Hashing to Elliptic Curvesの実装調査および安全な利用方法の考察

石橋 拓哉[1,a)]　国井 裕樹[1]

概要：RFC 9380「Hashing to Elliptic Curves」は，任意の文字列を楕円曲線上の点に一意かつ安全にマッピングするアルゴリズムを定めた標準規格である．パスワードやパスフレーズ等で利用している文字列をHashing to Elliptic Curves によって楕円曲線上の点に安全にマッピングすれば，公開鍵暗号や電子署名での鍵ペアとして利用することも可能となる．一方で，パスワードを利用する場合には，既存の攻撃である辞書攻撃等への耐性について検討する必要がある．本研究では，上記の目的から RFC 9380 準拠の実装を提供する主要ライブラリの調査および，Certificate Transparency や仮想通貨トランザクションに含まれる公開鍵の抽出・解析を行う．これらの調査から RFC 9380 に基づいて生成した公開鍵を実際のサービスで安全に運用するための方法と留意点を検討する．

キーワード：Hashing to Elliptic Curves, RFC9380，楕円曲線，公開鍵

A study of “Hashing to Elliptic Curves” implementation and its usage

Takuya Ishibashi[1,a)]　Hiroki Kunii[1]

Abstract: RFC 9380 “Hashing to Elliptic Curves,” defines an algorithm that securely and uniquely maps arbitrary character strings to points on an elliptic curve. When strings used in passwords or passphrases are securely mapped to points on an elliptic curve using this algorithm, they can be used as key pairs in public-key cryptography and digital signatures. However, when considering passwords, it is necessary to consider resistance to existing attacks, such as dictionary attacks. To this end, this study examines major libraries that provide RFC 9380-compliant implementations and analyzes public keys included in Certificate Transparency and Cryptocurrency transactions. Based on these findings, we will examine methods and considerations for securely operating public keys generated based on RFC 9380 in real-world applications.

1. はじめに

楕円曲線暗号（Elliptic Curve Cryptography: ECC）の応用範囲が広がる中で，任意の文字列を楕円曲線上の点に安全かつ一意にマッピングする技術への関心が高まっている．RFC9380「Hashing to Elliptic Curves（HEC）」[1]は，このようなニーズに応える標準規格として策定され，文字列を楕円曲線上の点に変換するための安全なアルゴリズムを定義している．

この技術を活用することで，従来のパスワードやパスフレーズなどを用いた文字列ベースの認証情報を，公開鍵暗号や電子署名における鍵ペアとしてより安全に利用することが可能となると考えられる．しかしながら，パスワードのような人間が記憶・入力する文字列を対象とする場合，辞書攻撃などの既存の攻撃手法への耐性について慎重な検討が必要である．

本研究では，HEC とブレインウォレットとの比較を通じて，HEC を使用する場合の安全性と実装上の利点を明らかにするとともに，RFC9380 に準拠した実装を有するライブラリの状況を調査する．また，Certificate Transparency（CT）ログやビットコイントランザクションに含まれる公開鍵を対象に，HEC を用いた辞書攻撃の可能性について

[1] セコム株式会社 IS 研究所
Intelligent Systems Laboratory, SECOM CO.,LTD.
[a)] taku-ishibashi@secom.co.jp

予備調査を行う．これらの調査・考察を通じて，HEC を実サービスにおいて安全に運用するための方法と留意点を明らかにすることを本研究の目的とする．

2. 背景・準備

本節では，研究背景と既存研究について説明する．次に，本研究に用いる技術や手法に関する概説・比較を行う．

2.1 研究背景

楕円曲線暗号において，ユーザのパスワードから秘密鍵・公開鍵を導出する方式は利便性が高い一方で，パスワードのエントロピーが低い場合にオフライン辞書攻撃に脆弱になるという問題を抱えている．例えばビットコインの「ブレインウォレット（Brain Wallet）」では，ユーザがパスフレーズをハッシュして秘密鍵を生成する手法が用いられたが，攻撃者はブロックチェーン上の公開鍵（アドレス）を収集し，流出パスワードリストを用いて総当たりで秘密鍵を再現できることが実証されている．Vasek らの研究 [1] では，約 3000 億語に及ぶワードリストを用いて 2011 年から 2015 年のブロックチェーンを分析した結果，884 件（約 10 万ドル相当）のブレインウォレットが確認され，そのほとんどが短時間で資金を抜き取られていたことが報告されている．

パスワード由来の鍵生成における攻撃耐性を高めるために，近年はメモリハード関数などを用いた Key Derivation Function（KDF）の導入が推奨されている．Tippe らの研究 [2] では，KDF に用いられるメモリハード関数の一種である Argon2（RFC9106[3]）を評価している．その結果，OWASP 推奨設定（46MiB）などの比較的強いパラメータを用いることで，強めのパスワードに対する耐性は向上することが示されている．一方で，「RockYou」に代表される弱パスワード分布に対しては，Argon2 と従来の SHA-256 のいずれでも，依然として高い割合（約 96.9〜99.8%）で突破されるという結果も示されている．また実装調査では，多くのソフトウェアが推奨より弱いパラメータを使用している例が散見され，運用面での注意が必要であることも指摘されている．

2.2 ブレインウォレットの概要

ブレインウォレットは，ユーザーが覚えているパスフレーズからハッシュ値を生成して直接暗号通貨の秘密鍵を作り，その秘密鍵でアドレスを管理する方式である．物理的なデバイスや紙が不要で，記憶だけでウォレットを復元できる点が利点である．しかしながら，人が覚えるパスフレーズはエントロピーが低い点や単一のハッシュだけを用いる場合には非常に早く計算ができてしまうため，辞書攻撃や総当たり攻撃などに対する耐性がない場合が多い．そのため，現在では BIP-39[4] などの安全な方法を用いてウォレットを生成することが推奨されている．以下に単純なブレインウォレットの作成手順を示す．

手順 1　パラメータを決定する
楕円曲線暗号のパラメータとして，位数 n，生成元 G を用いる．

手順 2　ユーザがパスワードを決める
ユーザはパスワードとなる任意長の入力メッセージ $password$ を選択する．このとき $password$ は長いメッセージであることが望まれる．

手順 3　パスワードをハッシュして秘密鍵を生成する
sha-256 などのハッシュ関数 H を用いて変換した値 $h = H(password)$ を計算し，スカラー値にした h から秘密鍵となる $sk = h \bmod n$ を求める．この時，KDF として Argon2 などを用いるとさらに安全性が高くなる．

手順 4　秘密鍵 sk から公開鍵を計算する
公開鍵 $pk = sk * G$ を求める．この時公開鍵 pk は楕円曲線上の座標の値となる．

手順 5　公開鍵 pk からウォレットのアドレスを計算する
ビットコインでは公開鍵を SHA256 → RIPEMD160 → Base58Check の順に変換を行い，ビットコインアドレスを生成する．

2.3 Hashing to Elliptic Curves の概要

RFC9380「Hashing to Elliptic Curves（HEC）」[5] は任意のバイト列を楕円曲線上の点へ一貫して写像するための総合的な手順と推奨実装を規定する文書で，Crypto Forum Research Group (CFRG) の合意に基づいて IRTF ストリームで 2023 年に公開された．RFC9380 の目的は，用途ごとに散在していた「ハッシュ→曲線（hash-to-curve）」アルゴリズムを統一インタフェースで整理し，定数時間実装，アプリケーション毎のドメイン分離やコファクタ処理など実装上の細部を明確にすることにある．

以下に HEC の処理手順を示す．

手順 1　パラメータを決定する
楕円曲線暗号のパラメータとして，位数 n，生成元 G を用いる．また，ドメイン分離を行うための文字列である Domain Separation Tag（DST）を決定する．

手順 2　入力メッセージを決める
マッピングの入力となる任意長メッセージ m（例: パスワード）を用意する．m は事前に強力な KDF で伸張することが推奨される．

手順 3　入力メッセージから必要長バイト列を生成する
必要長の均一なバイト列を生成する `expand_message` を用いて m と DST からバイト列 b を生成する．入力に DST を用いているため異なる用途間での衝突が防止されている．

手順 4　バイト列を有限体上の点に変換する

バイト列を有限体上の要素に写像する hash_to_field を用いて，バイト列 b を入力に，$u \in \mathbb{F}_p$ を得る．

手順 5　有限体上の要素を曲線上の点にマッピングする

有限体上の要素を入力に，一意に曲線上の点にマッピングする map_to_curve を用いて，u を入力に曲線上の点 P に写像する．必要に応じてコファクタ処理により正しい部分群へ移す．

3 つの関数，expand_message，hash_to_field，map_to_curve を部品化することにより，それぞれの関数が安全に動作するよう設計されている．そのため，P-256，P-384，edwards448，curve25519/edwards25519，secp256k1，BLS12-381 など様々な曲線に対応できる．写像には Simplified SWU(SSWU) や Elligator 2 といった決定論的マッピングが採用されている．確率的手法は，タイミング攻撃などのサイドチャネル脆弱性を招くため非推奨である．

実装時には定数時間処理，expand_message を用いるなどの安全性要件遵守やコファクタ処理および例外点の正しい扱いが推奨されている．その中には，DST の使用による明確なドメイン分離も含まれており，DST を用いることによって異なる用途間でのハッシュ衝突を防ぎつつ，相互運用性を担保する実装が可能となっている．さらには，入力バイト列とソルトを入力とする KDF の使用も併せて推奨されている．これは，KDF を用いることにより辞書攻撃に対してより強固な耐性を持たせることが可能となるためである．ライブラリなどの実装も多数存在し，実サービスで利用する場合は RFC の指針に従いつつ曲線固有の最適化とサイドチャネル対策を施すことが推奨される．

2.4　HEC とブレインウォレットの比較

本節では，RFC9380 に準拠した Hashing to Elliptic Curves（HEC）と，従来のブレインウォレット方式を比較し，それぞれの技術的特徴や安全性について比較する．特に，実装時のセキュリティ対策，辞書攻撃や総当たり攻撃への耐性といった観点から，両者の違いを明確にすることを目的とする．

(1) **プロトコルの安全性**

HEC では，曲線上の大きな素数位数を持つ部分群に対して，点を均等に分布させる生成手法が採用されており，さらにコファクタ処理の手順も示されている．ブレインウォレットでは，パスワードからハッシュ値を生成する形による単純変換で実装している場合が多い．そのため，安全性が確保されていない可能性が存在する．

(2) **実装上の安全性**

HEC では DST を用いた用途ごとのドメイン分離や決定論的マッピング手法による定数時間実装が採用されており，サイドチャネル攻撃やクロスプロトコル攻撃に対する耐性を有する設計指針が示されている．また，テストベクターの提供がされており，実装ミスの発見も容易となっている．ブレインウォレットではドメイン分離や定数時間実装などは考慮されておらず，サイドチャネル攻撃やクロスプロトコル攻撃への対策がされていない実装が存在する．

(3) **辞書攻撃・総当たり耐性**

HEC では外部の KDF やソルトの使用を推奨しており，入力がエントロピーの低いバイト列だった場合においても，辞書攻撃・総当たり攻撃への耐性を持たせた運用を想定している．ブレインウォレットでは，KDF を用いるなどの対策が行われていない実装が存在する．そのため，辞書攻撃・総当たり攻撃への耐性を持たず，過去に辞書攻撃による仮想通貨流出が発生している．

表 1 にブレインウォレットと HEC の比較のまとめを示す．

3.　HEC における辞書攻撃の予備調査

本節では，HEC を用いた辞書攻撃が実際に成立し得るかを検証するために実施した，予備調査の概説とその結果について述べる．楕円曲線暗号を用いた公開鍵が多数記録されているオープンリポジトリとして，CT ログおよびビットコインのトランザクションが存在する．そこで，これらの CT ログおよびビットコインのトランザクションに記載されている公開鍵を対象に，予備的な調査を実施した．調査においては，HEC への入力文字列として，Github 上で公開されているパスワードリスト「Rockyou75」[*1] に含まれるパスワードを使用した．これらの文字列を HEC を用いて楕円曲線上の点にマッピングし，生成された値が既存の公開鍵と一致するかどうかを調査した．

3.1　調査対象データ

Certificate Transparency（CT）は，SSL/TLS 証明書の不正発行を防ぐための仕組みであり，証明書の発行履歴を公開ログに記録することで透明性を確保している．証明書発行機関（CA）が発行した証明書は CT ログに登録され，Signed Certificate Timestamp（SCT）を通じて検証可能となる．これにより，なりすましや中間者攻撃のリスクを軽減し，Web の安全性を向上させている．CT は RFC6962[6] および，RFC9162[7] で標準化されている．

ビットコインのトランザクションは，ネットワーク上でビットコインの所有権を移転するための記録である [8]．各トランザクションは「入力（Input）」と「出力（Output）」から構成され，入力には送金元の情報，出力には送金先のアドレスと金額が含まれる．送金元の所有者は秘密鍵を用

[*1] https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/rockyou-75.txt

表 1　ブレインウォレットと HEC の比較
Table 1　Comparison between Brain Wallets and HEC.

	ブレインウォレット	HEC（RFC9380）
プロトコルの安全性	対策なし	対策あり（コファクタ処理等）
実装上の安全性	考慮されていない	ドメイン分離・マッピング手法により各種攻撃に耐性あり
辞書攻撃・総当たりへの耐性	入力のエントロピーに依存	KDF（推奨）+`expand_message` で対策

表 2　調査した CT ログ，ビットコイントランザクション
Table 2　Survey targets and their results.

	nimbus2025 (Cloudflare)	argon2025h2 (Google)	トランザクション（ビットコイン）
種別	CT	CT	Transaction
件数	31,830	88,928	2,000,000
調査結果	衝突なし	衝突なし	衝突なし

いてトランザクションに署名し，対応する公開鍵によって正当性が検証される．この公開鍵はトランザクションの一部としてブロックチェーン上に記録され，誰でも閲覧可能である．公開鍵はセキュリティや匿名性の観点から重要であり，仮想通貨の分析や監視にも活用されている．

3.2　調査結果

今回の調査における実装条件を以下に記載する．

- 言語：JavaScript(noble-curves 使用)
- パスワードリスト：RockYou75 (59,186 件)
- 曲線：P-256，secp256k1，edwards25519，edwards448
- DST：RFC9380 にて各曲線ごとに用いられている値
- KDF：使用していない

表 2 は調査対象とした CT ログとビットコイントランザクションの一覧および，パスワードリストの値を HEC を用いて楕円曲線上にマッピングした値との調査結果である．CT ログは現在使用されている SSL/TLS サーバ証明書を取得し，さらに公開鍵に楕円曲線を用いているものを抽出した．ビットコインのトランザクションに関しては，100 万件を 2 回に分けて取得したものの中から公開鍵を抽出し調査を行った．

調査の結果取得した公開鍵全てにおいて，パスワードリストから変換した楕円曲線上の点座標が一致するものは発見されなかった．

4.　HEC に関する実装調査・考察

本節では，RFC9380 に準拠した HEC の実装状況と，それに基づく安全な運用に関する考察を行う．具体的には，主要なオープンソースライブラリにおける HEC の実装内容を調査し，DST や KDF の取り扱い状況を確認することで，実装上の課題や改善点を明らかにする．また，第 3 節で行った予備調査の結果を踏まえ，HEC を実サービスで安全に運用するために重要となる留意点について検討する．

4.1　HEC を実装したライブラリの調査

HEC では DST をアプリケーション・サービス毎に固有にすることや，KDF を用いることが推奨されている．本節では，RFC9380 に準拠した実装を有するライブラリを対象に，DST・KDF の推奨事項への対応に関する調査を行う．以下に今回の調査対象とした RFC9380 準拠の実装を持つライブラリを記す．

- **hash2curve(Go)** [9]
- **bls12381rs(Go)** [10]
- **h2c-rust-ref(Rust)** [11]
- **bls12_381(Rust)** [12]
- **noble-curves(JavaScript)** [13]
- **py_ecc(Python)** [14]
- **mcl(C++)** [15]
- **BoringSSL(C)** [16]
- **CIRCL(Go)** [17]

上記ライブラリの内，hash2curve，bls12381rs，h2c-rust-ref および，bls12_381 はそれぞれ HEC の実装をメインとして公開されている OSS ライブラリである．noble-curves，py_ecc，mcl，BoringSSL および，CIRCL はそれぞれ楕円曲線やペアリングを用いた演算を行う OSS ライブラリであり，その中の一部として HEC の実装が準備されている．上記のライブラリを調査した結果を表 3 に記す．

調査したライブラリはそれぞれ扱える曲線が異なっている．その中でも JavaScript 用のライブラリである，noble-curves は secp256k1, ed25519, NIST 曲線, bls12-381, bn254 等の幅広い曲線に対応しており，今回調査したライブラリの中で一番対応する曲線数が多いものとなっていた．また HEC の演算に用いられる DST に関しては，調査した全てのライブラリにおいて RFC に記載されているサンプル用の文字列を用いた実装がされていた．これはどのライブラリも，RFC に記載されているテストベクターを検証するための実装となっているからである．さらに KDF に関しても，実装に組み込まれているライブラリは見当たらなかった．しかし，noble-curves[13] においては一部の KDF の実装は用意がされており，HEC の演算に直接組み込まれてはいないものの，KDF を使用することを推奨する旨が記載されていた．

表 3　HEC 実装のあるライブラリ一覧
Table 3　List of libraries with HEC implementation.

ライブラリ	言語	対応曲線
hash2curve	Go	BLS12-381，NIST 曲線
bls12381rs	Go	BLS12-381
h2c-rust-ref	Rust	BLS12-381，NIST 曲線
bls12_381	Rust	BLS12-381
noble-curve	JavaScript	NIST 曲線，secp256k1，edwards 等
py_ecc	Python	BLS12-381
mcl	C ++	BN 系，BLS12-381
BoringSSL	C	NIST 曲線
CIRCL	Go	NIST 曲線

4.2　HEC における予備調査に関する考察

本節では，第 3 節で行った予備調査の結果に対する考察を行う．

(1) **比較的新しい標準化であること**

HEC は，2018 年 3 月にインターネットドラフトとして初めて提案され，2023 年 8 月に RFC9380 として正式に標準化された技術である．RFC としての公開からまだ約 2 年しか経過しておらず，実装の普及や実サービスへの適用は発展途上にあると考えられる．そのため，今回の調査において，既存の公開鍵と一致するものが確認されなかったのは，HEC の普及が十分に進んでいないことが一因である可能性がある．

(2) **使用している DST が RFC9380 内で使われているものであること**

今回の調査では，RFC9380 のドキュメント内で使用されている DST を流用して処理を行っている．DST はサービス・アプリケーション毎に変えるものであり，実際にサービスに用いられている DST を調査し処理に用いることにより，一致する公開鍵が発見できる可能性がある．

(3) **KDF を用いた演算処理を行っていないこと**

HEC では文字列から直接ハッシュ計算を行わずに，一度 KDF を用いた演算を行った後にハッシュ演算を行うことを推奨している．ただし，今回の実験では KDF を用いた演算を行わずに処理を行っているため，KDF を用いた場合に一致する公開鍵が発見できる可能性がある．

(4) **取得した公開鍵の個数・パスワードリストの母数が少ないこと**

今回の調査では CT ログ・トランザクションを併せて 200 万件強と，7 万件弱のパスワードリストを用いて調査を行った．しかしながら，公開鍵・パスワードリストともに調査範囲を広げて母数を増やすことにより一致する鍵の発見に至る可能性が高くなると考えられる．

4.3　HEC における DST の値に関する考察

DST は同じハッシュ処理を異なる用途・プロトコルで衝突なく安全に再利用するためのタグであり，これを用いることにより異なるプロトコル・サービス間での出力が独立な値となり安全性が保たれるようになっている．しかしながら，本節で行った調査の結果，HEC を実装しているライブラリでは，全て RFC9380 内で使用されている DST をそのまま使用する形で実装がされていることが判明した．実際のサービスで HEC を使用する場合には，HEC を利用するサービス提供者等が DST の値をサービス毎などに変更することを徹底する必要がある．

4.4　HEC における KDF の利用に関する考察

HEC では KDF を用いることにより，比較的エントロピーの低いパスワードなどを入力に取ったとしても，ソルトと共に演算した値を入力値として用いることにより安全に使用することが可能となる．しかしながら，今回調査したライブラリでは KDF を用いて変換した値を HEC の入力としているライブラリは確認できなかった．noble-curves では KDF の用意はされていたものの，実際のサービスに利用する場合には Node.js などでネイティブ実装されている KDF を使用することを推奨する旨の記載が確認できた．HEC では KDF の使用が推奨されているものの，「選択できる（OPTIONAL）」な項目となっている．しかし，実際にサービスなどで利用する場合には KDF を用いない場合には，入力値のエントロピーに安全性が左右されるといった危険性が懸念されるため，KDF の使用は確実に行うべきであると考えられる．

5.　まとめ

本研究では，RFC9380「Hashing to Elliptic Curves (HEC)」に準拠した実装の調査と，実サービスにおける安全な運用方法について検討・考察を行った．まず，HEC の技術的構成やブレインウォレットとの比較を通じて，HEC が提供する安全性や実装上の利点を明らかにした．HEC においては，DST によるドメイン分離や KDF の導入によって，辞書攻撃やクロスプロトコル攻撃への耐性を向上することができる．

HEC における DST や KDF の利用実態の調査を行う前に予備調査として，CT ログおよびビットコイントランザクションに含まれる公開鍵において，辞書攻撃可能な鍵が存在するか調査を行った．調査の結果，パスワードリストの値に一致する公開鍵は確認されなかった．

主要な HEC 実装ライブラリの調査では，RFC に準拠した DST の使用は確認されたものの，独自の DST の使用や KDF の導入はされていなかった．予備調査の結果と合わせても，HEC が実際のサービス等に使われている段階にはないと考えられる．一方で，将来的に新たなサービス

に HEC が利用されるケースにおいて既存ライブラリの実装がそのまま利用されると，DST のパラメータの変更や KDF の導入を見落とす可能性がある．この場合，ブレインウォレット等と同様に辞書攻撃の対象となってしまうため，サービスの安全性に懸念が出てしまう．

HEC を安全に活用するためには，DST の適切な設定と KDF の確実な導入が不可欠である．DST についてはサービスやアプリケーションごとに独自の値を設定し，異なる用途間でのハッシュ衝突を防ぎセキュリティを高める必要がある．また，KDF についても，入力文字列のエントロピーが低い場合に備えて，Argon2 などを用いた前処理を導入することで，辞書攻撃への耐性を強化することが重要である．

参考文献

[1] Vasek, M., Bonneau, J., Castellucci, R., Keith, C. and Moore, T.: The Bitcoin Brain Drain: Examining the Use and Abuse of Bitcoin Brain Wallets, *Financial Cryptography and Data Security - 20th International Conference, FC 2016, Christ Church, Barbados, February 22-26, 2016, Revised Selected Papers* (Grossklags, J. and Preneel, B., eds.), Lecture Notes in Computer Science, Vol. 9603, Springer, pp. 609–618 (online), DOI: 10.1007/978-3-662-54970-4_36 (2016).

[2] Tippe, P. and Berner, M. P.: Evaluating Argon2 Adoption and Effectiveness in Real-World Software, *CoRR*, Vol. abs/2504.17121 (online), DOI: 10.48550/ARXIV.2504.17121 (2025).

[3] Biryukov, A., Dinu, D., Khovratovich, D. and Josefsson, S.: Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications, RFC 9106 (2021).

[4] Palatinus, M., Rusnak, P., Voisine, A. and Bowe, S.: BIP-0039: Mnemonic code for generating deterministic keys, `https://bips.dev/39/` (2013).

[5] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S. and Wood, C. A.: Hashing to Elliptic Curves, RFC 9380 (2023).

[6] Laurie, B., Langley, A. and Kasper, E.: Certificate Transparency, RFC 6962 (2013).

[7] Laurie, B., Messeri, E. and Stradling, R.: Certificate Transparency Version 2.0, RFC 9162 (2021).

[8] Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System, `https://bitcoin.org/bitcoin.pdf` (2008).

[9] bytemare: hash2curve, `https://github.com/bytemare/hash2curve`.

[10] drand: bls12381rs, `https://github.com/drand/bls12381rs`.

[11] armfazh: h2c-rust-ref, `https://github.com/armfazh/h2c-rust-ref`.

[12] zkcrypto: bls12_381, `https://github.com/zkcrypto/bls12_381`.

[13] paulmillr: noble-curves, `https://github.com/paulmillr/noble-curves/tree/main`.

[14] ethereum: py_ecc, `https://github.com/ethereum/py_ecc/tree/main`.

[15] herumi: mcl, `https://github.com/herumi/mcl`.

[16] Google: BoringSSL, `https://boringssl.googlesource.com/boringssl/`.

[17] Cloudflare: CIRCL, `https://github.com/cloudflare/circl`.