgcrypt-1.9.4.tar.bz2 を展開したディレクトリ (libgcrypt-1.9.4ディレクトリを含むディレクトリ) に配置し、そのディレクトリ内で以下のコマンドを実行して計測用プログラムをビルドした。

```

gcc -o libgcrypt_benchmark libgcrypt_benchmark.c `./benchlib/bin/lib-
gcrypt-config --cflags --libs`
export LD_LIBRARY_PATH=$PWD/benchlib/lib

```

〈Win〉 libgcrypt_benchmark.c を以下のコマンドを実行して計測用プログラムをビルドした。

```

gcc -o libgcrypt_benchmark libgcrypt_benchmark.c `libgcrypt-config --cflags
--libs`

```

【計測用プログラムの実行方法】

以下のコマンドを実行した。

```
./libgcrypt_benchmark
```

④ Nettle

【OSS の入手】

〈Linux〉 〈Mac〉 GMP 6.2.1 と Nettle 3.7.3 を以下からダウンロードした。

<https://gmplib.org/download/gmp/gmp-6.2.1.tar.lz>

<https://ftp.gnu.org/gnu/nettle/nettle-3.7.3.tar.gz>

以下のコマンドを実行してライブラリをビルドし、benchlib にインストールした。

```
mkdir benchlib
tar --lzip -xf gmp-6.2.1.tar.lz
cd gmp-6.2.1
./configure --prefix=$PWD/../benchlib; make; make check && make install
cd ..
export LD_LIBRARY_PATH=$PWD/benchlib/lib
tar -xzf nettle-3.7.3.tar.gz
cd nettle-3.7.3
./configure --prefix=$PWD/../benchlib; make; make install
cd ..
```

〈Win〉 Nettle は Visual Studio でビルドできないため、以下のコマンドで MSYS2 MinGW 64-bit のパッケージおよび依存するライブラリをインストールした。

```
pacman -S mingw-w64-x86_64-nettle-3.7.3-1
```

依存するライブラリは Linux, Mac と同じバージョンであった。

【計測用プログラムの準備】

計測用プログラムは、ECDSA P-256, ECDSA P-384, ECDSA P-521, Ed25519, Ed448 について

- ECDSA P-256 の場合
 - `ecdsa_generate_keypair` 関数で鍵生成
 - `sha256_init`, `sha256_update`, `sha256_digest`, `ecdsa_sign` で署名生成
 - `sha256_init`, `sha256_update`, `sha256_digest`, `ecdsa_verify` で署名検証
- ECDSA P-384, ECDSA P-521 の場合

- `ecdsa_generate_keypair` 関数で鍵生成
- `sha512_init`, `sha512_update`, `sha512_digest`, `ecdsa_sign` で署名生成
- `sha512_init`, `sha512_update`, `sha512_digest`, `ecdsa_verify` で署名検証
- Ed25519 の場合
 - `yarrow256_random` 関数と `ed25519_sha512_public_key` 関数で鍵生成
 - `ed25519_sha512_sign` 関数で署名生成
 - `ed25519_sha512_verify` 関数で署名検証
- Ed448 の場合
 - `yarrow256_random` 関数と `ed448_shake256_public_key` 関数で鍵生成
 - `ed448_shake256_sign` 関数で署名生成
 - `ed448_shake256_verify` 関数で署名検証

をそれぞれ 10^4 回実行し、それぞれの 10^4 回分の処理時間（ナノ秒） t を測定する．

t を 10^9 で割ることでナノ秒から秒に変換し、得られた秒数で 10^3 [回]を割ることで、スループット（1秒あたりの実行回数）を求めた．計算式は

$$\frac{10^4}{(t/10^9)} = \frac{10^{13}}{t} [\text{回/秒}]$$

となる．

Nettle では Ed25519ph, Ed25519ctx, Ed448ph をサポートしていないため計測していない．

計測用プログラム `nettle_benchmark.c` (図 31) , `nettle_benchmark_p256.c` (図 32) , `nettle_benchmark_p384.c` (図 33) , `nettle_benchmark_p521.c` (図 34) , `nettle_benchmark_ed25519.c` (図 35) および `nettle_benchmark_ed448.c` (図 36) を作成した．

```

1 #include <nettle/yarrow.h>
2
3 /* to avoid warning of incompatible pointers between
4  * void*(void*,...) and
5  * void*(struct yarrow256_ctx*,...) */
6 void yrandom(void *ctx, size_t length, uint8_t *dst) {
7     yarrow256_random((struct yarrow256_ctx*)ctx, length, dst);
8 }
9
10 void p256(struct yarrow256_ctx *rx);
11 void p384(struct yarrow256_ctx *rx);
12 void p521(struct yarrow256_ctx *rx);
13 void ed25519(struct yarrow256_ctx *rx);
14 void ed448(struct yarrow256_ctx *rx);
15
16 int main(int argc, const char *argv[]) {
17     /* Init random context */
18     struct yarrow256_ctx rx;
19     yarrow256_init(&rx, 0, NULL);
20     uint8_t seed[YARROW256_SEED_FILE_SIZE] = {};
21     yarrow256_seed(&rx, YARROW256_SEED_FILE_SIZE, seed);
22
23     p256(&rx);
24     p384(&rx);
25     p521(&rx);
26     ed25519(&rx);
27     ed448(&rx);
28
29     /* don't clear yarrow256_ctx */
30
31     return 0;
32 }

```

图 31 nettle_benchmark.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #include <nettle/yarrow.h>
6 #include <nettle/ecc-curve.h>
7 #include <nettle/ecdsa.h>
8 #include <nettle/dsa.h>
9 #include <nettle/sha2.h>
10
11 #define MSG "Hello World!"
12 #define MSGLEN 12
13 #define LOOP 10000
14 #define SEC(spec) ((double)spec.tv_sec + (double)spec.tv_nsec / 1000000000.0)
15 #define SEC_DIFF(st, ed) (SEC(ed) - SEC(st))
16
17 void yrandom(void *ctx, size_t length, uint8_t *dst);
18
19 void p256(struct yarrow256_ctx *rx){
20     /* Init public key */
21     struct ecc_point pub;
22     ecc_point_init(&pub, nettle_get_secp_256r1());
23
24     /* Init private key */
25     struct ecc_scalar key;
26     ecc_scalar_init(&key, nettle_get_secp_256r1());
27
28     /* Init signature */
29     struct dsa_signature sig;
30     dsa_signature_init(&sig);
31
32     /* digest for MSG */
33     uint8_t dgst[SHA256_DIGEST_SIZE];
34
35     struct timespec st, ed;
36
37     clock_gettime(CLOCK_MONOTONIC, &st);
38     for (int i = 0; i < LOOP; ++i) {
39         /* * * Generate keypair * * */
40         ecdsa_generate_keypair(&pub, &key, rx, yrandom);
41     }
42     clock_gettime(CLOCK_MONOTONIC, &ed);
43     printf("ECDSA P-256 KeyGen : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
44
45     clock_gettime(CLOCK_MONOTONIC, &st);
46     for (int i = 0; i < LOOP; ++i) {
47         /* * * Generate Signature * * */
48         struct sha256_ctx ctx;
49         sha256_init(&ctx);
50         sha256_update(&ctx, MSGLEN, MSG);
51         sha256_digest(&ctx, SHA256_DIGEST_SIZE, dgst);
52         ecdsa_sign(&key, rx, yrandom, SHA256_DIGEST_SIZE, dgst, &sig);
53     }
54     clock_gettime(CLOCK_MONOTONIC, &ed);
55     printf("ECDSA P-256 Sign : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
56
57     clock_gettime(CLOCK_MONOTONIC, &st);
58     for (int i = 0; i < LOOP; ++i) {
59         /* * * Verify Signature * * */
60         struct sha256_ctx ctx;

```

```

61 ^     sha256_init(&ctx);↵
62 ^     sha256_update(&ctx, MSGLEN, MSG);↵
63 ^     sha256_digest(&ctx, SHA256_DIGEST_SIZE, dgst);↵
64 ^     ecdsa_verify(&pub, SHA256_DIGEST_SIZE, dgst, &sig);↵
65 ^ }↵
66 ^ clock_gettime(CLOCK_MONOTONIC, &ed);↵
67 ^ printf("ECDSA P-256 Verify : %.08f\n", (double)LOOP / SEC_DIFF(st, ed));↵
68 ↵
69 ^ /* Check verify result */↵
70 ^ int result = ecdsa_verify(&pub, SHA256_DIGEST_SIZE, dgst, &sig);↵
71 ^ printf("ECDSA P-256 Result : %s\n", result ? "valid" : "invalid");↵
72 ↵
73 ^ dsa_signature_clear(&sig);↵
74 ^ ecc_scalar_clear(&key);↵
75 ^ ecc_point_clear(&pub);↵
76 ^ }↵

```

图 32 nettle_benchmark_p256.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #include <nettle/yarrow.h>
6 #include <nettle/ecc-curve.h>
7 #include <nettle/ecdsa.h>
8 #include <nettle/dsa.h>
9 #include <nettle/sha2.h>
10
11 #define MSG "Hello World!"
12 #define MSGLEN 12
13 #define LOOP 10000
14 #define SEC(spec) ((double)spec.tv_sec + (double)spec.tv_nsec / 1000000000.0)
15 #define SEC_DIFF(st, ed) (SEC(ed) - SEC(st))
16
17 void yrandom(void *ctx, size_t length, uint8_t *dst);
18
19 void p384(struct yarrow256_ctx *rx){
20     /* Init public key */
21     struct ecc_point pub;
22     ecc_point_init(&pub, nettle_get_secp_384r1());
23
24     /* Init private key */
25     struct ecc_scalar key;
26     ecc_scalar_init(&key, nettle_get_secp_384r1());
27
28     /* Init signature */
29     struct dsa_signature sig;
30     dsa_signature_init(&sig);
31
32     /* digest for MSG */
33     uint8_t dgst[SHA512_DIGEST_SIZE];
34
35     struct timespec st, ed;
36
37     clock_gettime(CLOCK_MONOTONIC, &st);
38     for (int i = 0; i < LOOP; ++i) {
39         /* * * Generate keypair * * */
40         eccdsa_generate_keypair(&pub, &key, rx, yrandom);
41     }
42     clock_gettime(CLOCK_MONOTONIC, &ed);
43     printf("ECDSA P-384 KeyGen : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
44
45     clock_gettime(CLOCK_MONOTONIC, &st);
46     for (int i = 0; i < LOOP; ++i) {
47         /* * * Generate Signature * * */
48         struct sha512_ctx ctx;
49         sha512_init(&ctx);
50         sha512_update(&ctx, MSGLEN, MSG);
51         sha512_digest(&ctx, SHA512_DIGEST_SIZE, dgst);
52         eccdsa_sign(&key, rx, yrandom, SHA512_DIGEST_SIZE, dgst, &sig);
53     }
54     clock_gettime(CLOCK_MONOTONIC, &ed);
55     printf("ECDSA P-384 Sign : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
56
57     clock_gettime(CLOCK_MONOTONIC, &st);
58     for (int i = 0; i < LOOP; ++i) {
59         /* * * Verify Signature * * */
60         struct sha512_ctx ctx;

```

```

60 ^     struct sha512_ctx ctx;↵
61 ^     sha512_init(&ctx);↵
62 ^     sha512_update(&ctx, MSGLEN, MSG);↵
63 ^     sha512_digest(&ctx, SHA512_DIGEST_SIZE, dgst);↵
64 ^     ecdsa_verify(&pub, SHA512_DIGEST_SIZE, dgst, &sig);↵
65 ^ }↵
66 ^ clock_gettime(CLOCK_MONOTONIC, &ed);↵
67 ^ printf("ECDSA P-384 Verify : %.08f\n", (double)LOOP / SEC_DIFF(st, ed));↵
68 ↵
69 ^ /* Check verify result */↵
70 ^ int result = ecdsa_verify(&pub, SHA512_DIGEST_SIZE, dgst, &sig);↵
71 ^ printf("ECDSA P-384 Result : %s\n", result ? "valid" : "invalid");↵
72 ↵
73 ^ dsa_signature_clear(&sig);↵
74 ^ ecc_scalar_clear(&key);↵
75 ^ ecc_point_clear(&pub);↵
76 }↵

```

图 33 nettle_benchmark_p384.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #include <nettle/yarrow.h>
6 #include <nettle/ecc-curve.h>
7 #include <nettle/ecdsa.h>
8 #include <nettle/dsa.h>
9 #include <nettle/sha2.h>
10
11 #define MSG "Hello World!"
12 #define MSGLEN 12
13 #define LOOP 10000
14 #define SEC(spec) ((double)spec.tv_sec + (double)spec.tv_nsec / 1000000000.0)
15 #define SEC_DIFF(st, ed) (SEC(ed) - SEC(st))
16
17 void yrandom(void *ctx, size_t length, uint8_t *dst);
18
19 void p521(struct yarrow256_ctx *rx){
20     /* Init public key */
21     struct ecc_point pub;
22     ecc_point_init(&pub, nettle_get_secp_521r1());
23
24     /* Init private key */
25     struct ecc_scalar key;
26     ecc_scalar_init(&key, nettle_get_secp_521r1());
27
28     /* Init signature */
29     struct dsa_signature sig;
30     dsa_signature_init(&sig);
31
32     /* digest for MSG */
33     uint8_t dgst[SHA512_DIGEST_SIZE];
34
35     struct timespec st, ed;
36
37     clock_gettime(CLOCK_MONOTONIC, &st);
38     for (int i = 0; i < LOOP; ++i) {
39         /* * * Generate keypair * * */
40         eccdsa_generate_keypair(&pub, &key, rx, yrandom);
41     }
42     clock_gettime(CLOCK_MONOTONIC, &ed);
43     printf("ECDSA P-521 KeyGen : %.08f\n", (double)LOOP / SEC_DIFF(st, ed));
44
45     clock_gettime(CLOCK_MONOTONIC, &st);
46     for (int i = 0; i < LOOP; ++i) {
47         /* * * Generate Signature * * */
48         struct sha512_ctx ctx;
49         sha512_init(&ctx);
50         sha512_update(&ctx, MSGLEN, MSG);
51         sha512_digest(&ctx, SHA512_DIGEST_SIZE, dgst);
52         eccdsa_sign(&key, rx, yrandom, SHA512_DIGEST_SIZE, dgst, &sig);
53     }
54     clock_gettime(CLOCK_MONOTONIC, &ed);
55     printf("ECDSA P-521 Sign : %.08f\n", (double)LOOP / SEC_DIFF(st, ed));
56
57     clock_gettime(CLOCK_MONOTONIC, &st);
58     for (int i = 0; i < LOOP; ++i) {
59         /* * * Verify Signature * * */
60         struct sha512_ctx ctx;

```

```

61 ^ sha512_init(&ctx);↵
62 ^ sha512_update(&ctx, MSGLEN, MSG);↵
63 ^ sha512_digest(&ctx, SHA512_DIGEST_SIZE, dgst);↵
64 ^ ecdsa_verify(&pub, SHA512_DIGEST_SIZE, dgst, &sig);↵
65 }↵
66 clock_gettime(CLOCK_MONOTONIC, &ed);↵
67 printf("ECDSA P-521 Verify : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));↵
68 ↵
69 /* Check verify result */↵
70 int result = ecdsa_verify(&pub, SHA512_DIGEST_SIZE, dgst, &sig);↵
71 printf("ECDSA P-521 Result : %s\n", result ? "valid" : "invalid");↵
72 ↵
73 dsa_signature_clear(&sig);↵
74 ecc_scalar_clear(&key);↵
75 ecc_point_clear(&pub);↵
76 }↵

```

图 34 nettle_benchmark_p521.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #include <nettle/yarrow.h>
6 #include <nettle/eddsa.h>
7
8 #define MSG "Hello World!"
9 #define MSGLEN 12
10 #define LOOP 10000
11 #define SEC(spec) ((double)spec.tv_sec + (double)spec.tv_nsec / 1000000000.0)
12 #define SEC_DIFF(st, ed) (SEC(ed) - SEC(st))
13
14 void ed25519(struct yarrow256_ctx *rx){
15     /* public key, private key and signature */
16     uint8_t pub[ED25519_KEY_SIZE];
17     uint8_t priv[ED25519_KEY_SIZE];
18     uint8_t sig[ED25519_SIGNATURE_SIZE];
19
20     struct timespec st, ed;
21
22     clock_gettime(CLOCK_MONOTONIC, &st);
23     for (int i = 0; i < LOOP; ++i) {
24         /* * * Generate keypair * * */
25         yarrow256_random(rx, ED25519_KEY_SIZE, priv);
26         ed25519_sha512_public_key(pub, priv);
27     }
28     clock_gettime(CLOCK_MONOTONIC, &ed);
29     printf("Ed25519 KeyGen : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
30
31     clock_gettime(CLOCK_MONOTONIC, &st);
32     for (int i = 0; i < LOOP; ++i) {
33         /* * * Generate Signature * * */
34         ed25519_sha512_sign(pub, priv, MSGLEN, MSG, sig);
35     }
36     clock_gettime(CLOCK_MONOTONIC, &ed);
37     printf("Ed25519 Sign : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
38
39     clock_gettime(CLOCK_MONOTONIC, &st);
40     for (int i = 0; i < LOOP; ++i) {
41         /* * * Verify Signature * * */
42         ed25519_sha512_verify(pub, MSGLEN, MSG, sig);
43     }
44     clock_gettime(CLOCK_MONOTONIC, &ed);
45     printf("Ed25519 Verify : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
46
47     /* Check verify result */
48     int result = ed25519_sha512_verify(pub, MSGLEN, MSG, sig);
49     printf("Ed25519 Result : %s\n", result ? "valid" : "invalid");
50 }

```

图 35 nettle_benchmark_ed25519.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4
5 #include <nettle/yarrow.h>
6 #include <nettle/eddsa.h>
7
8 #define MSG "Hello World!"
9 #define MSGLEN 12
10 #define LOOP 10000
11 #define SEC(spec) ((double)spec.tv_sec + (double)spec.tv_nsec / 1000000000.0)
12 #define SEC_DIFF(st, ed) (SEC(ed) - SEC(st))
13
14 void ed448(struct yarrow256_ctx *rx){
15     /* public key, private key and signature */
16     uint8_t pub[ED448_KEY_SIZE];
17     uint8_t priv[ED448_KEY_SIZE];
18     uint8_t sig[ED448_SIGNATURE_SIZE];
19
20     struct timespec st, ed;
21
22     clock_gettime(CLOCK_MONOTONIC, &st);
23     for (int i = 0; i < LOOP; ++i) {
24         /* * * Generate keypair * * */
25         yarrow256_random(rx, ED448_KEY_SIZE, priv);
26         ed448_shake256_public_key(pub, priv);
27     }
28     clock_gettime(CLOCK_MONOTONIC, &ed);
29     printf("Ed448 KeyGen : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
30
31     clock_gettime(CLOCK_MONOTONIC, &st);
32     for (int i = 0; i < LOOP; ++i) {
33         /* * * Generate Signature * * */
34         ed448_shake256_sign(pub, priv, MSGLEN, MSG, sig);
35     }
36     clock_gettime(CLOCK_MONOTONIC, &ed);
37     printf("Ed448 Sign : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
38
39     clock_gettime(CLOCK_MONOTONIC, &st);
40     for (int i = 0; i < LOOP; ++i) {
41         /* * * Verify Signature * * */
42         ed448_shake256_verify(pub, MSGLEN, MSG, sig);
43     }
44     clock_gettime(CLOCK_MONOTONIC, &ed);
45     printf("Ed448 Verify : %.06f\n", (double)LOOP / SEC_DIFF(st, ed));
46
47     /* Check verify result */
48     int result = ed448_shake256_verify(pub, MSGLEN, MSG, sig);
49     printf("Ed448 Result : %s\n", result ? "valid" : "invalid");
50 }

```

図 36 nettle_benchmark_ed448.c

〈Linux〉 〈Mac〉 nettle_benchmark.c, nettle_benchmark_p256.c, nettle_benchmark_p384.c, nettle_benchmark_p521.c, nettle_benchmark_ed25519.c, nettle_benchmark_ed448.c を nettle-3.7.3.tar.gz を展開したディレクトリ (nettle-3.7.3 ディレクトリを含むディレクトリ) に配置し、以下のコマンドを実行して計測用プログラムをビルドした。

```
gcc -o nettle_benchmark nettle_benchmark.c nettle_benchmark_p256.c net-  
tle_benchmark_p384.c nettle_benchmark_p521.c nettle_benchmark_ed25519.c  
nettle_benchmark_ed448.c -Ibenchlib/include -Lbenchlib/lib -lhogweed -lnet-  
tle -lgmp  
export LD_LIBRARY_PATH=$PWD/benchlib/lib
```

〈Win〉 nettle_benchmark.c, nettle_benchmark_p256.c,
nettle_benchmark_p384.c, nettle_benchmark_p521.c, nettle_bench-
mark_ed25519.c, nettle_benchmark_ed448.c を同一ディレクトリに配置し、以下のコ
マンドを実行して計測用プログラムをビルドした。

```
gcc -o nettle_benchmark nettle_benchmark.c nettle_benchmark_p256.c net-  
tle_benchmark_p384.c nettle_benchmark_p521.c nettle_benchmark_ed25519.c  
nettle_benchmark_ed448.c -Ibenchlib/include -Lbenchlib/lib -lhogweed -lnet-  
tle -lgmp
```

【計測用プログラムの実行方法】

以下のコマンドを実行した。

```
./nettle_benchmark
```

⑤ OpenSSL

【OSS の入手】

〈Win〉 〈Linux〉 〈Mac〉 openssl 3.0.0 を以下からダウンロードした。

<https://www.openssl.org/source/openssl-3.0.0.tar.gz>

ダウンロード後に展開した。

Windows の場合は展開先パスにマルチバイト文字を含めないようにした。

OpenSSL に同梱されている計測用プログラムは鍵生成を測定しないため、計測用プ
ログラムを改造後にライブラリと計測用プログラムのビルドを行った。

【計測用プログラムの準備】

コマンドのソース apps/speed.c に対して図 37 の改造を行った。ソースが長い
ため差分のみを示す。

```

$ diff apps/speed.c apps/speed_org.c | more
434c434
< static double ecdsa_results[ECDSA_NUM][3]; /* 3 ops: keygen, sign then verify */
---
> static double ecdsa_results[ECDSA_NUM][2]; /* 2 ops: sign then verify */
442c442
< static double eddsa_results[EdDSA_NUM][3]; /* 3 ops: keygen, sign then verify */
---
> static double eddsa_results[EdDSA_NUM][2]; /* 2 ops: sign then verify */
470,471d469
< const char *ecdsa_curve[ECDSA_NUM];
< EVP_PKEY *ecdsa_pkey[ECDSA_NUM];
475,476d472
< EVP_PKEY_CTX *eddsa_pctx[EdDSA_NUM];
< EVP_PKEY *eddsa_pkey[EdDSA_NUM];
931,950c927
< static long ecdsa_c[ECDSA_NUM][3];
< static int ECDSA_keygen_loop(void *args)
< {
<     loopargs_t *tempargs = *(loopargs_t **) args;
<     EVP_PKEY **ecdsa_pkey = tempargs->ecdsa_pkey;
<     const char *curve = tempargs->ecdsa_curve[testnum];
<     int count;
<
<     for (count = 0; COND(ecdsa_c[testnum][0]); count++) {
<         ecdsa_pkey[testnum] = EVP_EC_gen(curve);
<         if (ecdsa_pkey[testnum] == NULL) {
<             BIO_printf(bio_err, "ECDSA keygen failure\n");
<             ERR_print_errors(bio_err);
<             count = -1;
<             break;
<         }
<     }
<     return count;
< }
<
---
> static long ecdsa_c[ECDSA_NUM][2];
960c937
< for (count = 0; COND(ecdsa_c[testnum][1]); count++) {
---
> for (count = 0; COND(ecdsa_c[testnum][0]); count++) {
981c958
< for (count = 0; COND(ecdsa_c[testnum][2]); count++) {
---
> for (count = 0; COND(ecdsa_c[testnum][1]); count++) {
1011,1030c988
< static long eddsa_c[EdDSA_NUM][3];
< static int EdDSA_keygen_loop(void *args)
< {
<     loopargs_t *tempargs = *(loopargs_t **) args;
<     EVP_PKEY_CTX **eddsa_pctx = tempargs->eddsa_pctx;
<     EVP_PKEY **eddsa_pkey = tempargs->eddsa_pkey;
<     int ret, count;
<
<     for (count = 0; COND(eddsa_c[testnum][0]); count++) {
<         ret = EVP_PKEY_keygen(eddsa_pctx[testnum], &eddsa_pkey[testnum]);
<         if (ret == 0) {
<             BIO_printf(bio_err, "EdDSA keygen failure\n");
<             ERR_print_errors(bio_err);
<             count = -1;
<             break;
<         }
<     }
<     return count;
< }
<
---
> static long eddsa_c[EdDSA_NUM][2];

```

```

1040c998
<   for (count = 0; COND(eddsa_c[testnum][1]); count++) {
---
>   for (count = 0; COND(eddsa_c[testnum][0]); count++) {
1061c1019
<   for (count = 0; COND(eddsa_c[testnum][2]); count++) {
---
>   for (count = 0; COND(eddsa_c[testnum][1]); count++) {
2542,2559d2499
    for (i = 0; st && i < loopargs_len; i++) {
<         /* EVP_EC_gen not support NIST curve names. So use NIDs. */
<         loopargs[i].ecdsa_curve[testnum] = OSSL_EC_curve_nid2name(ec_curves[testnum].nid);
<     }
<     if (st) {
<         pkey_print_message("keygen", "ecdsa",
<                             ecdsa_c[testnum][0], ec_curves[testnum].bits,
<                             seconds.eddsa);
<
<         Time_F(START);
<         count = run_benchmark(async_jobs, ECDSA_keygen_loop, loopargs);
<         d = Time_F(STOP);
<         BIO_printf(bio_err,
<                    "mr ? "+R5:"%ld:%u:%.2f\n"
<                    : "%ld %u bits ECDSA keygens in %.2fs\n",
<                    count, ec_curves[testnum].bits, d);
<         ecdsa_results[testnum][0] = (double)count / d;
<         op_count = count;
<     }
2580c2520
<         ecdsa_c[testnum][1], ec_curves[testnum].bits,
---
>         ecdsa_c[testnum][0], ec_curves[testnum].bits,
2589c2529
<         ecdsa_results[testnum][1] = (double)count / d;
---
>         ecdsa_results[testnum][0] = (double)count / d;
2611c2551
<         ecdsa_c[testnum][2], ec_curves[testnum].bits,
---
>         ecdsa_c[testnum][1], ec_curves[testnum].bits,
2620c2560
<         ecdsa_results[testnum][2] = (double)count / d;
---
>         ecdsa_results[testnum][1] = (double)count / d;
2720a2661
>     EVP_PKEY_CTX *ed_pctx = NULL;
2736,2739c2677,2680
<     if ((loopargs[i].eddsa_pctx[testnum] = EVP_PKEY_CTX_new_id(
<         ed_curves[testnum].nid, NULL)) == NULL
<         || EVP_PKEY_keygen_init(loopargs[i].eddsa_pctx[testnum]) <= 0
<         || EVP_PKEY_keygen(loopargs[i].eddsa_pctx[testnum], &ed_pkey) <= 0) {
---
>     if ((ed_pctx = EVP_PKEY_CTX_new_id(ed_curves[testnum].nid,
>         NULL)) == NULL
>         || EVP_PKEY_keygen_init(ed_pctx) <= 0
>         || EVP_PKEY_keygen(ed_pctx, &ed_pkey) <= 0) {
2741,2742c2682
<     EVP_PKEY_CTX_free(loopargs[i].eddsa_pctx[testnum]);
<     loopargs[i].eddsa_pctx[testnum] = NULL;
---
>     EVP_PKEY_CTX_free(ed_pctx);
2744a2685
>     EVP_PKEY_CTX_free(ed_pctx);
2767,2782d2707
<     if (st) {
<         pkey_print_message("keygen", ed_curves[testnum].name,
<                             eddsa_c[testnum][0],
<                             ed_curves[testnum].bits, seconds.eddsa);
<
<         Time_F(START);

```


- EVP_DigestVerify 関数で署名検証

を、それぞれ 10 秒間繰り返して繰り返し回数 n を測定する。 n を 10 秒で割ることで、スループット（1 秒あたりの実行回数）を求めた。計算式は

$$\frac{n}{10}[\text{回/秒}]$$

となる。

OpenSSL は Ed25519ph, Ed25519ctx, Ed448ph をサポートしていないため計測していない。

〈Linux〉 〈Mac〉 改造した OpenSSL に対して、tar.gz ファイルを展開したディレクトリ内で以下のコマンドを実行してライブラリおよび測定用プログラムをビルドした。

```
./Configure  
make  
make test
```

なお、システムにインストール済みの OpenSSL を破壊しないように make install は行わない。

〈Win〉 ビルドに必要な

OpenSSL 推奨の perl : Strawberry Perl 5.32.1.1 64bit²⁵

OpenSSL 推奨のアセンブラ : Netwide Assembler (NASM) 2.15.05²⁶

をインストール後、Windows の環境変数 PATH に C:\Program Files\NASM を追加した。

スタートメニューの Visual Studio 2022 ディレクトリから x64 Native Tools Command Prompt for VS 2022 を管理者権限で起動し、コマンドプロンプトで tar.gz ファイルを展開したディレクトリに移動した。

改造した OpenSSL に対して、以下のコマンドを実行してライブラリおよび測定用プログラムをビルドした。

```
perl Configure VC-WIN64A  
nmake  
nmake test
```

なお、システムにインストール済みの OpenSSL を破壊しないように nmake install は行わない。

²⁵ <https://strawberryperl.com>

²⁶ <https://www.nasm.us>

【計測用プログラムの実行方法】

〈Linux〉 〈Mac〉 以下のコマンドを実行した.

```
export LD_LIBRARY_PATH=$PWD
./apps/openssl speed ecdsap256 ecdsap384 ecdsap521 ed25519 ed448
```

〈Win〉 以下のコマンドを実行した.

```
apps\openssl.exe speed ecdsap256 ecdsap384 ecdsap521 ed25519 ed448
```

⑥ wolfSSL

【OSS の入手】

〈Win〉 〈Linux〉 〈Mac〉 wolfSSL 5.0.0 を以下からダウンロードした.

<https://github.com/wolfSSL/wolfssl/archive/refs/tags/v5.0.0-stable.zip>

ダウンロード後に展開した.

Windows では Visual Studio を使用したビルド方法は開発元から示されているが, この方法でビルドした wolfSSL は EdDSA が含まれないことが分かった. また, MSYS2 では wolfSSL をディストリビューションで配布していないため, wolfSSL 開発者の示した MSYS2 MinGW 64-bit 上での方法でビルドすることとした.

〈Linux〉 〈Mac〉 〈Win〉 展開したディレクトリ内で以下のコマンドを実行してライブラリおよび測定用プログラムをビルドした.

```
mkdir ../benchlib
./autogen.sh
./configure --prefix=$PWD/../benchlib --enable-eccustcurves --enable-
ed25519 --enable-ed448
make
make install
export LD_LIBRARY_PATH=$PWD/../benchlib
```

【計測用プログラムの準備】

wolfSSL に同梱されている計測用プログラムを利用した.

この計測用プログラムでは

- ECDSA P-256, ECDSA P-384, ECDSA P-521 の場合
 - `wc_ecc_make_key_ex` 関数で鍵生成
 - `wc_ecc_sign_hash` 関数で署名生成
 - `wc_ecc_verify_hash` 関数で署名検証

- Ed25519 の場合
 - wc_ed25519_make_key 関数で鍵生成
 - wc_ed25519_sign_msg 関数で署名生成
 - wc_ed25519_verify_msg 関数で署名検証
- Ed448 の場合
 - wc_curve448_make_key 関数で鍵生成
 - wc_ed448_sign_msg 関数で署名生成
 - wc_ed448_verify_msg 関数で署名検証

を、それぞれ 1 秒間繰り返して繰り返し回数 n を測定してスループット（1 秒あたりの実行回数）を求めた。

Libgcrypt 等のライブラリでサポートしていないため、Ed25519ph, Ed25519ctx, Ed448ph は計測していない。

【計測用プログラムの実行方法】

〈Linux〉 〈Mac〉 〈Win〉 以下のコマンドを実行した。

```
./wolfcrypt/benchmark/benchmark -ecc-all -ed25519-kg -ed25519 -ed448-kg -ed448
```

5.4.3. 性能測定結果と評価

測定結果はスループット（回/秒）で統一した。

① 鍵ペア生成

鍵ペア生成のスループット測定結果を表 10 に示す。

表 10 鍵ペア生成 スループット測定結果（回/秒）

		Bouncy Castle	OpenJDK	Libgcrypt	nettle	OpenSSL3.0.0	WolfSSL
Windows	ECDSA P-256	4566.2	1408.5	51.1	24791.9	23885.2	681.9
	ECDSA P-384	2155.2	734.8	34.8	10621.8	1003.9	240.9
	ECDSA P-521	1347.7	404.5	21.0	5576.7	486.1	144.8
	Ed25519	9345.8	2132.2	75.9	31620.0	26747.0	49900.8
	Ed448	4424.8	769.8	59.1	9272.0	2745.9	12989.2
Linux	ECDSA P-256	5586.6	1721.2	172.6	30224.9	40558.6	797.7
	ECDSA P-384	2702.7	889.7	88.9	13615.5	1458.4	279.2
	ECDSA P-521	1569.9	478.2	44.2	7090.8	608.4	167.6
	Ed25519	14492.8	2816.9	498.3	35703.3	36800.2	49860.2
	Ed448	5780.3	910.7	311.2	11478.2	5589.3	14574.6
Mac	ECDSA P-256	5263.2	1626.0	114.4	28981.1	25298.1	849.4
	ECDSA P-384	2481.4	825.1	62.4	13533.5	1335.2	272.3
	ECDSA P-521	1402.5	458.3	31.1	6693.2	588.0	151.2
	Ed25519	11627.9	2631.6	465.9	35966.0	27178.0	54304.9
	Ed448	4975.1	874.9	294.7	10901.0	20436.6	11809.1

ECDSA P-256 と Ed25519 の鍵ペア生成におけるスループット比較のグラフを図 38, 図 39 および図 40 に示す.

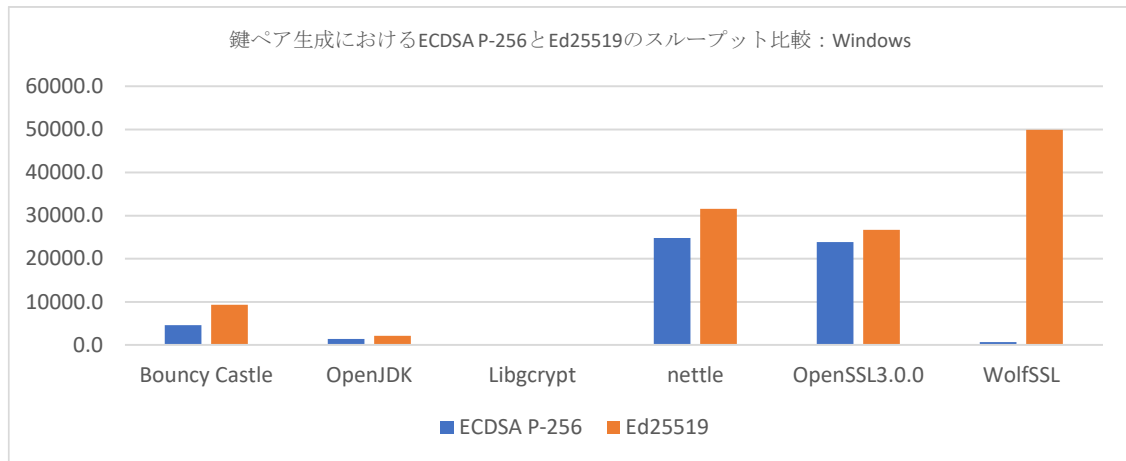


図 38 鍵ペア生成における ECDSA P-256 と Ed25519 のスループット比較：Windows

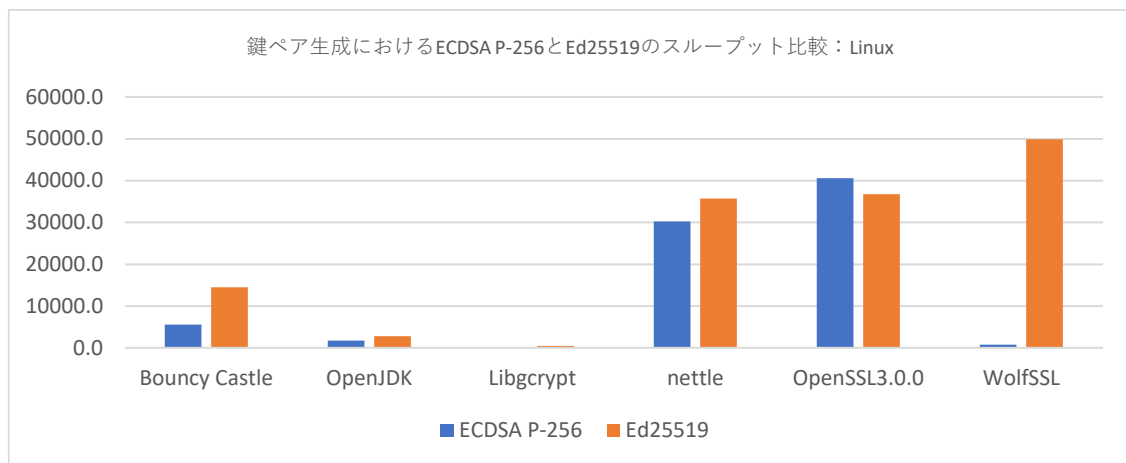


図 39 鍵ペア生成における ECDSA P-256 と Ed25519 のスループット比較：Linux

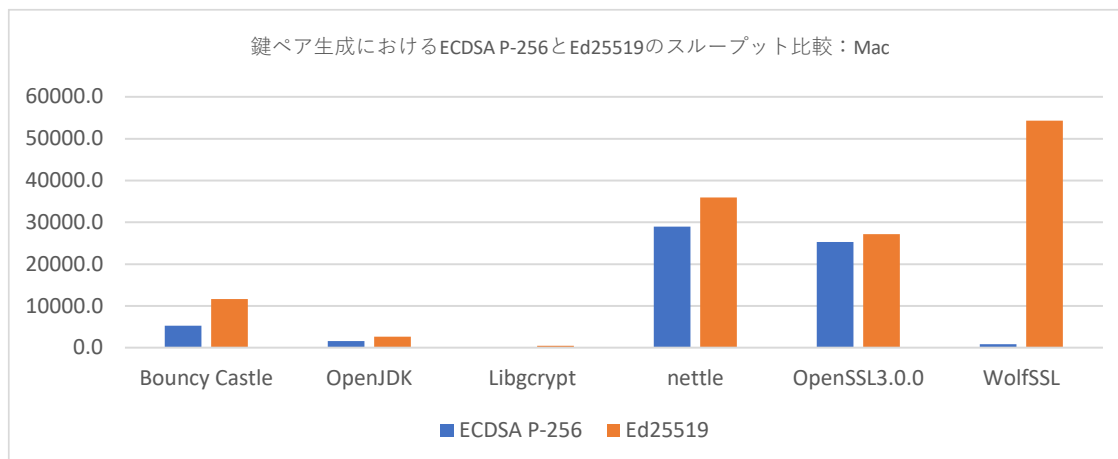


図 40 鍵ペア生成における ECDSA P-256 と Ed25519 のスループット比較：Mac

wolfSSL の ECDSA P-256 と Ed25519 の性能差が非常に大きいですが、プラットフォームによらず同じ傾向となっていることから、誤差ではないと考えられる。C 言語の wolfSSL の ECDSA P-256 が Java の OpenJDK より低いスループットとなっていることから、wolfSSL の ECDSA P-256 のスループットが C 言語実装のわりに低いと考えられる。wolfSSL の ECDSA P-256 のスループットが仮に Nettle や OpenSSL 並であれば、Ed25519 との比率は 1～2 倍となり不自然さはなくなると思われる。

同じ方式を実装したにもかかわらず OSS によって性能にバラツキはあるが、どのプラットフォームでも最も高いスループットのアルゴリズムは Ed25519 であり、これらの OSS を比較した限りでは、同程度の安全性の ECDSA P-256 に比べ、Ed25519 の鍵生成は高スループットの実装となる傾向があると言える。

ECDSA P-384, ECDSA-P521, Ed448 の鍵ペア生成におけるスループット比較のグラフを図 41、図 42 および図 43 に示す。

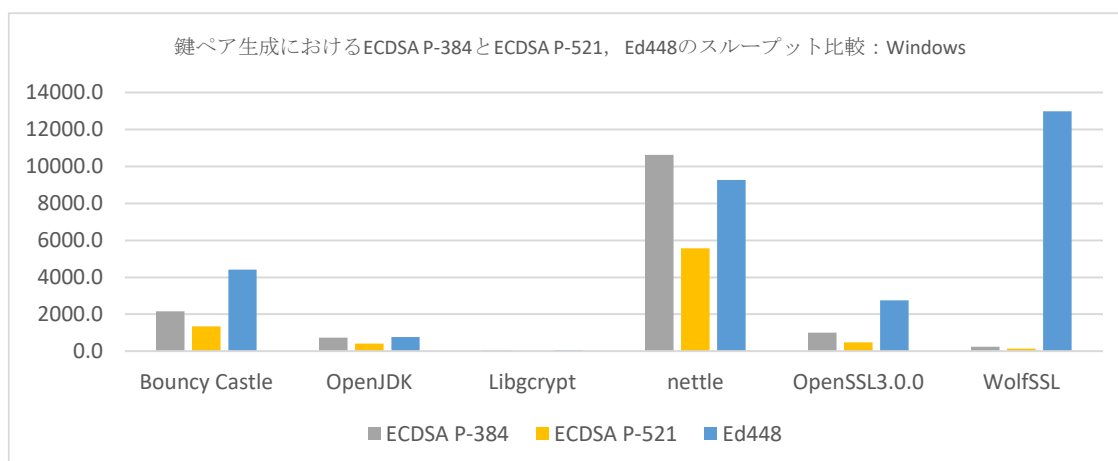


図 41 鍵ペア生成における ECDSA P-384 と ECDSA P-521, Ed448 のスループット比較：Windows

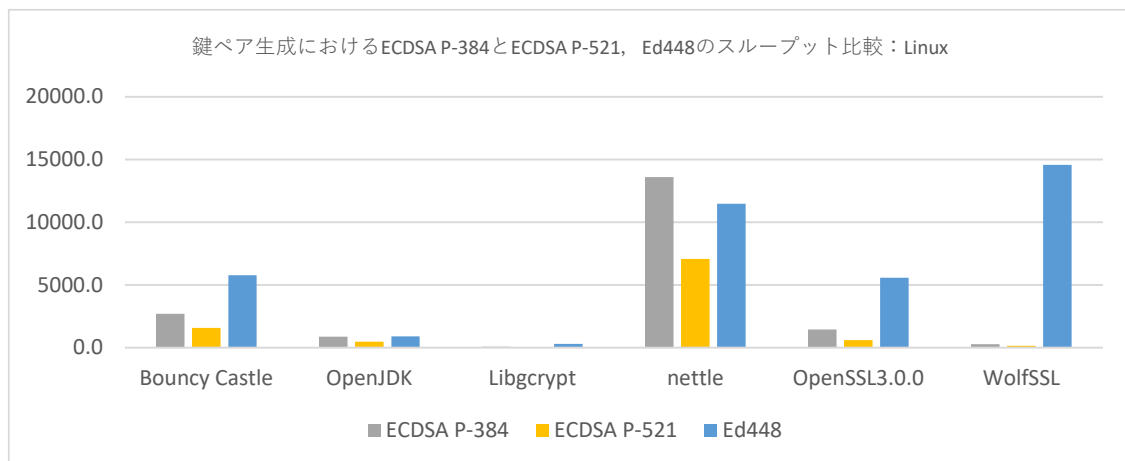


図 42 鍵ペア生成における ECDSA P-384 と ECDSA P-521, Ed448 のスループット比較 : Linux

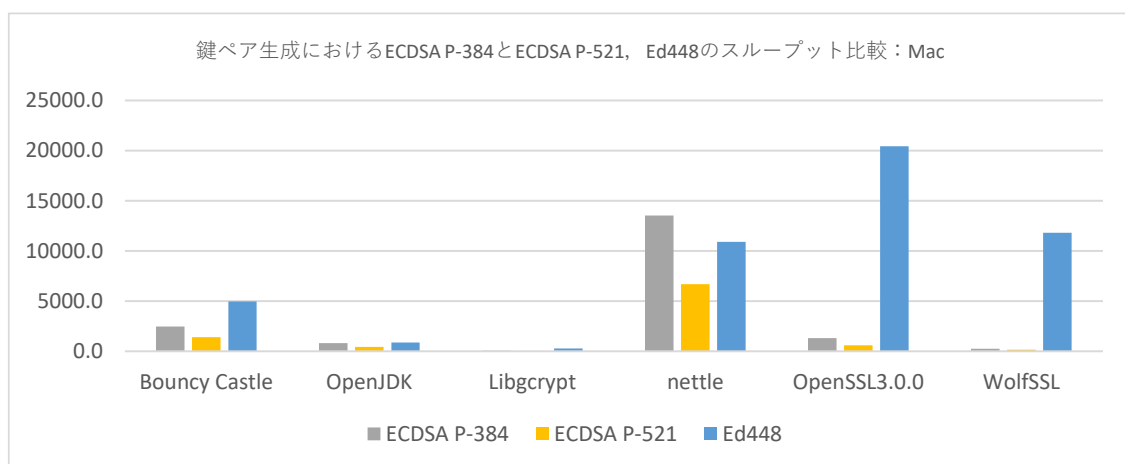


図 43 鍵ペア生成における ECDSA P-384 と ECDSA P-521, Ed448 のスループット比較 : Mac

wolfSSL と OpenSSL の ECDSA P-384, ECDSA P-521 と Ed448 の性能差が非常に大きい
が、プラットフォームによらず同じ傾向となっていることから、誤差ではないと考え
られる。C 言語の wolfSSL と OpenSSL の ECDSA P-384, ECDSA P-521 が Java の OpenJDK
より低い同程度のスループットとなっていることから、wolfSSL と OpenSSL の ECDSA
P-384, ECDSA P-521 のスループットが仮に Nettle 並であれば、Ed25519 との比率は同
程度となり不自然さはなくなると思われる。

Mac の OpenSSL3.0.0 の Ed448 のスループットが Windows の OpenSSL3.0.0 の Ed448
もスループットに比べて 7 倍以上となっている。CPU が異なるとはいえ、同じソース
からビルドしているため、これほどの差がついたのは、アルゴリズムや実装の仕方によ
らない原因（例えばコンパイラの違いによる最適化）による可能性がある。

同じ方式を実装したにもかかわらず OSS によって性能にバラツキはあるが、どのプ
ラットフォームでも最も高いスループットのアルゴリズムは Ed448 であり、これらの

OSS を比較した限りでは、低い安全性の ECDSA P-384 に比べても、Ed448 の鍵生成は同程度かやや高スループットの実装となる傾向があると言える。

② 署名生成

署名生成のスループット測定結果を表 11 に示す。

表 11 署名生成スループット測定結果 (回/秒)

		Bouncy Castle	OpenJDK	Libgcrypt	nettle	OpenSSL3.0.0	WolfSSL
Windows	ECDSA P-256	2331.0	1661.1	457.7	17197.3	43939.4	1309.6
	ECDSA P-384	1129.9	733.1	240.8	7856.7	1011.1	460.8
	ECDSA P-521	561.2	371.3	107.8	4055.1	474.4	280.8
	Ed25519	20408.2	1312.3	2207.8	30745.1	28579.8	44471.8
Linux	Ed448	6024.1	408.2	760.2	9535.6	2735.8	12663.3
	ECDSA P-256	3300.3	1893.9	742.6	21508.9	57375.9	1543.0
	ECDSA P-384	1210.7	863.6	340.9	9945.1	1439.3	540.1
	ECDSA P-521	628.9	422.1	157.1	5235.4	621.1	295.1
Mac	Ed25519	21739.1	1647.4	2528.6	35597.5	39167.9	42388.9
	Ed448	6493.5	475.3	823.5	11294.6	5572.8	13725.5
	ECDSA P-256	2604.2	1782.5	432.8	20405.7	44969.4	1617.9
	ECDSA P-384	1068.4	800.0	226.9	9715.1	1337.2	521.8
	ECDSA P-521	595.9	398.6	106.8	4910.3	573.1	295.4
	Ed25519	20000.0	1545.6	2161.9	35485.0	29554.7	48203.1
	Ed448	5917.2	449.6	781.6	10657.8	20479.0	11403.7

ECDSA P-256 と Ed25519 の署名生成におけるスループット比較のグラフを図 44、図 45 および図 46 に示す。

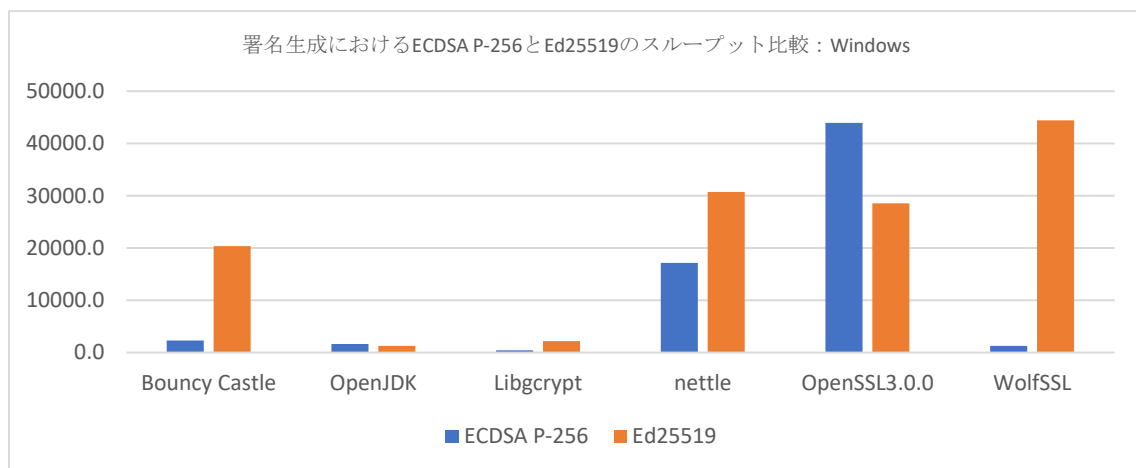


図 44 署名生成における ECDSA P-256 と Ed25519 のスループット比較 : Windows

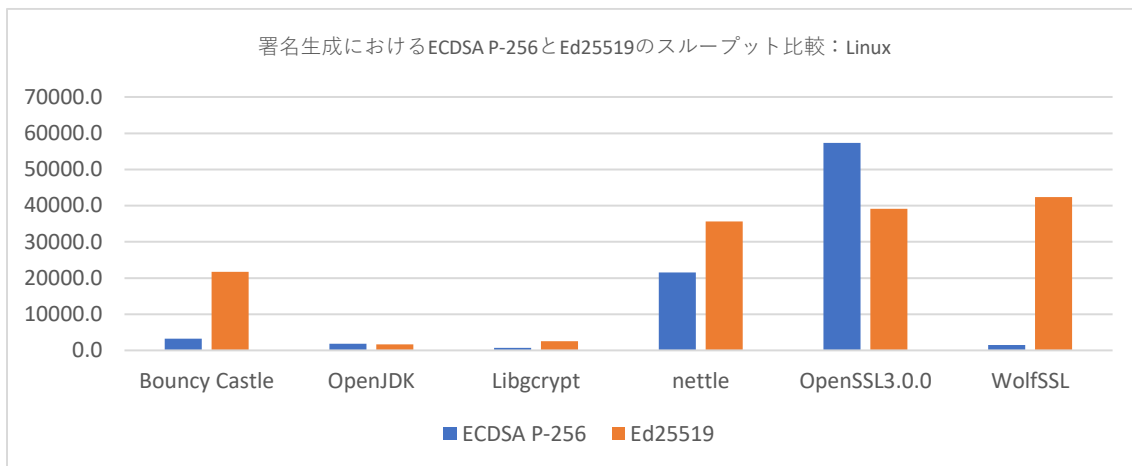


図 45 署名生成における ECDSA P-256 と Ed25519 のスループット比較：Linux

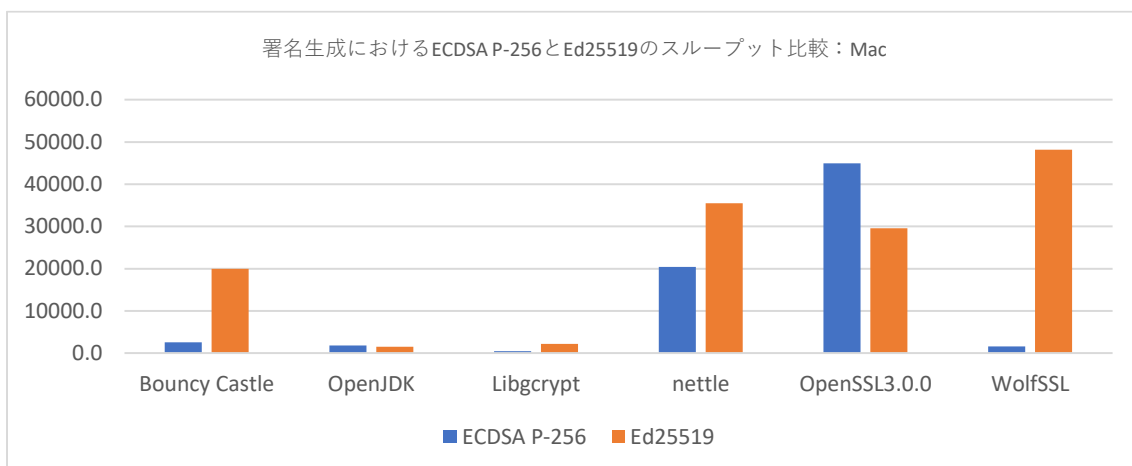


図 46 署名生成における ECDSA P-256 と Ed25519 のスループット比較：Mac

Bouncy Castle と wolfSSL の ECDSA P-256 と Ed25519 の性能差が非常に大きいですが、プラットフォームによらず同じ傾向となっていることから、誤差ではないと考えられる。C 言語の wolfSSL の ECDSA P-256 が Java の OpenJDK より低いスループットとなっていることから、C 言語実装の wolfSSL の ECDSA P-256 のスループットが低いと考えられる。wolfSSL の ECDSA P-256 のスループットが仮に OpenSSL 並であれば、同程度のスループットとなり不自然さはなくなると思われる。

OpenJDK と OpenSSL ではどのプラットフォームでも ECDSA P-256 が高いスループットだった。一方で、それ以外の 4 つの OSS では Ed25519 が高スループットであり、OSS によって分かれた形となった。特に OpenSSL の ECDSA P-256 と wolfSSL の Ed25519 はほぼ同じ程度で高いスループットとなっている。

同じ方式を実装したにもかかわらず OSS によって性能にバラツキはあるが、高速な実装をすれば ECDSA P-256 と Ed25519 は同程度のスループットとなるが、アルゴリズムを素直に実装すれば、同程度の安全性の ECDSA P-256 に比べ、Ed25519 の署名生成は高スループットの実装となる傾向があると言える。

ECDSA P-384, ECDSA P-521 と Ed448 の署名生成におけるスループット比較のグラフを図 47, 図 48 および図 49 に示す.

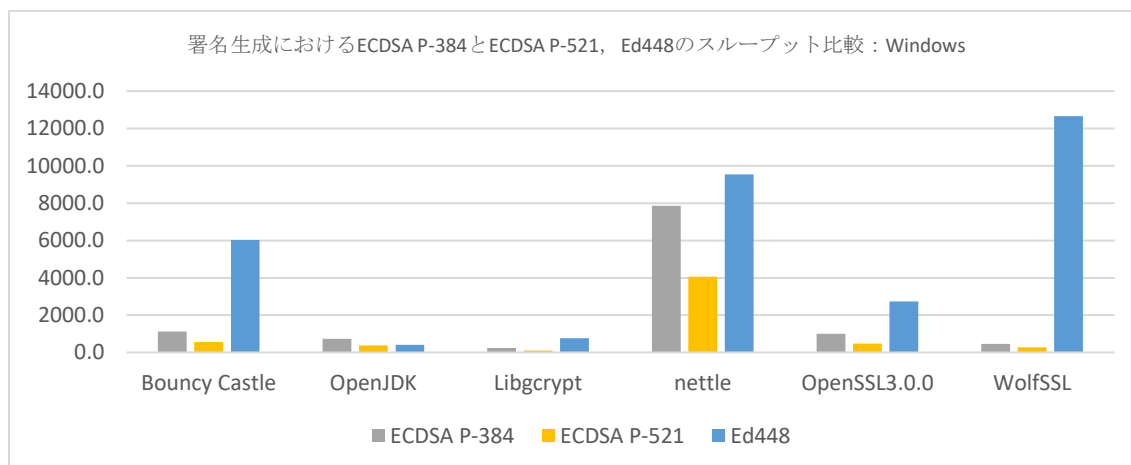


図 47 署名生成における ECDSA P-384 と ECDSA P-521, Ed448 のスループット比較 : Windows

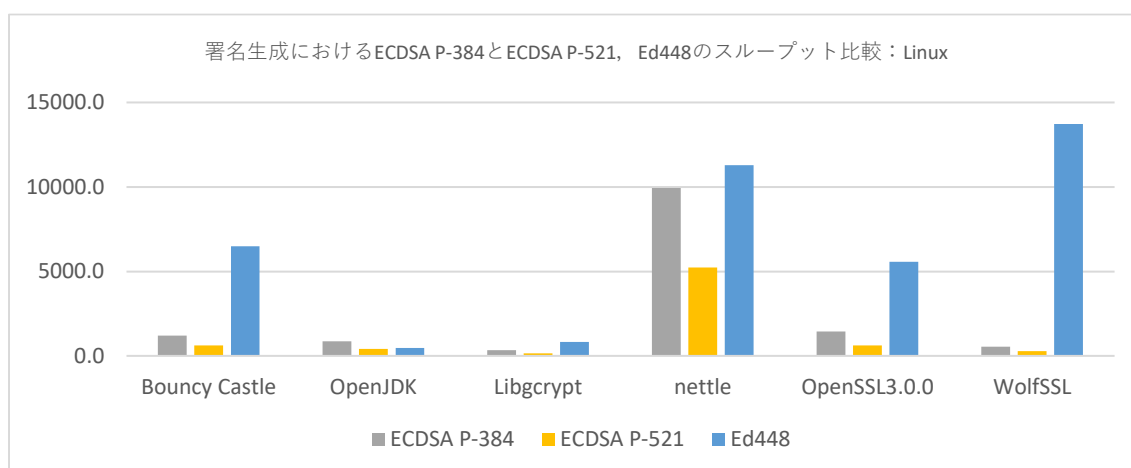


図 48 署名生成における ECDSA P-384 と ECDSA P-521, Ed448 のスループット比較 : Linux

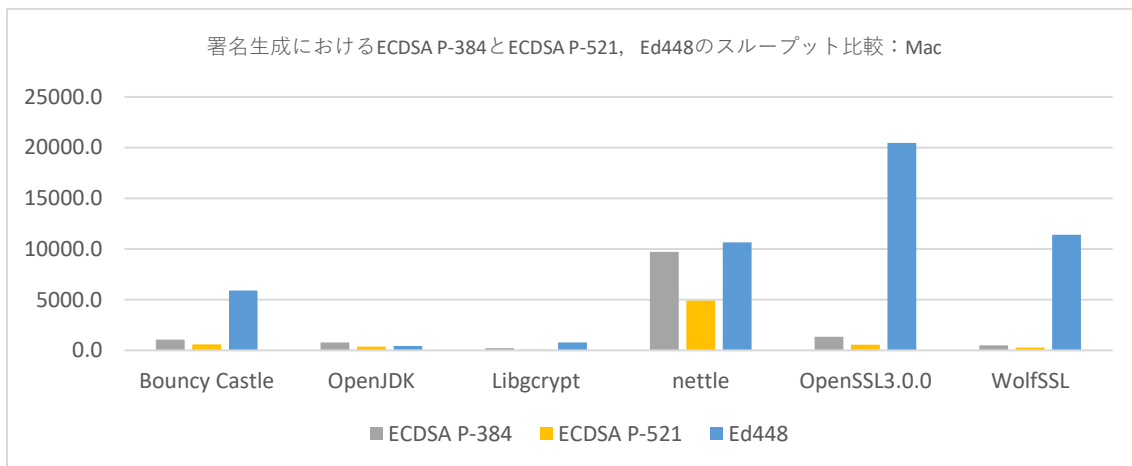


図 49 署名生成における ECDSA P-384 と ECDSA P-521, Ed448 のスループット比較：Mac

Bouncy Castle, wolfSSL の ECDSA P-384, ECDSA P-521 と Ed448 の性能差が非常に大きいですが、プラットフォームによらず同じ傾向となっていることから、誤差ではないと考えられる。

Mac の OpenSSL3.0.0 の Ed448 のスループットが Windows の OpenSSL3.0.0 の Ed448 もスループットに比べて 7 倍以上となっている。CPU が異なるとはいえ、同じソースからビルドしているにもかかわらず、これほどの差がついたのは、アルゴリズムや実装の仕方によらない原因（例えばコンパイラの違いによる最適化の違い）による可能性がある。

C 言語の wolfSSL の ECDSA P-384, ECDSA P-521 が Java の OpenJDK より低いスループットとなっていることから、C 言語実装の wolfSSL の ECDSA P-384, ECDSA P-521 のスループットが低いと考えられる。wolfSSL の ECDSA P-384, ECDSA P-521 のスループットが仮に Nettle 並であれば、比率は 1～2 倍程度となり不自然さはなくなると思われる。

同じ方式を実装したにもかかわらず OSS によって性能にバラツキはあるが、どのプラットフォームでも最も高いスループットのアルゴリズムは Ed448 であり、これらの OSS を比較した限りでは、低い安全性の ECDSA P-384 に比べても、Ed448 の署名生成は高スループットの実装となる傾向があると言える。

③ 署名検証

署名検証のスループット測定結果を表 12 に示す。

表 12 署名検証スループット測定結果（回/秒）

		Bouncy Castle	OpenJDK	Libgcrypt	nettle	OpenSSL3.0.0	WolfSSL
Windows	ECDSA P-256	4201.7	891.3	408.2	5678.9	17006.0	1884.1
	ECDSA P-384	1329.8	399.4	212.4	2621.1	1200.8	706.1
	ECDSA P-521	639.8	202.6	92.9	1311.6	593.1	431.4
	Ed25519	7246.4	1293.7	1223.6	8867.3	9995.1	18295.9
	Ed448	1919.4	427.0	423.7	2719.4	2048.1	4644.7
Linux	ECDSA P-256	3937.0	1065.0	659.6	7110.8	19853.7	2234.6
	ECDSA P-384	1550.4	464.0	301.3	3152.9	1740.9	747.9
	ECDSA P-521	741.3	233.6	134.8	1741.0	762.5	496.8
	Ed25519	7518.8	1584.8	1358.5	10797.9	12043.9	20916.8
	Ed448	2375.3	486.6	450.9	3510.4	5387.5	5120.1
Mac	ECDSA P-256	3144.7	1006.0	393.5	6541.3	18093.7	2375.6
	ECDSA P-384	1506.0	433.7	202.3	3073.8	1606.8	804.9
	ECDSA P-521	698.8	220.1	92.7	1592.8	726.1	458.7
	Ed25519	6622.5	1483.7	1196.5	10074.1	10811.0	18492.3
	Ed448	2036.7	455.2	428.0	3242.5	6800.2	3788.6

ECDSA P-256 と Ed25519 の署名検証におけるスループット比較のグラフを図 50、図 51 および図 52 に示す。

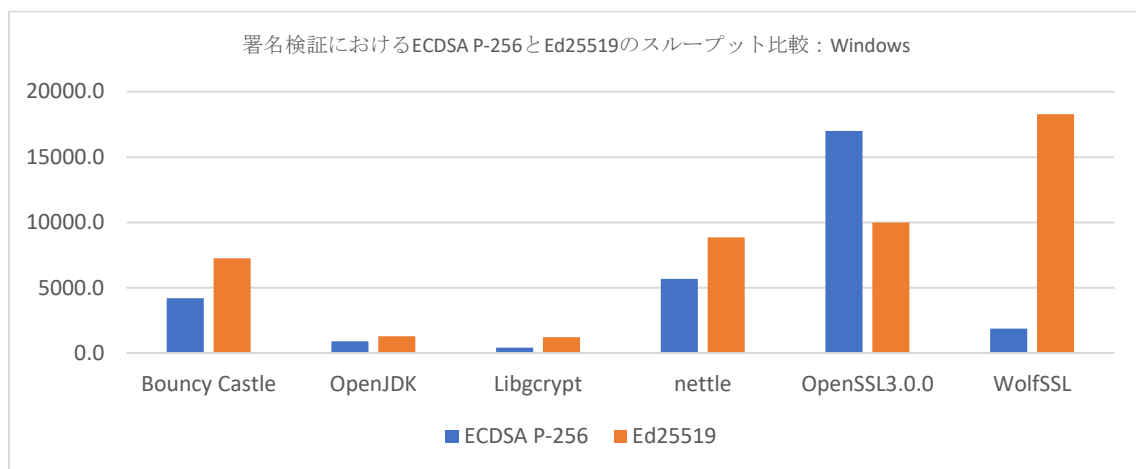


図 50 署名検証における ECDSA P-256 と Ed25519 のスループット比較：Windows

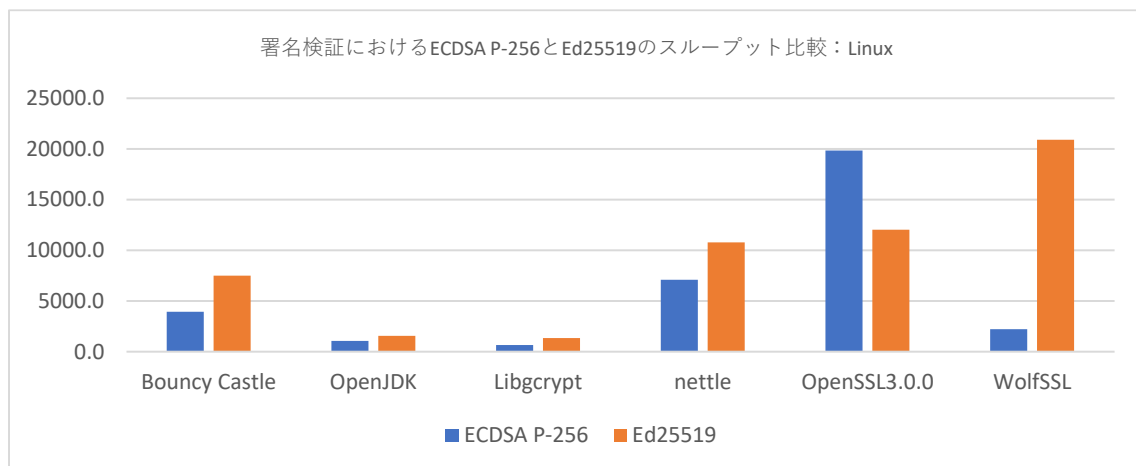


図 51 署名検証における ECDSA P-256 と Ed25519 のスループット比較：Linux

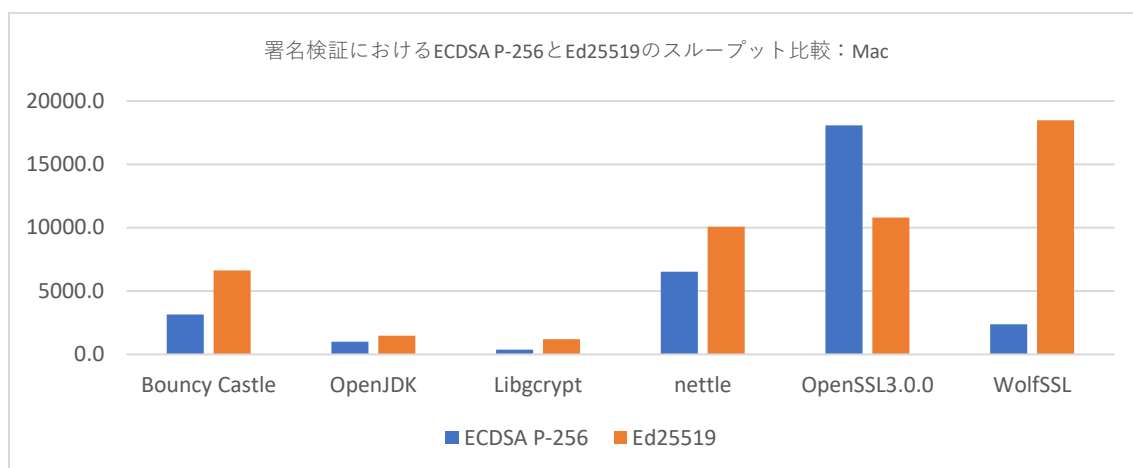


図 52 署名検証における ECDSA P-256 と Ed25519 のスループット比較 : Mac

wolfSSL の ECDSA P-256 と Ed25519 の性能差が非常に大きいですが、プラットフォームによらず同じ傾向となっていることから、誤差ではないと考えられる。C 言語の wolfSSL の ECDSA P-256 が Java の Bouncy Castle より低いスループットとなっていることから、C 言語実装の wolfSSL の ECDSA P-256 のスループットが低いと考えられる。wolfSSL の ECDSA P-256 のスループットが仮に OpenSSL 並であれば、同程度のスループットとなり不自然さはなくなると思われる。

OpenSSL では ECDSA P-256 が高いスループットだった。一方で、それ以外の 5 つの OSS では Ed25519 が高スループットであり、OSS によって分かれた形となった。特に OpenSSL の ECDSA P-256 と wolfSSL の Ed25519 はほぼ同じ程度で高いスループットとなっている。

同じ方式を実装したにもかかわらず OSS によって性能にバラツキはあるが、高速な実装をすれば ECDSA P-256 と Ed25519 は同程度のスループットとなるが、アルゴリズムを素直に実装すれば、同程度の安全性の ECDSA P-256 に比べ、Ed25519 の署名生成は高スループットの実装となる傾向があると言える。

ECDSA P-384, ECDSA-P521, Ed448 の署名検証におけるスループット比較のグラフを図 53, 図 54 および図 55 に示す。

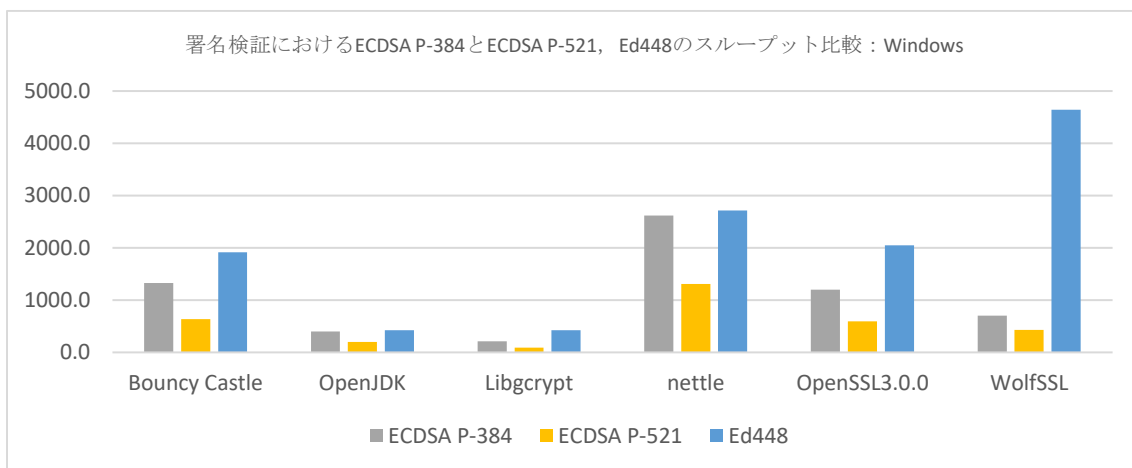


図 53 署名検証における ECDSA P-384 と ECDSA P-521, Ed448 のスループット比較：Windows

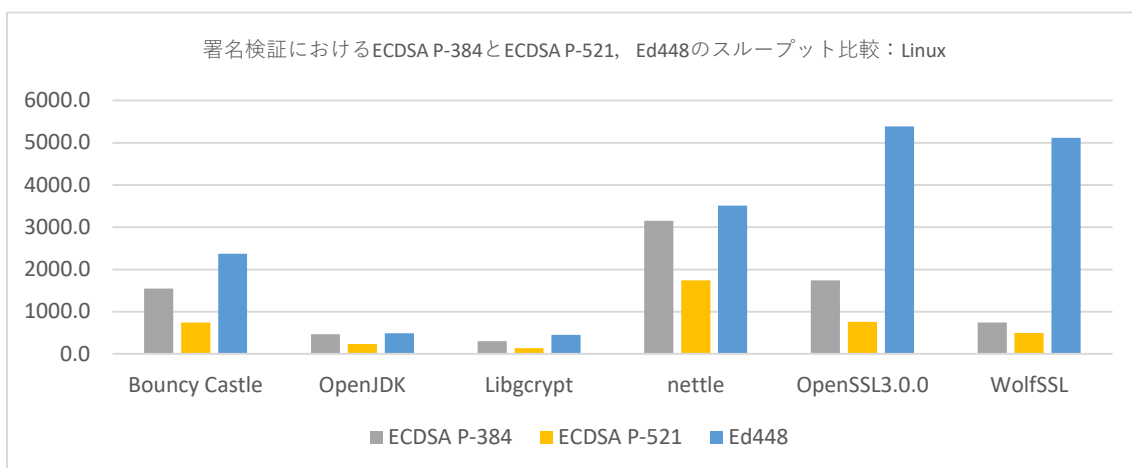


図 54 署名検証における ECDSA P-384 と ECDSA P-521, Ed448 のスループット比較：Linux

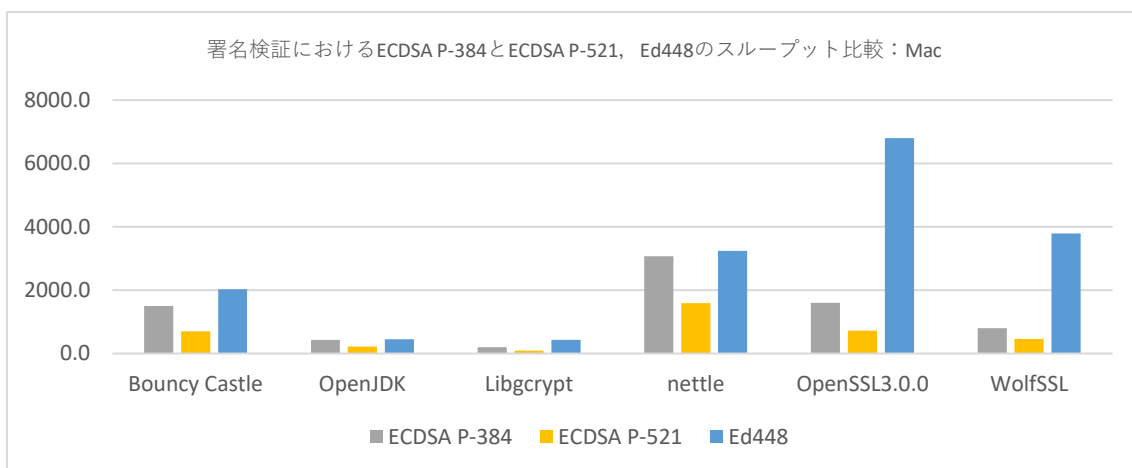


図 55 署名検証における ECDSA P-384 と ECDSA P-521, Ed448 のスループット比較：Mac

wolfSSL の ECDSA P-384, ECDSA P-521 と Ed448 の性能差が非常に大きいですが、プラットフォームによらず同じ傾向となっていることから、誤差ではないと考えられる。

Mac の OpenSSL3.0.0 の Ed448 のスループットが Windows の OpenSSL3.0.0 の Ed448 もスループットに比べて 3.3 倍以上となっている。CPU が異なるとはいえ、同じソースからビルドしているにもかかわらず、これほどの差がついたのは、アルゴリズムや実装の仕方によらない原因（例えばコンパイラの違いによる最適化の違い）による可能性がある。

C 言語の wolfSSL の ECDSA P-384, ECDSA P-521 が Java の Bouncy Castle より低いスループットとなっていることから、C 言語実装の wolfSSL の ECDSA P-384, ECDSA P-521 のスループットが低いと考えられる。wolfSSL の ECDSA P-384, ECDSA P-521 のスループットが仮に Nettle 並であれば、比率は 1~2 倍程度となり不自然さはなくなると思われる。

同じ方式を実装したにもかかわらず OSS によって性能にバラツキはあるが、どのプラットフォームや OSS でも高いスループットのアルゴリズムは Ed448 であり、これらの OSS を比較した限りでは、低い安全性の ECDSA P-384 に比べても、Ed448 の署名生成は高スループットの実装となる傾向があると言える。

6. EdDSA に関する実装評価まとめ

本調査では、RFC8032 [2]に記載された EdDSA のアルゴリズムとしての特徴を確認し、世間での採用の動向を調査し、実際の性能測定し、実装性能を評価した。

EdDSA のアルゴリズムとしての特徴は以下のようであった。

- 一般的な EdDSA と比較して、推奨パラメータが固定であることや推奨される座標系の特性により乗法逆元の処理が不要となり処理が単純化されているため高速化しやすい。
- 一般的な EdDSA と比較して、条件分岐等を考慮する必要がないため実装がしやすい。
- 実装するアルゴリズムが 1 つで条件分岐等を考慮する必要がなく分岐がないため、SCA 対策として有用である。

つまり、RFC8032 [2]で詳細について触れられていない有限体の演算を除くと、推奨される処理手順のみでも安全性や性能、実装コストの抑制がある程度、担保される。

また、RFC8032 [2]として標準化された EdDSA は、Draft 版を含めると 41 件の標準が採用し、少なくとも 20 件の OSS の暗号ライブラリが採用していた。2017 年のアルゴリズム発表から 4 年が経過して、署名方式として定着しつつあることが分かった。さらに、2 件の Draft 版の標準で参照していた FIPS Pub 186-5 (Draft) [12]にも同じアルゴリズムが含まれており、RFC8032 [2]と FIPS Pub 186-5 (Draft) [12]に共通して含まれる方式として今後もさらに利用が広がると予想される。

ただし、Ed448 は採用する標準や OSS の暗号ライブラリが Ed25519 に比べて少なかった。

調査した標準 46 件のうち、Ed25519 を採用していたものが 46 件、Ed448 を採用していたものが 28 件であり、EdDSA を採用している OSS の暗号ライブラリ 20 件のうち、20 件すべてが Ed25519 を使用している一方で、Ed448 を採用していたのは 8 件であった。

OSS の暗号ライブラリでの実装について品質という観点から、脆弱性に関する報告についても調査すると 2 件が見つかった。

OSS の暗号ライブラリの第三者による実装性能の測定値から

- Ed25519 は ECDSA P-256 と同等の安全性でありながら高速
- Ed448 は ECDSA P-384 より高い安全性でありながら高速

という傾向が読み取れた。

C 言語の 4 個、Java の 2 個の OSS の暗号ライブラリを、Windows, Linux, Mac の 3 つの OS 上でそれぞれ動かした計 18 個の環境で、実際に性能測定をしたところ、各環境の結果は以下ようになった。

- Ed25519
 - 鍵ペア生成

- ✧ 17 環境で ECDSA P-256 より高スループットだった。
1 環境では ECDSA P-256 より低スループットだったが 50%未満の極端な差はなかった。
 - 署名生成
 - ✧ 12 環境で ECDSA P-256 より高スループットだった。
6 環境では ECDSA P-256 より低スループットだったが 50%未満の極端な差はなかった。
 - 署名検証
 - ✧ 15 環境で ECDSA P-256 より高スループットだった。
3 環境では ECDSA P-256 より低スループットだったが 50%未満の極端な差はなかった。
- Ed448
 - 鍵ペア生成
 - ✧ 15 環境で ECDSA P-384 より高スループットだった。
3 環境では ECDSA P-384 より低スループットだったが 50%未満の極端な差はなかった。
18 環境で ECDSA P-521 より高スループットだった。
 - 署名生成
 - ✧ 15 環境で ECDSA P-384 より高スループットだった。
3 環境では ECDSA P-384 より低スループットだったが 50%未満の極端な差はなかった。
18 環境で ECDSA P-521 より高スループットだった。
 - 署名検証
 - ✧ 15 環境で ECDSA P-384 より高スループットだった。
3 環境では ECDSA P-384 より低スループットだったが 50%未満の極端な差はなかった。
18 環境で ECDSA P-521 より高スループットだった。

以上の結果から、今回の測定した範囲では、Ed25519 は ECDSA P-256 と同等の安全性でありながら高速であると評価でき、Ed448 は ECDSA P-384 より高い安全性でありながら同等または高速であると評価できる。また、この結果は第三者の測定結果の傾向とも一致する。

以上の調査結果から、EdDSA は、方式そのものに安全性担保、高速化、実装がしやすい特性があると評価でき、実際に OSS で実装された暗号ライブラリの性能は、同程度の安全性の ECDSA や高い安全性の ECDSA と比べても遜色のないものであった。実装された OSS の品質、高速化技法は洗練されているとは言い切れないが、歴史のある ECDSA と比べても性能で劣ることはないと考えられる。

7. 参考文献

- [1] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schewabe , Bo-Yin Yang, “High-speed high-security signatures,” 2011.
- [2] Internet Research Task Force (IRTF), “RFC8032 Edwards-Curve Digital Signature Algorithm (EdDSA),” 2017.
- [3] Harold M. Edwards, “A normal form for elliptic curves,” In Bulletin of the American Mathematical Society, 2007.
- [4] Daniel J. Bernstein , Tanja Lange, “Faster addition and doubling on elliptic curves,” 2007.
- [5] Claus P. Schnorr, “Efficient Identification and Signatures for Smart Cards,” 1990.
- [6] 新保 淳, 岡田 光司 , 丹羽 朗人, “ECDSA 評価報告書,” 2001.
- [7] Internet Research Task Force (IRTF), “RFC7748 Elliptic Curves for Security,” 2016.
- [8] 藤崎 英一郎, “デジタル署名 EdDSA の構成の安全性に関する調査および評価,” 2020.
- [9] 安田 雅哉, “デジタル署名 EdDSA で使われている曲線の安全性に関する調査及び評価,” 2020.
- [10] Steven D. Galbraith, “CRYPTREC Review of EdDSA,” 2020.
- [11] John M. Pollard, “Monte carlo methods for index computation (mod p),” *Mathematics of computation*, 第32巻 143号, p. 918-924, 7 1978.
- [12] National Institute of Standards and Technology, “FIPS Pub 186-5 (Draft) Digital Signature Standard (DSS),” 2019.
- [13] National Institute of Standards and Technology, “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions FIPS PUB 202,” 2015.
- [14] Internet Engineering Task Force (IETF), “RFC6234 US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF),” 2011.
- [15] Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter , Ed Dawson, “Twisted Edwards Curves Revisited,” Springer-Verlag, 2008.
- [16] VAMPIRE lab, “SUPERCOP,” <https://bench.cr.yp.to/supercop.html>.