Computer Security Symposium 2020
26 - 29 October 2020

BLS署名の実装とその応用

光成 滋生[1,a)]

概要：2001 年に提案された BLS 署名は決定的アルゴリズムであり，閾値署名や集約署名が可能である．2016 年に DFINITY が初めて BLS 署名を利用したブロックチェーン上の合意アルゴリズムを提案した．現在 IETF でそのペアリング曲線，BLS 署名，および楕円曲線へのハッシュ関数の仕様が策定中である．著者は 2016 年から BLS 署名の実装を行い，そのライブラリは Ethereum などの様々なプロジェクトで採用されている．本論文ではその実装方法について紹介する．

キーワード：BLS 署名, 閾値署名, ブロックチェーン, Ethereum

Implementation and application of BLS signature

Mitsunari Shigeo[1,a)]

Abstract: BLS signature proposed in 2001 is a deterministic algorithm and use for threshold signature and aggregate signature. DFINITY proposed a random beacon algorithm on a blockchain by BLS signature in 2016. Now, the standardization of pairing parameters, BLS signature and hash function to an elliptic curve are discussed in IRTF. We started to implement a BLS signature since 2016 for DFINITY and the library is adopted by several projects such as Ethereum 2. This paper introduces details of the implementation and applications.

Keywords: BLS signature, threshold signature, blockchain, Ethereum

1. はじめに

BLS 署名 [5] は決定的アルゴリズムであり，閾値署名や集約署名に利用できる [3]．2016 年に DFINITY が BLS 署名をブロックチェーン上のランダムビーコンアルゴリズムに利用する．その後，Harmony が BLS 署名を利用した合意形成アルゴリズム BFT（Byzantine Fault Tolerance）を提案したり [12]，Ethereum が署名検証時間削減利用したりしている [8]．また，相互利用のために標準化が進められている．

著者は 2016 年から DFINITY で利用する BLS 署名ライブラリを実装してきた．そのライブラリは Harmony, Ethereum, ... など様々なプロジェクトで利用されている．ターゲット CPU は x64，aarch64，ARM, MIPS，利用言語も，Go, Rust, WebAssembly, C#などはば広い．

高速性と移植の容易さを両立させるための手法を紹介する．

[1] サイボウズ・ラボ株式会社
Cybozu Labs, Inc.
[a)] herumi@nifty.com

2. BLS 署名

2.1 記法

G_1, G_2 を生成元が P_1, P_2 の素数位数 r の加法巡回群，G_T を位数 r の乗法巡回群とする．$H_i : \{0,1\}^* \to G_i$ をハッシュ関数とする．

$e : G_1 \times G_2 \to G_T$ を type-3 の非対称ペアリングとする．$g := e(P_1, P_2)$ は G_T の生成元である．群 G_i の添え字 (I, I') は $(1,2)$ または $(2,1)$ のどちらかを指すものとする．

2.2 BLS 署名のアルゴリズム

- **鍵生成**：$s \in \mathbb{F}_r$ をランダムにとり署名鍵，$Q := sP_{I'} \in G_{I'}$ を検証鍵とする．

© 2020 Information Processing Society of Japan

- **署名**： $m \in \{0,1\}^*$ をメッセージとして $\sigma := sH_I(m) \in G_I$ とする．
- **検証**： 署名 $\sigma \in G_I$，検証鍵 $Q \in G_{I'}$，メッセージ $m \in \{0,1\}^*$ に対して

$$(I, I') = (1,2) \text{ のとき } e(\sigma, P_2) = e(H_1(m), Q),$$
$$(I, I') = (2,1) \text{ のとき } e(P_1, \sigma) = e(Q, H_2(m))$$

が成り立てば受理，そうでなければ破棄する．

表 1 BLS 署名の群の選び方

方式	署名鍵（秘密鍵）	検証鍵（公開鍵）	署名
添え字 $(1,2)$	$\mathbb{F}_r$	G_2	G_1
添え字 $(2,1)$	$\mathbb{F}_r$	G_1	G_2

添え字のとり方で 2 種類の BLS 署名を構成できる．検証鍵を小さくしたい場合は添え字 $(2,1)$ を，署名を小さくしたい場合は添え字 $(1,2)$ を選ぶ．type-3 ペアリング上の BLS 署名は (G_1, G_2) における co-DHP* が困難で H_i がランダムオラクルのときに安全である [7]．

BLS 署名は検証鍵が $xP_{I'}$，署名が $xH_I(m)$ とどちらも「署名鍵 × 群要素」の形をしていて，乱数を含まない．このため閾値署名や複数の公開鍵や署名の集約が可能となる．

2.3 閾値署名

t-of-n 閾値署名は n 人からなるグループがあり，そのうちの任意の t 人の署名を集めるとマスター検証鍵で受理可能な署名を作成できる．BLS 署名を用いたアルゴリズムは次のとおりである．

- **鍵配付**：グループの各メンバー u_i（$i = 1, ..., n$）に対して相異なる ID $x_1, \cdots, x_n \in \mathbb{F}_q$ を決める．ディーラーが $d := t-1$ 次 $\mathbb{F}_r$ 係数多項式 $f(x)$ をランダムに選ぶ．$s := f(0)$ がマスター署名鍵，$Q := sH_I(m)$ がマスター検証鍵である．ディーラーはマスター検証鍵を公開し，マスター署名鍵は破棄する．
  ディーラーは各メンバー u_i に秘密の通信経路を用いて鍵のシェア $s_i := f(x_i)$ を配付する．ユーザ u_i の署名鍵が s_i，検証鍵が $Q_i := s_i P_{I'}$ である．
- **署名**：ユーザ u_i はメッセージ m に対して BLS 署名 $\sigma_i := s_i H_I(m)$ を作る．
- **復元**：グループのメンバーのうち任意の t 人の集合を S とする．与えられた署名 $\{\sigma_i : i \in S\}$ に対して，それぞれの署名の正当性を確認したあと Shamir の秘密分散の手法を用いて $f(0)H_I(m)$ を復元できる．この署名はマスター検証鍵 Q を用いて受理できる．

DFINITY は決定的に定まるが予測できない乱数（ビーコン）を利用してブロックチェーン上の合意形成アルゴリズムを提案した [11]．ビーコンの構成に閾値署名を利用し，鍵配付は信頼できるディーラーではなく DKG（Distributed key generation）を利用することで，非中央集権的アルゴリズムとなっている．

2.4 集約署名

集約署名とは n 人のグループのそれぞれがした署名を集約し，検証コストを減らす仕組みである．BLS 署名を用いたアルゴリズムは次の通りである．

- **鍵配付**：グループの各メンバー u_i（$i = 1, ..., n$）がそれぞれ BLS 署名の鍵生成を行い，検証鍵 $Q_i := s_i P_{I'}$ を公開する．
- **署名**： メッセージ m に対してそれぞれが署名 $\sigma_i := s_i H_I(m)$ を公開する．
- **集約**：署名と検証鍵をそれぞれ加算する．$\sigma := \sum_i \sigma_i$．$Q := \sum_i Q_i$．集約された (σ, Q) を用いて BLS 署名と同じ検証方法で検証する．

この方法は，Rogue 攻撃に対して安全ではないので，各メンバー u_i が検証鍵に Q_i に対応する署名鍵 s_i を持っていること KOSK（Knowledge of the Secret Key）を証明する必要がある．たとえば Q_i に対して s_i で署名し，それを相手に検証してもらう方法がある [1]．Boneh たちは KOSK 不要な集約署名を提案している [2]．

Harmony は BFT（Byzantine Fault Tolerance）合意形成に集約署名を利用している [16]．Ethereum の Casper と呼ばれる PoS（Proof of Sake）による合意形成プロトコルは，多数の署名の検証が必要である．それを軽減するために集約署名を利用して検証の負荷を下げている [8]．

3. 標準化

128 ビット安全性を持つ曲線として 256 ビットの BN 曲線が使われてきたが Menezes たちの攻撃により安全性が 100〜110 ビットに下がった [13]．そのため 128 ビット安全性を持つ新しいパラメータとして BLS12-381 が提案された [6]．楕円曲線の定義体の位数は 256 ビットから 384 ビットに増えたが巡回群の位数は 256 ビットの範囲に収まる．多数のプロジェクトや環境で利用されることが増えたため，相互運用のためにペアリング曲線と BLS 署名曲線の標準化が IETF で進められている [14][4]．また BLS 署名に使われるハッシュ関数の標準化も進められている [9]．

著者によるペアリングと BLS 署名の実装 mcl/bls[*1] を含め，様々なライブラリがこれらの標準仕様に対応している．

3.1 BLS12-381

BLS12-381 曲線は次のように定義される．

[*1] https://github.com/herumi/bls

$$z = -2^{63} - 2^{62} - 2^{60} - 2^{57} - 2^{48} - 2^{16},$$
$$r = z^4 - z^2 + 1,$$
$$p = (z-1)^2 r/3 + z$$

とする．r は 255 ビット，p は 381 ビットの素数である．E を $y^2 = x^3 + 4$ で定義される $\mathbb{F}_p$ 上の楕円曲線，G_1 を $E(\mathbb{F}_p)$ の位数 r の部分群とする．$\mathbb{F}_p$ の拡大体を次の方法で定義する：

$$\mathbb{F}_{p^2} = \mathbb{F}_p[u]/(u^2 + 1),$$
$$\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - u - 1),$$
$$\mathbb{F}_{p^{12}} = \mathbb{F}_{p^6}[w]/(w^2 - v).$$

E' を $y^2 = x^3 + 4(u+1)$ で定義される $\mathbb{F}_{p^2}$ 上の楕円曲線，G_2 を $E'(\mathbb{F}_{p^2})$ 上の位数 r の部分群とする．$\varphi : E \ni (x, y) \mapsto (x', y') \in E'$ を

$$\varphi(x, y) = (w^2 x, w^3 y)$$

とすると $\varphi^{-1} : G_2 \to E(\mathbb{F}_{p^{12}})$ は中への同型写像となる．

$R \in E(\mathbb{F}_{p^{12}})$，整数 n に対して $f_{n,R}$ を因子が $\mathrm{div}(f_{n,R}) = n(R) - (nR) - (n-1)(O)$ となる有理関数とする．$P \in G_1$, $Q \in G_2$ に対してペアリングを

$$e(P, Q) = f_{z,\varphi^{-1}(Q)}(P)^{(p^{12}-1)/4}$$

とする．

3.2 G_i へのハッシュ関数

文字列 $m \in \{0,1\}^*$ から G_i へのハッシュ関数はいくつか考えられている．よく知られた例としてカウンタ c と文字列 m に対して $\mathbb{F}_p$ へのハッシュ値をとったあと，それを G_i の x 座標とみなし，楕円曲線の定義方程式を満たすまでカウンタ c の値を 1 ずつ増やす方法がある．しかし定数時間アルゴリズムではない．現在標準化が進められている方法は [15], [10], [17] などを元にした決定的，定数時間アルゴリズムである．mcl は BLS12-381 曲線の G_1, G_2 へのハッシュ関数をサポートしている．

4. mcl の設計と実装

4.1 mcl の設計

著者によるペアリングの実装ライブラリ mcl[*2]の設計について述べる．mcl はもともと Intel の x86-64（以下 x64 と表記）CPU 専用であった ate-pairing[*3]を元に，高速性を保ちつつ汎用性も高めて再設計したライブラリである．利用言語は C++であり，以下の特長を持つ．

1. Windows, Linux, macOS, Android, iPhone, Node.js などの環境をサポートする．
2. x64, aarch64, ARM, MIPS などの CPU や WebAssembly をサポートする．
3. C++, C, Go, Rust, JavaScript, C#などの言語で利用できる．
4. 組み込み環境でも利用しやすいよう C++の例外，実行時型識別 RTTI や malloc や new などのメモリ管理機構，および標準入出力ライブラリを利用しない．

4 番目の特長は，iPhone や Android だけでなく, 制限された IoT 機器でも動作してほしいという要望に対応するためである．gcc や clang で`-std=c++03`, `-fno-exceptions`, `-fno-rtti`, `-fno-threadsafe-statics`, `-nostdinc++` といったオプションでビルドできる．たとえば `ARM/Aarch64` ベースの `Raspberry Pi` や `ConsenSys` の `MIPS` ベースハードウェアウォレット `Latice1` 上でも動作する[*4]．

ペアリングライブラリの高速化のためにはアセンブリ言語レベルでの最適化が必要となる．`mcl` は複数の方法で素体の演算の実装をしている．

1．`JIT` ライブラリ `Xbyak` を用いた `x64` 専用ルーチン
2．多倍長演算ライブラリ `GMP` を用いたルーチン
3．`LLVM` を用いたルーチン
4．`C++`を用いた汎用ルーチン

以下それぞれの実装について解説する．

4.2 Xbyak を用いた x64 専用ルーチン

`Xbyak`[*5]は `x64` アセンブリ言語を実行時に生成可能な `C++` のヘッダライブラリである．一般的な `C++`コンパイラの独自拡張である `intrinsic` 関数やインラインアセンブラ構文と異なり，構文はあくまでも通常の `C++`の範囲である．`C++`と連携しやすく，実行時の `CPU` に応じた拡張命令を利用するコードを記述しやすくするために開発した．`x64` 環境では `Xbyak` を用いた素体の演算を利用するのが最も高速である．

たとえば `n` 個の `uint64_t` 型配列 `px` と `py` の多倍長整数としての加算結果を `pz` で示されるポインタに保存するコードは `src 1` のように記述できる．ここで `RegExp` は `rax` や `rsp+64` といった `ModRM/SIB` で示される型，`Reg64` は 64 ビット汎用レジスタを指す．実行時に与えられた `n` に応じてループ展開されたニーモニックに対応する機械語が生成される（`src 2`）．

src 1

```
1 void gen_raw_add(
2   const RegExp& pz,
3   const RegExp& px,
4   const RegExp& py,
```

[*2] https://github.com/herumi/mcl
[*3] https://github.com/herumi/ate-pairing
[*4] https://gridplus.io/lattice
[*5] https://github.com/herumi/xbyak

```
  const Reg64& t,
  int n)
{
  mov(t, ptr [px]);
  add(t, ptr [py]);
  mov(ptr [pz], t);
  for (int i = 1; i < n; i++) {
    mov(t, ptr [px + i * 8]);
    adc(t, ptr [py + i * 8]);
    mov(ptr [pz + i * 8], t);
  }
}
```

src 2

```
gen_raw_add(rdi, rsi, rdx, rax, 3)の実行例
mov rax, [rsi]
add rax, [rdx]
mov [rdi], rax
mov rax, [rsi + 8]
adc rax, [rdx + 8]
mov [rdi + 8], rax
mov rax, [rsi + 16]
adc rax, [rdx + 16]
mov [rdi + 16], rax
```

4.3 多倍長演算ライブラリ GMP を用いたルーチン

GMP（The GNU Multiple Precision Arithmetic Library）[*6]は x86，x64，ARM，Aarch64 などさまざまな CPU 向けに最適化された多倍長演算関数を提供するライブラリである．mcl では mpn_で始まる GMP の中でも最も高速な低レベル関数[*7]を利用している．動的メモリ確保は行われない．最近は他言語との連携のしやすさのため，GMP を使わないモードでビルドされることが多い．

4.4 LLVM を用いたルーチン

LLVM[*8]とは様々なプログラミング言語の開発に使われることを想定したコンパイラやツールチェインの開発基盤の総称である．LLVM は独自の仮想機械とその表現（ビットコード）を持ち，ビットコードから実際に動作する環境の CPU に向けた機械語を生成する．2020 年 8 月現在，ARM64(aarch64)，ARM，MIPS，MIPS64，PPC，PPC64，SPARC，x86/x64，RISC-V，WebAssembly などに対応する．

LLVM は任意の長さの固定長の整数型を持つ．構文は全ての変数に型を指定し，再代入ができない静的単一代入形（SSA）である．たとえば 192 ビット整数 x，y を足して z に代入する（繰り上がりを無視した）関数 add は src 3 のように記述する．

src 3

```
define void @add(
  i192* %pz,i192* %px,i192* %py) {
  %x = load i192, i192* %px
  %y = load i192, i192* %py
  %z = add i192 %x, %y
  store i192 %z, i192* %pz
  ret void
}
```

乗算命令はターゲット CPU の命令サイズを超える乗算命令はエラーになるため自分で実装する必要がある．

4.5 LLVM を出力する DSL

前節の例のように LLVM では全ての変数と命令に型を記す必要がある．変数の型情報は不変であるため，i192* %pz と変数を宣言した時点で load や add などに付随する i192 は推論で確定するため冗長な表記である．

また素体のサイズに応じて 256 ビット，384 ビットなどさまざまな加算，乗算関数を実装する必要がある．これらの実装で SSA で記述する必要があり変数の使い回しができず，必要な変数が増えて可読性が低下し，メンテナンスコストが高い．

そのため mcl ではレジスタサイズを抽象化し，複数代入可能な C++の DSL を開発した．

DSL で記述されたコードを実行すると対応する LLVM ビットコードが生成される．DSL は変数の複数代入をサポートし，DSL を用いた記述した固定多倍長乗算や Montgomery 乗算から，各素体サイズや CPU のワードサイズ（32 or 64）に応じた LLVM ビットコードを生成する．

MIPS への対応も，Makefile でクロスコンパイル方法の指定や clang の CPU の taget 指定を追加するだけで完了した．

4.6 C++を用いた汎用ルーチン

標準的な C++03 でコンパイルできる．GMP や LLVM に対応していない環境でも動作するために用意されている．

4.7 ベンチマーク

表 2 で x64[*9]，Aarch64[*10]における Xbyak（x64 環境のみ），LLVM，GMP を用いたときのペアリングの速度のベンチマークを計測した．WebAssembly の数値は Emscripten

[*6] https://gmplib.org/
[*7] https://gmplib.org/manual/Low_002dlevel-Functions
[*8] http://llvm.org/
[*9] Intel Core i7-8700 3.2GHz, clang 8.0.1
[*10] Arm v8-A, clang-10.0.1

1.39.11[*11]でコンパイルしたものを Node.js 12.18.3 で実行したものである．WebAssembly は現在，64 ビット ×64 ビット→ 128 ビットの乗算命令を持っていないため，32 ビット乗算命令を組み合わせて実行しなければならない．そのためかなり遅い．

表 2　BLS12-381 上のペアリング（msec）

mode	x64	Aarch64
Xbyak	0.598	-
LLVM	0.694	3.466
GMP	1.253	6.393
WebAssembly	6.359	20.16

GMP の npm_関数を使った場合に比べて LLVM の DSL で記述したバージョンは x64，Aarch64 環境共に約 1.8 倍高速となった．x64 や Aarch64 専用アセンブリコードを直接書かなくてもこのような高速化を達成したのは興味深い．

5. 言語バインディング

言語バインディングとは C/C++以外の言語から利用しやすいようにインタフェースを提供することである．現在 mcl は Go，C#，Rust，Node.js 用の言語バインディングを用意している．C API を提供しているので他言語からの利用も難しくはないだろう．

以下，いくつかの言語バインディング開発時に発生した問題点の解決方法を紹介する．

5.1 Go

Go[*12]は Google が開発しているプログラミング言語である．Go から C の関数を呼び出すためには Cgo[*13]を利用する．GitHub などのオープンソース開発プラットフォームに公開されたライブラリを Go get コマンドで利用できるためにはコンパイル済みライブラリを同伴しなくてはならない．そのため https://github.com/herumi/mcl/bls-eth-go-binary でそれらを含めたパッケージを公開している．

5.1.1 環境依存のディレクトリ指定方法

Windows（mingw64），Linux for x64/Aarch64，macOS など様々な環境用のコンパイル済み static ライブラリを用意する．Go は GOOS，GOARCH という環境変数に OS 種別（linux，darwin，windows など），CPU 種別（amd64，arm64）などが設定される．したがってこれらの環境変数に応じたディレクトリを作り，その中に static ライブラリ bls384_256.a を配置した．

```
lib/darwin/amd64/libbls384_256.a
lib/android/arm64-v8a/libbls384_256.a
lib/android/x86_64/libbls384_256.a
lib/android/armeabi-v7a/libbls384_256.a
lib/linux/arm64/libbls384_256.a
lib/linux/amd64/libbls384_256.a
lib/windows/amd64/libbls384_256.a
```

ライブラリの位置を Go に伝えるのは Cgo を使って次のように記述する．

```
package bls

/*
#cgo CFLAGS:-I${SRCDIR}./include -DBLS_ETH
#cgo LDFLAGS:-lbls384_256 -lstdc++ -lm
#cgo ios LDFLAGS:-L${SRCDIR}/lib/ios
#cgo linux,amd64 LDFLAGS:-L${SRCDIR}/lib/linux/amd64
#cgo linux,arm64 LDFLAGS:-L${SRCDIR}/lib/linux/arm64
#cgo windows,amd64 LDFLAGS:-L${SRCDIR}/lib/windows/amd64
...
#include <mcl/bn_c384_256.h>
#include <bls/bls.h>
*/
import "C"
```

「#cgo <os>,<cpu> LDFLAGS:...」の<os>や<cpu>が GOOS，GOARCH にマッチしたときだけ処理される．それ以外の部分は全ての環境で処理される．

5.1.2 配列の共用

mcl はライブラリ内で動的メモリ確保を行わない．可変長の配列を扱う場合は，ライブラリ利用者が確保し，そのバイト列の先頭アドレスを受け渡しする方法をとっている．これは，ガベージコレクション GC を持っていたり，デストラクタが無い言語では言語特有のメモリ管理機構を扱う必要が多く，扱いづらいことが多いためである．

Go，C#，Rust などの言語は C の構造体に対応するものを持ち，その配列はメモリ上連続配置される．そのためこの方法で効率よく C の API を呼び出せる．

mcl の過去のバージョンでは各オブジェクトのポインタの配列を扱う API も用意し，Go で利用していた．しかし Go 1.5 以降では GC の仕様変更により禁止された．そのためそのような API 設計は汎用的ではないと判断し，現在ではサポートしていない．

5.2 WebAssembly

WebAssembly は最近のモダンブラウザでは標準で利用

[*11] https://emscripten.org/
[*12] https://golang.org/
[*13] https://godoc.org/github.com/gophersjp/go/src/cmd/cgo

できる．また JavaScript 実行環境 Node.js でも利用できる．mcl はどちらの環境にも対応している．

5.2.1 暗号学的擬似乱数生成

秘密鍵の生成で乱数を使うとき，Linux や Windows 環境ではそれぞれ専用の暗号学的擬似乱数生成関数を利用する．WebAssembly 環境自体にはそのような機能は存在しない．したがって，ブラウザや Node.js それぞれに用意された暗号学的擬似乱数生成関数を用いて生成した乱数を mcl に渡す方法をとっている．

5.2.2 配列の受け渡し

JavaScript が管理する配列のメモリと WebAssembly で扱うメモリは分離されている．

そのため Go や C#などのように配列の共用が難しい．一般的な C API ではオブジェクトの生成関数（create）と破壊関数（destroy）を用意してユーザがそれらを確実に呼び出すことを要求する方法をとることがある．

src 4

```
1 struct X *x = X_create(); # malloc
2 # call APIs for x
3 X_destroy(x); # free
```

このような API を用意すると JavaScript でも同様のコードを記述する必要がある．JavaScript プログラマはそのようなオブジェクトの生存期間を意識したコードを書くことは少ないため，mcl ではユーザの負担の少ない方法をとることにした．

関数呼び出しの中で JavaScript が管理するメモリを WebAssembly で扱うメモリにコピーし，関数呼び出し終了時にその領域を破棄する．src 5 では署名の加算（Signature.add）の実装内部で現在のオブジェクト（this）と加算する署名（y）のそれぞれの JavaScript のメモリを WebAssembly で管理するメモリにコピーする（3，4 行目）．実際の C API を呼んだ（5 行目）あと，不要になったメモリを開放し（6 行目），結果を元のオブジェクトに反映している（7 行目）．

src 5

```
1 # ライブラリ内部の実装概略
2 Signature.add(y) {
3   const xPos = this._allocAndCopy()
4   const yPos = y._allocAndCopy()
5   mod.mcl_addSignature(xPos, yPos)
6   _free(yPos)
7   this._saveAndFree(xPos)
8 }
```

この方法は関数呼び出しの負荷は大きくなるが，オブジェクトの寿命の管理が通常の JavaScript のクラスト同じためライブラリ利用者は扱いやすい（src 6）．

src 6

```
1 # ライブラリ利用者の例
2 let sig1 = new bls.Signature()
3 let sig2 = new bls.Signature()
4 # sig1, sig2 への操作
5 # ...
6 sig1.add(sig2) # sig1 に sig2 を足す
```

6. まとめ

現在，BLS12-381 曲線やハッシュ関数，BLS 署名は標準化の議論が進行中であり，様々なプロジェクトで利用されている．著者が開発中の BLS 署名ライブラリ mcl/bls もそれらの仕様に準拠している．mcl/bls の設計方針や，高速化の手法，他言語からの利用方法について紹介した．

参考文献

[1] A. Boldyreva. Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In *PKC*, 2003.

[2] D. Boneh, M. Drijvers, and G. Neven. Compact multi-signatures for smaller blockchains. Cryptology ePrint Archive, Report 2018/483, 2018. https://eprint.iacr.org/2018/483.

[3] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT*, 2003.

[4] D. Boneh, S. Gorbunov, R. Wahby, H. Wee, and Z. Zhang. draft-irtf-cfrg-bls-signature-02, 2020. https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-02.

[5] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *ASIACRYPT*. Springer Berlin Heidelberg, 2001.

[6] S. Bowe. BLS12-381: New zk-SNARK elliptic curve construction, 2017. https://electriccoin.co/blog/new-snark-curve/.

[7] S. Chatterjee, D. Hankerson, E. Knapp, and A. Menezes. Comparing two pairing-based aggregate signature schemes. In *Des. Codes Cryptogr*, 2010.

[8] J. Drake. Pragmatic signature aggregation with BLS, 2018. https://ethresear.ch/t/pragmatic-signature-aggregation-with-bls/2105.

[9] A. Faz-Hernandez, S. Scott, N. Sullivan, R. Wahby, and C. Wood. Hashing to elliptic curves, 2020. https://tools.ietf.org/html/draft-irtf-cfrg-hash-to-curve-09.

[10] P.-A. Fouque and M. Tibouchi. Indifferentiable hashing to barreto--naehrig curves. In *LATINCRYPT*, 2012.

[11] T. Hanke, M. Movahedi, and D. Williams. DFINITY technology overview series consensus system, 2018. https://arxiv.org/pdf/1805.04548.

[12] R. Lan. BLS-based practical BFT consensus, 2018. https://talk.harmony.one/t/bls-based-practical-bft-consensus/131.

[13] A. Menezes, P. Sarkar, and S. Singh. Challenges with assessing the impact of NFS advances on the security of pairing-based cryptography. In *Mycrypt*, 2016.

[14] Y. Sakemi, T. Kobayashi, T. Saito, and R. Wahby. Pairing-friendly curves, 2020. https://tools.ietf.org/html/draft-irtf-cfrg-pairing-friendly-curves-07.

[15] A. Shallue and C. E. van de Woestijne. Construction of rational points on elliptic curves over finite fields. In *ANTS*, 2006.

[16] H. Team. Harmony technical whitepaper, 2019. https://harmony.one/whitepaper.

[17] R. S. Wahby and D. Boneh. Fast and simple constant-time hashing to the bls12-381 elliptic curve. *CHES*, 2019.